Addressing Weaknesses in the Domain Name System Protocol

Christoph L. Schuba and Eugene H. Spafford

COAST Laboratory

Department of Computer Sciences

Purdue University

West Lafayette, IN 47907-1398

{schuba,spaf}@cs.purdue.edu

ABSTRACT

Schuba, Christoph. M.S., Purdue University, August 1993. Addressing Weaknesses in the Domain Name System Protocol. Major Professor: Eugene H. Spafford.

The Domain Name System (DNS) is a widely implemented distributed database system used throughout the Internet, providing name resolution between host names and Internet Protocol addresses.

This thesis describes problems with the DNS and one of its implementations that allow the abuse of name based authentication. This leads to situations where the name resolution process cannot be trusted, and security may be compromised.

This thesis outlines the current design and implementation of the DNS. It states the main problem both on a high level and as applied to the DNS in a more concrete fashion. We examine the weaknesses in the DNS and exploit a method to abuse the DNS for system break–ins.

We demonstrate these weaknesses by describing the necessary modifications in authoritative DNS data and Domain Name System code. We list experiences gained during experiments with several setups of name servers and trusting hosts in a local area network.

Too weak assumptions during the authentication processes cause many security breaches. We state the security considerations in the official design documents and analyze the algorithms used in the DNS protocol looking for weak assumptions. Using a wide variety of criteria, we discuss several approaches to solve the main problem in the Domain Name System protocol. Two of these solutions, hardening the name server and using cryptographic methods for strong authentication, receive more attention than the other solutions.

DISCARD THIS PAGE

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

We would like to thank the German-American Fulbright Commission for a scholarship that made this work possible. Thanks to Steven Bellovin whose valuable comments are most appreciated and Dan Trinkle who showed us how to master some of the subtle difficulties of the DNS.

1. INTRODUCTION

The Internet is a widespread conglomeration of hundreds of thousands of interconnected heterogeneous networks and hosts. The design of the Internet is based on a protocol hierarchy. There exist multiple implementations of these protocols.

Computers communicate with each other on the basis of different types of addresses; on the physical layer using low–level physical addresses like Ethernet[1] card addresses, on the data link to presentation layer using host addresses such as IP addresses[2], and on the application layer using high–level, pronounceable host names.

One of the management tasks in the Internet is the mapping of lower level addresses to host names. A first naive approach is to collect all name–to–address mappings in a single file. That was also the first approach taken in the Internet. The file "HOSTS.TXT" contained the name–to–address mapping for every host connected to the ARPANET.

The task of naming hosts and network domains is addressed by creating a hierarchical relation between domains, with hosts as the furthest descendants from an artificial root domain. By appending the domain labels one after the other to the host labels on the path up to the root in the hierarchical tree, a unique, memorizable, and usually pronounceable identifier is created: the host name.

The mapping, or binding, of IP addresses to host names became a major problem in the rapidly growing Internet. This thesis does not deal with the mapping between addresses on the physical layer and transport layer, which is solved by ARP[3] in the UNIX[4] protocol suite, but with the mapping between host names and IP addresses.

[1]Ethernet is a registered trademark of Xerox Corporation

[2]"32-bit addresses assigned to hosts that want to participate in a TCP/IP internet" [Com91]

[3]"Address Resolution Protocol – used to dynamically bind a high level IP address to a low level physical hardware address" [Com91]

[4]UNIX is a trademark of AT&T Bell Laboratories

This higher level binding effort went through different stages of development up to the currently used Domain Name System. The Domain Name System, with its Berkeley UNIX implementation called BIND[5], is a distributed naming resolution system used by most network services available throughout the Internet. It works transparently for the user who sends email, accesses another host via "telnet" or "rlogin," or transfers some files via "ftp" from another site to his own machine. The Domain Name System provides name binding in both directions: given a host name, it returns the appropriate IP addresses, and vice versa.

Before hosts grant network services to users, an authentication process takes place, where the users' access rights, and the identity of connecting hosts get scrutinized, according to provider policies. These examinations are usually based upon identification by login name, password and host name. In some cases it is sufficient to provide the right names, and access is granted without specifying any password at all.

Some Berkeley "r–commands" offer network services for which it is sufficient to verify user name and host name to grant complete access. As the remote user name is specified by the connecting site, the authentication is based upon the name of the connecting machine. A machine that offers services can acquire information about the socket that is used by the connecting site. A socket is a tuple consisting of IP address, port, and protocol used by the remote site. To verify the host name, it is the task of the Domain Name System to map the IP address on the host name. We examine this case more closely later in this thesis.

Because the Domain Name System is distributed among many thousands of hosts, it can be a critical mistake to blindly trust the resolved binding. This thesis shows that under some assumptions it is no major effort to falsify the host name and authorization for a system.

Although this problem has been known for some years now, not many publications deal with it. [Bel90b] is the main paper we can mention as related work. It demonstrates the subversion of system security using the Domain Name System and

[5] Berkeley Internet Name Domain

discusses possible defenses against the attack and limitations on their applicability. An earlier paper by Steven Bellovin ([Bel89]) has already mentioned the possibility of abuse of the Domain Name System. That paper follows suggestions from Paul V. Mockapetris, the designer of the Domain Name System.

The main body of this thesis consists of three chapters followed by a final chapter drawing conclusions and giving suggestions for future work.

The first of these three chapters, Chapter 2, describes the position and role of the Domain Name System in its frame, the Internet. It gives a short historical sketch of the Internet and describes the Domain Name System on a high level. In that section we go into as much detail as necessary to build up the necessary background for the succeeding chapters. We introduce the technical terms and explain the mechanisms central to the understanding of the Domain Name System and the exploitation of its weaknesses. We give an example of a name resolution and the description of the data structures and algorithms used by name servers and resolvers.

Chapter 3 states precisely the main problem we are addressing. We explain the main problem in several stages, giving more details from section to section. First we describe the problem at a high level. Then we show the existence of the problem with the Domain Name System. We express the assumptions and examine the weaknesses in the Domain Name System that lead to the possibility of gaining unauthorized access to a certain type of remote host. In Chapter 3 we demonstrate the exploitation of the security flaws by giving details of an artificial setup that leads stepwise to an unauthorized login on another host. We close the chapter with experiences gained during our experiments.

Concluding the main body of this thesis, Chapter 4 analyzes the current security features in the Domain Name System and presents solutions to the given problem. The first part contains the security considerations in the RFC and a security analysis of the name server and resolver algorithms. Some of the solutions in the second part are already implemented and running in patched versions of system software, or are followed by organizational policies; others are still in an early stage of development.

Each of the solutions presented is discussed in this chapter and evaluated using a wide variety of criteria.

The approach, and its discussion, of combining partial solutions to a dense network, are part of the concluding chapter. Even if these interwoven solutions do not guarantee the security of a system, at least they increase the confidence in it.

2. THE DOMAIN NAME SYSTEM

This chapter describes the position and role of the Domain Name System in its frame, the Internet. We start off by talking about the Internet, the TCP/IP protocol suite, Internet services, routing, and finally the need for name resolution. It follows an outline of the historical development of the Domain Name System that led to the current system. We describe the design goals of the current system for name resolution in the Internet and its interacting entities. We also talk about forward and reverse mapping trees, and recursive and iterative resolving techniques. The following section contains some additional remarks about topics that were already mentioned but deserve a more detailed treatment.

Before describing the concrete data structures and algorithms used by name servers and resolvers we give an example of a name resolution. This example should provide a good understanding of the algorithms and the interaction of all participating entities in the distributed Domain Name System.

Wherever it is necessary to provide more specific descriptions of concepts or the implementation of the Domain Name System, we cover the respective topics in greater detail.

2.1 Introduction

To understand the role that the DNS plays, we start by introducing the Internet in general (see [Com91, Preface and chapter 1]).

Data communication has become a fundamental part of computing. Hosts gather information worldwide and their users want to exchange data and use remote services for different purposes. Common interests, shared by people that live and work thousands of miles away from each other, created the need for efficient and reliable data communication. What started before 1960 with the development of information theory, the sampling theorem, and the field of signal processing, became around the mid 1960s the question of how to transmit data packets in local area networks. The Internet contains and provides even more: internetwork technologies, protocol layering models, and datagram and stream transport services between hosts on possibly different networks, that together constitute an interconnected architecture that functions as a single unified communication system.

2.1.1 The TCP/IP Protocol Suite

The need and importance of internet technology was recognized by government agencies, which resulted in its development by DARPA[1]. The DARPA technology includes network standards that specify details and conventions of computer communication, network interconnection, and traffic routing. "TCP/IP[2]," an abbreviation of the official name "TCP/IP Internet Protocol Suite," can be used to set up communication between any set of interconnected hosts or networks. It is noteworthy that TCP/IP is one of many possible technologies that could be used to compose interconnected networks; one that has demonstrated its viability on a large scale.

2.1.2 Internet Services

Users are usually not interested in the underlying technologies of the Internet – their interest is the utilization of network services. The layered design of TCP/IP provides the necessary means for transparency in communication and hiding details from the high level applications. Services can be partitioned into application level internet services and network level internet services. Examples of application level services are electronic mail, file transfer, and remote login. The network level services "connectionless packet delivery service" and "reliable stream transport service" are used by the network application programmer and remain hidden from the application

[1] Defense Advanced Research Projects Agency

[2] named after its major standards TCP (Transmission Control Protocol) and IP (Internet Protocol)

end user. These two services are based on the transmission of data packets, units of data sent across a packet switching network. The collection of packets that belongs to one connection composes the data communication.

2.1.3 Packet Routing

Packets that are sent from one host to another usually have to traverse more than one physical link between these hosts. In a complex network with many thousands of machines it is not a trivial task to direct a packet from its source to its destination.

In an internet[3] there are specially dedicated machines that attach two or more networks and transmit packets from one to the other. These machines are called "gateways." While traversing the network from source to destination host, a message is likely to pass through one or more gateways. If the topology of the network allows several paths for the message to reach its destination, these gateways have to make decisions about which route to choose for the packet.

In a TCP/IP internet the basic unit of data transmission is the IP datagram. The process of choosing a path over which to send a datagram from source to destination is referred to as routing; any computer making such a decision is called a router.

Gateways in the function of routers compose a cooperative, interconnected structure. Datagrams originated at the source are passed from router to router until they reach a gateway that can deliver the datagram directly to its destination.

2.1.4 Name Resolution

Early systems supported point–to–point connections between computers and used low level hardware addresses to specify machines. Internetworking introduced universal addressing as well as protocol software to map universal addresses into low-level hardware addresses. There is also the notion of a host name — a high level address

[3] "Physically, a collection of packet switching networks interconnected by gateways along with protocols that allow them to function logically as a single, large, virtual network. When written in upper case, Internet refers specifically to the connected Internet and the TCP/IP protocols it uses." [Com91]

— a pronounceable identifier for hosts. The universal addresses can be mapped into host names.

Mapping processes can also be called "name binding" or "name resolution." This thesis is based on the name resolution process between high level addresses, the host names, and universally assigned lower level IP addresses.

Name resolution is a general concept. The current protocol in the TCP/IP protocol suite dealing with this concept and solving the problems that arise from it is the Domain Name System.

2.2 Historical Development

Around 1970, the ARPANET and the TYMNET were introduced. They were the first large–scale, general–purpose data networks that connected geographically distributed computer systems.

As the community contained only a few hundred hosts, name resolution was managed using a single text file: HOSTS.TXT. This file contained name–to–address mapping for every connected host. The administration, maintenance, and distribution was done by the SRI[4]– NIC[5].

Whenever some application had to resolve a host name and get the corresponding IP address, or vice versa, the resolver function called simply looked up the name (or IP address) in a local copy of the master HOSTS.TXT file and returned the associated value.

The enormous growth rate of the Internet was by no means predictable. Therefore it took several years until serious problems became apparent:

- System administrators used to e–mail changes to the NIC and periodically contact the SRI-NIC to obtain the latest copy of HOSTS.TXT. Network traffic and processor load became unacceptably high for the NIC.

[4]Stanford Research Institute in Menlo Park, California
[5]Network Information Center

- Names assigned to hosts have to be unique. As the NIC had no authority over host name assignments, name collisions became a problem.
- With the growth of the Internet and the irregularity of database updates the consistency of the name space was no longer guaranteed.

All of these problems arose because the original approach scaled poorly.

In 1984 the network community switched to the Domain Name System. Paul Mockapetris was responsible for the design of the architecture of the new system. The original RFCs[6] describing the Domain Name System are [Moc83a] and [Moc83b]. They have been obsolete since the release of the current specifications [Moc87a] and [Moc87b] in November 1987 ([LR93] and [BG92]).

2.3 Design Goals

The effort of designing the Domain Name System was directed towards several goals, which had the main influence on determining the current structure. The aim was to create a system with the following objectives in mind:

- Data Consistency
- Efficiency
- Distributed Character
- Generality
- Independence

P. Mockapetris states in [Moc87a] the design objectives that led to the current system:

[6] RFCs are a series of technical reports called Requests for Comments

2.3.1 Data Consistency

The primary goal was to provide a consistent name space to be used to refer to resources. In particular, the name space should not depend on any network identifiers, and therefore be totally independent of routing information or network topology.

2.3.2 Efficiency

The growth of the Internet in number of machines and subnetworks called for the introduction of a naming resolution system that could handle not only the immense volume of machines and resolution requests, but could also respond efficiently. To obtain these desired effects, the system was built in a hierarchical, distributed manner using the technology of caching.

In an internet, access to machines in local networks is more likely than remote access via many links. Therefore, far more name resolution requests are made locally. The knowledge about the requested bindings in the local network is available in the form of the local database. These facts suggests the use of the hierarchical organizational format in which local resolution requests are resolved efficiently by a local entity, and infrequent resolution requests about remote mappings are dealt with by an interaction of local and remote entities. The clear and clean structure that results in seeing the name space as a tree also favors this approach.

The creation of host names by appending node labels from the leaves to the root of this tree served the need for pronounceable, easily rememberable names for machines. The distributed arrangement of the system contributes to cutting the huge name space into pieces that can be managed efficiently. Caching information locally that was received from remote sites is another mechanism to obtain efficiency. Because of the dynamics of the system, the cached information is qualified with an additional time to live (TTL) parameter to ensure the goal of data consistency.

2.3.3 Distributed Character

The choice of implementing this large scale client–server paradigm in a geographically distributed set of machines was supported by the need for increased reliability through the existence of redundant data bases in secondary name servers. In the case of any kind of failure in one of the name servers for a zone, the redundant backup servers will still be able to provide the mapping service. Therefore the occurrence of a failure at a single site cannot lead to the denial of the resolution service.

Local authorities could administer their own domains and zones, keeping the data base consistent, providing autonomous control of name assignment, and taking away the load from central authorities. Authority passes down the edges of the tree, whereas information flows across the hierarchies from one host to another. The conceptual arrangement of domain name servers in a tree resembling the name structure is in fact a more realistic arrangement, namely a shallow tree.

2.3.4 Generality

Pragmatic reasons called for generality. Implementation costs and the amount of administrative effort in supporting the system dictated a general usefulness. Therefore the system does not contain any unnecessary restrictions regarding its purpose or applications. This goal can be reformulated as the desire to allow augmentation of the data basis by new data structures.

2.3.5 Independence

The system was designed to be independent of underlying hardware, be it of the local machine or the network interface. Furthermore, the transactions should be independent of the communication system that carries them. Therefore, all possible kinds of packet switching are suitable, such as store–and–forward switching using datagrams, virtual circuits, or possibly hybrid approaches.

2.4 DNS Entities

The Domain Name System consists of several entities: resolvers, name servers, and resource records (RR). We first describe the domain name space and resource records that are sections in DNS messages. They serve for the exchange of data between the interacting name servers and resolvers. We then describe purposes and features of name servers and resolvers.

2.4.1 Domain Name Space

The Domain Name Space is the specification of a tree–structured name space. The root of the tree is the root domain followed by its children, the top–level domains, which can contain several levels of subdomains. Figure 2.1 shows the structure of such a tree. Host names consist of a concatenation of the labels of each node on the path from the leaf that represents the actual host up to the root. Adjacent labels are separated by a dot. Domains are simply subtrees of the Domain Name Space. In our example "purdue.edu" is a domain name.

