Selected Aspects of Computer Security in Open Systems

Danny Smith
Australian Computer Emergency Response Team
c/- Prentice Centre
The University of Queensland
Qld. 4072. Australia

+61 7 365 4417
auscert@auscert.org.au

8 November, 1993

Abstract

The past five years have seen a lot more attention being focused on computer and network security. This heightened awareness has given rise to a better understanding of the types of threats a computer system might face. It has also seen the development of new tools and techniques to combat those threats.

This report examines those security threats, and details what tools and techniques are available to combat them. These threats are categorised using standard criteria. In addition, an examination of common authentication techniques is provided, including a comparison between X.509 Strong Authentication and Kerberos.

The suite of available tools is complemented, through the development of a one time password system for OpenVMS. This development is also detailed.

1. Introduction

In November 1988 a "worm" program was released to the Internet, which (although not entirely intended by the worm's author) brought much of the network to a standstill [Spa88]. Many sites disconnected from the network until a solution could be found. This had an added disadvantage as many of the first fixes for the problem were distributed via the network.

In March 1991 a ship was lost in the Bay of Biscay. A weather forecasting satellite was not working as the integrity of a computer system at the European Weather Forecasting Centre in Bracknell, Berkshire had been broken by intruders[1]. The storm that caused the ship to sink had not been predicted [Aus93].

In mid-1993, a number of sensitive medical test results were changed from negative to positive by intruders. Various people around that country were then led to believe that they had cancer. It was up to Police to advise them that the results had been changed and were incorrect [Aus93].

Often when the topic of computer security is discussed, particularly when it relates to intruders gaining unauthorised access, a wide range of viewpoints can be expressed. Some people view intruders as "fighters for freedom of information"; others view them as "joyriders in the electronic space"; others view them as "intellectual joyriders". Mostly the intruders do not cause harm, but when intruders perform acts like the sample described above, it is difficult to tell the difference between the "good" intruder and the "bad" intruder. Therefore, these intruders must now be treated as criminals.

This report examines the topic of computer security for Open Systems. In this report, an Open System is one that is connected to a global network without any network access restrictions. There are many other more restrictive definitions for an Open System, but this one will suffice for the purposes of this report.

Over the past five years, computer vulnerabilities and the way they are exploited has changed character. In 1988, many intruders were simply exploiting poor password choices, poor system configuration, or software vulnerabilities [CER92]. However, in 1993, those types of attacks are still common, with the addition of people viewing network information, either through modifying the network software or looking at data that traverses the network links. Intruders are now also actively examining source code for operating systems and utilities in an attempt to determine any potential vulnerabilities, or attempting to exploit poor protocol design to try and fool servers into performing some action.

This report examines the common threats faced by computer systems, and looks at various tools and techniques that have been developed to combat these threats. Its aim is to examine what tools and techniques are available to combat the known threats. As a result of this examination, an extension is provided to the security on one open system (OpenVMS) where a deficiency is perceived.

---

[1] This report uses the term "intruder" to indicate a person who enters a computer system, or who accesses or modifies data, and does not contain the appropriate authorisation to do so. Other terms are used in the public domain such as "hacker" and "cracker", and these terms are continually debated for appropriateness. The term intruder is chosen here, as it indicates the criminal and antisocial intent of the activity, as viewed by the author. Other terms which would also been seen to be acceptable to the author are "criminal", "vandal", or "data terrorist".

Whilst this report concentrates heavily on the Unix operating system, many of the vulnerabilities and techniques to combat them apply to all operating systems. The approach taken is not a formal one, but a comparison of services provided from a user perspective. There are many papers that formalise the analysis of security (for example, see [LAB92]).

This report commences with an examination of authentication techniques, as authentication forms the backbone which other security services build upon. It is for this reason that the author believes that more attention should be devoted to it. This examination is found in Chapter 2.

Chapter 3 examines common threats faced by computer systems today. These threats involve more than authentication services, and special attention is devoted to categorising them based upon standard criteria. In particular, this chapter captures the wider spectrum of computer security, as some of the detailed threats bypass authentication, rendering them potentially more dangerous than they might first appear.

Chapter 4 contains an examination and evaluation of what tools are readily available to combat the common threats outlined. These tools have been developed to address the problems of computer security.

Once the general problems of security are detailed and solutions proposed, an extension is proposed to an existing system (OpenVMS) for which a common solution (one time passwords) to a standard problem (network monitoring) is not freely available. A design and implementation is supplied in Chapter 5 for the proposed extension. All source code and header files are found in Appendix A.

2. Authentication Techniques

This chapter reviews computer security services in broad terms, and indicates why authentication is important. It examines the threats that authentication faces, and then compares some of the commonly used authentication techniques in use, looking at their strengths and weaknesses. These techniques generally involve the use of a password to effect the authentication.

Some of the newer techniques use public and private key encryption techniques to protect and enhance the authentication sequence. These authentication techniques are also examined, looking at their strengths and weaknesses.

2.1. What is Security?

Businesses today require the processing and exchange of information to conduct their business dealings. The Twentieth Century has been described as "The Information Age". Security of that information is now paramount if business is to survive in this new world [CLS91], [Edw90].

In order to protect against perceived threats, various security services need to be provided. These can include [ISO92]:

- *Authentication*: the act of identifying that an entity is who or what they are claiming to be;
- *Access Control*: the act of deciding whether an entity has the rights or authority to perform the task they are seeking to do;
- *Data Confidentiality*: the act of protecting data from unauthorised disclosure;
- *Data Integrity*: the act of protecting data from manipulation;
- *Non-repudiation*: the act of proving the integrity and origin of data in an unforgeable relationship which can be verified by a third party at any time.

When imposing access controls, judgment must be reliable and predictable , and the contents of an audit trail should be accurate to resolve any problems that might occur. Data Integrity and Confidentiality may be questioned if the identity of the entity being communicated with is in doubt [JM91]. Correct identification is impossible if the act of authenticating the entity is deficient, and doubts are cast on the integrity of the authentication[2].

2.2. What are the Threats to Data Security?

The act of authenticating an entity must be trustworthy if there is to be any security in the computer system at all. This means that the authentication facility must correctly identify the entity in all cases, regardless of the role played by the entity, the location of the entity, and the presence or absence of potential enemies. The commonly known threats [ISO92] to data security include identity interception, masquerade, reply, data interception, manipulation, repudiation, denial of service, misrouting, and traffic analysis. These threats are more fully defined in Section 3.1.

This remainder of this chapter examines a number of authentication schemes and indicates if any protection is provided for Identity Interception, Masquerade, Replay, and Repudiation threats.

---

[2]Then again, can the authentication be believed if the access control and data integrity are insufficient to protect the authentication data?

One of the aspects of authentication that is often overlooked is that it is a two-way street. When a user authenticates to a computer, it is often just as important that the computer authenticates to the user. This has been highlighted in recent times with people starting "trojan horse" programs that imitate the login sequence, but in reality capture the account and password for use by an enemy. Another example is an automatic teller machine that captures account numbers and PINs for later use. The ability to authenticate the machine will help to prevent these types of attack.

2.3. How is Authentication Achieved?

Authentication is generally achieved through one or more of three items [CLS91]. These are:

> Something you *know*. This could for example be a password or Personal Identification Number (PIN);
> Something you *possess*. This could be an identity card, an access card for electronic locks, or a smart card;
> Something you *are*. This involves something personal about the subject such as a retina scan. It may also involve the role played when seeking authentication. For example, a person's rights to perform an action are vastly different if they acting as a Lecturer in a Department, as opposed to acting as the Head of that Department. The role of Head may be delegated via authentication.

Most authentication techniques make use of one of these items (such as knowing a password, possessing a key to an office, or signing of a name as the Head of Department). Whilst in many circumstances this is sufficient, the authentication can be enhanced by combining two or more of these items together. Examples of this might be possessing an identity card indicating the status of Head of Department which must be shown to security personnel, plus possessing a key to that office. If an attempt to enter the office of Head of Department with an identity card that indicates the status of a lecturer is attempted, then the security personnel should forbid this.

Another example in common use is the possession of a magnetic stripe card from the bank, and knowing the PIN for that card.

The following sections investigate different authentication techniques, describing their attributes according to this criteria, and looks at how well they cope with the threats detailed above.

2.4. Passwords

2.4.1. Normal Passwords

This is by far the most common authentication technique used on computer systems today. It is categorised by "something you know" which is a user account and the password for that account. If this information is revealed in any way, then an enemy can authenticate to the computer system, pretending to be a legitimate user.

Passwords are stored and handled in many different ways by different computer systems.

<u>2.4.1.1. Plain Text, Non-world Readable</u>

International Business Machine's VM/CMS stores its passwords in the directory file in plain text (EBCDIC), but to access this file requires special privileges and permissions which are normally only granted to a small number of users [IBM89].

Strengths

This system is very easy to implement, easy to maintain, and easy to logically prove that it is behaving correctly (that is, it can be trusted).

Should a user forget their password, it is relatively easy for an authorised person to find out what the password was, and reveal this information to the user.

Weaknesses

If access is gained to the password file by an unauthorised person, then *every* account and password on the system is revealed. This means that the Virtual Machine concept that so effectively keeps users separated is compromised, and any and all data on the system is available to the enemy.

Since all passwords are stored in plain text, then security personnel for that system will have access to every password. This would include passwords for the Head of Department, leading researchers, system administrators, and general users. It would be relatively easy to access any information on the system, masquerading as the owner of that information, thus bypassing the access control and auditing facilities.

2.4.1.2. World Readable, but Encrypted

This is the technique mostly used on UNIX systems. The password is encrypted using a one way function (one that is computationally difficult to reverse), and the encrypted version stored in a world readable file. The encryption is further protected by "salting" the algorithm with a seed chosen from 4096 possible values. This means that a single password may produce up to 4096 possible encryptions [GS91].

Strengths

The algorithm to encrypt passwords operates by encrypting a null string several times over with an altered version of the DES [FIP77] algorithm using the plain text password as the key. To date, nobody has successfully reversed this algorithm in a reasonable time.

By having the password file world readable and the algorithm publicly available, it simplifies the design of applications that require authentication services. Therefore, it is simpler to prove that these applications are performing correctly (that is, they can be trusted).

The UNIX authentication system has been subject to much attention due to its public nature. It has been studied, targeted, tested, and analysed by many people. This should give a user confidence that the algorithm is computationally strong enough for the present moment (provided a good password is chosen - see [SER93]).

Weaknesses

The *major* weakness of this authentication technique is the combination of the world readable file, and human nature. Whilst the encryption algorithm has not been reversed, much success has been gained by guessing the plain text password, encrypting it, and checking the encrypted value against the encrypted password stored in the password file. These password guessing attacks are notoriously successful because human users often pick passwords that are easy to remember, which often means that they are easy to guess [CLS91].

Another weakness also involves brute force attacks. With the speed of computers ever increasing in leaps and bounds, it may be feasible one day to just try all possible passwords against the encrypted version. The current estimates for encryption speeds indicate that 6,400,000 passwords encryptions may be achieved *per second*. This enables all combinations of six character passwords to be tested in less than one hour [Mad93]. The ability to check all eight character (maximum length) passwords may not be all that far in the future[3]. Therefore, it is imperative that the password be changed regularly, well chosen, and not reused.

2.4.1.3. Non-world Readable Passwords

One of the major problems with UNIX style passwords is that the encrypted password is freely available. Many attacks are performed by taking a copy of the encrypted password to another machine, and applying attacks such as dictionary attacks in an attempt to guess the password. Often the remote machine belongs to another site, and has been compromised previously. The crackers use the resources of one organisation to gain access to another organisation.

One mechanism to prevent the encrypted password from becoming known is to store it in a non-world readable file. This concept is called shadow passwords on UNIX systems. The utilities that require access to the encrypted password require special privileges to perform their tasks. The encrypted passwords can still be read by privileged users, but by protecting the shadow password file appropriately, then the normal system user cannot gain access to the information. This means that even if a password is guessed, and access gained to the system, the rest of the encrypted passwords are still secure.

Digital Equipment Corporation's OpenVMS has had the concept on non-readable authorisation files for many years now [DEC90]. Access can be gained to the authorisation file through the use of special system calls and utilities. Access to some parts of the authorisation file are freely available, whilst access to others is restricted to privileged users.

This still does not prevent password guessing attacks. However, since the password file is now not world readable, the password guessing attacks will need to be performed on the machine under attack. Both OpenVMS and UNIX implement mechanisms to prevent or limit the success of such attacks[4].

Ultimately, the success of the password mechanism is up to the individual user to ensure that their password is well chosen, and not easy to guess. Analysis of how to form a well chosen password can be found in many places. Some examples are [SER93], [Cur90], and [Bis92a].

Strengths

It now requires privileged access to copy the password file and take it away to another machine to run password guessing programs on it. This means that if one user allows access to the system via an easy to guess password, the rest of the passwords are not at risk by that user's actions.

Since the encrypted password is no longer freely available, password guessing attacks need to be performed on the local machine (by attempting to login to the account

[3] At this speed, the time required to check all eight character passwords is 31 years.

[4] These facilities are beyond the scope of this report.

using different passwords). This allows the machine to take action against this type of attack, and enforce a consistent security policy.

Weaknesses

Now that it requires privileged code to access the encrypted password, it requires special care must be taken to ensure that extra security holes are not created. This means that it is becoming more difficult to prove that the security system is behaving as expected (that is, it can be trusted).

Such systems are sometimes not standard, as shipped with the original operating system code. Therefore, they are security "add-ons" which are provided by a third party. This means that they may not have received a thorough testing, or may be inconsistent with existing operating system code. Generally however, most of these types of security systems have been around long enough to have the major bugs fixed in them.

The password is still transmitted in plain text on the network, thus rendering this scheme susceptible to replay attacks.

2.4.2. Passwords - Secure?

Are passwords all that secure then? It is possible to protect them from being revealed, either while they are stored on the system or being transmitted as part of an authentication sequence.

Klein [Kle90] completed a study of password security by attempting to "guess" passwords that were in use on various machines. After 12 CPU months, more than 25% of the passwords were guessed[5]. Of those passwords guessed, more than half of them were six characters or less. This, coupled with the huge advancements in computer speed almost brings a brute force attack into the realms of reality - no password will be safe [Alv90]!

Guessing just ONE password is enough to allow access to a system. Having achieved this, an intruder has taken the first step to gaining further access to the computer system [Che92]. Some suggest that a hit rate of over 50% is not uncommon the first time a password guessing program is run [KC90]. Of five documented security incidents at Columbia University, all were as a result of bad passwords [BKS90].

The problem with passwords is not the password itself, but the people choosing them [Sto89]. If all passwords were a MINIMUM of eight characters, and chosen well, then this type of attack would at best, yield minimal results [Spa92].

In the past, it was always considered poor practice to write a password down. Now instead of writing it down, it is simpler for a user to choose a password that is easy to remember (which most likely means it is also easy to guess). Maybe it is time to start rethinking the policies, and suggest that it *is* alright to write a password down, provided that the password is stored in a safe place. The literature is full of papers where passwords are guessed, but no article indicated that someone's wallet was stolen to retrieve the password. Even if it is, the owner will know that their password has been compromised, rather than relying on chance to discover that someone has guessed their password [Bra90].

[5] 21% were guessed in the first week, and 368 (2.7%) in the first 15 minutes of testing.

The use of default passwords and poorly chosen passwords has caused many problems over the years [CER92]. This is the major flaw that allowed the WANK and OILZ worms to operate [LS93].