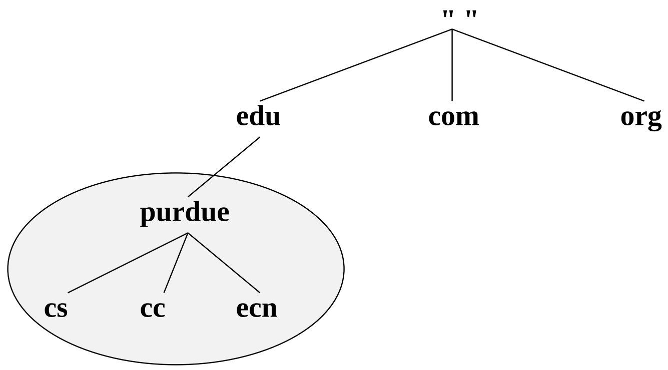

Figure 2.1 Domain purdue.edu

A part of the Domain Name Space that is controlled completely by a name server is called a zone. The delicate difference between a domain and a zone is that a zone contains all the domain names and data that a domain contains, except for the domain names and data that are delegated elsewhere (see Figure 2.2). Viewing the domains (nodes) and hosts (leaves) as the conceptual arrangement yields a tree with greater height than viewing the zones as nodes. The latter is a more realistic layout of the tree in terms of efficiency.

An example for the difference between domain and zone is the following scenario. A local authority manages the domain “alpha.dom”. “alpha.dom” has three subdomains “phi,” “chi,” and “psi” that contain several hosts, but no further subdomains. If the authority for subdomain “psi” is transferred to “psi.alpha.dom,” two zones are the result. The authority for “alpha.dom” could additionally transfer the authority for “chi” to the same authority that administers “psi”. This example shows that zones do not have to be connected by edges in the tree structured domain tree.

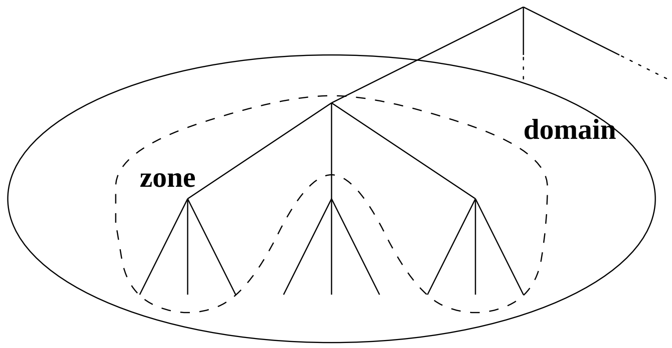

Figure 2.2 Domain vs. zone

2.4.2 DNS Messages

DNS messages are the data units that are transmitted between name servers and resolvers. A DNS message consists of the header and up to four sections (see Figure 2.3). The header contains the following fields:

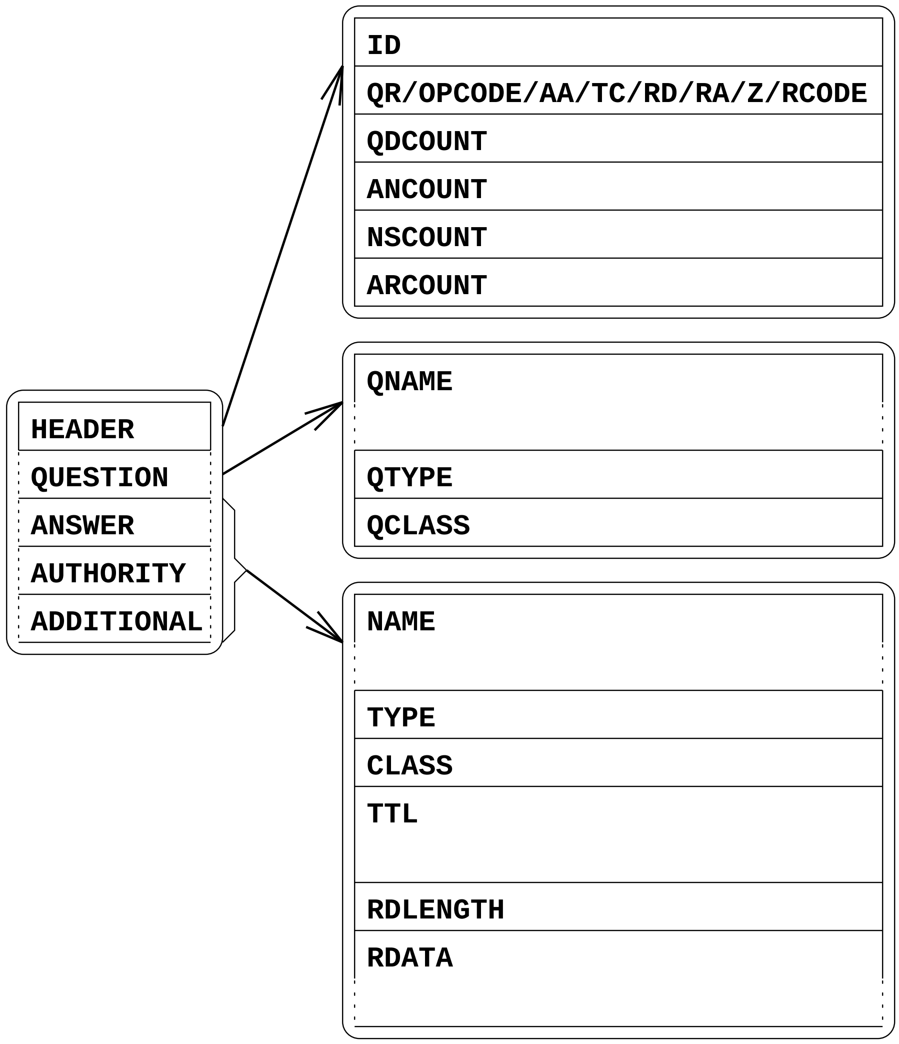

Figure 2.3 DNS message

- a 16 bit identifier is assigned by the program that generates any kind of query
- the "QR" bit specifies whether the message is a query (value 0) or a response (value 1)
- the "OPCODE" is a four bit field that specifies the kind of query in the message. It can contain the following values:
  - 0 for a standard query (QUERY)
  - 1 for an inverse query (IQUERY)
  - 2 for a server status request (STATUS)
  - 3 - 15 reserved for future use
- the next bit "AA" is only valid in a response and specifies that the responding name server is an authority for the domain name in the question section
- the "TC" bit specifies if a message was truncated
- the "RD" bit specifies if recursion is desired by a query
- the "RA" bit specifies if recursion is available
- the following three bits in the "Z" field are reserved for future use
- the last four bits determine the response code "RCODE". Possible values for the response code are:
  - 0 for "No Error Condition"
  - 1 to indicate a "Format Error"
  - 2 to indicate a "Server Failure"
  - 3 to indicate a "Name Error"
  - 4 to indicate that the requested feature is "Not Implemented"

  - 5 to indicate that the name server "Refused" to perform the specified operation
  - 6 - 15 are reserved for future use

- The following four unsigned 16 bit integer values specify the number of entries in the following question, answer, authority, and additional sections.

The contents of these four sections serve different purposes. The order of these section is always the same. Some of the sections can be empty in a DNS message. The format of the answer, authority and additional section is the same.

The question section carries query name, query type and query class. Valid query types are all the codes for resource record types, which we will explain in the following Section 2.4.3, and some more general ones for zone transfer, mail handling tasks, and wild–carding.

The following class mnemonics and values are currently defined:

- 1 for "IN" – Internet
- 2 for "CS" – CSNET
- 3 for "CH" – CHAOS
- 4 for "HS" – Hesiod
- 255 for wild–carding

The answer section carries resource records that directly answer the query, the authority section carries resource records that describe other authoritative servers, and the additional section carries resource records that are not explicitly requested but might be helpful in using the resource records in the other sections.

The authoritative section contains name server data in the following case: if a name server tries to resolve a name and he knows of an authoritative name server for the domain in which the name lies that has to be resolved, he puts the name server's name into the authority section of the reply. This is the approach in the DNS to refer clients to others servers in the not recursive mode.

The additional section plays an important role in the same case. If a name server refers a resolver to another name server, he better also provides the address of the other name server, because that is the next information the resolver needs in order to proceed with his queries. Another reason to have the additional section is to have space for extra, not requested information. If a resolver receives additional records, and caches them, he might be able to use them later. That would result in an increased performance of the system, because the resolution of data that is already in the local cache is considerably more efficient than a remote resolution that requires network traffic.

These three types of DNS message sections share the same format. They have:

- a name
- a type as in a query
- a class as in a query
- a 32 bit time to live field given in seconds (TTL)
- an unsigned 16 bit integer that specifies the length of the RDATA field in bytes
- a variable length string of bytes that describes the resource.

2.4.3 Resource Records

Data that is associated with the nodes and leaves of this tree is exchanged in the RDATA portion of the last three sections in a DNS message. These resource records are tagged according to the type of data they contain. We mention only those types that provide necessary information for understanding this thesis. A complete list of types and classes can be found in RFC 1035 ([Moc87b]).

- an "A" record contains a host address; a 32-bit Internet address when the class is "IN"

- an "NS" record specifies a host which should be authoritative for the specified class and domain
- an "SOA" record is the first entry in each of the database files and specifies a server to be the authoritative source of information within the domain
- a "PTR" record provides a pointer to another location in the domain name space
- an "HINFO" record identifies the CPU type and operating system type used by a host
- a "CNAME" record specifies the canonical or primary name for the owner – the owner is an alias
- a "MX" record specifies a host willing to act as a mail exchange for the owner name and a preference given among other resource records at the same owner
- an "X25" record contains a character string which identifies a public switched data network address
- an "ISDN" record contains a character string which identifies an ISDN[7] number of the owner and the DDI (Direct Dial In), if any

2.4.4 Name Servers

The whole database is divided into zones that are distributed among the name servers. The essential task of a name server is to answer queries using data in its zone. To ensure a higher degree of reliability of the system, the definition of the Domain Name System requires that at least two name servers contain authoritative data for a given zone. Some sites run more than two name servers: one of them usually outside of the affected network to guarantee name service if the network is unreachable for some reason. The main name server is called the primary name server,

[7]Integrated Services Digital Network

Table 2.1 Subset of QTYPEs

QTYPE	value	meaning
A	1	a host address
NS	2	an authoritative name server
SOA	6	start of authority
PTR	12	a domain name pointer
HINFO	13	host information CPU and OS
CNAME	14	canonical name (alias)
MX	15	mail exchange
X25	19	public switched data network address
ISDN	20	integrated services digital network

and the backup servers are called secondary name servers. Secondary authoritative name servers update the data base for their zone periodically with data polled from their primary servers. Primary name servers load the database files provided by the zone administrator and maintain a cache of data that was acquired through resource records. Servers want to adapt dynamically to changes in the setup of the name space of other authorities. Therefore, each resource record contains a time to live field which ensures that name servers do not cache data without time bound.

The actual algorithm name servers use depends on the local operating system and data structures used to store resource records. A basic outline can be found in [Moc87a, section 4.3.2] and in section 2.9.2 of this thesis.

2.4.5 Resolvers

The interface between the Domain Name System and user programs is the name resolver. In the simplest case, a resolver receives a request from a user program in the form of a system call or subroutine call and returns the desired information. The resolver is located on the same machine as the user program, but contacts one or more name servers on (usually) remote machines if the requested data is not obtainable from the local cache.

The typical resolver–client interface has a triple functionality: host name to IP address translation, IP address to host name translation, and a lookup of general information specifying query name, type, and class. The following results can be obtained after the resolver performed the indicated function: the data requested, a name error in case the referenced name does not exist, or a data not found error.

To obtain higher efficiency, it is reasonable to have all resolvers on one machine share their cache. An algorithm outline for the resolver can be found in [Moc87a, section 5.3.3] and in section 2.9.3 of this thesis.

2.5 Forward and Inverse Mapping Tree

The Domain Name Space consists of a hierarchy of domain names. As the decimal numbers in the dotted quad notation for IP addresses can be viewed as names, it is only one step to construct a tree that consists of these numbers as domain names. This inverse mapping tree is mounted on the domain in-addr.arpa. The IP address 128.46.152.78 for zoo.ecn.purdue.edu has the corresponding name 78.152.46.128.in-addr.arpa which maps back to zoo.ecn.purdue.edu (see Figure 2.4).

The reason for the numbers of the IP address appearing in reverse order in the reverse mapping tree is the following: Domain names read from left to right get less specific, whereas IP addresses get more specific from left to right (see Figure 2.5). The task of delegating authority for in-addr.arpa domains to zone administrators would be impossible if the entries appeared in the original order.

In case someone wanted to index an arbitrary piece of data in the domain space (something aside from IP addresses or host names), an additional subdomain such as the in-addr.arpa domain is necessary. A so called inverse lookup (an exhaustive search of the whole domain name space), is also possible, but not feasible for regular usage. Any one name server only knows about part of the overall domain name space.

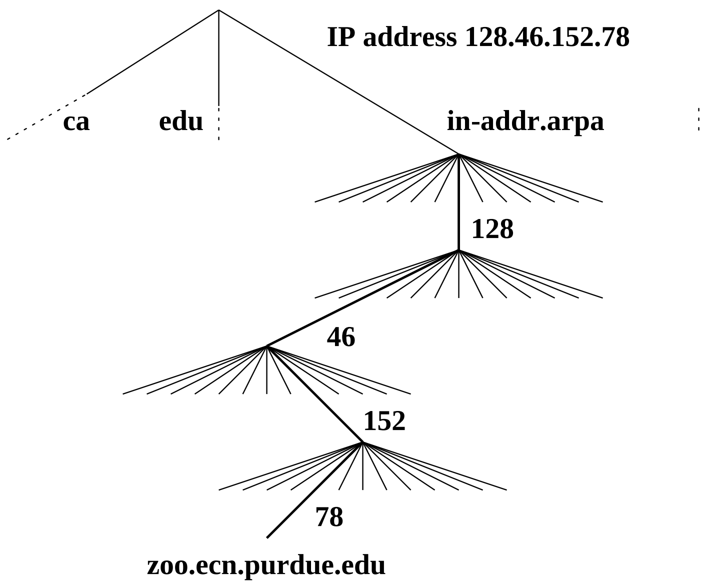

Figure 2.4 The in-addr.arpa domain

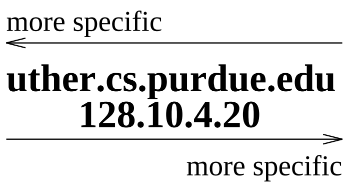

Figure 2.5 Degree of specification

Therefore, an inverse query is never guaranteed to return an answer. If a name server receives an inverse query for an IP address it knows nothing about, it cannot return an answer; but it also does not know if the IP address does not exist, because it has only its part of the DNS database to work with. Additionally, the implementation of inverse queries is optional according to the DNS specification.

2.6 Recursion and Iteration

When there is the need for resolving a name in the Domain Name System, the following steps are taken. Whoever wants to resolve a name invokes a local client program, the resolver. The resolver formulates a query according to the DNS protocol and contacts its local name server.

These queries can come in two different flavors: "recursive" and "iterative".

In recursive resolution, a resolver sends a recursive query to a name server. The queried name server then has the obligation to respond with the answer to that query or with an error code. The name server cannot refer the resolver to another name server. In case the queried name server is not authoritative for the requested data, it has to resolve the query again; recursive or iterative. Current implementations resolve the query iterative and do not pass the work to another server.

Iterative resolution does not require nearly as much work on the part of the queried name server. In iterative resolution a name server simply returns the best answer it is capable of giving. No additional querying of other name servers is required. The queried name server only consults its local data looking for the data requested. If the data is not there, it makes its best attempt to give the querier data that will help it continue the resolution process. This data usually contains names and addresses of name servers that are "closer" to the data its seeking.

After possibly many referrals, the local name server queries the authoritative name server, which returns an answer or an error code.

2.7 Filling in the Blanks

This section contains features that were briefly touched in the previous sections, but that need further explanations: the central role of caches for system performance enhancement, the role of administrative authorities, and the types of errors that can occur during name server operation.

2.7.1 Role of Caches

The whole resolution process may seem convoluted and cumbersome compared to simple seeks through a host table database. However, it is fast, speeded up considerably by caching.

As our example in Section 2.8 shows, name servers may need several DNS messages to find the answer to a query. During successive resolution attempts name servers discover information about the Domain Name Space. This information can be used for future resolutions. If a name server caches the data, it builds up a data base that helps speed up the processing of further querying. The next time a resolver queries the name server for data about a domain name the name server knows something about, the process is shortened considerably. Even if a name server does not have the answer to the query in its cache it might have learned the identities of the authoritative name servers for the zone the domain name is in, and it might be able to resolve them directly.

It is difficult to determine the optimal time to live value for data that is to be cached. There is a trade-off between enhanced performance once data is cached and the possibility that the cached data might be out of date by the time it is used.

2.7.2 Role of Authorities

Manageability of the administration of the Domain Name Space is an important issue because of the large number of hosts in the Internet. The key concept to solve this problem is the delegation of authority along the edges of the Domain Name Space tree. Local authorities administer their own zones. They keep the data base consistent and have autonomous control of name assignments. This delegation scheme takes away the load from central authorities.

It is important to understand that the organizational tool of delegation of authority includes the responsibility for the delegated entity. There is no delegation without responsibility.

2.7.3 Occurrence of Errors

Several error situations can occur during name server and resolver operation. The header section of every DNS message contains the field "RCODE," a 4 bit field that is part of a response (see section 2.4.2). The contents of the "RCODE" field determines which error has occurred while processing the query:

- if a name server is unable to interpret a query, it flags a "Format Error"
- if a name server is unable to process a query because of a problem with that server, it flags a "Server Failure"
- if an authoritative name server for a zone determines that the referenced name does not exist, a "Name Error" is flagged.
- if a server does not support the requested kind of query, it returns a "Not Implemented" error
- if a name server does not want to provide the information a resolver asked for in a query, it returns the "Refused" code. This is one example of the server refusing to perform a specified operation for policy reasons

2.8 Example: Name Resolution

This section contains a simple example for a name resolution using a mechanism based on the client–server paradigm. A generic resolution example is shown in Figure 2.6 with a short explanation of the steps in table 2.2.

A resolver forms a query of some kind and wants to retrieve the response containing the answer to its query from the name server A. This name server A could be running on the same host with the resolver software, on a host in the local network of the resolver, on a host somewhere in the net, or on one of the hosts serving the root domains. Assuming that A does not know the requested information, it tries to retrieve it from other name servers. The selection of which name servers to contact

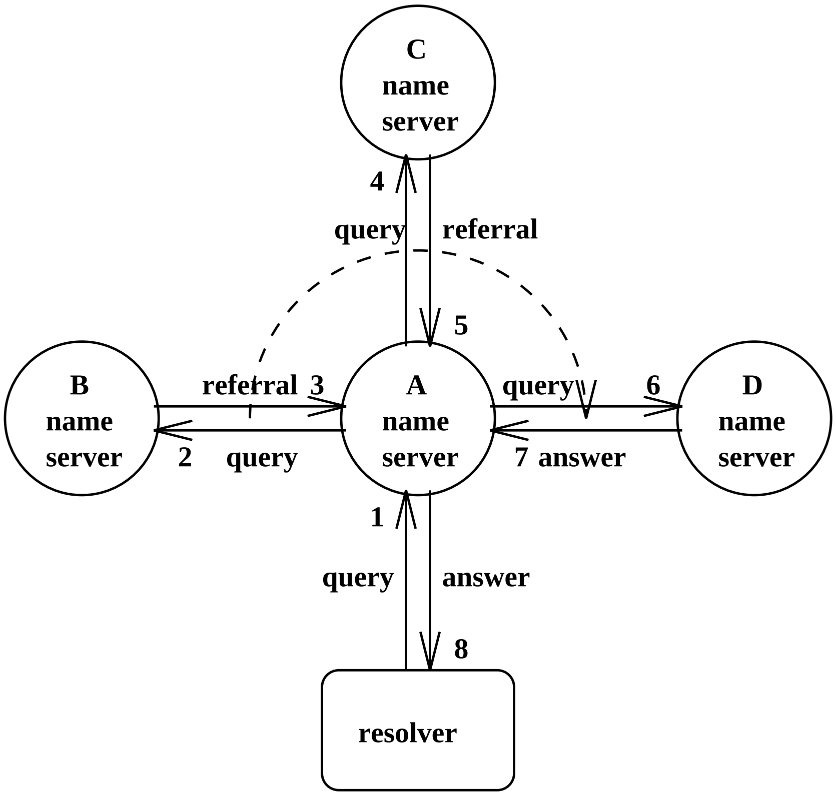

Figure 2.6 Example name resolution

depends on the name to be resolved. The decision process about this choice is given in sections 2.9.2 and 2.9.3 where we explain the algorithms used by name servers and resolvers.

The contacted name servers return an answer to the query to the requesting name server, or they return a referral to another name server that is more likely to know the answer. We neither consider the occurrence of exceptions or errors in this example, nor caching issues. Possible return codes in responses are given in section 2.4.2 and are further explained in section 2.7.3.

As soon as one of the contacted name servers returns an answer to A, A responds to the original query of the resolver with the retrieved answer.

Table 2.2 Example steps in name resolution

Step	Action
1	Name server A receives a query from the resolver
2	A queries B
3	B refers A to other name servers, incl. C
4	A queries C
5	C refers A to other name servers, incl. D
6	A queries D
7	D answers
8	D returns the answer to the resolver

2.9 The Domain Name System Protocol

The official design documents [Moc87a] and [Moc87b] state and describe concepts and facilities, implementation and specification. In the following sections, we will discuss topics related to the data structures and data organization, and present the name server and the resolver algorithm on a fairly high level. We get into more detail where it is necessary to examine the weak points of the protocol.

The data structures and the algorithms are the basis for the analysis of the protocol later in this thesis.

2.9.1 Data Structures

Two principal kinds of data appear in the Domain Name System: zone data and cache data.

A zone contains a complete database for a particular pruned subtree of the domain name space. This data can be authoritative if it is the original database managed for this particular zone by a primary or secondary name server. Otherwise it is non–authoritative data. Secondary servers maintain zone data as copies from the master files. Name servers check periodically for changes (for a changed serial number in the SOA records) and update their data by reading the master files, or via zone transfer operations.

As we will describe in Section 2.3.2, the technology of caching is a key concept in the Domain Name System. The cached data usually represents only an incomplete view of zone information. It improves the performance of the retrieval process when non–local data is repeatedly accessed. Zone data is eventually discarded by a timeout mechanism.

The implementation of the Domain Name System is not limited to a certain data structure, but is free to choose any internal data structure. However, it is suggested by the standard that a separate instance of the data structure be used for each zone, a data structure for the catalog, and one for the cached data. It is important that resolver and name server can concurrently access the same cache when they are on the same machine. In Section 2.10.1 we go into more detail on this point.

2.9.2 Name Server Algorithm

The implementation of the name server algorithm, which is given in Figure 2.7 depends on the local operating system and data structures used to store RRs. The algorithms of the name server and the resolver assume an organization of the data as described in the previous section: several tree structures, one for each zone.

In the following presentation of the algorithm we stay close to the outline specified in [Moc87a].

1. Set or clear the RA bit in the response depending on whether the name server is willing to provide recursive service. If recursive service is available and requested via the RD bit in the query, branch to step 5, otherwise step 2.

2. Search the available zones for the zone which is the nearest ancestor to the queried name. If such a zone is found, branch to step 3, otherwise step 4.

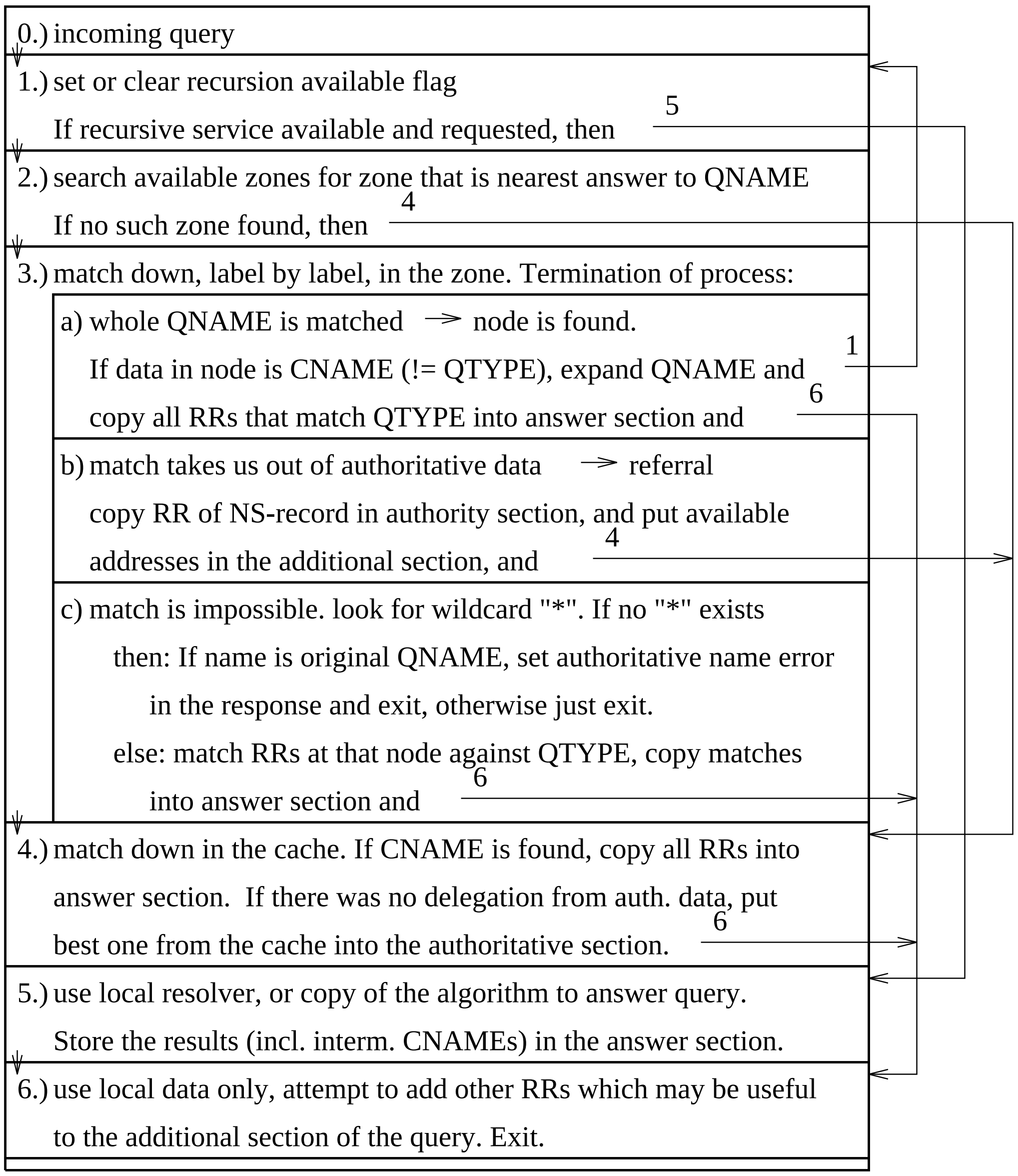

Figure 2.7 Name server algorithm

3. Start matching the name in the zone, label by label. The matching process can terminate several ways:

(a) If the whole queried name is matched, we have found the node.

If the data at the node is a canonical name, and the queried type was not CNAME, copy the canonical name resource records into the answer section of the response, change the queried name to the canonical name in the CNAME RR and go back to step 1.

Otherwise copy all resource records which match the queried type into the answer section and go to step 6.

(b) If a match would take us out of the authoritative data, we have a referral. This happens when we encounter a node with name server resource records marking cuts along the bottom of a zone.

Copy the name server resource records for the subzone into the authority section of the reply. Put whatever addresses are available into the additional section, using glue resource records if the addresses are not available from authoritative data or the cache. Go to step 4.

(c) If at some label, a match is impossible, look to see if a "*" label exists.

If the "*" label does not exist, check whether the name we are looking for is the original name in the query, or a name we have followed because of a CNAME. If the name is original, set an authoritative name error in the response and exit. Otherwise just exit.

If the "*" label does exist, match resource records at that node against the queried type. If any match, copy them into the answer section, but set the owner of the resource record to be the queried name, and not the node with the "*" label. Go to step 6.

4. Start matching down in the cache. If the name is found in the cache, copy all resource records attached to it that match the query type into the answer section. If there was no delegation from authoritative data, look for the best one from the cache, and put it into the authoritative section. Branch to step 6.

5. Use the local resolver or a copy of its algorithm to answer the query. Store the results, including any intermediate canonical names, in the answer section of the response.

6. Use local data only, attempt to add other resource records which may be useful to the additional section of the query. Exit.

2.9.3 Resolver Algorithm

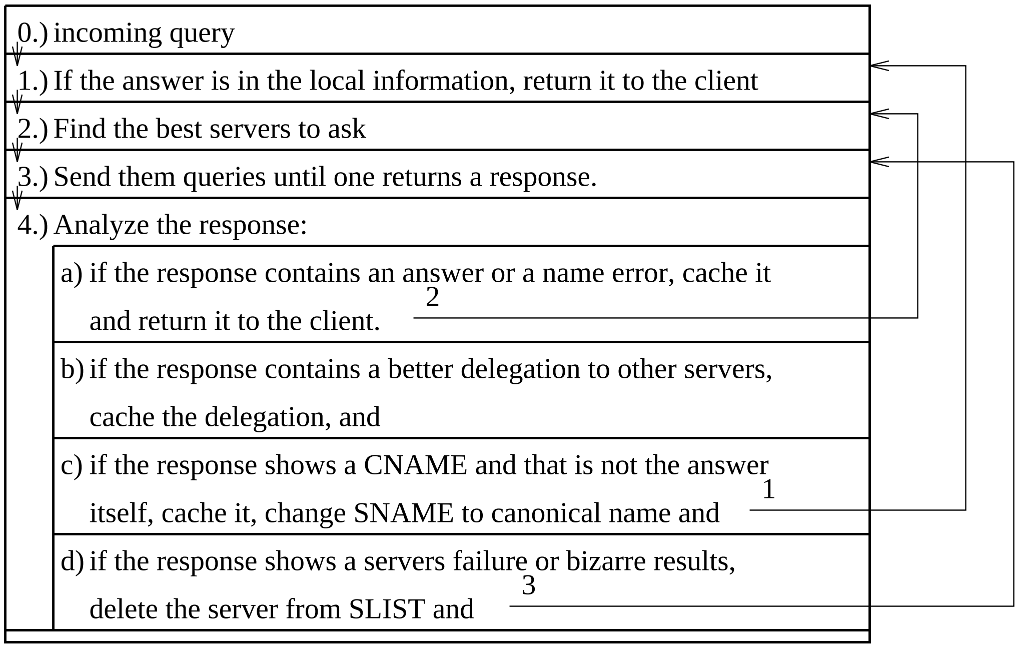

Figure 2.8 Resolver algorithm

The resolver acts as the interface between a user program and the name server described in Figure 2.9 and performs three main actions to map the query to an answer. The algorithm (see Figure 2.8 and the following list for details) tries to find the information locally first. If that does not succeed, it sends the query to the best server to ask. As soon as a reply returns, it checks for answer, name error, delegation, canonical name expansion, or failure of the server and reacts properly. The following steps describe the algorithm in more detail. They are derived from [Moc87a]:

1. See if the answer to the query is in the local information, and if so, return it to the client.

2. Find the best servers to ask.

3. Send them queries until one returns a response.

4. Analyze the response:

   (a) if the response answers the question or contains a name error, cache the data as well as return it to the client.

   (b) if the response contains a better delegation to other servers, cache the delegation information, and go to step 2.

   (c) if the response shows a CNAME which is not the answer itself, cache the CNAME, change the queried name to the canonical name in the CNAME RR and go to step 1.

   (d) if the response shows a server failure or other bizarre contents, delete the server from the server list and go back to step 3.

2.10 Interaction of Name Server and Resolver

Name server and resolver interact mainly by passing data back and forth. There is at most indirect control flow at step five in the name server algorithm (see Section 2.9.2). In the case that a resolver requests recursive name resolution and the name server provides this service, the name server passes the query to the local resolver. This can be seen as pure data flow, but because the execution of the whole query is passed to the resolver, we interpret it as control flow.