The choice of passwords can be improved by preventing users from changing their password to a known "bad" password. The checks for a bad password are often done at the time the user changes their password, using a form of pro-active password checker (Some examples are [Bis92a], [Kur90], and [Spa92]).

2.4.3. One Time Passwords

One-time password generators generally employ a combination of hardware and software to effect the authentication. This technique employs "something you possess" and in some systems, "something you know". The hardware devices are usually known as "tokens" or "password tokens". There are several techniques in use [Ell92], [CER92], [Lau92].

One system operates by issuing a challenge which consists of a seven digit number (in phone number format). The user enters their PIN number and the challenge number into the hand held device which returns a seven digit response code to reply with.

Other systems use a changing, non-reusable password system. Each time the user authenticates, a new password is supplied by the hand held device. There is no challenge-response system, and the user must keep in synchronisation with password usage to prevent a denial of service. Some of these systems can support single use password generation for up to eight separate host systems. Some of these systems require that a PIN be entered before the next password is issued.

Another system displays the password continuously, changing it every minute or so. The host must not only keep the user's key (for generating the same sequence), but also a synchronised clock. This system is vulnerable to replay attacks if the password is captured and replayed prior to it changing. Some systems combat this by only allowing each password to be used once.

Some of the systems that employ the use of a PIN allow a second PIN to be specified. This may be useful for example to alert security staff that the user is being held and forced to access the computer system against their will. The second PIN may allow the user access (in a restricted sense), but also signal alarms!

S/Key is a software only system that requires the user to carry a list of generated passwords with them [KHW93]. S/Key is discussed in Section 4.4.9 and Chapter 5.

Strengths

The one-time password system is extremely effective in preventing replay attacks, provided the enemy does not know the sequence of generated passwords (either by guessing, or possession of a similar device and key). Many attacks in the present time will be instigated by monitoring authentication sequences [CER92].

Extra techniques can be employed that allow limited access, whilst also generating an alarm to indicate that this authentication sequence is being forced by an enemy.

Weaknesses

One of the major disadvantages is that to authenticate, the user must carry the hardware or password list with them at all times. If they do not possess the hand held device, then authentication cannot take place.

Some of the systems have a requirement for synchronised clocks. This suffers the same problems that Kerberos does, with denial of service due to clocks slipping, or an attacker targeting the clock synchronisation protocols to set the time to any desired value.

One particular system used generated keys that were so short, that it was easier to guess the password "1111" than it was to guess the correct password for the generated key [Bra90].

Systems that make use of a password that changes at time intervals may be prone to replay attacks if special precautions are not taken to prevent them.

2.5. Kerberos

The following analysis is drawn from [Ste90], [BM91], and [KCS90].

The Kerberos authentication system was produced at MIT as a part of Project Athena. It is a system that uses protocols which allow authentication to take place, even under the assumption that the network is under the complete control of an enemy. Kerberos uses a private key cryptosystem to protect the information from disclosure and modification. The user interface is the same as that for normal passwords, so this authentication technique is categorised as "something you know".

Kerberos authenticates a principal by providing evidence of that principal's identity. The principal can be a user or a particular service on some machine. A principal consists of the tuple:

<primaryname, instance, realm>

If the principal is a user (a person), then the primary name is the login identifier, and the instance is either null or represents particular attributes of the user. For a service, the service name is used as the primary name and the machine name is used as the instance. The realm is used to distinguish among different authentication domains. It is therefore possible to have several Kerberos databases serving an organisation rather than one large and universally trusted one.

In the following discussion, the symbols used are defined as:

c	client principal
s	server principal
tgs	ticket granting server
K_x	private key of "x"
$K_{c,s}$	session key for "c" and "s"
$\{ info\}K_x$	"info" encrypted in key K_x
$\{ T_{c,s}\}K_s$	Encrypted ticket for "c" to use "s"
$\{ A_c\}K_{c,s}$	Encrypted authenticator for "c" to use "s"
addr	client's IP address

Kerberos principals may obtain tickets for services from a special server known as the ticket-granting server or TGS. A ticket contains assorted information identifying the principal, encrypted using the private key of the service. `timestamp1` indicates the time the ticket was issued, and `lifetime` the amount of time the ticket is valid for.

$$\{ T_{c,s}\}K_s = \{ s, c, addr, timestamp1, lifetime, K_{c,s}\}K_s$$

Since only Kerberos and the service share the private key K_s, the ticket is known to be authentic. The ticket contains a new private session key, $K_{c,s}$, known to the client as well (when it obtains the ticket); this key may be used to encrypt transactions during the session which protects them from disclosure and modification.

To guard against replay attacks, all tickets presented are accompanied by an authenticator:

$$\{ A_c\}K_{c,s} = \{ c, addr, timestamp2\}K_{c,s}$$

This is a string encrypted in the session key and containing `timestamp2` (the current time); if the time does not match the current time within the (predetermined) clock skew limits, the request is assumed to be fraudulent. This clock skew limit is usually in the order of five minutes.

For services where the client needs bidirectional authentication, the server can reply with:

$$\{ timestamp2 + 1\}K_{c,s}$$

This demonstrates that the server was able to read timestamp from the authenticator, and hence that it knew $K_{c,s}$; that in turn is only available in the ticket, which is encrypted using the server's private key.

Tickets are obtained from the TGS by sending a request:

$$s, \{ T_{c,tgs}\}K_{tgs}, \{ A_c\}K_{c,tgs}$$

In other words, an ordinary ticket/authenticator pair is used; the ticket is known as the ticket-granting ticket. The TGS responds with a ticket for server `s` and a copy of $K_{c,s}$, all encrypted with a private key shared by the TGS and the principal:

$$\{ \{ T_{c,s}\}K_s, K_{c,s}\}K_{c,tgs}$$

The session key $K_{c,s}$ is a newly-chosen random key.

The key $K_{c,tgs}$ and the ticket-granting ticket itself, are obtained at session-start time. The client sends a message to Kerberos with a principal name; Kerberos responds with:

$$\{ K_{c,tgs}, \{ T_{c,tgs}\}K_{tgs}\}K_c$$

The client key K_c is derived from a non-invertible transform of the user's typed password. Thus all privileges depend ultimately on this one key (the user's password).

Note that servers must possess private keys of their own, in order to decrypt tickets. These keys are stored in a secure location on the server's machine. The security of the tickets and ultimately, the session key depends upon the security of the server private keys.

Strengths

The major strength of Kerberos is that the password is never transmitted on the network in plain text. This reduces the likelihood of the password being captured and replayed.

The tickets and authenticators include a timestamp which goes a long way to preventing replay attacks. If an authenticator is captured and replayed, it will not be valid the second time around provided sufficient time has passed by. To use this type of attack successfully requires either breaking the encryption algorithm, or possessing the password, the session key, or the server key.

This style of authentication was designed with the distributed or networking environment in mind. It is well suited to the client-server model often used in networking applications.

Since both the ticket and authenticator contain the network address of the client, another workstation cannot use stolen copies without changing their network address to be the same as the one that owns the stolen copy.

Weaknesses

The timestamps are critical to the successful operation of Kerberos. The times on the source and target machines must be closely aligned, or it will be possible that a valid ticket will be rejected as fraudulent because the time is incorrect. Typically, a clock drift of five minutes will cause a denial of service.

Relying on the time to be in synchronisation means that one should also protect the protocols that set the time, so that an enemy cannot adjust the time to their will via this mechanism.

Tickets are reusable. However, after having been authenticated for a long period (typically eight hours), it is necessary to generate a new ticket. This is because every ticket has a lifetime so once it expires, a new ticket must be generated by entering the login name and password to Kerberos again.

Within the Project Athena environment (and hence, Kerberos), the primary need is for user to server authentication. That is, when a user sits down at a workstation, that person needs access to private files residing on a server. The workstation itself has no such files, and hence has no need to contact the server or even identify itself. This is in marked contrast to a typical UNIX system's view of the world. Such systems do have an identity, and they do own files. Assorted network daemons transfer files in the background, clock daemons perform management functions, electronic mail and news is transferred, and so on. If such a machine relied on servers to store its files, it would have to prove its identity when talking to these servers. This means that Kerberos is designed to authenticate the end-user - a human being sitting at a keyboard. It is not a peer to peer system, intended to be used by one computer's daemons when contacting another computer.

In a workstation environment, it is quite simple for an intruder to replace the login command with a version that contains a trojan horse (captures accounts and passwords). Such an attack negates the primary strength of Kerberos, that passwords are not transmitted in plain text over a network. While this problem is not restricted to Kerberos environments, the Kerberos protocol makes it difficult to employ the standard countermeasure: one-time passwords.

The authenticator relies on the use of a timestamp to prevent against replay. Given that the lifetime of an authenticator is typically five minutes, a window of opportunity exists where a stolen live authenticator could be used to fraudulently gain access to a server. It has been suggested that the proper defence is for the server to store all live authenticators so that a replay could be detected. However, on UNIX systems, TCP-based servers generally operate by forking a separate process to handle each incoming request. Since the child and the parent do not share any memory, it is not convenient to communicate to the parent (or any other child processes) the value of any authenticator that is presented. UDP-based query servers generally use a single process to handle all incoming requests, but may have problems with legitimate retransmissions of the client's request if the answer gets lost.

Whilst the Kerberos system guards against having to send the password in plain text, the passwords chosen are much the same as the standard UNIX password, and therefore suffer the same fate of normal passwords in a password guessing attack. At least the encrypted password is not freely available (it is stored on the Kerberos server), so the success of such an attack is much the same as the system where the passwords are not world readable. To succeed with such an attack would require the password to be guessed, rather than just replaying the authentication sequence again. This would allow an enemy to masquerade as another user, without using replay.

Tickets are based upon a system's IP address. On multi-homed systems (systems with more than one network interface and IP address), this may cause a problem as the ticket will only be valid through one of those interfaces.

Tickets are stored in `/tmp` which does not work very well for multi-user systems.

2.6. X.509 Authentication

X.509 is a CCITT Recommendation (as well as being Part 8 of ISO International Standard 9594) [ISO92]. This standard forms the Authentication Framework, initially for use by the X.500 Directory Service. The concepts and facilities provided by this framework could be used by other OSI applications, or by network applications in general that require such facilities.

The strong authentication techniques of X.509 make use of public key cryptosystems (PKCS), unlike Kerberos. The recommendation states the requirements of the PKCS, without dictating any particular algorithm. It also includes the mechanisms for specifying the algorithm in use, and therefore no single algorithm is required to successfully use X.509 services. One particular algorithm that is commonly known, and satisfies the criteria dictated by X.509 is RSA [RSA78].

Whilst X.509 could be categorised by "something you know" (the password), it is possible that the secret key for the PKCS (when using strong authentication) could be so complex that it is necessary to store it on something like a "smart card". This system might require a smaller password to unlock the smart card. In this case, the authentication is categorised by "something you possess" and "something you know".

The recommendation describes more facilities than just authentication (such as Certificates, Certification Authorities, and Digital Signatures), but the scope of this report is limited to the use of digital signatures and authentication facilities. The authentication facilities range from very simple authentication, to extremely strong mechanisms.

2.6.1. Digital Signatures

Information can be *signed* by appending to it an encrypted summary of the information. The summary is produced by means of a one-way hash function while the encrypting is done by applying a public key cryptosystem and using the secret key of the signer[6]. This can be shown as [ISO92]:

$$X\{\text{Information}\} = \text{Information}, Xs[h(\text{ Information})]$$

[6]It is possible to use secret key encryption to implement digital signatures, but in general they are not as efficient. This report only addresses digital signatures that use public key encryption.

The signature can be verified by applying the one-way hash function to the information, and comparing the result with that obtained by decrypting the signature using the public key of the signer[7] [Law93].

2.6.2. Simple Authentication

Simple Authentication is intended to provide local authorisation based upon the name of a user (their username) and an agreed password. Utilisation of Simple Authentication is primarily intended for local use only, and will not be very effective in a wider scope. There are two means by which Simple Authentication can be achieved:

2.6.2.1. Unprotected Simple Authentication

The procedure for achieving unprotected simple authentication is:

1. an originating user A sends their username and password to a recipient user B (the target machine);
2. B consults a local database of usernames and passwords where the password is checked against that held for user A;
3. the success or failure of authentication may be conveyed to A.

Unprotected Simple Authentication involves the transfer of the username and password in clear text to the recipient for evaluation. It is therefore prone to attack from network traffic monitors. As well, the passwords are stored in plain text, and may be discovered if privileged access to the database is gained.

Unprotected Simple Authentication could be compared with the plain text, non-world readable password mechanisms currently in use, such as that used in VM/CMS. If the password is stored in the database in encrypted form, then this mechanism would be comparable to the authentication systems found in Operating Systems such as UNIX and OpenVMS.

2.6.2.2. Protected Simple Authentication

The procedure for achieving protected simple authentication is:

1. an originating user *A* sends their protected identifying information (`Authenticator1`) to a recipient user*B* (the target machine).

   Authenticator1 = timestamp1, random1, username, Protected1
   Protected1 = $f1$(timestamp1, random1, username, password)

   where `random1` is a random number with an optional counter included (where used), `timestamp1` consists of timestamps (where used), `username` is the login identifier for the user seeking authentication, and `Protected1` is `timestamp1`, `random1`, `username`, and `password`, all processed through the one-way function $f1$;
2. *B* verifies the protected identifying information offered by *A* by generating (using the information included in `Authenticator1`, plus a local copy of the password for *A*) a local copy of `Protected1`. *B*

[7] It is not strong enough to encrypt a hash value of the information if the hash value is itself weak. Data manipulation may still occur, producing the same value for the hash function. In this case, the signature will be verified (providing data origin authentication), but the message will have been changed (voiding the data integrity requirements).

compares for equality the locally generated copy of `Protected1` with the one supplied by *A*;
3. *B* confirms or denies the identity of *A* based upon this verification.

The procedure can be further protected by passing `Protected1` through another one-way function $f2$, and including a second random number and/or timestamp.

Authenticator2 = timestamp1, timestamp2, random1, random2, Protected2.
Protected2 = $f2$(timestamp2, random2, Protected1)

There is no requirement for $f1$ and $f2$ to be the same, or different. The principle for verifying the identity of *A* is the same as before.

The use of random numbers and timestamps is to minimise the chance of a successful replay attack, and to conceal the password from a cryptographic attack.

Protected Simple Authentication provides similar facilities to Kerberos in that it prevents the password from being transmitted in plain text, and provides some protection against replay attacks. However, Protected Simple Authentication does not provide bidirectional authentication which Kerberos can.

Strengths

One of the strengths of this technique is that it is relatively easy to implement, and to prove that it is functioning correctly without vulnerabilities. The more complex the system, the harder it is to determine the lack of vulnerabilities.

Another advantage is that it is quick to determine if the authentication attempt is successful or not. For transaction style operations, the time to authenticate a user may become an important issue.

The use of timestamps and random numbers reduces the chance of a successful replay attack.

Weaknesses

Since the username and password are transmitted in plain text, simple authentication can suffer from network monitoring and replay attacks.

The password is stored in the password database in plain text, so it is relying on the database security to protect the password from being revealed.

2.6.3. Strong Authentication

Strong Authentication is defined as "Authentication by means of cryptographically derived credentials" [ISO92]. The approach taken by X.509 makes use of public-key cryptosystems. These systems use two keys, one secret and one public, rather than a single key in private key cryptosystems.

Each user is identified by their possession of the secret key. A second user or system is able to determine if a partner is in possession of the secret key without either learning that secret key, nor revealing their own secret key to anyone.

Strong Authentication can be 1-way, 2-way, or 3-way as described below.

2.6.3.1. One-way Strong Authentication

One-way authentication is used when user *A* wishes to authenticate to user *B* (a system or user). *A* sends the following message:

$$B \rightarrow A, A\{ t^A, r^A, B\}$$

where $B \rightarrow A$ is a chain of certificates (certification path from *B* to *A*) which ultimately verifies the public key for *A*, t^A is a timestamp consisting of one or two dates (generation time and expiry time), r^A is a non-repeating number, and $A\{ t^A, r^A, B\}$ is the authentication information digitally signed by user *A* (by appending an encrypted summary to the end). The timestamps are used to determine the life of the token, and help to detect replay attacks. r^A is valid until the expiry date indicated in t^A.

It is possible to include extra information such as `sgnData` which will be used to provide data origin authentication by digital signatures, and `encData` which will be used as a secret key for the cryptosystem specified in `sgnData`. `encData` is encrypted using the public key of *B*, so that only *B* can read it.

$$B \rightarrow A, A\{ t^A, r^A, B, sgnData, Bp[encData]\}$$

Note: This last form of message (authentication token) in certain circumstances is subject to a form of manipulation attack. For example, suppose a token accompanies a confidential message which is encrypted with a key that is contained in `encData`. An opponent *C* might intercept the transmission of the token plus encrypted message from *A* to *B* and substitute a transmission consisting of the token:

$$C, C\{ t^C, r^C, B, sgnData, Bp[encData]\}$$

plus the original encrypted message. *B*, upon receipt of this substitute transmission might, in its response to *C*, disclose confidential information from the original encrypted message (which would have otherwise be inaccessible to *C*). One method that can avert such a threat is to include additional information in the encrypted message that accompanies the token (e.g., A, t^A, and r^A).

2.6.3.2. Two-way Strong Authentication

Two-way authentication is used when user *A* wishes to authenticate to user *B* (a system or user), and wishes *B* to authenticate to *A* in return. The initial authentication sequence is the same as one-way strong authentication, but *B* then returns to *A*:

$$B\{ t^B, r^B, A, r^A\}$$

where t^B is a timestamp defined in the same way as t^A, r^B is a non-repeating number used for similar purposes to r^A, and $B\{ t^B, r^B, A, r^A\}$ is the authentication information digitally signed using *B*'s secret key.

It is possible to include extra information such as `sgnData` which will be used to provide data origin authentication by digital signatures, and `encData` which will be used as a secret key for the cryptosystem specified in `sgnData`. `encData` is encrypted using the public key of *A*, so that only *A* can read it.

$$B\{ t^B, r^B, A, r^A, sgnData, Ap[encData]\}$$

Note that this means that separate keys may be used to provide data confidentiality in each direction of data transfer.

2.6.3.3. Three-way Strong Authentication

Three-way authentication is used to establish the same properties as two-way authentication, but does so without the need for association timestamp checking. It involves a further transfer from *A* to *B*.

The first two exchanges between *A* and *B* are the same for two-way authentication except that the timestamps may be zero, and they need not be checked. *A* then sends to *B*:

$$A\{ r^B, B\}$$

B needs to check that the received r^B is the same as the one sent. $A\{ r^B, B\}$ is the authentication information digitally signed using the secret key of *A*.

Strengths

One of the strengths of X.509 is that it is not hampered by existing systems in its operation and interface. It has the ability to use any facility at its disposal to perform its function well. However, this is a double-edged sword in that by trying to specify a system that is usable by all, it may be that the system is not usable by anybody. This may be reflected by the lack of widespread deployment of this system.

X.509 protects the authentication sequence from replay through the use of timestamps and random numbers. The inclusion of the random number means that it is not required that the token be valid for any longer than necessary. The authentication instance is defined by a combination of timestamp and random number, which will be unique in a defined time period. All that is required is that the entity performing the authentication cache the random numbers for at least the period of validity of the timestamp.

X.509 protects the authentication sequence from disclosure through the use of public key encryption. This allows a third party to verify the correct credentials of a purported entity, without that entity ever having to exchange any secret information, or revealing that entity's secret key. Thus authentication can be undertaken between mutually distrusting parties, provided a trusted entity verifies that the public keys of those parties are correct.

X.509 does not mandate the use of any particular public key encryption algorithm. One algorithm that matches the requirements of X.509 is RSA [RSA78]. This algorithm has to date been difficult to break, provided that keys of a suitable size are chosen. One advantage of the RSA algorithm is that should a certain key size be determined to be insufficient, it is possible to increase the key size to attain the desired level of security. This would also have the effect of slowing the encryption down, and increasing complexity.

Weaknesses

The only public key encryption algorithm that has been currently used with X.509 is RSA [RSA78]. This algorithm uses keys that can be difficult to manage. Private key encryption can use keys that are only 56 bits long, and therefore, can be easy to remember ASCII text. RSA works on large prime numbers, which can be 100 digits long! Some people can be hard pressed remembering seven digit phone numbers!

The X.509 standard details some other potential threats that are specific to the strong authentication method [ISO92]. These are:

- *Compromise of the user's secret key*: The identity of an entity is determined by possession of the secret key. Compromise of that key would mean that a rogue user could authenticate themself fraudulently;
- *Compromise of the Certification Authority's (CA) secret key*: Certification Authorities are a trusted source that can be consulted to determine a user's public key. This key is transmitted inside a certificate (signed using the CA's secret key). If an invalid certificate is issued, than a rogue user could use an invalid secret/public key pair to authenticate as another user, and present what appears to be a valid certificate verifying the incorrect public key thus tricking an entity into believing that the invalid secret key in possession is the real one;
- *Misleading a CA into producing an invalid certificate*: In general, CAs are kept off-line to the main processing, and this offers some protection. It is up to the CA to verify the credentials of a user before issuing them a certificate that verifies their public key;
- *Collusion between a rogue CA and user*: This represents a basic breakdown of the trust relationship required in the X.509 method. There must be at least one entity that can be trusted to verify the credentials of all other (delegated) authorities. There are techniques to combat this type of threat. The reader is referred to [HY92] and [Smi93].
- *Forging of a certificate*: As the certificate is signed by the CA, the only way a certificate can be forged is if the CA's secret key is compromised, or the encryption technique is broken;
- *Forging of a token*: As the token is signed by the sender, the only way a token can be forged is if the sender's secret key is compromised, or the encryption technique is broken;
- *Replay of a token*: This is protected against by using random numbers and timestamps;
- *Attack on the cryptographic system*: "*the likelihood of effective cryptanalysis of the system, based on advances in computational number theory and leading to the need for a greater key length are reasonably predictable.*" [ISO92].

Lack of practical experience with X.509 means that it is difficult to prove that it does provide the services it promises effectively and efficiently. A system such as Kerberos has been in use in real situations for many years now, and its deficiencies are relatively well known. X.509 *may* contain hidden weaknesses that are yet to surface.

2.7. Authentication Techniques versus Threats

This section details which of the authentication techniques above provide protection against a subset of the threats outlined in Section 3.1. The threats chosen from the detailed list are Identity Interception, Replay, Masquerade, and Repudiation. Data Interception and Manipulation are viewed as extensions of the previous list, and are not considered. Denial of Service, Misrouting, and Traffic Analysis form a special type of threat which generally only applies to systems with extreme security (such as Government, or Military).

Table 1 shows which authentication scheme provides protection for a subset of defined security threats.

Table 1 - Threats handled by authentication schemes

	Identity Interception	Replay	Masquerade	Repudiation
plaintext passwords - world readable	x	x	x	x
encrypted passwords - world readable	x	x	x	x
encrypted passwords - read protected	x	x	x	x
one time passwords	x	√	√	x
Kerberos	x	√	√	x
X.509 simple unprotected	x	x	x	x
X.509 simple protected	x	x	x	x
X.509 strong - 1-way	√	√	√	√ (see Note 1)
X.509 strong - 2-way	√	√	√	√ (see Note 1)
X.509 strong - 3-way	√	√	√	√ (see Note 1)

Note 1: Non-repudiation is not provided by the authentication facilities of X.509 alone, but is, when coupled with other facilities such as Digital Signatures. Public key cryptosystems (under certain circumstances) can help to provide non-repudiation [Tan89] (as opposed to private key). By having a user sign data using their secret key, data integrity and data origin services are provided. This coupled with ensuring that only the specified user can view the data (by encrypting it using that user's public key - only they can decrypt it with their secret key) and having them reply using some part of that message signed by them, provides two way non-repudiation. The sender cannot claim not to have sent it (as only they could have signed it), and the receiver cannot claim that it was corrupted (it was signed) nor that they didn't receive it (as they replied using the contents of the message).

2.8. Comparison of Kerberos and X.509 Strong Authentication

X.509's one-way and two-way strong authentication techniques provide basically the same services that Kerberos provides, but uses different techniques. The basic differences between the two systems are shown in Table 2.

Table 2 - Comparison between Kerberos and X.509

Kerberos	X.509 Strong Authentication
Uses private key encryption	Uses public key encryption
Requires keys to be exchanged prior to authentication	Requires a trusted third party to verify the public key of the entity seeking authentication

Private key encryption relatively "easy" and "fast" to use	Public key encryption relatively "difficult" or "expensive" to use
Key management more difficult	Key management easier
Requires synchronised clocks	Three-way strong authentication does not require synchronised clocks (one-way and two-way still require synchronised clocks)
Uses clocks to prevent replay attacks - a window of opportunity exists where an attack may be successful - can be reduced if the server caches previously used information	Uses clocks and random numbers to prevent replay attacks - does not provide a window of opportunity provided the server caches previously used information
Is in current use in a variety of situations	Currently not many implementations
Required to fit into an existing framework of operation - the interface is determined by pre-existing systems	Provides services that clients must determine how best to use - no preconceptions on the interface

A number of development implementations and projects using concepts from X.509 have been undertaken over the last few years, but these systems are currently not in wide use. The reader is referred to [UCL92], [OSI92] and [TAP90] for further information.

3. Security Threats in Existing Operating Systems

This section describes the types of threats that a computer system may face. Note that it simply describes the class of attack, but does not detail how to actively exploit particular examples of this type of attack. This is deliberately done in the interests of those systems that may not be fully secure.

3.1. Classifying Threats

The X.509 standard details a number of threats that computer networks may face, particularly relating to authentication exchanges. These are detailed below [ISO92].

Identity Interception - this is the threat that the identity of one or more of the users involved in some form of communication is observed for later misuse. Generally, this is not a serious problem unless the nature of the parties communicating is in the political arena (for example, the leader of a rival company is observed to be communicating with one of the top employees in another company).

Masquerade - this is the threat where one user pretends to be another user. This usually takes place during an authentication sequence. Once the user has convinced the remote person or computer that they possess a different identity, the user then inherits all the access rights of the masqueraded user. When that masqueraded user is privileged, then this type of attack can be devastating.

Replay - this is a very simple form of attack where some sequence of events or commands is observed, and then replayed in the attempt to trick the user or machine into performing some act. This is mostly seen during authentication, by replaying another user's username and password, thus ultimately performing a masquerade attack. This is a special form of masquerade attack as it is the most common form in use; hence it is in a category of its own.

Data Interception - this type of threat can be observed at any time during network communications. The threat here is that some form of user data may be observed by another user that is unauthorised to see that data. This has many implications for confidentiality, or maintaining a secret during an basic authentication sequence.

Manipulation - if data can be manipulated either during transmission, or whilst it is stored on disk, then the potential for destructive action is enormous. In this case the integrity of all the data (not just that portion which was changed) is brought into question.

Repudiation - this is the denial of one or many of the users in having participated in a communication. This is more a problem in the business world where directives or agreements need the ability to be proven to have taken place.

Denial of Service - A denial of service attack is insidious as it often conflicts with other forms of security protection mechanisms. Denial of service can be categorised as the prevention or interruption of access to a service, or the delay of time-critical operations.

Misrouting - this is where a communication intended for one person is rerouted to another person.

Traffic Analysis - this is when the communication between two parties is observed, and information about that communication is obtained (such as absence/presence, frequency, direction, sequence, type, amount, and so on). This might be of importance for example to the military or government, particularly in an international arena.

The next section examines different types of attacks that have occurred in the past, and details how they they may be categorised using the above criteria.

3.2. Characteristics of Common Threats

3.2.1. Morris Worm

"*On the evening of 2nd November 1988, a program was executed on one or more hosts connected to the Internet. This program collected host, network, and user information, then broke into other machines using vulnerabilities present in existing software. After breaking in, the program would replicate itself, and the replica would also attempt to infect other machines. The program spread quickly, and the scope of the breakins came as a great surprise to almost everyone.*" [Spa88].

The "worm" exposed some security flaws in standard system services. These flaws are now patched, with the patches widely available. The first flaw was a bug in the `fingerd` daemon. This program made a call to the `gets()` routine. `gets()` makes no checks on the bounds of the buffer involved, and hence it is possible to overflow that buffer (`scanf()`, `sscanf()`, `fscanf()`, and `sprintf()` also have this problem). The particular problem in question allowed the input buffer to be overflowed thus by careful selection of the input data, a privileged login shell could be obtained and any commands executed.

A simple change to prevent unbounded input was all that was required. This particular vulnerability is a typical example of poor coding practices. When a program is to be executed by a privileged or an unauthenticated user, extreme care must be taken to minimise any damage as a result of error conditions or extraneous events[8]. In this case, the buffer overflow was not detected and a security risk was the result.

The second vulnerability exploited by the Morris Worm was in `sendmail`. `sendmail` is a mailer designed to route mail in a heterogeneous network. It is extremely large and complex. Configuring the sendmail system is a long and difficult task. Therefore, the sendmail developers allowed a DEBUG feature to be turned on to aid with configuring the system. Unfortunately, it was such a useful feature that it was often left turned on by default. When DEBUG was enabled, it allowed a user to issue a set of commands instead of the user's address as the recipient of a message. Since `sendmail` was running as a privileged program, this allowed those commands to operate in a privileged environment.

This is a classic example of poor configuration, particularly relating to extremely complex computer subsystems. Despite this being a well documented problem, automated attacks targeting this vulnerability are still being seen today, suggesting that perhaps some limited success is still being found using this mechanism [CER93].

The key attack of the worm involved attempts to discover user passwords. Since this is such a common problem, it is addressed as a separate issue in a later section.

---

[8]This does not mean that a program be executed by an unprivileged, authenticated user does not also require the same level of care!

Simple precautions would have prevented the rapid and widespread deployment of the Morris Worm. However in 1988, computer security did not have the high profile that it has today.

The Morris Worm was aimed at Unix based systems, specifically those based around the BSD variants. The next section shows that this type of attack is not a Unix-only problem.

3.2.2. WANK and OILZ Worms

The following analysis is drawn from [LS93].

During October and November of 1989, a different work attacked the SPAN and HEPnet networks, targeting VMS systems linked via DECnet. Whenever a system was infected with WANK or OILZ, the damage ranged from annoying users with messages and practical jokes to more major damage in terms of Trojan Horses and penetrated system accounts. After about two weeks, the attacks by WANK subsided. However, a new version of this worm, OILZ, attacked hundreds of additional systems. OILZ fixed a number of errors and problems that were present in WANK.