2.10.1 Data Flow

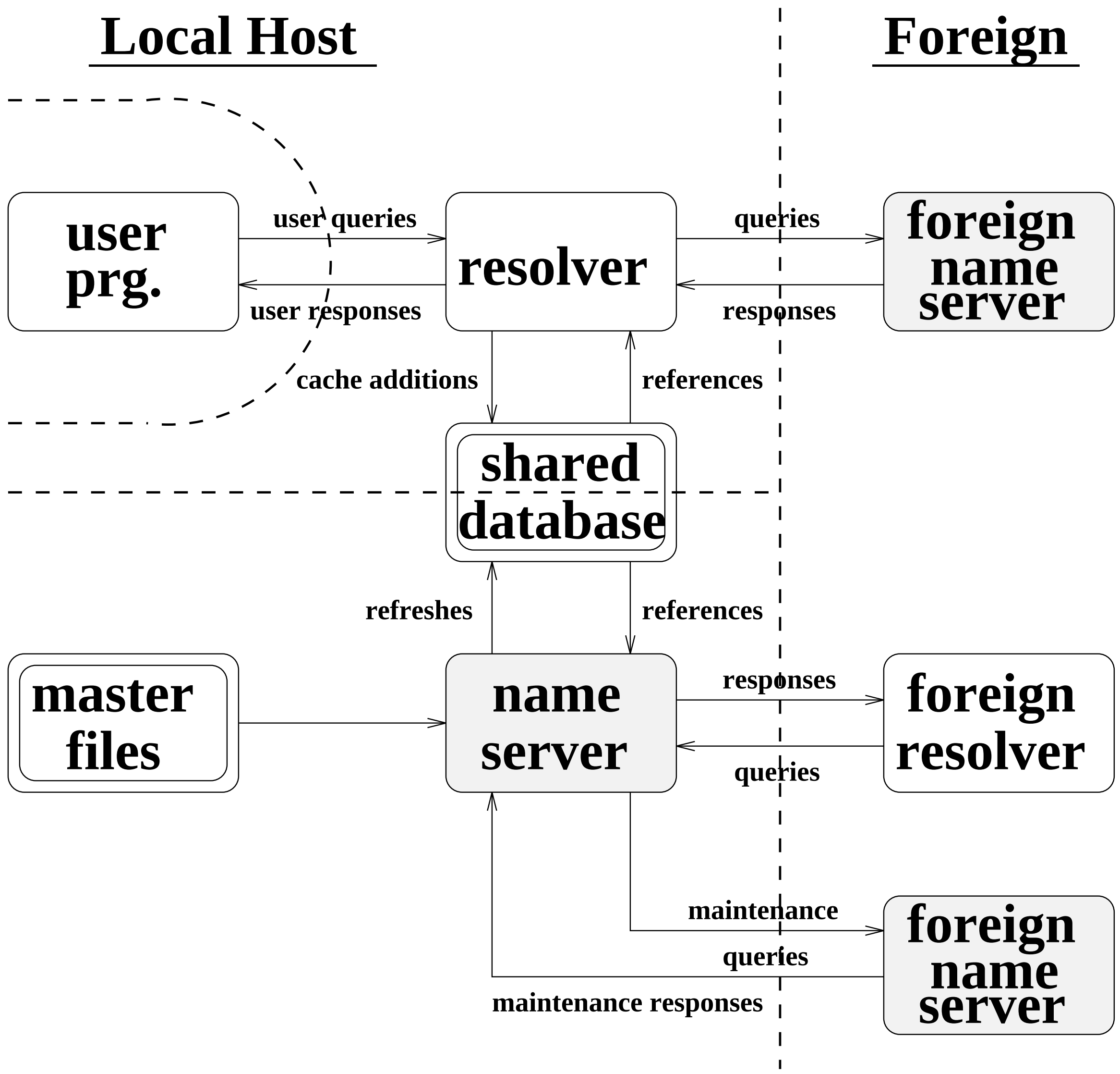

Figure 2.9 Data flow between DNS entities

The data flow between Domain Name System entities is not limited to simple queries and responses, illustrated in Figure 2.9. We distinguish among four parts that interact with each other: the user program, the resolver, the name server, and an unknown subnet that can contain foreign name servers and resolvers.

User program and resolver exchange user queries and user responses. In the BIND implementation of the Domain Name System, this exchange is done by calling the system calls "gethostbyaddr()" and "gethostbyname()". As can be seen here, the usage of the Domain Name System is completely transparent to the user who requests name resolution. The same system call interface can be used when the Domain Name System is replaced by another mapping mechanism (for example static mapping).

Local resolvers communicate with foreign name servers via the exchange of queries and responses, as does a local name server with foreign name servers or resolvers. Queries are always sent to a name server and responses go the reverse direction. When name servers communicate, they exchange zone data or maintenance queries and responses. Under the assumption that the local name server is a primary server, it gets its primary zone data from the master files.

Both name server and resolver usually maintain a cache. It is not unusual for a name server and a resolver that run on a single host to share this database.

2.10.2 Shared Information

A shared cache can be accessed by resolver and name server. Resolvers provide as cache additions whatever they learn from the responses to their queries. They also consult the cache and retrieve data from it. Name servers also reference the cache to answer queries and provide refreshes from local authoritative data.

A database that is shared concurrently by many processes must be protected by synchronization mechanisms. The additional complexity in dealing with the problems a shared database brings with it is amortized by the gain in performance and efficiency of the system in total. It is obvious that successful lookups in the local cache are preferred over sending queries to remote machines with no bounds on how long it will take them to reply. Maintaining a larger cache shared between two entities increases the probability of finding a match in the cache.

3. DESCRIPTION AND DEMONSTRATION OF WEAKNESSES

This chapter concentrates on the description and demonstration of the central problem of this thesis.

We first give an abstract statement of the problem. We state it again in the following section, but in a more concrete fashion directly related to the Domain Name System. We talk about the general features in the Domain Name System that facilitate the exploitation of the problem.

The following section gives details of regular remote machine access and several approaches of how to exploit the problem to gain unauthorized access. We then talk about our implementation test environment and describe the experiments we performed to support the claim that this security flaw is exploitable. The concluding section of this chapter presents the experiences we gained from our experiments.

Figure 3.1 shows the setup of machines and their names. It serves as a running example in this chapter. A detailed description of this setup is given in Section 3.5.1.

3.1 Statement of the Problem

Authenticity is based on the identity of some entity. This entity has to prove that it is genuine. In many network applications the identity of participating entities is simply determined by their names or addresses. High level applications use mainly names for authentication purposes, because address lists are much harder to create, understand, and maintain than name lists.

Assuming an entity wants to spoof the identity of some other entity, it is in some cases enough to change the mapping between its low level address and its high level name. That means that an attacker can fake the name of someone by modifying the association of his address from his own name to the name he wants to impersonate.

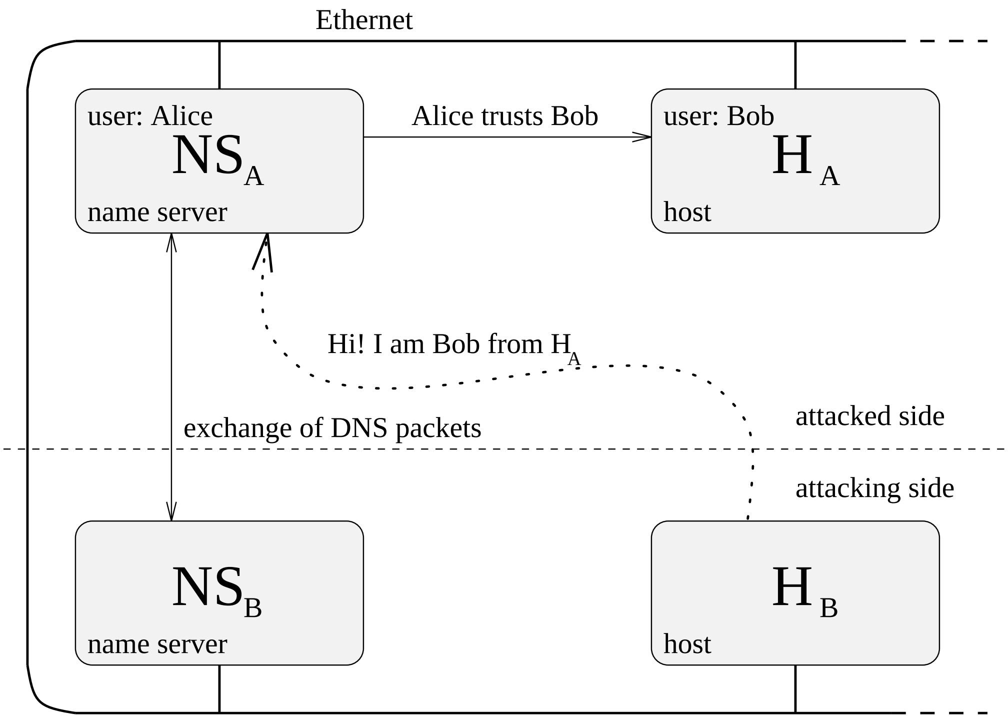

Figure 3.1 Experimental setup

Once an attacker has done that, an authenticator can no longer distinguish between the true and the faked entity.

This describes the fundamental problem on which this thesis is based. If the binding process between names and addresses cannot be trusted fully, no one can rely on an authentication process on a high level.

3.2 The Problem in the DNS

Many security problems of the TCP/IP protocol suite rely on the ability of the attacker to spoof the IP address of a trusted machine, as described in [Bel89]. As hosts trust each other, usually on the basis of host names, an attacker can take the easier approach and spoof a host's name instead of its IP address.

If a host named H_A accesses another host named NS_A, host NS_A accepts the connection and retrieves address information about the connecting host H_A. Host NS_A reads host H_A's IP address and converts it into a regular host name. To bind the right name to the IP address, host NS_A starts a Domain Name System query in the reverse lookup tree.

For a pair of machines NS_B and H_B under the power of an attacker, with NS_B running a primary name server for a certain zone, and H_B trying to fake H_A's identity, it is easy to make NS_A believe H_B was H_A. H_B connects to NS_A and claims to be H_A, NS_A retrieves H_B's IP address 111.22.33.4 and queries the name 4.33.22.111.in-addr.arpa from the Domain Name System. One single entry in the authoritative data for the reverse lookup tree for NS_B's zone specifies the IP address–to–name mapping between 4.33.22.111.in-addr.arpa and H_B. If the attacker replaces this line by a mapping between 4.33.22.111.in-addr.arpa and H_A, NS_A's resolution attempt will finally grant H_B access to NS_A.

This shows the simplicity of an attack that is based upon trust placed in the data provided by DNS. It is based on a weakness in the DNS, not an easily fixable bug in the implementation of a particular network service.

One widely accepted way of dealing with this problem is the Berkeley software patch described in section 4.5. However, adding an additional Domain Name System query of the determined host name to the server code and comparing the returned IP addresses against the original IP address for a match only adds to the quality of security; it does not provide complete security. An attacker can piggyback additional resource records to the answer packet to the first query. Doing so, the attacker poisons the victim's cache with false information, such that the forward lookup would not disclose the attack. In Section 3.5.6 we go into more detail on this issue when we describe our concrete approach of cache corruption.

3.3 Weaknesses

In this section we describe the conditions that must hold to facilitate a break–in. The Domain Name System is weak in several places. We examine the problems of name–based authentication processes, trusting information that comes from an untrustworthy authority, and accepting additional, possibly incorrect information that was not requested, but that seems to provide advantages for runtime performance.

3.3.1 Assumptions to Facilitate Break–ins

In our setup we assume that the attacker has complete control over machine NS_B running a legitimate primary name server for a DNS zone. This strong assumption does not always need to be satisfied. It is simply the easiest way for an attacker if he controls a primary name server, because of its capabilities and the fact that other machines believe name servers.

Depending on the topology of a real network it is sufficient if an attacker controls one of the authoritative name servers for the particular zone; the one that is queried first by the remote resolver. It is not much easier for an attacker to satisfy this second assumption than the first one.

The control must include the associated inverse mapping tree. The attacker might have successfully subverted such a machine or simply be a renegade system administrator. Both have happened in the past (i.e. [Sto89, Mad92]).

We can relax this assumption further. If an attacking machine manages to somehow obtain the ID number of a current DNS query to a legitimate name server, it could run some code (e.g. a tool that constructs the response packet and uses the source route option to send it to the originator of a query) to answer the query and supply additional records to poison the cache. The ID number prediction could be based on previously received queries and knowledge on how a resolver modifies the identifier. An attack based on TCP sequence number prediction to construct a TCP packet sequence that allows an attacker to spoof a trusted host's identity on a local network was described in [Mor85]. This example shows the feasibility of ID number prediction.

In the following discussion we will assume that the attacker has indeed superuser access to a primary name server. With that assumption in place we decrease the complexity of the following discussions.

3.3.2 Authentication via Host Names

We explained in the introduction that users have to be authorized by network service providers before they can use the service. This authentication is usually based on the verification of the user's login name along with the associated password and the host name of the machine on which the user starts his requests. Networks may be classified into different partitions[1]: Closed Networks, Open Networks, and Trusted Networks [PL91]. Closed Networks can be accessed only within certain boundaries. Sessions are controlled and secured in accordance with the rules implied by an organization's business goals. In a Closed Network, the location of all resources is well known and specified.

Open Networks are regions separated by boundaries from their surroundings, but the transfer of information across these boundaries is admitted. They are augmented by publicly accessible parts or connections to networks owned by other companies or organizations. These two extensions make this type of network vulnerable to external threats.

Trusted Networks introduce the concept that network access is controlled at the entry node. In the case of large international networks, maintainability and controllability are important issues. Adopting the Trusted Network concept allows the decomposition of a large network, growing towards an unmanageable complexity, into relatively small national or regional networks, each supported by local staff, and each provided with its own network access control. The advantages are increased controllability, maintainability, manageability, and simplification of change management. A

[1]A very similar classification is applicable to systems in general.

Trusted Network can be regarded globally as a single Closed Network, but from a local point of view, the interconnected networks stand widely open with all the applicable security threats.

The Internet is a system of Trusted Networks within Open Networks. This allows the danger that once someone has falsely gained access to one machine, it is much simpler to subvert others. Within Trusted Networks users are authenticated solely by their login name and connecting host name. The login name is specified by the connecting site, and therefore can be falsified, such that the only "reliable" information left for the addressed machine is the connecting machine's IP address that is provided by an operating system call. The addressed machine then maps the IP address into a host name using the Domain Name System. If an attacker manages to subvert this name binding call, he can falsify the name of a machine within the Trusted Network and therefore succeed in his attack.

3.3.3 Trusting a Not Trustworthy Source

Using the Domain Name System to map the IP address provided by lower level protocol layers into the applicable host name, the addressed host blindly trusts the information that is provided by the Domain Name System. Information that comes from sources outside of the trusted area is trusted. That is a severe violation of the partitioning concept. Only truly authoritative information should be trusted.

3.3.4 Believing Additional, Not Authoritative Information

Efficiency is one of the stated goals of the Domain Name System, as we saw in Section 2.3.2. The DNS packet contains an additional answer section (see Figure 2.3), where name servers can provide resource records containing information that could come in handy in future requests, but that were not explicitly requested. There are situations where these additional records yield in system efficiency, for example after the lookup of "NS" records when "A" records specifying the addresses of the queried name servers are found in the additional answer section. That saves the lookup of the IP addresses, once the name of the applicable name server is found. Additional resource records are cached for future use.

As we rely on the correctness of these additional records once we use them, we trust information that comes from a source possibly outside of the trusted scope. That is another violation of the partitioning concept.

3.4 Exploiting the Flaws

The following sections are the most concrete description of how to exploit the security flaw in the Domain Name System. In this chapter we concentrate on the "rlogin" command of Berkeley UNIX. We do not explain the whole "rlogin" protocol in detail, but only state the parts and commands that are related to our interest.

3.4.1 Regular Access

Table 3.1 Regular access

host NS_A (rlogind)	Bob@H_A
	rlogin NS_A *-l Alice*
`getpeername()` $\rightarrow IP_{H_A}$	
`gethostbyaddr(`IP_{H_A}`)` $\rightarrow H_A$	
find entry H_A Bob in ~Alice/.rhosts	
grant access	

Table 3.1 gives the procedure followed during a regular remote login. Time proceeds from top to bottom of the table. User Bob on machine H_A wants to log into machine NS_A. The underlying protocols create a connection between the "rlogin" program and the "rlogind" daemon. During the authentication process the daemon retrieves the IP address of the connecting machine: IP_{H_A}. It then uses the Domain

Name System to map this address to a host name. The call of "gethostbyaddr(IP_{H_A})" does that and returns H_A.

The daemon then checks whether the user from the machine with name H_A is allowed access by scanning the entries in the ".rhosts" file of user Alice. If the appropriate entry is found, access is granted. If the system administrator of system NS_A has installed the "/etc/hosts.equiv" file and entered the name of host H_A, then access is granted even without a user maintained entry in file ".rhosts."

3.4.2 The "Database Modification" Approach

Table 3.2 The "Database Modification" approach

host NS_A (rlogind)	Bob@H_B
	rlogin NS_A *-l Alice*
`getpeername()` $\rightarrow IP_{H_B}$	
`gethostbyaddr(`IP_{H_B}`)` $\rightarrow H_A$	
find entry H_A `Bob` in ~Alice/.rhosts	
grant access	

This is the first example of how an attacker can spoof someone else's host name. Host H_B behaves as if it were host H_A. The access pattern is very similar to the previous, regular one, except that the call of "getpeername()" now returns the IP address of host H_B. If the DNS database is modified by the attacker, the call of "gethostbyaddr()" does not return the name H_B as it would with a database in an unimpaired state, but the name H_A. Bob@H_B finally gets access to NS_A.