The WANK worm would use the network to examine remote systems with a view of learning what some usernames were on those systems. It would then attempt to guess a password for those accounts, initially only trying a small subset of passwords to help prevent detection. The worm would determine if it had penetrated a privileged account and further invade the system if possible.

The OILZ worm used the same techniques (correcting a few errors in the WANK worm), but also used the DECnet default account for penetration. This allowed the worm to spread to hundreds of additional systems.

Once the worm had penetrated the system using a privileged account, it would install back doors back into the system by editing any command file it could find to modify the password and privileges of a set account each time that command file was executed, it would disable mail to the system account (probably to prevent any notification of eradication scripts or other information), and it would modify user accounts to play a practical joke (which resulted in users deleting their own data).

These worms exploited standard attacks using already known vulnerabilities. The attacks that allowed the worms to propagate were the result of poor password choices, and default user accounts. Once again, it highlights that this is an extremely important security mechanism.

Similar worms have existed in the past which also exploited poor password choices and vendor supplied default passwords (such as the Father Christmas worm in 1988 [Cly93]).

A formal definition of worms can be found in [Coh92].

3.2.3. Trusted Systems

The discussion on this type of attack relates specifically around Unix, although it could easily apply to other Operating Systems (such as OpenVMS using the Network Proxy facility [Cly93] and DECnet [DEC88a]). Using a trusted system attack, many systems are compromised by using a trusted system on their local network that has itself been compromised [Cur90]. Turning this around, it means that once one system is compromised at a site, it is usually very easy to compromise many further systems at that site. There are a number of facilities that enable this to happen. The first is the file `/etc/hosts.equiv`. This file contains a list of all known hosts that this system trusts. In the past, many versions of the operating system were shipped by default as trusting all hosts on the global network. This was usually done as many of the workstations were being installed on closed networks, and not connected to the global network. It resulted in a system that was easy to install and get running in the local network. In the global network, this type of assumption is entirely inappropriate.

Another aspect of trusted systems attacks involves the use of the `.rhosts` file. This file works much the same as the `hosts.equiv` file, but only acts for specific users, rather than all users. The unfortunate part of this situation is that the file is entirely under the control of the user. This means that any individual user may lower the security of a system by allowing access to their account via any other system on the network.

The `.rhosts` file is used to grant access via the "r" commands (`rlogin`, `rcp`, `rsh`, `rexec`, `rdist`, and so on) [Sun90b]. However there are further vulnerabilities concerning these commands that have been discovered and rectified in the past. It was possible that the protocol exchange could be exploited to gain unauthorised access. This has the problem that the detection of this situation is usually out of the hands of the system administrator, and the system administrator now relies on third parties (such as an Incident Response Team) to detect and take responsibility for organising a software fix, and then disseminating the information to the sites.

The `.rhosts` file is used to identify the trusted system, by name. When all the host name to host address mappings were stored on the local system (for example, in the `/etc/hosts` file), this worked reasonably well. Now that these mappings are being performed by name servers that are usually out of the control of the local system administrator, it is possible to change the address to name binding to fool a system into thinking that the connection is coming from a trusted host [Ven92]. This represents a fundamental flaw in the design of these "r" commands.

The final style of vulnerability that falls under a trusted systems attack comes from inappropriate setting of the "secure" attribute on terminals [CER92]. This allows anyone using that terminal to login as `"root"` (the system privileged account on Unix). If the terminal is not set to `secure`, then the user must login on their own account first, and then change their identity to `root`, thus passing through more security checks and audit trails. Some operating systems by default had all of their terminals set to secure as it helped the system administrators when initially setting up their systems. Once again, this highlights the need for sensible default configurations when installing new software.

3.2.4. tftp

`tftp` stands for trivial file transfer protocol [Sun90b], [RFC783]. This works much the same as the normal file transfer protocol, but the word trivial is used to indicate that a number of key services are not implemented (such as user authentication!). `tftp` is mainly used by diskless workstations or smart terminals for boot information, configuration loading, or font loading. The major problem with `tftp` is that if care is not taken, it is possible to configure it in an inappropriate manner, thus allowing any world readable file on the system to be copied (generally, an intruder cannot overwrite a file, but even this is still possible under extreme circumstances). A typical `tftp` attack involves an intruder taking a copy of the `/etc/passwd` file to a remote site, and then attempting to guess the passwords. It is possible to automate these attacks, resulting in some widescale attempts at accessing password files (one single incident covered 65,000 individual hosts!) [CER92].

3.2.5. NIS

Sun Microsystem's Network Information Service (NIS)[9] is a distributed database system that lets many computers share password files, group files, host tables, and other files over the network [Sun90a]. The files are stored centrally on a NIS server, and are made available to the NIS clients. The files on the clients contain a special entry (that begins with a "+") to specify that the NIS server needs to be contacted for the rest of the information. This can have some unexpected effects. For example, the `/etc/passwd` file has an entry that looks like:

```
+::0:0:::
```

On some older versions of NIS, it was possible that this line would be treated as a normal password file line if the NIS server was unavailable [GS91], [Sho93]. Therefore, logging in with an account of "+" resulted in a privileged login without requiring a password [Spa93]. Another danger is that should the "+" ever be accidentally removed or changed, the line becomes a valid password file entry, allowing a privileged login without a password.

Other problems include the ability for any user outside of the organisation to obtain copies of the database files exported by the NIS server, by guessing the name of the NIS domain, binding to it using the `ypset` command, and requesting the databases [Mor90]. One major vulnerability here is that the password file may be disclosed, allowing password crackers to be exploited.

There are design flaws in the code of some NIS implementations that allow a user to reconfigure and fool the NIS system [GS91]. One of the major ways that this is performed is to establish a `ypserv` program that acts like a real `ypserv` program and responds to `ypbind` requests. The `ypbind` daemon can then be instructed to use that program rather than the real `ypserv` daemon. In this way, an attacker can supply their own version of the password file (containing any information they think is appropriate!).

Another form of attack is performed by analysing the NIS data packets as they pass on the network, and providing a valid response to NIS requests [Mor90]. This requires detailed knowledge of the NIS protocol. There are techniques around that ensure that the reply is received by the client before the valid reply. Since it is a datagram protocol, once the client receives a reply, it ceases listening, and the second reply is lost. This allows an attacker to enter a computer system by supplying any username (the client doesn't know it isn't a valid username), and that user can then become privileged (despite the fact that the terminals have been set secure, or that only a subset of people are allowed to execute the `su` command!).

Many implementations of NIS have been tidied up, and a lot of these problems have been addressed. The final problem relating to NIS is a simple configuration error with the password files. The "+" entry in the `/etc/passwd` file indicates that NIS is running and should be consulted for further passwords. A common mistake is to also include a "+" line at the end of the password file that is being exported by the NIS server. This has the effect of validating the username "+" as a real user account [GS91].

---

[9] Formerly "Yellow Pages".

3.2.6. NFS

The Network File System (NFS) allows different computers to share files over the network [Sun90a], [RFC1094]. NFS works using a client-sever model. Using NFS, clients can mount the disks on the server as if they were physically connected to themselves. Unlike other remote filesystems, NFS allows client users to read and change the contents of files stored on the server without ever having to log into the server or supply a server password. This characteristic is at the heart of NFS's security problems.

NFS can be made secure, however it is easy to incorrectly configure the system, often without even knowing. Most forms of NFS attacks take place on systems that are incorrectly configured. NFS unfortunately has a few "silent" features that may result in allowing any client on the network to mount the disk, if the configuration is not 100% correct. Inappropriate configuration allows any client to modify files on the server's disk, possibly installing malicious programs. The `showmount -e` command can be used to check a systems configuration to verify that what is set is appropriate.

Another security problem with NFS concerns file handles used to identify files [GS91]. These file handles consist of a filesystem id and an inode number, so it is relatively easy to guess valid file handles. The standard NFS server is unable to distinguish between falsified file handles from file handles that were created by the `mount` daemon. Therefore, the files on any NFS server are potentially accessible over the network by anyone who has the ability and determination to search for valid file handles.

3.2.7. Passwords

By far, the major strength (and weakness) in any system is the first line of defence - the authentication into the system. Any facility that allows *someone* access has the potential to allow *anyone* access. It is for this reason alone that special attention must be given to the authentication services. Unfortunately, this is not universally performed, and time after time, computer systems are being compromised as a result of a weak authentication service; in most cases a poor choice of password [CER92], [BKS90]. Sometimes the passwords are simply guessed, and with a small amount of knowledge it is usually easy to guess an inexperienced person's password. Mostly however, the password file is captured through some other means (perhaps `tftp`, or it may even be a local user with read access to that password file), taken to a remote site (often one that has been compromised!), and a variety of password guessing programs (password crackers) is run against it. It is a common belief that a young system intruder will never have sophisticated hardware to run password crackers on a password file. It is known that often the first thing the intruder does when compromising a system is to run a password cracker as a benchmark. If they decide the system is of sufficient speed, they will be back with a captured password file for cracking, possibly during the night hours only.[10]

Many systems support the concept of a "guest" account, generally to make the system administrator's life a little easier when they are required to generate a temporary account for someone who is temporarily visiting. Unfortunately, the passwords chosen for these guest accounts tend to be extremely poor, more so than normal user accounts. It is not uncommon for the password of a "guest" account to be "guest".

---

[10]This information is referenced in files that contain other information that is sensitive, and hence cannot be published.

This represents a major flaw in the system security as it not only allows access to the computer system, but also prevents any valid audit trails, as it can be easily argued that anyone could be using that account when it contains such an easy password to guess. [Gro93]

The final vulnerability relating to passwords is usually perpetrated by vendors and software manufacturers. Sometimes the vendor software requires an account and password to operate. Usually (always?), the choice of password for this account is arbitrary, and generally not required by any human user. Unfortunately, many vendors and manufacturers in the past tended to install a "default" password, which made their life easier. Many customers when installing products were either not aware of the problem, or never got around to rectifying it. This allowed many intruders to simply access the system with a known account and password, often with full system privileges. It is fortunate that now many vendors are becoming more security conscious, and this practice is falling into disfavour. It does still occur however. Special care must be taken after applying upgrades as well. It is possible that this vulnerability was closed by changing the account's password, but the upgrade silently set it back to a known value.

One reason that passwords remain a major source of vulnerability is that they are out of control of the system administrator. Passwords are chosen by users, and if those users are not educated on what forms a strong password, then they will tend to choose a password that is easy to remember (which is also easy to guess or crack). It is therefore not surprising that there are a large number and variety of tools which are solely aimed at improving password security (for example, [Bis92b] and [Hoo90]). Ultimately, the solution to the password problem is to make use of these tools coupled with educating users that security is a community response and they have just as much responsibility if the computer system is compromised.

3.2.8. Software Vulnerabilities

There are many techniques that a programmer must employ when writing privileged code (or at least, code that will run as the Id of another user) [Far91b]. In essence, it is important to ensure that the user cannot control the environment in which the program executes, and that all error status returns are checked and handled appropriately. Failure to do this may result in the software integrity being compromised. These techniques are not reserved for privileged code, but are generally good coding practices. They include simple tasks such as initialising the operating environment to a known state, checking the status returns of all system calls, and parsing all arguments internally and not trusting a third party to do it correctly. Much of the rest of this section is drawn from [Bis87] and [GS91].

One particular type of attack involves the IFS shell variable (Input Field Separator). This variable is used to indicate what characters separate input words to the shell. Whilst its functionality has been largely superseded, it lives on to cause unexpected results. For example, if a program calls the `system()` or `popen()` functions to execute a command, then that command is parsed by the shell first. If the user has control over the IFS environment variable, then they may cause unexpected results. A typical scenario might be if the program executes the following code:

```
system( "/bin/ls -l ");
```

If the IFS variable has been set to contain the "/" character and a malicious program called "bin" is placed in the path of the user executing the program, then that program will be executed as the shell will have parsed the line as:

```
bin ls -l
```

which executes the program `bin` (in the current path) passing two arguments `ls` and `-l`. It is for this reason that a program should not get the shell to parse command lines by using the `system()`, `popen()`, `execlp()`, or `execvp()` commands to run some other program.

Another form of environment attack is the use of the `HOME` environment variable. Normally the `csh` and `ksh` substitute the value of this variable for the ~ symbol when it is used in pathnames. Thus if an attacker is able to change the value of this variable, it might be possible to take advantage of a shell file that used the ~ symbol as a shorthand for the home directory.

For example, if a shell file was referencing the `~/.rhosts` file for the user running it, it is possible to subvert it by resetting the `HOME` environment variable before running it.

The final environment attack examined here (although these are not the only forms of environment attack, they do represent the classes of attack, and the methods to combat them are much the same), is the use of the `PATH` environment variable. This type of attack is characterised by the value and order of the files paths in the `PATH` variable. An inappropriate choice of path orderings may lead to unexpected results if a command is executed without reference to its absolute path. For example, consider the following `PATH` specification:

```
PATH = .:/usr/bin:/bin:/usr/local/bin
```

If someone had created a file called `ls` in a place that the current path is set to, then this would be executed in favour of the normal system `/bin/ls` command. If the file contained the following:

```
#!/bin/sh
(/bin/cp /bin/sh /tmp/.secret
/bin/chmod 4555 /tmp/.secret) 2>/dev/null
rm -f $0
exec /bin/ls "$@"
```

this would silently create a copy of `/bin/sh` which when executed would attain the identity of the person who executed the file. In addition, it would clean up any evidence and execute the real `/bin/ls` command so that the command would ultimately succeed without the user being aware that something else has happened. This is a typical example of a Trojan Horse (see Section 3.2.12).

In general, the use of relative path names either in referencing files or when executing programs should be considered as a poor programming practice.

Software vulnerabilities also are made apparent through the use of poor programming practices. These practices may be performed through ignorance or inexperience, or the programmer may simply be too lazy to do it right the first time.

A prime example of this was exploited by the Morris worm. The vulnerability was that the `gets()` system call was used which does not perform any length checking. Instead the `fgets()` system call should have been used. This allowed a buffer to be overflowed under the user's control, and hence the program was forced to take inappropriate action. There are several other system calls that suffer this same fate, including `scanf()`, `sscanf()`, `fscanf()`, and `sprintf()`. Software system design may affect the use of `strcpy()`, `bzero()`, and `bcopy()` as well.

Often, the value of `umask` (the default file protection mask) is set to something that is inappropriate. Many programs fail to check the value of `umask`, and often fail to specify a protection for any newly created files. Even if the program created a file, and then changed its protection to make it secure, a window of opportunity exists where an attacker could interrupt the program and possibly gain access to a writeable file. Therefore, it is important to establish a value for `umask`, prior to opening any files.

Another typical programming practice that can cause problems is the lack of making the effort to check the status returns on every system call. This means that if an intruder can gain control of the environment in which the program is running, they may cause a particular system call to fail, which the program blindly assumes will always work. This may cause the program to take further inappropriate action.

The final area covered here (which certainly does not form a comprehensive list, but details the typical mistakes made) is that often a program fails to catch all the signals it possibly can, and react appropriately. This may allow an attacker to set their `umask` value to something inappropriate, and the send a signal to a privileged program, causing it to dump its core. When this happens, the core file is owned by the effective UID of the running program, but protected with the value specified by `umask`. The dangers of this should be apparent!