3.4.3 The "Cache Poisoning" Approach

In this approach the "rlogind" daemon tries to enhance security by calling the function "gethostbyname()" to verify the mapping from IP_{H_B} to H_A. The attacker

Table 3.3 The "Cache Poisoning" approach

host NS_A (rlogind)	Bob@H_B
	rlogin NS_A *-l Alice*
`getpeername()` $\rightarrow IP_{H_B}$	
`gethostbyaddr(`IP_{H_B}`)` $\rightarrow H_A$	
and $H_A \rightarrow IP_{H_B}$ mapping	
`gethostbyname(`H_A`)` $\rightarrow IP_{H_B}$	
find entry H_A Bob in ~Alice/.rhosts	
grant access	

however has a way of subverting this additional security feature. He can send the additional mapping of H_A to IP_{H_B} along with the answer to the query for IP_{H_B}. By the time the daemon calls "gethostbyname()," it already has the necessary mapping information in its cache. The daemon believes the cached data and again grants the attacker access.

3.4.4 The "Ask Me!" Approach

In the previous sections we exploited the security weakness of the Domain Name System according to S. Bellovin's suggestions.

We thought of another way to exploit the weakness. If some entity sent a source routed datagram, containing a DNS message with false additional resource records to a name server, would that name server accept the data? The idea here is to poison a name server's cache with all necessary information (for reverse and forward lookup) before the "rlogin" attack is launched.

We will explain in Section 4.1 why this cannot work using source routed DNS messages directly. This deprives us of the chance of eliminating the basic assumption of the attacker having superuser priority on a primary name server in order to launch an attack.

Nevertheless, the idea can be exploited in another way, on a higher level, and far more elegantly than creating and sending datagrams manually. Imagine the following scenario:

The attacker on name server NS_B whishes to give NS_A wrong information about the mappings

- $IP_{H_B} \rightarrow H_B$.sub.domain.dom

and

- H_B.sub.domain.dom $\rightarrow IP_{H_B}$.

NS_B wants NS_A to believe the mappings

- $IP_{H_B} \rightarrow H_A$.domain.dom

and

- H_A.domain.dom $\rightarrow IP_{H_B}$.

As NS_B cannot simply send the false information to NS_A it could ask NS_A to resolve a mapping that only NS_B can resolve. NS_B would then append the additional incorrect information to the response to NS_A's query. Doing so, NS_A's cache would be poisoned with the necessary information to allow H_B to impersonate H_A and log into NS_A.

We call this the "Ask Me!" approach, because name server NS_B implicitly tells name server NS_A to send a query to NS_B. NS_B therefore tells NS_A to ask him a question.

We did not implement this attack. Using the standard tool "nslookup," NS_B can force NS_A to create a query, and using the name server modifications described in 3.5.6, NS_B can append the two false resource records to the additional section of the response to the query.

3.5 Implementation and Experiments

This section describes our main experiment step by step. We start with the description of the setup of our test zones and the machines used. We continue with the name server and resolver setups. The UNIX concept of trusted hosts is fundamental in exploiting this flaw. We explain this particular instance of the Trusted Network concept followed by the authentication process using the Berkeley "r–commands." Then we describe the manipulation in the authoritative data of the name server's reverse lookup tree. We also describe the final step, the cache corruption, in the case that the Berkeley patch is already installed.

3.5.1 Domain and Zone Setup

The setup of our experimental field consisted of two zones (see Figure 3.1). All machines, the attacked machine NS_A, the imitated machine H_A, and the attacker machines NS_B and H_B, were part of the domain sub.domain.dom. However, NS_A and H_A contacted another name server (NS_A) than NS_B and H_B (NS_B).

In reality the attacker and attacked hosts would not reside in the same domain, but because we are solely observing the Domain Name System protocol between name servers, it did not make a difference as long as the authoritative data that had to be corrupted remained in the attacking name server's zone, outside the attacked machine's zone.

3.5.2 Name Server and Resolver Setup

Name server NS_A was set up to contain primary information about the domain domain.dom, whereas name server NS_B contained primary information about the domain sub.domain.dom. The resolvers of NS_A and NS_B were set up to contact the name servers running on the local hosts exclusively. This kept the information requests on controllable, well–known paths.

3.5.3 Trusting Hosts

In Berkeley UNIX and derivatives, system administrators and users have the option to trust other systems, or to trust certain user accounts on remote systems by providing a "remote authentication" database. We introduced "trust" in section 3.3.2. The "/etc/hosts.equiv" file applies to the entire system, while individual users can maintain their own ".rhosts" files in their home directories.

The file "/etc/hosts.equiv" is maintainable only by the superuser. It can contain host names from which users can remotely access local accounts without having to provide a password for authentication. The user has to have the same login id on both machines. Access is granted on basis of the login name and the host name of the connecting machine.

Each user can create a file named ".rhosts" in his home directory. In this file he can specify trusted users on other machines. It is also possible to force remote users to always supply a password when using the "r–commands," by prefixing entries in ".rhosts" by a dash.

> These files bypass the standard password-based user authentication mechanism. To maintain system security, care must be taken in creating and maintaining these files. [Sun91, HOSTS.EQUIV(5)]

These features have caused many security breaches in the past, but still most system administrators do not disable them. Trust in networks is a transitive relation, in the sense that if A trusts B, and B trusts C, then A trusts C. This relationship can do great harm. Once an intruder has successfully subverted one machine, he can hop to other machines, exploiting trust. Examining the trade–off between convenience and possibly unauthorized access, most system administrators decide in favor of convenience.

In our setup, host NS_A trusts host H_A via the file "/etc/hosts.equiv" containing host H_A's host name.

3.5.4 Authentication in Berkeley "r–Commands"

The main two "r–command" applications we deal with are "rlogin" and "rsh," both of which consist of a client and a server side. [Ste90, Chapter 14] gives an overview of remote command execution under UNIX and [Ste90, Chapter 15] gives many details about the remote login procedure.

Examining the source code for the client "rlogin" and the server "rlogind" yields the following security check procedure:

1. Check if the client uses a reserved TCP port. Abort if not.
2. Check for a password file entry on the server for the specified server–user–name. Abort if not.
3. If not root login: Check the "/etc/hosts.equiv" file for the client's system.
4. If not root login: Check the ".rhosts" file in the home directory of server–user–name for the client's system.
5. If root login: Check the "/.rhosts" file for the client's system.
6. Prompt user for his password if none of the tests 3-5 passed.

It may seem that a system is much safer if only ".rhosts" files exist with no "/etc/hosts.equiv" file, because ".rhosts" files create the additional constraint that user login names have to match: the user name on the attacking host and the one on the attacked host. That is not the case. In Section 3.6.1 we will discuss how to acquire information about which host name and which user name to impersonate. Once we have that information, it makes no difference at all. In the "rlogin" protocol, the client connects to port IPPORT_LOGINSERVER[2] of the remote host and sends a packet consisting of <local–user–name>, <remote–user–name>, and <command> to the server. Because the client is under full control of the attacker, it is not difficult for

[2]in "netinet/in.h" currently specified as TCP port 513

the attacker to modify the "rlogin" code, such that local–user–name and remote–user–name contain the appropriate values. The attacker can then recompile the "rlogin" code and use the modified version instead of the original one.

3.5.5 Reverse Lookup Tree Manipulation

Because the attacker controls the primary domain sub.domain.dom, he can modify the data of the reverse lookup tree of his domain. In the "rlogin" protocol, the server retrieves the IP address of the connecting site with the system call "getpeername()". The server then maps the IP address into the host name with the system call "gethostbyaddr()". In Section 2.5 we explained that the IP address 111.22.33.4 gets converted into the name 4.33.22.111.in-addr.arpa, which is then queried in the reverse lookup tree via the Domain Name System protocol. In an unimpaired state of the database, the lookup returns the name of the attacker H_B. But if one single record in the reverse lookup tree is changed from

4.33.22.111.in-addr.arpa IN PTR H_B.sub.domain.dom

to

4.33.22.111.in-addr.arpa IN PTR H_A.sub.domain.dom

the query yields the name of H_A after the zones are reloaded into the name server.

3.5.6 Cache Corruption

Section 3.1 already mentioned the Berkeley software patch that adds a higher degree of security to the remote login procedure. The patch works as follows: the system call "gethostbyaddr()" in "rlogind" and "rshd" is implemented by a DNS request for a PTR record. The server that supplies the PTR record is under control of the attacker and can return a falsified record. The system call "gethostbyname()" requests A records from the name, server which is not controlled by the attacker. If the comparison of the retrieved IP addresses and the original IP address fails, the patch has succeeded in detecting an attempted impersonation. Figure 3.2 shows an overview of the algorithm used in the patch.

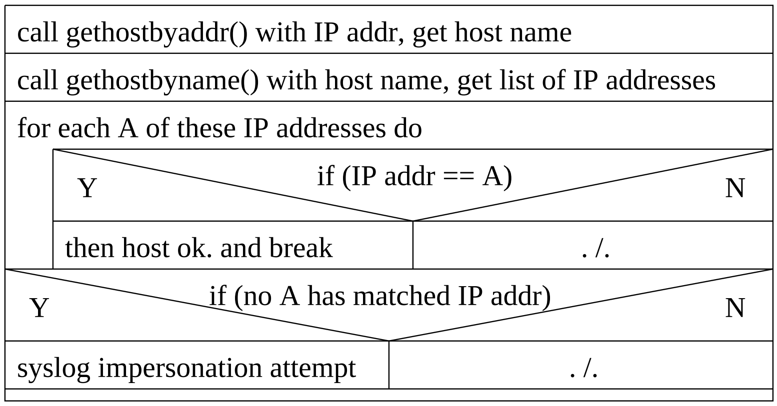

Figure 3.2 Algorithm of the Berkeley patch

In the case that the attacked site has the patch in place, the attacker has to use a more sophisticated approach to succeed with his intrusion attempt. The second query goes to the local machine's name server first. This name server has a cache which can be poisoned by the attacker by adding a false "A" record to the DNS message containing the PTR record. This additional "A" record makes the remote site believe the reverse lookup was correct.

In our setup, we modified the name server code of the attacking machine. We added statements to determine when the reverse lookup query for the mapping of 4.33.22.111.in-addr.arpa was issued. To the response to that query we added an additional record providing a faked forward mapping from 111.22.33.4 to H_A – not H_B. Figure 3.3 shows the contents of the additional record. It was important to piggyback the unrequested record on an otherwise valid packet, because a name server examines received packets for their id number and other criteria before it accepts the packets at all (we will examine these criteria in Section 4.1. For now it is enough to know that although a name server does not blindly accept anything, it is nevertheless easy to fool). To camouflage the attack, we supplied a short time to live value in the resource record. However, the BIND code contains a hard–coded constant that limits

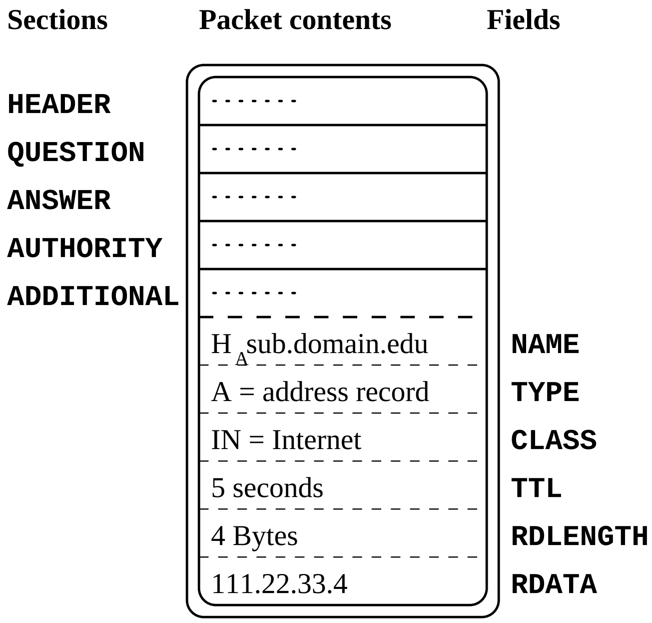

Figure 3.3 Additional false resource record

the minimum time to live value to "min_cache_ttl"[3]. In case the remote site NS_A contacts the attacking name server NS_B again within these five minutes, NS_B could overwrite the faked records by supplying new ones with the correct information.

We included the feature that the name server can understand an additional user issued signal. Using this toggle signal, the attacker can switch on the malicious code before the attack starts, and switch off the distribution of the malicious records right after access was granted by the attacked site. This ensures a directed attack and minimum possible unwanted auditing.

3.6 Experiences Gained

This section states the pieces of information necessary to launch an attack and describes the experiences gained while working with the test environment.

[3]in BIND version 4.8.3 (5*60) seconds = five minutes

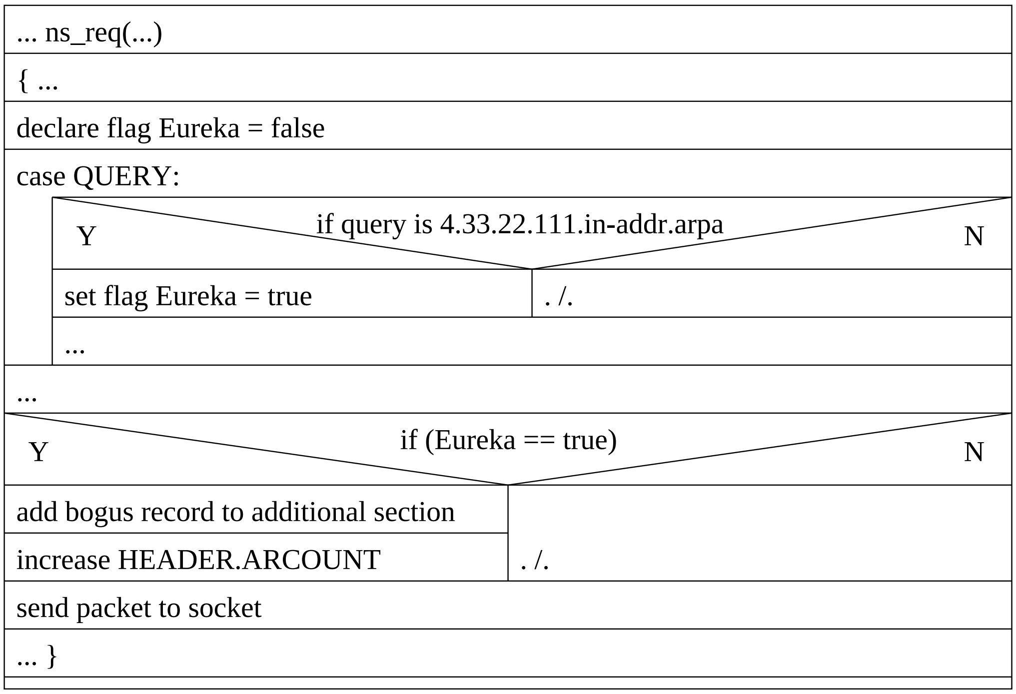

Figure 3.4 Modifications in name server code

3.6.1 Acquiring Information

An attacker needs to have three pieces of information before he can launch an attack:

- target host name NS_A
- user name(s) on hosts NS_A and H_A to impersonate
- host name H_A trusted by target host

In some environments, the local and remote login names for one user are identical. A user has the possibility to specify other user names as trusted users of his account. In that case, the login names are most likely different.

In our setup, we were not in need of acquiring host name pairs and the appropriate login names. Section 4.7 provides methods to obtain this information, followed by a discussion.

3.6.2 Complexity of Modifications

Most of the work that was done during the experiments went into the setup of the zones for the name servers, the source code modifications of the remote login and the name server, and some shell scripts to automatize the break–in. The modifications to facilitate a break–in are minimal in the simpler case that the Berkeley patch is not installed. Only one record in the database for the reverse lookup tree must be changed.

If, however, the patch is installed, the name server code must be changed to enter the false resource record into the additional answer section. These changes are not difficult, but they require a good understanding of the Domain Name System protocol and the name server source code.

Furthermore, there are some changes to the "rlogin" program. In the case that user Alice on host NS_A trusts user Bob on host H_A, the attacking host would need a legitimate user Bob that logs into NS_A. But that would require adding a new user id to the attacking system every time the attacker wants to impersonate a different user name, regardless of the viewable changes in the password file. A much neater approach requires few changes in the "rlogin" code. For the target host it is not important that the remote user Bob exists; it is sufficient to pass Bob's login name in the first packet (see section 3.5.4) from the "rlogin" client to the "rlogind" server to make the target host believe Bob is "real".