3.2.9. Network Monitoring and Data Capture

One of the threats now being faced by computer systems is the ease at which data can be captured while it is being transmitted between computers [Bro93]. In the past when large central systems were the norm, this did not pose a large threat, but the advent of heterogeneous computer networks that span the globe mean that sensitive data may travel beyond the control of an organisation. There are many packages available that can be used to monitor data as it passes by on a network link. Particularly vulnerable are the bus style networks (such as Ethernet) where data destined for a particular host can be viewed by any host connected to the network.

This now means that any data can be captured and used for different purposes. This does not only include sensitive data, but may also include protocol exchanges (such as a login sequence, including the password). Authentication sequences are one of the most vulnerable exchanges as they are the critical decision point when granting access privileges.

Data does not always have to be captured from the network itself. By installing a Trojan Horse in the network software or the application, data may be captured and saved on the disk for later examination. This may defeat the standard defence of encrypting data that is to be transmitted across the network.

3.2.10. Examining Source Code

Since there are now known coding practices to avoid when writing programs (particularly ones that will run in a privileged mode), experience suggests that attacks are being launched at particular programs, usually after careful examination of the source code. In general, the source code is freely available and this allows anyone to examine it looking for potential flaws. This was not the usual form of attack up to five years ago, but it is becoming more common place these days [CER92]. In some aspects, this will result in some good as the programming errors will be finally rectified, and programming methodologies will become better understood.

This threat relates significantly to Section 3.2.8. It differs from that section in that most software vulnerabilities in the past have been discovered by accident. Now, the intruders are actively examining source code looking for any unknown vulnerabilities, which represents a new style of exploiting poor coding practices.

3.2.11. Protocol Weaknesses

Over the past few years, computer intruders have increased in number, age, and intelligence [Aus93]. Instead of exploiting old weaknesses and programming errors, they are now starting to target the actual design of distributed systems, in an attempt to fool one system into believing that the attacker is the systems partner. This may mean actively studying the source code of a system.

One of the more vulnerable areas of a systems design is when the system must interact with the external environment, particularly when requesting some service from another system. This exchange between the client and server systems may happen inside the same computer, or may be distributed across local and wide area networks. When the exchange is between two computer systems, some form of protocol is required so that the cooperating processes can indicate what function they wish performed, and to retrieve the results.

Many of the protocols in use for client-server models are now being actively examined by intruders in an attempt to find some weakness [CER92]. The intruders now have the tools to actively monitor network links looking for particular protocol exchanges, and then by careful programming, may insert a valid response to a request in an attempt to fool the requesting system into believing that it has received a correct response [GS91].

This is the type of attack detailed for NIS, but is certainly not restricted to just NIS.

3.2.12. Trojan Horses

Trojan Horses are named after the legend of the same name. Computer style trojan horses resemble normal programs that a user wishes to run, such as editors, login programs, or games. While the program may be appearing to do what the user wishes it to do, it is actually doing something completely different (such as deleting files, storing passwords for later use, reformatting disks). By the time the user is aware of a problem, it is far too late [KC90].

Trojan Horses can appear in many different places. They can be found inside programs that have been compiled, or in system command files executed by system administrators. Other forms of trojan horse include sending commands to people as part of a message (such as electronic mail or a message to a terminal). Some mail handlers allow the user to escape to the shell and execute commands. This can be done by activating the feature when the message is read. Sending a particular message to some terminals can have a command sequence stored in the terminal, and then have that command played back as though it was typed on the keyboard. Editor initialisation files are also a favourite place for storing trojan horses.

Trojan horses are unfortunately, very common [GS91]. As soon as a system is compromised, the attacker usually modifies the system in many ways to ensure that if the original intrusion is discovered, they can always get back into the system. This has the added effect of raising the cost of recovering from a compromise, as the *entire* system must be painstakingly checked for the presence of any Trojan Horses.

3.2.13. Spoofing Mail

A user receives electronic mail from their manager, telling them to change the privileged account password to "abcdefgh" as the manager suspects that one of the staff may be illegally using the account. The manager stresses to the user not discuss this matter with anyone. What does the user do?

"Spoofing" or falsifying electronic mail is extremely easy to do. Electronic mail is becoming an extremely important tool for efficient communication, and having a shadow of possible mail forgery hanging over its head has not necessarily slowed down its deployment, but it may have limited its effectiveness in many situations.

If business transactions are conducted using electronic mail in its basic form, then the potential for chaos and confusion is enormous. The possibility of forged orders, denied receipts, and challenging the issuing of directives means that electronic mail in this form is not a viable option. The ability to verify the source of electronic mail is becoming a very important issue. This will help establish services such as non-repudiation (the ability to prove that a user sent the message, and the ability to prove that a user received a message).

3.2.14. suid Files

Unix has the ability to set a special permission on files called SUID and SGID (set user identity, and set group identity). The meaning of these permissions is to change the effective identity of the user executing the program, away from their real user identity, to the identity or group of the owner of the program. This allows for example, a user to be temporarily privileged to perform some certain function, whilst being confined in an enclosed and controlled environment.

This is inherently dangerous unless the programmer is extremely careful. Whilst the program is executing as a different identity, subtle coding errors may allow the user executing the program to take advantage of the different user identity. If that user identity is privileged (`root` for example), then the potential for disaster is huge.

Many Trojan Horses make use of the SUID facility, by making a copy of the `/bin/sh` program (or similar), and setting the permission to SUID. This means that when that program is executed, a new shell is started, but it is now executing as the user that owns that shell. If that shell file is owned by `root`, then when the shell is executed, `root` privileges are attained.

3.2.15. World Writeable Files

World writeable files are files that can be written to by anyone who is validly authenticated to the computer system. This type of file presents a potential security risk as any information in those files can be replaced perhaps by installing a Trojan Horse to perform some other function, or used to gain access to that user's account and their privileges. A particularly vulnerable file is the `.login` file (or any such similar file) as it is automatically executed every time the user logs in to the system. Other files require the user to actively execute them before a Trojan Horse can be initiated.

On Unix, the devices are viewed merely as another type of file. If these devices are world writeable, then there is also potential for malicious use. Consider if the device associated with the hard disk is world writeable, an attacker could change any file on that disk by changing the physical block where that file is stored. Particularly vulnerable would be the `/etc/passwd` file, where an attacker could change the contents to anything they desired.

3.2.16. X Windows Security

X windows represents a different form of security vulnerability, as it more applies to an application rather than to a computer system. However, as a result of this, the system may then be compromised. Earlier versions of X windows had a crude form of security. With this, a system administrator could allow or deny particular hosts to display X clients on the X server. This was not user based, so once access was granted to a host, anyone on that host had access to the X server [MP92]. This resulted in many practical jokes in which people would randomly display pictures on the screen[11], or use a program to make the screen "melt" or "crumble" away. These practical jokes had more of a nuisance value than a security risk.

There are now programs that will attach to the X server screen, and so the intruder can see exactly what is being displayed on the screen at any point in time. Other programs capture every keystroke (including shift and control keys). The potential for damage here is that it is usually easy to see a password being typed in. There are now facilities in place to provide per-user authentication, but they tend to be so cumbersome that the less-security conscious users tend to avoid them. Work in this area is still progressing[12].

3.2.17. Software Interaction and Configuration

Ultimately, the underlying reason for security problems can be reasoned to stem from the fact that computer systems and the software that run them are becoming more complex each day. Given that generally no single person writes the entire system, it is impossible to predict the interaction of several components of a system, especially at the border conditions and in obscure error cases. Programming methodologies are being tightened up, but it is definitely a case of "too little, too late".

As well as the complexity of software interaction, programmers are giving the system administrator a huge range of choices. The job of configuring a system is becoming so complex that simple errors may lead to subtle security problems. Finding staff who are experienced in a wide range of applications and system configurations is difficult, and they have mostly gained that experience the difficult way. There is a trend towards electronic testing of computer systems security; evaluating the ability to penetrate the system through the use of programs, both external and internal to the system itself. This however only reports on *known* vulnerabilities, and does little to detect new vulnerabilities.

3.2.18. Concluding Remarks

System intruders are definitely becoming more sophisticated. It was stated in a conference recently that the average age of the intruder is increasing, and law enforcement are no longer dealing with wayward teenagers. They are now arresting alleged intruders who have graduated with First Class Honours degrees [Aus93]. This is also reflected in the types of attacks that are being witnessed over time.

In 1988, the majority of computer system intrusions resulted from exploiting poor passwords, and system vulnerabilities.

In 1993, the techniques of five years ago are *still* in common use (and are *still* successful!). However, intruders are now exploiting protocol weaknesses in attempt to fool servers into performing some service, there is more network sniffing looking for valuable information, and many of the intruders are found with original source code most likely with a view of examining it for more flaws.

---

[11] The author's personal favourite was "Ronbo" - a picture of Ronald Reagan's head on Rambo's body!

[12] The Australian Security Emergency Response Team will be publishing an Advisory shortly which relates to X Windows security. This advisory is designed to educate the X windows user on what security facilities are presently available to them. This information is still being developed, and not yet available for public release.

As the sophistication of intruders grows, so does the sophistication of the system administrators *and their tools* need to grow. It has been suggested in brochures distributed by the Australian Computer Society that knowledge in the Information Technology industry is doubling every five years. This appears to correlate with the experiences shown above.

3.3. Summary of Threats and Classification

Having detailed a number of typical threats that computer systems have faced and are still facing, it is interesting to note how they can be categorised under standard criteria. Using the threats defined in X.509 [ISO92], Table 3 attempts to perform this analysis. However, it turns out to be quite difficult to do this while adhering to the strict definition of the standard threat criteria.

Due to these difficulties, structuring the information in this manner does not provide as much information as perhaps some other format. Further work in this area is required.

Table 3 - Categorisation of threats

	Data Interception	Masquerade	Replay	Manipulation	Repudiation	Misrouting
Worms						
Morris	x	√	x	√	√	x
WANK	x	√	√	√	√	x
OILZ	x	√	√	√	√	x
Trusted Systems						
hosts.equiv	x	√	x	√	√	√
.rhosts	x	√	x	√	√	√
secure terminal	x	√	√	x	√	x
tftp	√	√	√	x	√	x
NIS						
+ in passwd	x	√	x	x	√	x
ypset	√	√	√	√	√	√
ypserv	√	√	√	√	√	√
NIS Packets	√	√	√	√	√	√
NFS						
exports	√	√	x	√	√	x
file handles	√	√	√	√	√	√
Passwords						
guessing	x	√	x	x	√	x
cracking	√	√	x	x	√	x
guest	x	√	√	x	√	x
default	x	√	√	x	√	x

Software						
IFS	x	√	x	√	x	√
HOME	√	x	x	√	x	√
PATH	√	√	x	√	x	√
umask	√	x	x	√	x	x
error returns	x	x	x	√	x	√
signals	√	x	x	√	x	√
Monitoring						
network	√	x	x	x	√	x
trojan horse	√	√	x	√	x	x
Source Code	√	x	x	√	√	x
Protocol Weakness						
spoofing	√	√	√	√	√	√
Trojans						
part of message	√	√	x	√	√	√
in programs	√	√	x	√	√	x
Spoofing Mail	√	√	√	√	√	√
suid Files	x	√	x	√	√	x
World writeable files						
files	√	√	x	√	√	√
devices	√	√	x	√	√	√
X Windows	√	√	√	√	√	√
Interaction Configuration	√	√	√	√	√	√

4. Analysis of Security Tools

The threats that are detailed in the previous chapter are generally well known, and solutions are in place to combat them. However, it is impossible to say that a system is entirely secure, and so careful vigilance is required to ensure the integrity of a computer system, its software, and the data stored there.

A number of tools and techniques are available to help the system administrator and system programmer with their task. This chapter includes a wide selection of them with a discussion on what they do and what threats they are attempting to combat. They are broken into five broad sections: Cryptographic Tools, Monitoring Tools, Security Assessment Tools, Security Enhancement Tools, and Programming Methodologies.

4.1. Cryptographic Tools

4.1.1. DES

One of the most widely used encryption systems today is the Data Encryption Standard developed in the 1970s by IBM [FIP77], [CLS91]. DES is basically a bit permutation, substitution, and recombination function performed on blocks of 64 bits of data and 56 bits of key (8 characters of 7 bits). The algorithm is structured in such a way that changing any bit in the input has a major effect on almost all of the output bits.

The DES algorithm can be used in four modes:
- Electronic Code Book (ECB);
- Cipher Block Chaining (CBC);
- Cipher Feedback (CFB);
- Output Feedback (OFB).

Each mode has particular advantages in some circumstances, such as transmitting data over a noisy channel, or when it is necessary to decrypt only a portion of the file. The four modes are described below.

ECB Mode - In electronic codebook mode, each block of input is encrypted using the key, and the output is written as a block. This method is simple encryption of a message, a block at a time. This method may not indicate when portions of a message have been inserted or removed. It works well with noisy transmission channels, as alteration of a few bits will only corrupt a single 64-bit block.

CBC Mode - In cipher block chaining mode, the plaintext is XOR'ed with the encrypted value of the previous block. The result is then encrypted using the key. Because bits everywhere in the message propagate through the entire encrypted output, the last block can be used as a signature that the encrypted text has not been altered. As well, long runs of repeated characters will be masked in the output.

CFB Mode - In cipher feedback mode, the output is fed back into the mechanism. After each block is encrypted, part of it is shifted into a shift register. The contents of this shift register are encrypted with the user's key using ECB mode, and this output is XOR'ed with the data stream to produce the encrypted result. This method is self-synchronising, and enables the user to decipher just a portion of a large database by starting a fixed distance before the start of the desired data.

OFB Mode - In output feedback mode, the output is also fed back into the mechanism. A register is initialised with some known value. This register is the encrypted with ECB mode using the user's key. The result of this is then used as the key to encrypt the data block (using an XOR operation), and it is also stored back into the register for use on the next block.

DES uses the same key to encrypt the data and decrypt the data. Therefore, it is essential to use techniques that keep the secrecy of this key intact. Poor key management will lead to the reduced effectiveness of DES.

4.1.2. MD2

The MD2 Message Digest Algorithm [RFC1319] was created as part of the Privacy Enhanced Mail package (see Section 4.1.5). MD2 generates a 128-bit signature (fingerprint or message digest) from a given block of text. The signature is designed to prevent someone from being able to determine a block of valid text from a given signature or allowing them to modify the block of text while maintaining a valid signature. It uses 18 rounds of table substitution. The MD2 signature is a relatively slow algorithm compared with MD4 or MD5.

4.1.3. MD4

The MD4 Message Digest Algorithm [RFC1320] generates a 128-bit signature from a given block of text. It was designed to exploit to 32-bit RISC architectures to maximise its throughput, and does not require large substitution tables. "*Because MD4 was designed to be exceptionally fast, it is "at the edge" in terms of risking successful cryptanalytic attack.*" [RFC1320]. MD4 uses three rounds of transformations.

It has been shown that if MD4 is stripped of its last round, then for 99.99% of all initial values, it is possible to find two messages differing in only 3 bits which hash to the same output value. If MD4 is stripped of its first round, it is possible to find for a given initial input, two different messages which hash to the same value [BB91]. It is claimed that as a result of this attack, changes were made to MD4 to develop MD5 [BB93].

4.1.4. MD5

The MD5 Message Digest Algorithm [RFC1321] is a proposed data authentication standard. MD5 attempts to address potential security risks found in the speedier but less secure MD4. MD5 is a more conservative algorithm that attempts to reduce the risks of a successful cryptoanalysis attack by not adding further complexity. MD5 also produces a 128-bit signature, and uses four rounds of transformations to ensure pseudo-random output.

4.1.5. PEM

PEM is Privacy Enhanced Mail [RFC1421], [RFC1422], [RFC1423], [RFC1424], a standard for allowing transfer of encrypted electronic mail. It is actually a protocol that is designed to allow the use of private key (symmetric) and public key (asymmetric) cryptographic techniques. PEM is designed to provide message integrity checking, originator authentication, non-repudiation of origin, and data confidentiality. These are offered through the use of end to end cryptography between the originator and the recipient. There are no special requirements imposed on the message transfer system. This allows incorporation of the PEM system without impact on existing systems.

In general, the message is encrypted using a randomly chosen private key (the Data Encryption Key), and that key is then encrypted using the recipients public key. In that way, only the recipient of the message can decrypt the chosen private key (using their asymmetric secret key), and therefore decrypt the message. To provide message integrity and data origin authentication, the originator generates a Message Integrity Code and signs this with the private component of a PKCS key pair.

4.1.6. RIPEM

RIPEM is one implementation of the Internet's Privacy Enhanced Mail [Hey93a]. It allows electronic mail to have the four security facilities provided by PEM:

disclosure protection;
originator authenticity;
message integrity;
non-repudiation of origin.

RIPEM is not a complete implementation of PEM as described in the RFCs. PEM specifies certificates for authenticating keys, which RIPEM currently does not handle.

In RIPEM, key management is performed in three ways:

a central server;
distributed finger servers;
a flat file.

None of these facilities provides perfect security and is a recognised weakness of RIPEM [Hey93b]. For a discussion on certificates and key management, see [Smi93].

This differs to TIS/PEM which makes use of certificates to bind the name to the public key. RIPEM is freely available within the United States and Canada, but due to export restrictions, the cryptographic sections are not available outside those countries.

4.1.7. TIS/PEM

TIS/PEM is a different implementation of PEM which was developed by Trusted Information Systems Inc [TIS93]. It is a reference implementation developed in cooperation with RSA Data Security Incorporated, and is sponsored by ARPA. TIS/PEM is freely available within the United States and Canada, but due to export restrictions, the cryptographic sections are not available outside those countries.

TIS/PEM uses certificates to bind names to RSA public keys and is capable of generating the required certificates. Joining the Internet certification hierarchy makes it easier to verify others' mail and for them to verify yours.

4.1.8. PGP

PGP is a cryptographic mail program called Pretty Good Privacy [Zim92]. PGP makes use of RSA, but does not conform to the PEM RFCs. It is not a "standard" as such and is therefore not compatible with anything else but itself. PGP does not make use of RSAREF, but its own implementation of RSA which has not been licenced in the United States of America. This has recently raised a number of legal issues which are still unfolding.

Whilst PGP has more key management facilities than RIPEM currently does, it still does not provide the hierarchical structure of certificates, which is required to determine ultimate trust.

PGP is currently involved in legal issues, and it is not compatible with other systems. This will possibly reduce its lifetime as a viable solution.

4.2. Monitoring Tools

The Monitoring Tools can be used to examine network traffic, looking for set patterns of data or connection attempts.

4.2.1. TAP

TAP is a SunOS specific tool at present [Ney92]. It is a loadable kernel module that examines data as it passes through a stream and captures it for examination. This is performed by inserting the TAP module between two kernel modules (e.g., ttcompat and ldterm), and having these modules pass their M_DATA messages to TAP. TAP makes a copy of the data and sends it off to a specific driver, and then forwards the message onto the next kernel module for processing.

Given that TAP requires a module to be loaded into the kernel, it is inherently a dangerous tool. If any coding errors or kernel module incompatibilities exist either in this version or future versions of any software, then the likely result is a system panic. Another problem is that the knowledgeable user may be able to detect that they are being monitored. If TAP is used to monitor a malicious intruder, then this may be all the encouragement the intruder needs to proceed with destroying the compromised system.

Most techniques of monitoring people by making use of the computer system that the person is logged into can usually be detected if the person being monitored is vigilant. Therefore, it is much better to take the monitoring facility away from the host, and move it somewhere that the intruder cannot access (such as another machine connected to the network). This can be particularly effective as these days, most data travels across a network of some form, and it also allows several systems to be monitored at once.

4.2.2. tcpdump

`tcpdump` is a tool that listens to the network, and saves any packets that meet some specified criteria [MLJ92]. If the network controller is an Ethernet controller, it is placed into promiscuous mode first, so that all data on this bus network is examined by the networking software. Note that on some systems, it is not required to be privileged to place the controller into promiscuous mode.

`tcpdump` has two main modes of operation: the first is to display the packets on the screen as they are captured, and the second is to log the packets to disk in a binary format for later processing (using `tcpdump` again). In general this tool is designed for examining network problems, and does a good job of displaying the protocol headers. However, it is possible to specify how many bytes of each packet are to be saved, and this can be increased to include the data in a packet as well, therefore allowing anyone to view any network data.

This package can be tailored to only display or save relevant data. When the program is executed, a number of filters can be specified. These filters can select packets based upon the following criteria:

- a single host, or a range of hosts;
- source address, destination address, or either;
- a particular port number;
- a particular protocol type;
- less than or greater than a specified length;
- broadcast or multicast packet.

These primitives may be combined with boolean operators to form a complex expression. For example:

```
host 1.2.3.4 and net 1.4.7 and (port telnet or port mail)
```

Saving larger packets (or more data from each packet) increases the chances that a packet will be dropped due to insufficient resources. Despite this, `tcpdump` is a useful tool, not only for analysing a TCP/IP network.

4.2.3. TAMU etherscan

`etherscan` is a network monitoring tool that monitors certain TCP/IP services for activity that indicates possible intruder presence [SSH93]. It scans all packets and their contents looking for a specific set of intrusion signatures, such as `root` login attempts from remote sites. `etherscan` also acts as a complement to the `drawbridge` filter package (see Section 4.4.12) by covering areas of weakness inherent in bridging filter arrangements.

4.2.4. TAMU netlog

`netlog` contains three programs to log traffic from TCP and UDP connections [SSH93]. It can be used to locate suspicious network traffic. The `tcplogger` logs all TCP connections on a subnet, the `udplogger` logs all the UDP connections on a subnet, and `extract` processes the log files created by `tcplogger` and `udplogger`. These logger programs record a one line summary for all connection attempts. The associated analysis programs report on suspicious connections or patterns of connections.

4.2.5. Supervisor

Supervisor is a VMS only tool at present [Goa92]. It displays all I/O to a terminal device to another terminal device. Supervisor offers two monitoring modes: Observer and Advisor.

Observer mode allows a privileged user to view every character typed and printed at another user's terminal. The effect is that both users, initiator and target, see the same screen at the same time.

Advisor mode allows a terminal user to type command lines, comments, and program input into the target user's terminal process. The effect is the same as if the target user had typed the characters.

Photo is a similar utility that allows users to log their own terminal sessions into a file. While Photo is running, each character sent to and received from the terminal is recorded in a file and may be redisplayed later using the `playback` command.

4.3. Security Assessment Tools

This section describes tools that can be used to assess the security of the computer system. Each tool addresses different aspects f computer security.

4.3.1. COPS

COPS (Computer Oracle and Password System) is a collection of programs than individually examine a particular aspect of Unix computer security [Far91a]. These programs make no attempt to rectify any vulnerabilities they encounter, but merely report them. COPS examines at least the following list:

- file, directory, and device permissions/modes;
- poor passwords (using a simple password cracker);
- content, format, and security of password and group files;
- the programs and files run during system startup, and those run at regular intervals;
- the existance of any `root` SUID files, their writeability, and whether they are shell scripts or not;
- a CRC check of important binaries and key files to detect any changes;
- writeability of users home directories and startup files (`.profile`, `.cshrc`, and so on);
- anonymous `ftp` configuration;
- unrestricted `tftp`, decode alias in `sendmail`, SUID `uudecode` problems, hidden shells inside `inetd.conf`, `rexd` running in `inetd.conf`;
- miscellaneous `root` checks (current directory in path, a "+" in `hosts.equiv`, unrestricted `NFS` mounts, ensuring `root` is in `/etc/ftpusers`, and so on);
- dates of released vulnerabilities versus key files in an attempt to determine if there are binaries that are known to contain problems;
- the Kuang expert system which takes a set of rules and tries to determine if the system can be compromised.

The Kuang Rule-based Security Checker is supplied as part of the COPS package. The heart of the Kuang program is a set of rules that describe the Unix protection system from the point of view of an attacker. For example, one rule says that if an attacker can write the file `/etc/passwd`, then that attacker can gain super-user access. Given a goal like "become super-user", the Kuang system examines all the rules to produce a list of subgoals that would be sufficient to meet its goal. Recursively, each subgoal can be processed to produce a list of further subgoals. The process continues until there are no new goals. At each step, if a goal can be achieved using the initial privileges of the attacker, Kuang prints out a message that describes the sequence of steps that leads from the initial goals to the target goal. An example of the output from Kuang is:

```
member project1, write ~tom/.cshrc, member staff, write
/etc, replace /etc/passwd, become root
```

This means that the attacker had access to the group `project1`, which all members of `project1` belonged to. The `project1` group had write access to one of the startup command files executed automatically when the user `tom` logged in. Anyone on the `project1` group could add commands to Tom's startup file, and have these executed with Tom's privileges next time he logged in. Tom is a member of the `staff` group, which can write to the directory that contains the `passwd` file. Write access to this directory allows members of the `staff` group to delete and recreate the `passwd` file, which may contain a rogue entry for the super-user account. Therefore, the attacker could gain access to super-user privileges.

The Kuang program attempts to make the task of verifying the security decisions easier for the security manager. It examines the relationship between the various aspects of the protection mechanisms supplied by Unix, to determine if it can achieve a goal based upon achieving a number of subgoals.

4.3.2. CRACK

Crack is a fast UNIX password cracking program designed to assist site administrators in ensuring that user's use effective passwords [Muf92]. It is approximately 16 times faster than standard DES routines, enabling one to check more passwords in a given time. Newer versions of the `crypt()` function may increase this speed by a factor of one to three, depending upon the configuration of the hardware.

Crack takes as input a series of password files and source dictionaries. It merges the dictionaries, turns the password files into a sorted list, and generates lists of possible passwords from the merged dictionary or from information obtained from the password file.

Crack makes several passes over the password entries. Each pass generates password guesses based upon a sequence of rules. These rules are specified in a simplistic language. The instructions in the rule are followed in order and are applied to the dictionary words as they are loaded. There is a simple pattern matching primitive so that select words can be ignored (thus saving time and memory).

There are two groups of rules. The first is a set of rules that is applied to the `gecos` field of the password file (which stores useful information about the owner of that account). This field might for example store the name, department, and telephone extension of the account owner. The `gecos` rules create passwords based upon permutations of that information.

Once a pass has been made over the passwords based on the `gecos` information, the package makes further passes by first applying the dictionary rules to the word lists supplied. The rules are applied to each word first to generate a memory resident dictionary which is sorted and uniqued to prevent any repetition. As well as standard dictionary words, a file can be supplied which details common passwords that are non-dictionary words. Examples are `qwerty` or `drowssap` (password backwards).

If Crack successfully guesses a password, it marks the user as done and does not waste further time on trying to break that user's password. Crack also has the facility where passwords that were guessed on previous runs will not be checked again if they have not been changed yet. These passwords are trivially and instantly cracked.

Crack has the ability to automatically spread the load of password cracking to several machines on a network. The hosts are specified with a "power" rating which aids in determining how much load to pass to each of the networked machines. The power is determined by assigning the power of one (1) to the slowest machine, and rating every other machine relative to it. More accurate figures can be obtained by using a program to determine how many password encryptions a machine can perform in one second, and using that figure as the relative power of the machine.

Once Crack enters the second pass over the passwords, it periodically saves its state. This state can be used to recover from that point should a host crash, or Crack aborted. This helps to avoid the problem of programs with long run times from never completing if the relative uptime of the machine they are running on is less than the expected runtime for the program.

4.3.3. Tripwire

Tripwire is a file integrity checker using a number of cryptographic checksumming algorithms in parallel for added security [KS92]. Algorithms such as CRC-16 and CRC-32, commonly used to checksum data packets for transmission across a network [Tan89], do not provide sufficient strength to protect the integrity of files against a determined intruder. There are public domain tools that will help to "recreate" a valid checksum on files, while still maintaining file size. This is especially true of system binaries.

Tripwire makes use of several message digesting algorithms. These are:

- MD5
- MD4
- MD2
- Snefru
- CRC-32
- CRC-16

The use of more than one of these algorithms in parallel greatly decreases the chances of an intruder being able to modify a monitored file without detection. To run Tripwire, a reference database of results needs to be built, immediately after the installation of the operating system and any products, and prior to reconnecting to the network. In this way, it is possible to be sure that the files have not already been modified. The output of Tripwire (as well as Tripwire itself) should ideally be kept on a hardware write protected disk to prevent modification. Tripwire should then be run at regular intervals to verify the integrity of key system files. Another alternative to using hardware protected media is to print out a copy of Tripwire's results. In this way, an intruder must gain physical access to the premises to adjust the original data from Tripwire. This also helps if there is any suspicion on the integrity of the Tripwire database.

Note that it would be meaningless to use Tripwire to protect a file such as the system password file as users have the ability to change their password at any time, and thus the file checksums will also change.

4.3.4. TAMU tiger

`tiger` is a set of scripts that scan a Unix system looking for security problems, in the same fashion as COPS [SSH93]. In addition, it includes a file checksumming facility using Xerox's cryptographic checksum programs to check for both modified system binaries, as well as the presence of required security related patches. This is similar in operation to Tripwire.

4.4. Security Enhancement Tools

Security Enhancement Tools can be used to improve the security of a computer system, particularly to combat a known threat.

4.4.1. TCP Wrapper

TCP Wrapper (also known as LOG TCP) is a package that is used to monitor incoming IP connections, log them, and provide a number of add-on services including a limited form of access control and some sanity checks [Ven92].

The first function is to log connections. Any connection to an IP service that has TCP Wrapper enabled for it will write a connection record to the `syslog` daemon, containing the time and the source of the connection.

If the access control has been enabled, the list will be checked to see if the source of this connection has been allowed or denied access to that IP service. If the service is denied, the connection is aborted. If the service is allowed, then the normal daemon is executed.

If the name checking has been turned on, the wrapper will verify that the name to address mapping is the same as the address to name mapping. If there is any discrepancy, the wrapper concludes that it is dealing with a host that is pretending to have someone else's name (as in an attack on the "r" commands). If this is detected, it is logged and the connection aborted.

TCP Wrapper is an extremely simple, and yet effective tool. It is very useful in preventing connections from outside an organisation from approaching the systems. It is possible to allow certain connections (for example, mail) to the systems, while restricting others. Even if an intruder learns an account and password for the system, they must first penetrate a "trusted" system before they can gain access to the system. It is therefore imperative that users do not use the same password on all systems.

4.4.2. socks

The socks package is a firewall enhancer which allows users to pass out through the firewall transparently [KK92]. Many networks are now moving towards the concept of firewall systems - systems designed to act as an entry portal to the network. Using a firewall system means that no other systems on that network are directly connect to the global network. To access the global network requires accounts on the firewall system, and a two hop process. Granting a large number of users access to the firewall system reduces its effective security, and therefore its performance as a security tool.