Overall, the attack requires only a few changes and can be achieved easily. What makes the break–in difficult is obtaining the necessary information about remote users and machine names, having superuser privileges on a system with a primary name server, and having the proficiency of making the changes in the name server database and code.

3.6.3 Detecting a DNS based Break–in

During an attack, an attacker usually wants to operate as furtively as possible. After an attack, an attacker wants to leave behind as few clues as possible that could point to him or his actions.

We distinguish between where the attacker's presence or his actions can be detected or observed: On the attacked machine and on the attacker's machine.

In the following we assume that the attacker has not (yet) done any obvious harm to the attacked system. In our examination we only treat the detection of the break-in directly, not of its consequences, once an attacker has gained access. The false record in the cache has a minimum lifetime of currently five minutes and can be detected only in that short period of time. The false mapping could be detected by examining a cache dump of the name server, or in case a user tried to resolve one of the names involved in the tampering.

The simple fact that the attacker is logged in could be observed. In an environment where many users access a system at the same time, this seems unlikely. However, if the compromised machine is watched closely by a system administrator or users, the chance of detecting the login is higher. If the attacker logs in as superuser, the chances of detection are even higher, because logins of privileged users are logged separately.

It is also possible to modify the "rlogin"–code to log all remote logins to gather more information about connections involving the own host.

On the attacker's machine, we have to distinguish between the possible identities of an attacker. If he is a rogue system administrator and has no higher authority above him in his organization, there is hardly any chance that anyone on his system could detect his malicious deeds.

If he has subverted the system and has gained the necessary superuser privileges on the attacking machine, the chances of detecting him are better, though still pretty small. Because the attacker has subverted the attacking machine in the first place, everything we said about the possibilities of detecting anything on an attacked machine is applicable here as well. We could also observe the modified executable files, that are necessary for the "rlogin" and the modified name server operation. But all changes in binaries can be made using local copies of the source code that is readily available. Some sites run monitors that detect on a daily basis if binaries were changed or touched. Using local copies avoids detection by this type of monitor. The executables can even be started from local directories, well–hidden from others. The name server that is already running has to be replaced by the local copy, but that is a job that takes less than a second.

Tampering with the log files also aids the attacker in staying undetected. With the modified "rlogin" version, there are no additional password file entries necessary, which otherwise could be observed.

Overall, the attacker has very good chances of hiding his activities completely. Most of these methods of getting a glimpse of his doing seem farfetched to us and their odds of success are quite small. The highest chances of detecting the tampering is by catching the false record during its short lifetime or by simply finding the attacker logged in.

4. SECURITY ANALYSIS AND SOLUTIONS

Most of the proposed "solutions" in this chapter are not complete solutions to the problem. Some of them are valid under additional assumptions that cannot always be met; others are applicable to parts of the problem.

Because many factors contribute to the security breach encountered in this thesis and all of them are necessary, it is sufficient to eliminate one of them. That sounds easy to accomplish, but is a difficult task in practice, because eliminating any one of the factors brings a trade–off with functionality, efficiency, or simply convenience with it.

We present for each of our solutions the necessary background, if it was not already given in one of the previous chapters, followed by a description of the idea of the solution. The solution is then examined and discussed using criteria such as feasibility of its implementation, quality of the solution, complexity of the idea, and compatibility with the original design goals.

It is important to view these solutions as not stand alone. In different combinations they achieve several degrees of security. The concluding chapter of this thesis contains a high level discussion about combinations of our solutions, to obtain, if not absolute security, at least a high level of confidence in the security of the Domain Name System.

4.1 Security Considerations in the RFC 1035

In the design of the Domain Name System, security considerations were not forgotten, and the RFCs show that the integrity of the cache was an important issue. The eagerness to improve performance led to the nasty logic bomb of adding unauthorized records to the additional section and — in absence of strong authentication — believing their correctness.

Before responses are further processed, a number of preprocessing steps takes place. These include a check for the plausibility of the header (id number check), the correctness of the resource records' format, and time to live values. If a time to live value exceeds one week, the specification allows the implementor to discard this record, or limit its lifetime to one week. The id in the header of the response must match the id of the query. A name server expects the reply from the same IP address where he sent the query. This can cause some confusion if replies come from multihomed hosts that use other ports for sending the response, because of local routing information. This was a common bug in name servers.

The standard states several situations in which data should not be cached. If a packet is truncated (TC flag in the header is set), its resource records should not be cached, although they can be used for the current mapping. The reason for this is that a cache should not contain incomplete information. The information in a cache might be out of date which will eventually be corrected; but the cache stays always in a consistent state, because incomplete mappings are never entered. A cache should never prefer cache data over authoritative data. Responses to inverse queries are also taboo because of their incomplete information character. Name servers or resolvers have to do all correctness checks before they can cache data. Responses of dubious reliability have to be examined carefully. It is however not easy to decide criteria such as "dubious origin," or "reliable source."

Before caching a newly received record, the name server should check for an existing record in the cache. Depending on the circumstances, either the data in the response, or the cache is preferred, but the two should never be combined. If the data in the response is marked as authoritative data in the answer section, it should always be preferred.

4.2 Analysis of the Name Server Algorithm

In this section we review the name server algorithm stated in section 2.9.2 and analyze it step by step. We are especially looking for weak assumptions that do not always hold. These assumptions could be exploited by attackers.

1. In step one the algorithm determines if a recursive name resolution is requested and available. If so, it branches to step five, where a copy of the resolver algorithm or the local resolver is invoked. When the resolver returns an answer, the name server algorithm believes this answer to be correct and copies it as is into the according answer sections of the own reply. This answer could contain false records not only in the additional section, but also in the answer or authoritative section. This is a weak assumption because the response of an arbitrary name server cannot always be trusted.

2. In step two the name server searches the available zones for the nearest ancestor. It assumes that its zone data is accurate. This should usually be the case. But there is a possibility that its data base is not consistent. This inconsistency can lead to malfunction as it has in the past, and in the worst case to a security threat.

3. In step three the server tries to match the query in its own authoritative data base. In principle the same problem as in the previous step exists.

4. Step four is responsible for finding data in the cache once the matching phase in step three is not successful. If the QNAME is found in one of the cached records, all resource records matching the QTYPE of the query are copied into the answer section. If there is no delegation found in its authoritative data, the algorithm puts the best referral found in the cache into the authoritative section. In these cases, the algorithm believes the data that it retrieves from the cache to be unimpaired. As we showed, this does not necessarily hold.

5. Step five is the call to another resolver. The problem here is that the response is blindly believed, cached and used.

6. Step six does not contain a flaw itself, but it demonstrates how easy it is to add records to the reply, and that a name server accepts that without many constraints.

4.3 Analysis of the Resolver Algorithm

In this section we review the resolver algorithm stated in section 2.9.3 and analyze it step by step. We are especially looking for weak assumptions that do not always hold. These assumptions could be exploited by attackers.

1. Step one in the resolver's algorithm shows one of the security flaws in the protocol. The resolver searches the cache for the desired data. If the data is in the cache, the resolver "assumes" it to be good enough for regular use. This assumption can lead to the use of false records and aid an attacker in his unauthorized attempt to access another machine.

   Some resolvers offer the option at the user interface to force the resolver to ignore cached data and always consult an authoritative server. Although this approach would solve the problem, it is not recommended as the default, as this is very inefficient.

2. In step two the resolver looks for a name server to ask for the required data. The general strategy is to look for locally available name server resource records, starting at SNAME, towards the root. The resolver has many choices here and depending on which choice it makes it can contact sound name servers or the attacker's name server. However, if we assume, that the attacker has set up his zones such that his name server is the only one with the necessary information to answer the attacked machine's query, the resolver has certainly no other choice than finally contacting him.

3. Step three sends out queries until a response is received. The strategy is to cycle around all of the addresses for all of the servers with a timeout between each transmission.

4. In step four the resolver accepts answer packets from name servers it has contacted. These packets can contain records in the additional section. The resolver performs some preprocessing on these packets and the contained records (see 4.1 for detailed description), but very likely accepts them and caches their contents. Caching unrequested data provided by some unknown source can lead to a major problem but is also necessary to obtain a good overall system performance.

If the resolver has direct access to a name server's zone, it should check to see if the desired data is present in authoritative form, and if so, use the authoritative data in preference to the cache.

One could ask where exactly the problem lies: in believing the cached data in step one, or earlier in blindly caching additional information throughout step four. Obviously, the data should be correct before it is entered into the cache. That ensures the integrity of the internal data structures, which is an important precondition in most systems.

But this answer only shifts the question to the origin of these records. Where is the right point to ensure the integrity of transmitted resource records? In the name server that writes the records into the additional section? That can be violated by an attacker, as we have proved in our experiments. Or in the name server or resolver that accepts the resource records, before they are added to the cache? The problem here is that the receiving entity has no way of deciding what is reasonable to believe, and what can lead to trouble.

Neither of the approaches is feasible – the central dilemma in the current Domain Name System design.

4.4 Evaluation Criteria

The following sections present solutions that address the stated problem. Most of the solutions are based on the Domain Name System and are not solutions to the abstract problem.

As we have already mentioned, the presented approaches are not complete solutions to the problem. Most of them work only under certain additional assumptions, but then reliably. A good approach is probably to not limit a system to the application of one solution, but to implement a reasonable variety of them. This variety should cover as many cases as possible, with few overlaps. Some of the presented solutions are already in use in some systems, while others are in their early stages of design or development.

Our presentation of each solution contains a description and a discussion. We use several criteria that are important in an evaluation of solutions to our problem:

- The "quality" of the solution is a measurement of the radius of applicability of the solution. This value cannot easily be specified, because the set of applicable cases is not precisely given. We mention the cases that are covered by a solution and try to derive from that a judgement about the quality of the solution.

- The "feasibility of the implementation" of a solution determines how much effort is needed to apply the solution to an unmodified version of a state of the art name server.

- The "complexity of its implementation" measures if modifications in different areas are involved and how complicated their interaction is. A solution can have a very low degree of complexity, but require considerable implementation effort. A complex implementation does not has to result in a large amount of coding.

- In solving the problem we are striving for "compatibility with the original design." A solution that does not require changes to the DNS protocol is usually preferred over one that does – even if this conformity has other disadvantages.

- The Domain Name System is a system that resolves mappings on–line. The efficiency of the system and its performance are important factors of influence. The compliance of the solution's "efficiency" with that of the system is equally important.

- Some of the solutions involve users in general. For example if the solution requires a change in the user interface, or in an organization's policy of handling trust. The user has to learn to handle the changes, and his approval is a crucial point. We combine these aspects in the term "acceptability by the user."

- Solutions might not be applicable in every organizational environment. We call this criterion "applicability in an organization."

- An important point in the introduction of changes to systems is the "transition process" from the original state (before the solution is applied) to the new state. In case of minor changes this transition period can be very short – sometimes hardly noticeable. If changes of considerable degree are involved, this process plays a major role in the change management.

- The "transparency of the solution" involves the user interface and the software interface to the system. This point examines another notion than the "compatibility with the original design," which only involves the protocol issue — not the user.

4.5 The Berkeley Patch

We already mentioned the Berkeley software patch in some sections of this thesis and explained it in detail in Section 3.5.6.

This first attempted defense, developed at the University of Berkeley, CA , consists of modifications of the "rlogind" and "rshd" code. The idea is to validate the inverse mapping tree by looking at the corresponding node on the forward mapping tree. S. Bellovin describes the method used by the patch in [Bel92] as follows: "To detect this, we perform a cross–check; using the returned name, we do a forward check to learn the legal address for that host. If that name is not listed, or if the addresses do not match, alarms, gongs, and tocsins are sounded."

Refer to the description of the algorithm in Section 3.5.6 and Figure 3.2.

The fix is easily installed and not very complex. Its compatibility with the existing Domain Name System protocol is another advantage. The transition process to move to a name server that contains the patch is not difficult or complex. A few lines of code have to be inserted into the name server code, and the name server has to be recompiled and started.

Although we regard this patch as an obligatory modification to "rlogind" and "rshd," it is limited in its scope. It can easily be countered using the methods demonstrated throughout Section 3.5.6. Because a name server always prefers authoritative data over non–authoritative records, it is impossible to poison the cache of a primary or secondary server for a zone. Thus, an additional false A record cannot be inserted into the cache, and the cross–check will detect the tampering.

Overall, the patch is a true solution if trust can be extended only within the scope of authoritative data, and if the attacker does not use the more sophisticated attacking method. In case the attacker supplies the additional "A" record with the answer to the reverse lookup, and trust is extended to possibly untrustworthy sources, this method will fail.

4.6 Examining Berkeley "r–Commands"

The Berkeley r–commands extensively use the ".rhosts" and "/etc/hosts.equiv" files to increase convenient network access. In Section 3.5.3, we discussed the Trusted Network concept. R–commands such as remote login and remote shell offer the possibility to extend trust to other machines. Users and system administrators can build individual networks of trust. What looks like a good idea at the first glance proves very dangerous in some cases.

The existence of these structures of trust is necessary for the break–in to happen. Obviously, the break–in is prevented if we prohibit the usage of trusted hosts or users completely. It is technically possible to disallow the usage of "trust" in Berkeley commands. The choice can be made by the system administrator at compile time. However, being able to access other machines without passwords makes the work in a networking environment easier. Once used to the comfort, not many users agree to sacrifice their convenience for the prevention of "hypothetical" security concerns. The trade–off hereby would contain the loss of very convenient and in many cases necessary tools for trouble free connection to hosts that are accessed frequently.

A less "safe" solution would be to limit trust to locally administered zones, i.e. authoritative zones, where the Berkeley patch works reliably. As we discovered in Section 4.5, limiting trust to certain zones fixes the flaw. An organization could issue the policy that only local trust is allowed. In some organizations this can be considered a reasonable approach if hardly any remote accesses are originated outside of the "own" zone to the "own" zone. Additional tools would be necessary to enforce the policy, such as a script that periodically checks entries in ".rhosts" files. If periodic checks are still too weak, the r–command implementations could be changed in a way that users cannot directly modify their database of trusted machines (".rhosts"), but have to use a special program to manage trust–entries. The data must be kept in a protected data area of the operating system managed by the kernel. This program could filter out–of–zone entries at the time the user wanted to enter them. It would also contain the possibility of managing setup changes centrally. This solution actually proposes an automatized procedure to implement an organization's policy.

If the nature of connections allows a policy such as described above, implementing it is a major effort. Some system scripts have to be written to ensure proper usage, operating system code and r–command code must be modified, and a new user interface has to be developed. Users shall be trained how to apply the changed facility and have to be made familiar with the new policy and the new user interface (which could easily improve the existing one). Advantages of this new approach are the compatibility with the existing Domain Name System protocol and additional benefits in further security related issues.

Overall, a very weak point in the Berkeley derived UNIX systems is the usage of trust. This thesis exploits only one of several known flaws based upon trust. Using trust–based mechanisms requires thinking about a change in individual policies in dealing with granting trust to others. We can conclude, by citing S. Bellovin: "If a host trusts another host not named in a local zone, its name server cannot protect it." ([Bel90b])

Although we concentrate on the Berkeley "r–commands" in this section, we do not forget that there are other ways in exploiting the flaw. For example intercepting electronic mail is a target of attackers; especially electronic mail that is exchanged by security agencies and security related organizations.

4.7 Restricting Public Information Access

What makes the break–in possible in the first place is gathering necessary information about host names of trusting machines and user names on different systems trusting each other. This section discusses how to obtain the names and whether it is feasible or reasonable to restrict access to this information.

We are not discussing random patterns of trust that might exist between hosts, but two common patterns using a systematic approach. The following discussion is based on section 3 in [Bel90b]. In a cluster of time–sharing machines, each machine is likely to extend trust to all its peers. This pattern is not common to the general user population, but it is applicable to systems programming and operational staff. Another typical pattern is the occurrence of file servers that trust their clients, who serve as a source of extra CPU cycles. "Dataless" clients will frequently trust administrative machines to permit software maintenance.

There are several networking utilities that are generally available to all users on a system to spy out the wanted information.

A combined usage of "snmpnetstat" and "finger" can do the job. One might object that "snmpnetstat" is not always available and that some sites also restrict the usage of the finger daemon on their machines. But there are more common tools that can be abused.

Examination of mail or news headers gives us information about where mail originated and which path it took. The "Received:" fields contain a complete trace of the route. Sometimes this route contains workstation - server names that trust each other. A similar trick is possible using "traceroute" once we know a remote workstation name.