Socks automates the process of having a firewall host which is utilised as a transient point for global network access. Although socks itself does not enhance the security of the host it is running on, it does enhance the security of the firewall system by not requiring that every user needing global network access have an account on the firewall. Now the security of the firewall can be more effectively controlled.

From the point of view of a user behind the firewall host, there is no apparent difference between running socks and the regular client software on a host. All connections at the application level will appear to work the same, with the hidden difference that all traffic is passing through a daemon on the firewall host. This transparency is achieved through the socks library routines which applications use in place of the normal socket library calls.

Socks also maintains a security configuration file, which can filter connections based upon host address and service. This is very similar to what TCP Wrapper does, except TCP Wrapper operates on a host by host basis, and the socks package is designed to filter all connections to the local network that pass through a firewall system.

The socks daemon records logging information via `syslog`, which includes:

- access denied
- successful connection
- resource failure

Not e that one of the major issues relating to use of socks is the performance of the firewall system. The configuration of the firewall must be chosen carefully to cope with the burden of supporting all external network connections through the central point. However, since the major task of a firewall machine is to filter packets or allow them through (that is, it will now contain fewer user accounts), there will be more processing power available to perform this task adequately.

4.4.3. passwd+

`passwd+` is a replacement for the Unix password change program `/bin/passwd`, designed to prevent weak password choices before they are used [Bis92b]. `passwd+` takes a configuration file which defines a series of tests. If the proposed password passes any of these tests, it is rejected as being too weak. The tests are configurable by the system administrator to suit local conditions.

passwd+ has the ability to log information to a number of places:

- as input to a command;
- appended to a file;
- written to the system logger `syslogd`;
- written to standard error.

The level of logging may vary from logging errors with the program, to the result of the password tests and who used the system.

These suggested checks include:
dictionary words;
single case passwords;
minimum length;
patterns.

4.4.4. npasswd

`npasswd` is another replacement program for `/bin/passwd`, the program that changes a user's password [Hoo90]. This program deals both with normal password files, and with NIS password files. `npasswd` performs the following checks:
dictionary words;
single case passwords;
minimum length;
illegal characters in passwords (as defined by the system administrator).

`npasswd` checks the passwords and disallows them if they are considered to be insecure. It is possible to perform logging to the system logger `syslogd`.

`npasswd` is quite similar in functionality and intent with `passwd+`, however, `passwd+` offers a richer set of tests and configurations to `npasswd`.

4.4.5. secure_lib

secure_lib contains replacement routines for three SunOS kernel calls: `accept()`, `recvfrom()`, and `recvmsg()` [LeF92]. These replacements are compatible with the originals, with the added functionality that they check the Internet address of the machine initiating the connection to ensure it is authorised to connect. A configuration file defines what hosts are allowed for a given program. Once these replacement routines are compiled, they can be used when building a new shared `libc` library. The resulting `libc.so` can then be put in a special place. Any program that should be protected can then be started with an alternate `LD_LIBRARY_PATH`.

The secure_lib system is especially effective as it provides protection for all daemons, rather than a select few. This allows new daemons to be added to the system, and they can be protected with the same level of confidence. Only the three nominated SunOS kernel calls are replaced. Other kernel calls read data from the network, but only if data is read from a connected socket. Only `accept()` can generate file descriptors for connected sockets, so protecting `accept()` will also provide protection for these kernel calls.

Judicious use of this shared library may prevent rogue users from generating their own daemons, providing services from your system that were not intended.

4.4.6. Kerberos

Kerberos is a distributed authentication system designed to prevent many of the types of common attacks seen today. Kerberos has a number of design criteria which is used to combat some of these threats. The first is that passwords are not transmitted across the network in plain text. This prevents network sniffing of the password and its reuse. The second is that authenticators contain a timestamp which is used to help prevent replay attacks.

Section 2.5 provides a detailed description of Kerberos.

4.4.7. Password Ageing

Yet another problem in the long list of password usage deficiencies is the problem of users not changing their passwords regularly. Even if a secure password is chosen (one that is difficult to guess or crack), the longer the password remains in use, the more chance there is of having that password compromised.

Advances in computer technology are continually reducing the time required to attack a password using the brute force approach (by encrypting every possible password combination and checking them). New speed records place encrypting every possible six (6) character Unix password at less than one hour. [Mad93]

This computation time can only get smaller unless the algorithm is universally changed. Therefore, it is imperative that users choose passwords that are of sufficient length, and change them regularly.

One mechanism for ensuring this happens is to use pro-active password checkers (for enforcing the minimum length - see `passwd+` and `npasswd`), and applying a password ageing scheme to force users to regularly change their passwords. Some Unix systems support these schemes by default [Far91b], but others require an add-on package to implement this feature (password ageing is sometimes implemented as a part of other password enhancement packages, for example [Hei90]).

4.4.8. Token Generators

Token Generators are hardware packages that implement password "tokens", or one-time use passwords [Ell92], [CER92]. Token generators are implemented using a variety of schemes.

One system operates by challenging the user with a seven digit number (in phone number format). A PIN number and the challenge number are entered into the hand held device, and it gives a seven digit response code to reply with.

Other systems use a changing, non-reusable password system. Each time a user authenticates, a new password is supplied by the hand held device. There is no challenge-response system, and the user must keep in synchronisation with password usage to prevent a denial of service. Some of these systems can support single use password generation for up to eight separate host systems. Some of these systems require the user to enter a PIN before the next password is issued.

Another system displays the password continuously, changing it every minute or so. The host must not only keep the user's key (for generating the same sequence), but also a synchronised clock.

The one-time password system is extremely effective in preventing replay attacks, provided the enemy does not know the sequence of generated passwords (either by guessing, or possession of a similar device and key).

One of the major disadvantages is that to authenticate, a user must carry the hardware with them at all times. If they do not possess their hand held device, then authentication cannot take place.

Some of the systems have a requirement for synchronised clocks. This may cause the system to suffer a denial of service due to clocks slipping, or an attacker targeting the clock synchronisation protocols to set the time to any desired value.

4.4.9. S/Key

S/Key is a software system designed to implement a secure one-time password scheme [KHW93]. It uses 64 bits of information transformed by the MD4 message digest algorithm [RFC1320]. The 64 bits are supplied by the user in the form of six English words that are generated by a secure computer. Ultimately, this computer could be a pocket sized smart card, a standalone PC or Macintosh, or a secured machine at work.

The package consists of a number of programs. `keyinit` initialises a user to the S/Key database and requires them to enter a secret password. `key` is then used to generate a number of one-time passwords for future use. `key` requires the user to enter their secret password. Note that if the user does not correctly enter the secret password, a number of one-time passwords will still be generated, but they will not be valid.

There are also replacement programs for the `login` and `su` programs which prompt for the one-time password with a challenge. The output looks like the following:

```
host# ./keyinit
Adding root:
Reminder - Only use this method if you are directly connected.
If you are using telnet or dial-in exit with no password and use keyinit -s
Enter secret password:
Again secret password:

ID root s/key is 99 sh42277
SITU LEFT FLEW MALT MEL PUN
host#
host# key -n 3 99 sh42277
Reminder - Do not use key while logged in via telnet or dial-in.
Enter secret password:
97: MASK THIS WART RUE ANNA IRON
98: TALL TOY CALF AWN HOOK LIT
99: SITU LEFT FLEW MALT MEL PUN
host#
```

Now when a user wishes to use the one-time passwords, the following happens:

```
host> su
s/key 99 sh42277
Password:   !enter SITU LEFT FLEW MALT MEL PUN
host# ^D
host> su
s/key 98 sh42277
Password:   !enter TALL TOY CALF AWN HOOK LIT
host#
```

If the wrong password sequence is entered, it is treated as the same as an incorrect password.

keyinit also operates with a secure option if the user is logged in via an insecure network or communications link. This modifies the behaviour of keyinit by requiring a valid pre-existing one-time password, as follows:

```
host# keyinit -s
Updating root:
Old key sh42277
Reminder - You need the 6 english words from the skey command.
Enter sequence count from 1 to 10000: 99
Enter new key [default sh27276]: sh42277
s/key 99 sh42277
s/key access password: SITU LEFT FLEW MALT MEL PUN

ID root is 99 sh42277
SITU LEFT FLEW MALT MEL PUN
host#
```

From here it is possible to run key to generate a list of valid one-time passwords as before. It was not immediately obvious that the secure option provided much more security than the normal package. Initial observations were that the only difference was that the with the "insecure" option, a small secret password was entered, and with the "secure" option, a previously (any previously) valid one-time password id and sequence was entered. If someone is monitoring the network, they may already have seen a valid one-time password exchange, and therefore be armed with that information already. In fact, the "secure" option allows the user (requires the user) to specify the key, thus opening up a possibly previously used range of passwords for reuse. It was considered the "secure" option to actually be less secure than the "insecure" option for these reasons.

Discussions with the system developers indicate that the intention is not to use a previously valid password, but to generate a valid password, id, and sequence tuple which will allow the user to change to the new id over an insecure network without revealing their secret password. This generation of the next password is done whilst in the secure environment. In fact, once the secret password is set, it need never to transmitted over an insecure network again. It is considered that this was not clearly detailed in the documentation, allowing for confusion which could lead to a lowering of the security of this system.

Despite this misgiving, the S/Key system is an extremely useful and simple mechanism for implementing secure one-time passwords.

4.4.10. Shadow Passwords

One of the major problems with the Unix password system is that the password file is world readable. This is required as many applications need access to parts of the password file for a variety of reasons. Leaving the file world readable means that those applications do not need to be privileged to perform their task.

However, whilst the cryptographic algorithm that encrypts these passwords is still quite strong, the choice of the original password by users still leaves something to be desired. This means that by taking a copy of the password file to another system and running a password guesser on it, it is usually possible to guess at least one password, thus allowing access to the system.

There are a number of solutions to this problem. Shadow Passwords operate by not providing the intruder with a copy of the passwords in the first place. Therefore, the intruder is forced to guess the password by attempting to login, and this can be centrally detected and combatted.

With shadow passwords, the password field in `/etc/passwd` is replaced with some meaningless data (sometimes a "*", sometimes "##", sometimes the username), and the real passwords are stored on a non-world readable file elsewhere. The system programs that access the password field are privileged anyway, and they only need modification to know when to look in the shadow password file.

Shadow passwords are provided by default on some Unix systems [Arn93], but are required to be added-on for other Unix systems (for example, [Hei90]).

4.4.11. Security Extensions to TCP Transport Layer

Work is currently underway to develop security extensions to the TCP transport layer [Bro93]. This involves a modified implementation of the Secure Data Network System protocols [Din90]. The basic system provides authentication through a challenge-response mechanism, and encryption of the exchanged data. This helps to prevent network monitoring. Extensions have currently been added to the FTP and TELNET application layer protocols, and work is progressing to implement a system that works for all TCP and IP connections. This requires a greater modification to existing systems that those currently performed for the applications.

4.4.12. TAMU drawbridge

`drawbridge` is a bridging filter system that runs on a PC using two ethernet cards [SSH93]. It is composed of three different tools: filter, filter compiler and filter manager. By running this type of software on a local PC, it allows specific configurations to be tailored, without affecting all users on a multi-port router. Often, multi-port routers do not have sufficient capacity to route multiple protocols, while providing intelligent filtering functions. In addition, if the router's security is compromised, then the network security is not vulnerable. The PC is not designed as a multi-user machine with remote configuration abilities.

4.5. Programming Methodologies

Whilst some of the problems with security on computer systems are related to design, the proliferation of third party software packages has opened up a new world of security vulnerabilities. Often here, the problems relate to either inexperienced programmers or inadequate care when coding the system. The problem of how to write secure systems has been analysed for many years, and it is possible to write highly secure programs, if a number of basic mistakes are not performed. If it were not possible to secure computer systems, then they would not be connected to global networks.

Many of the solutions and programming styles simply come down to being as conservative as possible with programming, and never trusting the environment the program is operating in. The points detailed here are taken from [Bis87], [Far91b] and the author's own experience.

4.5.1. Close file descriptors

Close all unnecessary file descriptors before calling `exec()`. `exec()` has a documented feature than when the new program is called, "*descriptors open in the calling process remain open in the new process*" [Sun90c]. This means that if the first program was reading a sensitive file using privileges, and calls a user program via `exec()` without closing the file descriptor to that sensitive file, then the user program will also have access to that file.

4.5.2. Check the environment in which the process will run.

This basically means that the program should set up the *entire* environment to exactly how it requires it. This may involve setting known values in environment variables like `HOME`, `PATH`, and `IFS`, setting a valid `umask` value, and initialising all variables.

4.5.3. Never use relative filenames.

This is closely related to the previous problem of determining the environment. Relative filenames and program names allow an intruder to determine the exact location where the file will be read or written to, through control of the environment.

4.5.4. Catch all signals.

This one is not always obvious. When a process dumps core (terminates abnormally taking a copy of the memory image onto disk), the owner of the core file is the same user identity as the effective user identity of the running process. If this process is running as a SUID process, the user identity may be different to the person running the program. If the `umask` value has been set to something inappropriate, then it may be possible to write over the `core` file, while maintaining its original ownership. However, there may be race conditions introduced when catching signals. Signals can be safely ignored, but it may not always be appropriate to do this.

4.5.5. Make only safe assumptions about error recovery.

A privileged program can never assume that all will be well when performing operations. Subtle errors may occur that were never expected (such as not being able to access a file, or running out of file descriptors). These errors must be handled correctly. Do not attempt recovery unless the recovery is guaranteed. It is too easy to open the program up to taking inappropriate action once it loses control of its environment.

4.5.6. Verify all input data.

It is too easy to assume that just because one program wrote the input data, that it will always have the correct format and be valid. Data should be bounded, and verified for syntactic correctness, integrity, and origin if possible. Just because it is stored in a protected file is not sufficient grounds for waiving the responsibility for verifying the data. Any input data under the direct or indirect control of a user is particularly risky, and should always be treated with the utmost of suspicion.

4.5.7. Beware race conditions.

An example of this might be that the program creates a file, and in the next command protects it. In between the two commands, a window of opportunity may exist for an intruder to gain access to a poorly protected file that allows them to gain further access to the system. Attacks against race conditions (once identified) are generally automated, thus allowing the window of opportunity to be explored at a much greater rate than by hand. Never leave the configuration vulnerable, not even for a single instruction.

4.6. Tools use to Combat Threats

A useful exercise is to look at the threats outlined in Chapter 3, and determine which tools outlined in Chapter 4 can be used to combat those threats. Table 4 details that analysis.