We can also gain much insight using the Domain Name System itself. The SOA records contain a machine name and a host address of a privileged user. With the host name we can retrieve the IP address and then with a zone transfer obtain names of other machines in the network local to that machine. Even if the zone transfer is disabled, we could issue 254 reverse lookups to collect the names we seek. The HINFO records give additional information.

Further "help" is provided by "ftp" (some servers offer the service, only few workstations do), "smtp" (machines that run mail servers), and Sun's "rpcinfo" (what services are running?) Published material is available from some universities that describes the setup of their networks on a high level.

Some systems still use the same "/etc/hosts.equiv" files on many hosts just to simplify systems administration.

The mentioned collection of tools shows that it is a difficult task to limit information access without sacrificing the legitimate utilization of network services. Preventing someone from gathering the necessary information is nearly impossible. Too many services rely on address information, and most people would complain terribly if they were deprived of useful tools such as finger, email, and news. The idea of open systems requires open access to information services and address information. Therefore, most system administrators have decided that the benefits of these utilities outweigh the risks.

Overall, we think that shutting down well–known and widely used services is not a good idea. The lack of these services would hurt functionality and the purpose of the Internet to a considerable degree. There are too many ways to gather the necessary information; it would be a hopeless job to protect the Internet against abuse.

4.8 Adjusting DNS Update Intervals

Some sites have connections chiefly with machines outside of their zones that stay stable in the sense that host name to IP address mapping will stay the same for a long time. The idea is to enter long TTL values into the resource records, values that exceed the currently implemented threshold of 1 week. Limits could be increased up to 6, 12 months, or even longer, depending on the situation. If this data is entered with great care to ensure correctness of the mappings, the DNS based break–in is prevented.

This approach is limited by its scope of applicability, but it is a solution with many advantages. It goes with the current Domain Name System protocol and can be implemented without much effort, by simply changing the constant max_cache_ttl[1] in the name server code and recompiling the system. As all necessary entries are kept in the local cache, the system provides very quick replies to queries. It hardly ever uses the network and therefore saves bandwidth on the medium for other tasks.

This approach has the problem of validating the host name to IP address mappings before they are cached. How can it be ensured that the mappings are correct in the first place? Certainly, a false entry would stay for a long time, and the attacker's address would be finally noted. But does that really help, once mischief is done? It might aid in prosecution efforts, but only little in prevention.

One of the original reasons to introduce the Domain Name System was to manage the dynamic behavior of changes in the data base. This approach fixes mappings for a long time and uses a powerful distributed database system for an infrequently happening update process. Although we are not talking about a static mapping in

[1] in BIND version 4.8.3 (7*24*60*60) seconds = one week

this section, a well–maintained HOSTS.TXT file would do the job with less overhead. We will present the discussion about abandoning the Domain Name System and returning to the previous system in Section 4.9.

Overall, the approach of extending TTL values to a long period of time is a safe and feasible method in environments where the additional condition of static mappings with long lifetimes is given. However, in this case not the Domain Name System seems to be the right approach, but a locally well–administered static mapping mechanism.

4.9 Abandoning the Domain Name System

It could be suggested to abandon the DNS and either return to the previous system with a static host table, or move on to another system, that has yet to be developed. We are not going to talk about possible future development of the Domain Name System here, but about returning to the previous system. Abandoning the Domain Name System is not an extreme scenario of what we described in Section 4.8, as our solution there only assumed slow dynamic behavior.

This section suggests an again centralized management of the mapping data. In this approach, mappings can change frequently, but changes have to be reported to a central authority that manages the whole Domain Name Space in contrast to the Domain Name System approach of managing zones through delegated local authorities. This would not solve the problem, because the problem is not the DNS, but inadequate methods of host authentication.

IP addresses of trusted machines could still be imitated. This is a somewhat harder task, but the know-how has been published for quite some time (see [Mor85]).

Would it be safer to transmit updates to a central site? Email, telephone calls, or conventional paper are not necessarily a reliable way to transmit mapping information updates. The long time delay until centrally made changes are propagated through the network would condemn the database to be in an inherently inconsistent state. The system would again contain all the disadvantages described in Section 2.2, which were the reasons for developing the current Domain Name System.

But besides these obvious, technical, and well–known reasons, there is a significant argument why no one can possibly be in favor of reinstalling the previous system: the sheer size of the Internet. HOSTS.TXT was abandoned because 200,000 hosts was too much to be managed. Are currently about 1.5 million (see [Lot93]) easier to handle? Certainly not.

Overall, abandoning the Domain Name System would drag the name resolution task in the Internet out of a functioning state with a not easily exploitable security breach, into an unmanageable, not working state of prehistoric system design. We think that would do more harm than doing nothing at all.

4.10 Hardening Name Servers

This section contains a number of problems that we classify into two groups and a collection of possible modifications to the name server to provide (at least partial) solutions to these problems.

We thought about organizing this section in a way that solutions are stated directly in each section describing a problem. But then we discovered that most of the proposed solutions in hardening the name server are applicable to a variety of problems. In the same time, it is necessary to not only concentrate on how to deal with certain problems, but with all of them simultaneously. We therefore decided that a more general approach is to state a list of problems next to a list of solutions. This way we can relate problems to solutions and vice versa.

The following two sections are descriptions of the problems, grouped depending on whether a given problem exploits cache poisoning, or not.

4.10.1 Problems Not Exploiting Cache Poisoning

In Section 3.4.2 we saw a first example of how to exploit the weaknesses of the DNS. Simple changes in the database entries of a machine that is trusted, can lead to a break–in. As we showed in this thesis, it is not difficult to counter the attack based on database modification.

There are two more problems, that are related in their nature. In the first one, an attacker intercepts a query to another name server and provides the reply himself. If the reply contains a referral to some host that is under the attacker's control, the originator of the query will finally ask that name server and believe whatever is returned. If the time to live values for records supplied in that answer are zero, the originator will not cache the information, but use it for the current resolution process. The name server that was originally addressed, or its network connection, can be manipulated by the attacker in a way that they either not receive any query at all, or that their response gets lost (see [Mor85] for an example).

A similar attack is based on the fact that the standard for the DNS implicitly determines that the first answer a resolver receives to a query is returned to the user program. The standard states in [Moc87a] : "Get the answer as quickly as possible". If a query is answered by more than one host (and one of the hosts supplying an answer can be the attacker who has intercepted the query, like in the previously described problem) the fastest answer wins. This answer can again refer to another name server under the control of the attacker.

4.10.2 Problems Exploiting Cache Poisoning

In the Sections 3.4.3 and 3.4.4 we described two problems that exploit the fact that the cache of a name server can be poisoned. We describe two more problems in this section.

Imagine again the scenario we described in the previous section, where the originator of a query receives more than one response and one of the responses contains false information supplied by an attacker. The standard states in [Moc87b, 7.4] "When several RRs of the same type are available for a particular owner name, the resolver should either cache them all or none at all." The fact that the responses come from different IP addresses, does not matter to the originator. In [Moc87b] the standard deals with the fact that name servers are sometimes multi–homed hosts and respond to queries using another network interface than where the query arrived. We cite:

"That is, a resolver cannot rely that a response will come from the same address which it sent the corresponding query to."([Moc87b])

Under certain additional assumptions it is possible to poison some name server's cache by simply sending it a query that contains the corrupt information in the additional section. This should work in the following setup:

- an Attacker on host NS_B sends a query along with the false additional RR to a name server B it wants to compromise, requesting recursive resolution
- the name server on host NS_A does not cache incoming information according to the RFC, but it shares its cache with the local resolver on the same machine
- if the name server on host NS_A invokes its local resolver that will finally get back an answer from somewhere, this resolver on host NS_A will cache whatever data is provided according to the rules – including the additional record provided by the attacker.

The name server on host NS_A inherits the weakness of its own resolver.

4.10.3 Keeping Additional Information

A first idea is to log "rlogin" attempts with IP address and local and remote user names. Or even more: to tag cache entries with their origin. The latter is another easily achieved modification that costs additional memory space in the cache. This method makes it easier to track, for example, a false "A" record for the purpose of debugging wrong zone data or investigating a DNS based break–in.

4.10.4 Prevention of Cache Poisoning

Preventing the cache from contamination is probably not feasible from within the name server code, as there is no way of a priori determining if any given additional record is trustworthy or not. We could start treating special cases of when to allow or disallow additional information.

The default safe behavior would be to disallow the caching of unrequested information, and to allow it only in cases where the information is necessary, and then only for the current resolution.

4.10.5 Context Cache

But there are other, more sophisticated approaches possible: If some additional or authoritative records are returned together with a resource record, they should be interpreted only in the context of that resource record. The difference between the default safe behavior approach and this one is that in the first one resource records are only cached, when they were requested or necessary additional information, whereas in the second approach the new entries get cached, but can be retrieved from the cache only in the same context in which they were entered. For example, an “A” record in the additional section of a response to an “MX” record request should only be used for delivering mail. The information would not be acceptable for an “rlogin” to another host, or generally usable for other services.

A glue “A” record coming along with an “NS” record would only be used for domain hopping, because that is the context in which it was supplied.

“A” records along with “PTR” records should never be cached, because there is no legal context in which they have to be returned in a single response.

This whole approach leads to the question of whether we still need the additional section at all. If only certain combinations of resource records are allowed as a response to a query, why not consequently eliminate the idea of additional unrequested information completely, and adapt the protocol to accommodate the new ideas, namely a certain limited number of types of associations?

First of all, that would require a protocol change, which is something we try to avoid. Some of the original design goals of the Domain Name System also imply that eliminating the additional section would not be a good approach. The system would lose some of its generality, because the additional section might become very useful in future applications of the Domain Name System without containing any security threats. The system would certainly lose efficiency. Here we see again an important trade-off that we have already mentioned in several earlier sections: an increase in systems security and a decline in system performance vs. good system performance and a possible lack of security.

It is therefore justifiable to take the approach of hardening the name server by treating more special cases, and by increasing the complexity of the internal data bases, instead of hardening it by implementing the same ideas accepting protocol changes.

4.10.6 Authority Cache

A further approach would be to cache data only if the source of a record is known to be authoritative for that zone. We give an example for that: If a name server NS_A receives a "PTR" record from some host NS_B, and the DNS message also contains an "A" record in its additional section, then the name server NS_A would believe and cache this information only if it already knows that the source name server NS_B is authoritative for the according zone. A name server following this strategy would create its own tree of authoritative name servers. This tree would have to lose subtrees according to the expiration of the lifetime of some node (name server).

4.10.7 Conditional Cache Use

The Berkeley patch (see Section 4.5) can fail in the case that the cache is already poisoned. An idea to strengthen the Berkeley patch is to provide the possibility to resolve queries without using the cache. That could be used by the Berkeley patch. The system call executing the forward lookup would for example set a flag to indicate that the cache contents should not be used for the following resolution. This method again hits the efficiency of the system, but it prevents the exploitation of the weakness. One could also think of a system call to flush the cache followed by a reload of the database, similar to the signal SIGHUP that a system administrator can send to the BIND implementation of the name server to achieve the same.

4.10.8 Discussion

A very thorough analysis of the protocol is needed to determine the cases in which additional resource records are legal and cannot do any harm, or have to be stored in different contexts.

Hardening the system would require careful design, implementation, and testing and would lead to a higher complexity of the code and the system. Our analysis has to stress the higher complexity, because design, implementation and testing are a process that will be done at some point, but the complexity of a system is a feature that stays with it. Higher complexity usually goes along with greater insecurity. It is therefore important to keep the complexity in a manageable scope.

A decline in system performance would result from the fact that name servers would believe and therefore cache less data — data that might be needed later.

Overall, hardening name servers consists of several possible modifications, some of which seem promising, even though their application decreases the system's performance and increases its complexity, which might lead to further insecurity.

4.11 Cryptographic Methods for Strong Authentication

In this section we describe an architecture for an authenticated Domain Name System. The outline for the approach described below is only one of several possible scenarios. There are systems that provide access authentication in distributed environments. Some examples of systems that use tickets or security certificates are the Kerberos authentication service ([SNS88]) and project SESAME ([Par91]). They are not directly applicable to our problem.

Our approach contains three major features that are necessary to ensure the kind of security we are trying to obtain:

1. data integrity of a message

2. originator authentication

3. originator's proof of being an authoritative source by presenting credentials signed by the parent domain

In the following we will elaborate on these three features and present techniques and ideas for their possible implementation.

4.11.1 Data Integrity

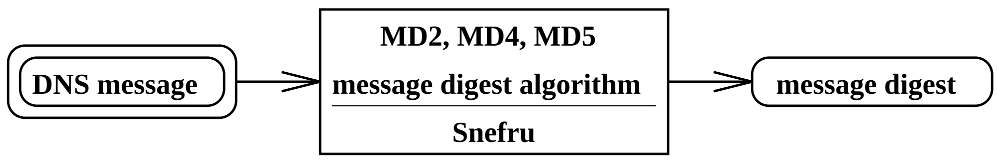

Figure 4.1 Application of a message digest algorithm

Integrity service means that a recipient is provided with assurance that the content of a received message is identical to the content of a message (including its header) sent by its originator (see [Ken93a]).

In our case, we want to ensure the integrity of transmitted DNS messages. There are several approaches to protect a message against unauthorized change: prevention techniques, avoidance techniques, and detection and recovery techniques. All these techniques have inherent advantages and disadvantages. We will not discuss them here, but concentrate on a certain technique to detect unauthorized message alteration. We stress this approach, because it is efficient and considerably secure. In case of alteration detection, recovery actions could be to ignore the DNS message and issue an additional query. Our approach is based upon message digest algorithms. They are one-way hash functions that compute a checksum of some data (in our case the DNS message — see Figure 4.1). They have the following features:

- they are easy to compute (examples are the MD2, MD4, and MD5 algorithms in [Kal92, Riv92a, Riv92b] and the Snefru algorithm in [Mer89])
- the signature (message digest or fingerprint) is only a few bytes per message
- they are computationally hard to invert
- they usually require a certain size of input data

An originator would calculate the message digest of a DNS message immediately before it is sent out. The recipient would recalculate the message digest and compare the resulting value with the one calculated by the originator. In case of a mismatch, the originator would conclude that he did not receive an unaltered DNS message. He would dispose of it.

How does the message digest calculated by the originator get to the receiver unimpaired? The message digest algorithms are publicly known and anyone tampering with a message could easily modify the associated message digest accordingly. To show how this can be prevented we discuss a method for originator authentication in the following section. A message digest together with an authorization service guarantee the integrity of transmitted data.

4.11.2 Originator Authentication

Originator authentication service permits the recipient of a message to reliably determine the identity of the originator of a message.

We demonstrate a procedure that guarantees the originator's authenticity. In an asymmetric (i.e. public key) cryptoalgorithm a pair of distinct, but mathematically related, keys are used for encryption and decryption. One key is private and kept secret by the sender, the other one is publicly known. Data encrypted with a sender's private key can be decrypted using his public key, and vice versa. These keys are usually large integer numbers, several hundred decimal digits long with special, mathematical properties. (ex. [Den82]). "RSA" is an example of a public key encryption algorithm ([RSA78]).

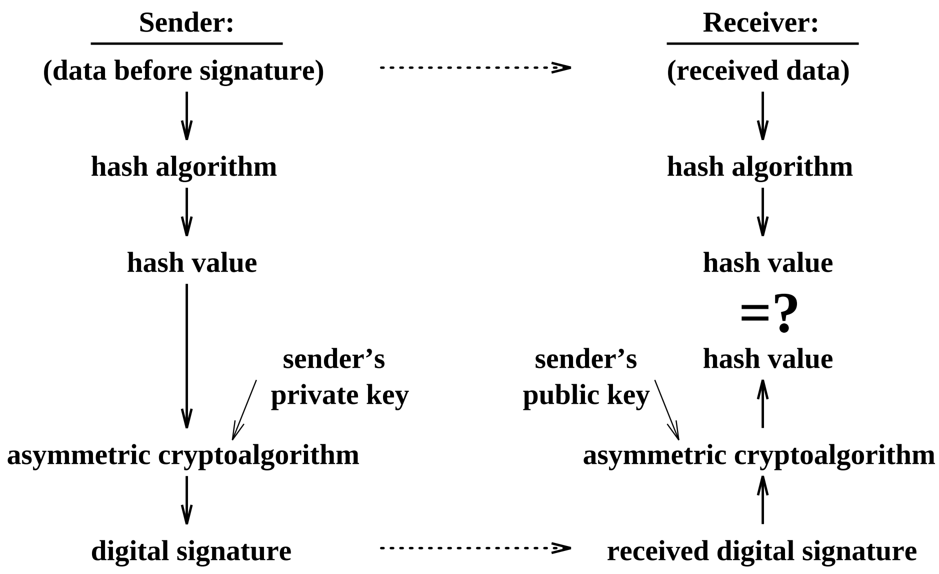

Figure 4.2 Digital signature generation and validation

The following procedure and Figure 4.2 outline how we would use the public key cryptoalgorithm to ensure originator authentication.

The procedure could work as follows:

- The sending name server creates the digital signature of the DNS message m: $s = hash(m)$
- The sending name server signs the message digest (the digital signature) s using its private key K_{priv}^{Sender} : $s' = E_{K_{priv}^{Sender}}(s)$
- The sending name server transmits (m, s')
- The resolver decrypts s' by applying the name server's public key K_{pub}^{Sender} : $s'' = D_{K_{pub}^{Sender}}(s')$
- The resolver recomputes the message digest $s = hash(m)$

- If $(s = s'')$ then the resolver has validated the integrity and the originator of the DNS message

Why do we calculate a message digest at all and not simply encrypt and then transmit the whole message? The main point here is the difference between the runtime costs of creating a message digest and encrypting a message, depending on the length of the original message.

Runtime costs for public key encryption are rather high. Many CPU cycles are needed. Therefore we want to fix the size of the data portion that has to be encrypted: in our case the fingerprint, the output of the message digest algorithm.

Runtime costs for the hash functions are rather small compared to those of public key encryption. It is therefore important to note, that it is more efficient to pad a short DNS message, calculate its fingerprint, and then encrypt the fingerprint, than simply to encrypt the whole DNS message. Message digest lengths are typically shorter than the typical DNS message.

4.11.3 Passing Credentials to Prove Authority

The name server sending the DNS message has to provide credentials signed by its parent domain, to convince the recipient of its authority over the domain for which it just resolved a mapping.

The use of such a certificate transforms the problem of establishing the credibility of one entity into the problem of establishing the credibility of the entity issuing the certificate. This problem is very closely related to the problem of distributing public key certificates. The CCITT recommendation X.509 shows a way to solve this problem. In X.509, a certificate binds a public key to a directory name and identifies a party that vouches for the binding.

We can adopt this mechanism, such that a certificate binds all name servers that are authoritative for a certain zone to this zone of authority and identifies the zone that vouches for the binding. X.509 imposes no constraints on the semantic or syntactic relationship between a certificate issuer and a subject. However, in our approach, the certification system takes the form of a single rooted tree. Each node represents a zone. Several name servers serve as certification authorities for each zone, because all servers that were introduced to increase the reliability of the database system are capable of valid referrals.

A certificate for a zone (for example sub.domain.dom) consists of all IP addresses of authoritative name servers for that zone, signed with the private key of the name servers for the parent domain (domain.dom). Any resolver that receives a DNS message receives as part of it this certificate. After obtaining the public key for the parent zone of the queried zone, the resolver can then verify the validity of the referral. But to verify the authority of the parent zone, the resolver has to ask this zone for credentials.

This validation process for certificates is done recursively up the tree, starting at the name server that provides the queried mapping. The recursion will stop at some point, either at the root, or at some intermediate node that was certified before. The certificates that a name server holds are subject to timeouts, just like the resource records that specify bindings of this name server. The certificate for the root must be transmitted by some trusted, out-of-band mechanism. For example, the root certificate could be published in a national newspaper.

Even if an attacker manages to get a valid certificate of a name server it wants to impersonate, and has the capability to also spoof this name server's IP address, it is still not possible for the attacker to impersonate another host. As we saw in the previous Section 4.11.2, a DNS message is encrypted with the name server's private key before it is sent out. The credentials are part of the message and are therefore also encrypted. An attacker cannot construct the correctly enciphered message without breaking the public key system used.

4.11.4 Example

We present an example to show how certificates are used in our approach. We assume that all hosts already have the public keys of the machines that participate in this example. Host "host.aim." wants to resolve the name–to–address binding for the name "host.domain.dom.". The example is not complete in the sense that all possibilities are not covered, or else reasons are given why a name server returns a certain referral and not another one. But it describes the overall interaction and stresses the use of certificates.

Table 4.1 contains a summary of the zones in Figure 4.3, and Table 4.2 interprets the abbreviations used through¡out the description of the resolution process.

Table 4.1 Example: certificate validation

Zone Name	Domain Name(s)	Name Server(s)
.	.	ns
	dom	ns
domain.dom	domain.dom	ns1.domain.dom
		ns2.domain.dom
aim	aim	ns.aim

Table 4.2 Example: legend of abbreviations

Name	Meaning
$MD(m)$	message digest (fingerprint) of message m
$K^{owner}_{pub/priv}$	key of owner – public/private
$E_K(s)$	s encrypted with key K
$D_K(s)$	s decrypted with key K

- "host.aim" queries "ns.aim" for name–to–address resolution of "host.domain.dom".

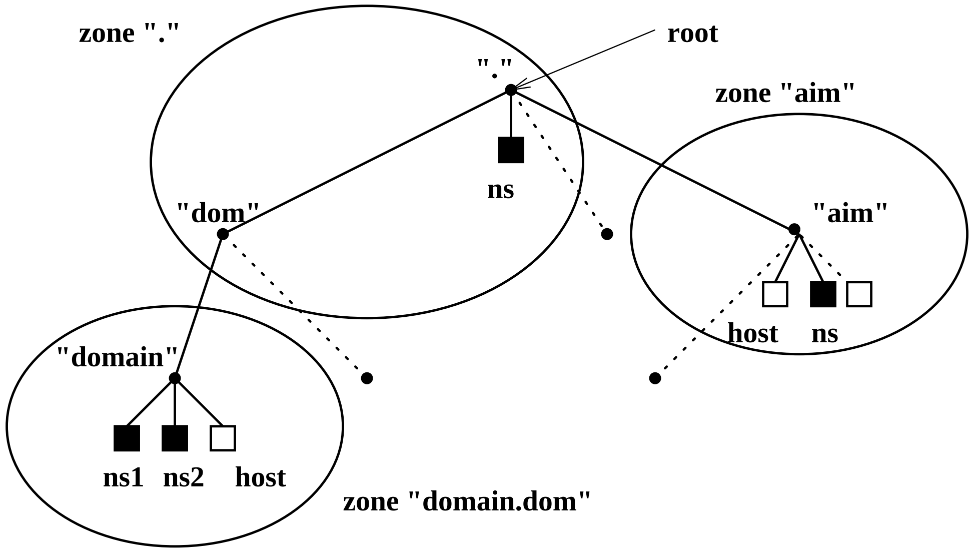

Figure 4.3 Example: certificate validation

- "ns.aim" replies with a referral to "ns2.domain.dom"

- "host.aim" queries "ns2.domain.dom" for name–to–address resolution of "host.domain.dom".

- "ns2.domain.dom" replies with (m,c,s), where

  m = mapping information "host.domain.dom" $\rightarrow$ $\mathrm{IP}_{\mathrm{host.domain.dom}}$

  c = credentials from "ns2.domain.dom"'s parent zone's name server "ns"

  = $E_{K^{ns}_{priv}}$ (list of all IP addresses of authoritative name servers for zone "domain.dom")

  s = encrypted message digest of m concatenated with c

  = $E_{K^{ns2.domain.dom}_{priv}}(MD(m|c))$

- "host.aim" receives (m,c,s), and then

- validates s, by calculating $s' = MD(c|s)$ and $s'' = D_{K_{pub}^{ns2.domain.dom}}(s)$ and comparing them.
  If they are equal $\rightarrow$ ok!
- validates c, by calculating $L = D_{K_{pub}^{ns}}(c)$ and checking if $\mathrm{IP}_{\mathrm{ns2.domain.dom}} \in L$.
  If so $\rightarrow$ ok!
- checks if "ns" is already validated (previously, or root name server).
  If "ns" were not a root name server, "host.aim" would request credentials from "ns"'s parent zone and validate them the same way

4.11.5 Discussion

The validation of integrity and originator of the message, and its underlying pattern of certifications stating trust are the features that make this approach secure. The following discussion shows its disadvantages. Some of them are serious enough to block an implementation of this approach at the current time.

The whole procedure is very time and space consuming. Many rather long public keys have to be stored (about 200 decimal digits long each to make the public key encryption reasonably strong). Obtaining memory for them, as well as additional cache memory for larger resource records, is not a problem in current architectures. The keys have to be obtained before they can be used. S. Kent describes in [Ken93b] certificate based key management; X.509 is the equivalent in the OSI[2]-world. We will not go into detail regarding the key distribution process. The registering process is rather cumbersome. The calculations to encrypt and decrypt message digests may take too long to support the goal of the Domain Name System of efficiency. The additional data that has to be transmitted would not degrade performance too badly, especially if faster transmission media becomes broadly available, but the calculation overhead for encryption and decryption cannot easily be amortized.

[2] "Open Systems Interconnection – A reference to protocols, specifically ISO standards, for the interconnection of cooperative computer systems." [Com91]

The implementation of such a solution is a major effort. The whole key management problem is complex and it also requires additional administrative effort. Resolver routines and name server routines have to be modified, along with the DNS protocol. The implementation is feasible, though very complex. Another drawback is the transition phase that is necessary because of protocol changes.

Overall, the method seems to be hardly feasible, because of its large computational overhead. Further drawbacks are the necessary protocol changes and the complexity of proper key and certificate management.

5. CONCLUSIONS AND OUTLOOK

The Domain Name System is the world's most distributed database, managing name resolution for about 1.5 million hosts. In this thesis we outlined and explained the current implementation of the Domain Name System.

We stated the main problem we are dealing with in this thesis: name based authentication, where the name resolution process cannot be trusted. We examined a method to abuse the Domain Name System for system break–ins and showed that this method exploits several weaknesses. All these weaknesses are necessary before the break–in is possible. We demonstrated the feasibility of the break–in by describing our implementation in an experimental network, set up to satisfy the necessary assumptions that match the real world situation in the crucial points.

We provided the security considerations found in the official design documents and analyzed name server and resolver algorithms with respect to security flaws or weak assumptions. Most of the solutions presented are not complete solutions to the problem in the sense that they cannot prevent the break–in unconditionally. However, a combination of some of the proposed solutions increases the security of the Domain Name System and gives a high confidence in security, although complete security is not achieved.

We consider the Berkeley patch to be mandatory. The current implementation of the Trusted Network concept in UNIX is far from being optimal from a security point of view. We propose major improvements in its design, which would also take care of other shortcomings in the security of systems.

Future work could implement some of the solutions we gave in the previous chapter. Experience with the implementation of policy based solutions would give deep insight into the applicability of these approaches.

An implementation of the solution presented in Section 4.11 (Digital Signatures and Public Key Encryption) would provide a test environment to determine runtime costs of that approach. These results in connection with results of the experiences gained with the PEM system could lead to surprising conclusions. Despite the many disadvantages we found, we still consider this solution worth some more thought and examination.

BIBLIOGRAPHY

BIBLIOGRAPHY

[AL92] Paul Albitz and Cricket Liu. *DNS and BIND.* O'Reilley & Associates, Inc. Sebastopol, CA., 1992.

[Bel89] Steven M. Bellovin. *Security Problems in the TCP/IP Protocol Suite.* AT&T Bell Laboratories, Murray Hill, New Jersey, April 1989.

[Bel90a] Steven M. Bellovin. Pseudo-Network Drivers and Virtual Networks. In *Proc. Winter USENIX Conference*, pages 229–244, Washington, D.C., 1990.

[Bel90b] Steven M. Bellovin. *Using the Domain Name System for System Break-ins.* AT&T Bell Laboratories, Murray Hill, New Jersey, 1990. (unpublished technical report).

[Bel92] Steven M. Bellovin. There Be Dragons. In *UNIX Security Symposium III Proceedings*, pages 1–16, Baltimore, MD, 1992.

[BG92] Dimitri Bertsekas and Robert Gallager. *Data Networks.* Prentice-Hall, Englewood Cliffs, New Jersey, second edition, 1992.

[CD88] George F. Coulouris and Jean Dollimore. *Distributed Systems.* Addison-Wesley Publishing Company, Inc., 1988.

[Com91] Douglas E. Comer. *Internetworking with TCP/IP.* Prentice-Hall, Englewood Cliffs, New Jersey, second edition, 1991.

[Den82] Dorothy E. Denning. *Cryptography and Data Security.* Addison-Wesley Publishing Company, Inc., 1982.

[DK84] Kevin J. Dunlap and Michael J. Karels. *Name Server Operations Guide for BIND, Release 4.8.* University of California, Berkeley, CA, May 1984.

[DOK92] Peter B. Danzig, Katia Obraczka, and Anant Kumar. An Analysis of Wide-Area Name Server Traffic. *Computer Communications Review*, 22(4):281–92, October 1992.

[GS91] Simson Garfinkel and Gene Spafford. *Practical UNIX Security.* O'Reilley & Associates, Inc. Sebastopol, CA., 1991.

[Hun92] Craig Hunt. *TCP/IP Network Administration.* O'Reilley & Associates, Inc. Sebastopol, CA., 1992.

[Kal92] Burton S. Kaliski. *RFC-1319 The MD2 Message-Digest Algorithm.* Network Working Group, April 1992.

[Ken93a] Stephen T. Kent. Internet Privacy Enhanced Mail. *Communications of the ACM*, 36(8):48–59, May 1993.

[Ken93b] Stephen T. Kent. *RFC-1422 Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management.* Network Working Group, February 1993.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *Programmieren in C.* Carl Hanser Verlag München Wien, second edition, 1988.

[Lot93] Mark Lottor. Internet Domain Survey Apr 93. SRI International, April 1993.

[LR93] Daniel C. Lynch and Marshall T. Rose. *Internet System Handbook.* Addison-Wesley Publishing Company, Inc., 1993.

[Mad92] Jørgen Bo Madsen. The greatest cracker-case in Denmark: The detecting, tracing and arresting of two international crackers. In *UNIX Security Symposium III Proceedings*, pages 17–40, Baltimore, MD, 1992.

[Mer89] Ralph C. Merkle. Snefru. Xerox Corporation, Palo Alto, CA, 1989.

[Moc83a] Paul Mockapetris. *RFC-882 Domain Names - Concepts and Facilities.* Network Working Group, November 1983.

[Moc83b] Paul Mockapetris. *RFC-883 Domain Names - Implementation and Specification.* Network Working Group, November 1983.

[Moc87a] Paul Mockapetris. *RFC-1034 Domain Names - Concepts and Facilities.* Network Working Group, November 1987.

[Moc87b] Paul Mockapetris. *RFC-1035 Domain Names - Implementation and Specification.* Network Working Group, November 1987.

[Moc89] Paul Mockapetris. *RFC-1123 Requirements for Internet Hosts – Application and Support.* Network Working Group, 1989.

[Mor85] R. T. Morris. A Weakness in the 4.2BSD UNIX TCP/IP Software. Computing Science Technical Report No. 117, AT&T Bell Laboratories, Murray Hill, New Jersey, February 1985.

[Par91] T. A. Parker. A Secure System for Applications in a Multi-vendor Environment (The SESAME project). In 14^{th} *NCSC Conference Proceedings*, 1991. Vol. II.

[PL91] R. Paans and H. de Lange. Auditing the SNA/SNI Environment. *Computer & Security*, 10(3):251–61, May 1991.

[Riv92a] Ronald L. Rivest. *RFC-1320 The MD4 Message-Digest Algorithm.* Network Working Group, April 1992.

[Riv92b] Ronald L. Rivest. *RFC-1321 The MD5 Message-Digest Algorithm.* Network Working Group, April 1992.

[RSA78] R. Rivest, A. Shamir, and L. Adleman. A Method for Obtaining Digital Signatures and Public Key Cryptosystems. *Communications of the ACM*, 21(2):120–6, February 1978.

[SNS88] J.G. Steiner, C. Neuman, and J.I. Schiller. Kerberos: An Authentication Service for Open Network Systems. In *Proceedings, Winter USENIX*, Dallas, Texas, 1988.

[Spa88] Eugene H. Spafford. The Internet Worm Program: An Analysis. Technical Report CSD-TR-823, Purdue University, West Lafayette, IN, 1988.

[Ste90] Richard W. Stevens. *UNIX Network Programming.* Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[Sto89] Clifford P. Stoll. *The Cuckoo's Egg: Tracing a Spy Through the Maze of Computer Espionage.* Doubleday, 1989.

[Sun91] Sun Microsystems. *manual pages*, 4.1 edition, January 1991.

[Tan92] Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

[Tho84] Ken Thompson. Reflections on Trusting Trust. *Communications of the ACM*, 27(8):761–3, August 1984.