Table 4 - Tools use to combat threats

Worms	
Morris	Secure Programming Methodologies, passwd+, npasswd, password ageing, S/Key, Token Generators
WANK	password ageing, S/Key, Token Generators
OILZ	password ageing, S/Key, Token Generators
Trusted Systems	
hosts.equiv	COPS, TCP Wrapper, socks, Kerberos, secure_lib
.rhosts	COPS, TCP Wrapper,socks, Kerberos, secure_lib
secure terminal	TCP Wrapper, socks, Kerberos, S/Key, Token Generators
tftp	COPS, TCP Wrapper, socks, secure_lib
NIS	
+ in passwd	COPS
ypset	TCP Wrapper, socks, secure_lib
ypserv	TCP Wrapper, socks, secure_lib
NIS Packets	socks, Security Extensions
NFS	
exports	COPS, TCP Wrapper, socks, secure_lib
file handles	Security extensions, TCP Wrapper, socks, secure_lib
Passwords	
guessing	CRACK, passwd+, npasswd, password ageing, Token Generators, S/Key
cracking	CRACK, passwd+, npasswd, shadow passwords, password ageing, Token Generators, S/Key
guest	password ageing, Crack
default	password ageing, Crack
Software	
IFS	Secure Programming Methodologies
HOME	Secure Programming Methodologies
PATH	Secure Programming Methodologies
umask	Secure Programming Methodologies
error returns	Secure Programming Methodologies
signals	Secure Programming Methodologies
Monitoring	
network	Kerberos, Token Generators, S/Key, DES, PEM, socks, Security Extensions
trojan horse	Tripwire,
Source Code	Secure Programming Methodologies
Protocol Weakness	
spoofing	Secure Programming Methodologies, TCP Wrapper, secure_lib

Trojans part of message in programs	 Secure Programming Methodologies Tripwire
Spoofing Mail	PEM, PGP, MD2, MD4, MD5, DES, RIPEM, TIS/PEM
suid Files	Secure Programming Methodologies, COPS, Tripwire
World writeable files files devices	 COPS COPS
X Windows	TCP Wrapper, socks, secure_lib
Interaction Configuration	COPS, TCP Wrapper, TAMU, secure_lib, Secure Programming Methodologies

5. Design of Security Enhancement for OpenVMS

This chapter details a design and implementation of the S/Key one time password system [KHW93] for Digital's OpenVMS operating system. OpenVMS is supplied with a number of standard security facilities including `breakin` detection and avoidance, a wide range of audit logging, and tailorable password control (such as lifetime and minimum length). However, OpenVMS is still susceptible to network monitoring attacks, possibly revealing passwords. There do exist products for OpenVMS that combat this type of threat, but these are generally registered products requiring initial purchase and a licence to run them. The author is unaware of any software which is provided free of charge in the public domain for performing this type of security enhancement to OpenVMS.

A simple solution to the problem where network monitoring may capture passwords is to use a one time password scheme. This may involve hardware/software solutions, but in general these are not as useful for providing wide spread use without requiring a sale of some form (e.g., the hardware devices). The optimal solution for enhancing the security of OpenVMS which can be used by anybody is to develop a totally software based solution. Such a solution already exists for Unix (S/Key), so an analysis of this was performed to determine its suitability for the OpenVMS environment. Section 5.1 details a description of the internal structure of the S/Key system.

A design was required for integrating the system into OpenVMS. This was met with a number of problems. OpenVMS provides a callout facility for `LOGINOUT` (the user authentication program) where a systems programmer may "replace" various facilities required by `LOGINOUT` (for example, the password authentication). The complexity of these callouts dictated that the S/Key system be integrated into OpenVMS using a prototype solution first to determine the viability of this extension.

The prototype design and implementation was completed as a two stage approach. The implementation of the first stage required porting the S/Key system from Unix to OpenVMS, and development of a prototype system that used S/Key. Upon completion of the first stage, a major security flaw was identified in this prototype. Further design work in the prototype was undertaken in an attempt to combat this security flaw. Both designs were implemented and tested. The design and implementation of the prototype system is detailed in Sections 5.2 and 5.3.

The demonstration of the modified prototype system indicated that the one time password system for OpenVMS would be an extremely useful addition to the security of OpenVMS. It would effectively combat the threat of network monitoring if it could be implemented in such a way that the identified security flaw in the prototype system could be overcome. This required full integration into the `LOGINOUT` authentication system for OpenVMS. This design and implementation is detailed in Section 5.4.

Section 5.5 identifies a number of areas where improvements in the final implementation could be made.

5.1. Description of S/Key

The S/Key system was developed at Bellcore in the United States. S/Key is a one time password system that is composed entirely of software. It has been designed mainly with Unix in mind (although it has been made available for `MS-DOS` and `Macintosh` as well).

S/Key uses the MD4 Message Digest algorithm [RFC1320] to provide the necessary security of the one time passwords. Each user is initialised into the system by providing an S/Key identification (which may be supplied by S/Key), and a secret password. These two items are concatenated together, and passed through MD4 to generate a 128 bit signature. This signature is collapsed to 64 bits, and forms the user's starting key. As the starting key has been processed through MD4, it is not possible to recreate all the information based upon the signature. Therefore, by storing the user's starting key and identifier (their `ident`) in a file, it is possible to verify the user by having them enter their secret password, and then verifying the starting key by the method detailed above.

The starting key is then processed through MD4, and the resulting 128 bit signature collapsed to 64 bits. This 64 bits is passed back into the MD4 function, and again the result collapsed to 64 bits. This continues until the desired range of passwords is reached. For example, if the user requires passwords 95 to 99 be generated, the one way function is performed 95 times, and then the results for the iteration number 95, 96, 97, 98, and 99 are displayed. These form the one time passwords. The next password to be used is the one with the highest unused number. In this example, that would be password number 99. Note that password number 99 was generated by passing password number 98 through MD4, and collapsing the output down to 64 bits. Therefore, it is not possible to determine password number 98 (the next password to be used) from knowledge of password number 99.

The 64 bit passwords are processed through a routine that maps set bit positions onto small English words. This allows for an easy display of the passwords for the users to interact with. For example, a password may look like:

```
SITU LEFT FLEW MALT MEL PUN
```

Passwords are displayed in this form, and are entered by users in this form. They are then converted back to the 64 bit representation for comparison.

The S/Key data file stores the identifiers for users, and the last valid S/Key one time password they successfully entered. When S/Key is used to verify a user, they are challenged for the password that is one less than the last successful one. This is entered by the user, converted to 64 bit binary, and then passed through the MD4 one way function. If the result of this matches the result stored in the file, then the user is considered to be valid. The entry for the user is updated to reflect the latest correctly entered password.

The S/Key data file must be protected from modification by unauthorised users. Therefore, the programs that access it for modification must be privileged in order for them to successfully update the file. On Unix, this is done via the `SUID` bit. On OpenVMS, this can be achieved by installing the programs with the `SYSPRV` privilege.

S/Key also provides a "secure" mode when reinitialising a user's S/Key entry. This does not require a user to generate a starting key by specifying a secret password (which may be monitored over an insecure network). Instead, the user should have previously generated a valid list of one time passwords while they are within their secure environment. From this point, the user may move to a new list by specifying the identifier, sequence number, and valid one time password. These items are entered into the system as being accurate without further checking. Only if these items are entered correctly can the next password to be used be verified successfully. The next password to be used will be the one with a sequence number one lower than the specified sequence number.

The S/Key system provides a simple yet effective solution to the problem of intruders monitoring a network for passwords.

5.2. Prototype Design

As such, there was very little design work in the initial stages to get S/Key to function under OpenVMS. A system already existed for Unix, so it was deemed to be prudent to remain as compatible with that as possible.

The design of the S/Key prototype integration was done entirely in user mode (did not require integration into the OpenVMS operating system). The program is designed to be called on *every* attempt at logging in to the system (including DECwindows, Internet access, and DECnet access). The design of the system is detailed below:

- Upon entry, the program immediately establishes an exit handler. This exit handler checks the state of a global variable, and if not favourable will kill the process. This prevents any problems where a user may attempt to interrupt the execution of the program or the program terminates due to an abnormal condition;
- If the terminal that the process is being created on does not have a physical device name, then this connection is not interactive and is a process related to the operation of other services such as DECwindows. These processes must be allowed to succeed, and the system can then proceed to check the next process when it logs in. DECwindows generally creates two processes when a new application is started. If the application is something like the calendar or clock, it will be unable to supply a password, and hence cannot be authenticated in this way. The global exit variable is set to a favourable state, and the program exited;
- If the user is found within the S/Key system, they are prompted for the next valid password. They will be given a set timeout period to enter this password in;
- If the correct password is entered successfully, then the program terminates favourably. Else, the process is killed;
- If there are any errors returned from any system calls, the process is terminated.

This system is designed to be run by every process that logs in, and it will detect which processes require challenging by S/Key. Therefore, the command to run this may be placed in the system login file `SYS$MANAGER:SYLOGIN.COM` which is run by every process being created.

However, if an intruder captures an account and password for a privileged user, and then suddenly discovers on logging in that this system is in use, they may then attack the computer system by using other forms of connection, such as network connections. This may allow an intruder to edit the `SYS$MANAGER:SYLOGIN.COM` file and remove the command to execute the S/Key system. Network connections cannot be challenged by the S/Key system as they cannot enter the valid password.

Therefore, the prototype design was modified to help combat this type of attack. The following changes were designed:

- If a user is challenged by the S/Key system and fails to provide the correct password, a log file is kept in the user's home directory, detailing the connect time and the source of the connection;
- When a process is executing the S/Key challenge system, the system now checks for the mode of the process. If the mode is not `INTERACTIVE` (as opposed to `BATCH`, `NETWORK`, `DETACHED`, and so on), then a check is made to see if a non-empty log file exists for the user. If this is found, the

process is terminated. The reasoning behind this is that if an intruder tries once using a captured account and login password but fails to supply the correct S/Key one time password, they may then attempt to defeat the system by accessing the computer via a non-interactive method such as `FTP`, DECnet `COPY`, and so on. If a non-empty log file exists for the user, these processes will be immediately terminated, thereby effectively preventing any further access to the system using the captured account and password;
- The only mechanism for using the account now is to login interactively, and successfully pass the S/Key system. If the correct S/Key one time password is entered, then any existing log file is printed out on the screen for the user. This serves to warn the user that their login password may have been used by an unauthorised person.

Whilst this modified design provides slightly more protection against defeating the S/Key challenge system, it is still relatively easy to circumvent if knowledge of the existance of the system is gained by the intruder. They could for example, immediately disable the system prior to login interactively. Another mechanism might be that they capture more than one account and password, and as soon as one account is disabled (thus alerting the intruder to the presence of the S/Key challenge system), they may then enter the system using a second account to defeat the system.

These deficiencies are obviously significant, and render the security provided the existing prototype S/Key challenge system as insufficient for most needs. It may however serve as a "tripwire" or early warning device that security problems may exist on the computer system.

A design and implementation for how to integrate S/Key into OpenVMS in a way that it cannot be tampered with is given in Section 5.4.

5.3. Prototype Implementation

The implementation of S/Key for OpenVMS first required porting the existing S/Key code (written in the C language) to OpenVMS. A number of incompatibilities were encountered, and these required solutions to be developed to allow the system to function. A number of Unix specific functions were accessed by S/Key, and replacements for them required development under OpenVMS. The following problems and steps were required to port the code to OpenVMS:

- The VAXC compiler [DEC89b][13] does not understand line breaks in a `#if` line. Therefore, some compiler functionality was required to handle this deficiency;
- S/Key made use of the `gethostname()`, `getuid()`, `getpwuid()`, `getpwnam()`, and `getopt()` Unix system calls. These are not provided by OpenVMS and required replacement routines to be developed;
- The function `readpass()` required replacement for OpenVMS;
- OpenVMS generally does not support strict case matching and hence most information is supplied to programs in uppercase. However, for Unix compatibility, the VAXC language supports case matching and all arguments are supplied to the program in the exact case they were typed in. This caused problems with case matching that required resolution;

[13] Due to be replaced by Digital soon.

- A bug existed in the existing S/Key system which caused an array overflow and variables to be destroyed. The bug fix has been communicated back to the authors of S/Key and has now been incorporated into their system;
- Checking for a valid username and the privileges of a user required adapting to the OpenVMS environment;
- The location of the data file required modification for OpenVMS;
- OpenVMS requires different error and program return codes to the existing Unix system.

All extensions to the S/Key system for OpenVMS have been conditionally included into the code using the

```
#ifdef VAXC
#endif
```

conditional compilation directives. The `VAXC` symbol was chosen over the `VMS` symbol as some of the assumptions made for OpenVMS were only relevant to the VAX architecture. The author did not have access to the Alpha AXP architecture to perform development work for OpenVMS Alpha.

It is possible to initialise users other than yourself into the S/Key system. On Unix, this can only be performed by the `root` user (UID 0) which is the privileged user. However, on OpenVMS privileges are specified with a finer distinction than Unix, allowing partial privileges to be present, but not all privileges. Possession of any of the following privileges was deemed to be sufficient to indicate a privileged user:

`SYSPRV`	may access objects via system protection
`SETPRV`	may set any privilege bit
`BYPASS`	may bypass object access controls

Whilst possession of other privileges may ultimately lead to the possession of any of those privileges above (such as `LOG_IO`, `PHY_IO`, `CMKRNL`), it is not necessary to check for the possession of these privileges as well. These privileges are designed to represent different access to the operating system and its objects. The fact that they can have a side effect of allowing other privileges to be obtained is of no consequence to this implementation.

Despite security flaws in the prototype design and implementation, it still demonstrates the usefulness of using S/Key to combat network monitoring, and improves security. For an intruder to combat this design, they must have already successfully captured a valid account and password of a privileged account. If the intruder is unaware of the presence of this S/Key prototype, then that captured account will be disabled when the intruder fails to supply the correct one time password. If the S/Key prototype is not in operation, then the security of the system has already been compromised severely. Ideally, the full integration of S/Key into the OpenVMS authentication procedures (`LOGINOUT`) is highly desired (see Section 5.4).

The modified S/Key code and the prototype implementation can be found in Appendix A.

5.4. Design and Implementation for Full Integration

To fully integrate the S/Key system securely into OpenVMS requires use of the new `LOGINOUT` Callouts facility [Cov90]. Only by using this mechanism can any confidence be gained that an attacker cannot modify or bypass the system.

OpenVMS has the concept of a second password for accounts. This is used for example, to provide two person authentication to a privileged account so that no single person can act without a second person also authenticating them into the system. Unfortunately, the S/Key system presently only supports a single password per account. Therefore, any accounts using two passwords are considered to be exempt from the S/Key system.

If the account does not possess any passwords, then no processing takes place and a success status is returned to `LOGINOUT`. Password prompting can only be performed on `INTERACTIVE` processes. Therefore, for any non-interactive processes, the S/Key callout returns a success status immediately, so that normal OpenVMS processing can continue. A `SUBPROCESS` of a parent cannot be prompted for a password, so like the previous situation, a success status is returned, and normal OpenVMS processing continued.

If any errors occur while processing the S/Key account or password, it is ignored, and the supplied password is considered to be an OpenVMS password. If subsequent verification of the OpenVMS password fails, the user is denied access to the system, and auditing and `breakin` detection is performed.

If the user is not found within the S/Key system, then they are given a standard OpenVMS password prompt. If they are found within the S/Key system, they are given an S/Key password prompt.

If the entered password fails to be authenticated using S/Key, it is checked using standard OpenVMS password checks. This is in case the user wishes to enter the system using their standard OpenVMS password (for example, if they are entering from within their secure environment).

As stated before, if the account uses two passwords, the second OpenVMS password is prompted for, and verified. By this stage, the first OpenVMS password will have been verified. Standard OpenVMS errors are passed back to `LOGINOUT` in reply to any problems with the authentication.

If any error occurs, the status is returned to `LOGINOUT` for processing. Otherwise, the status `LGI$_SKIPRELATED` is returned which indicates that the user has been fully authenticated and no further OpenVMS supplied authentication is required.

The DECwindows interface also calls the `LGI$ICR_AUTHENTICATE` callout to do user authentication. This is done after normal username and password prompting has been completed. Therefore, another callout is established which if called by the DECwindows initialisation routines, will disable all other callouts so that normal OpenVMS DECwindows authentication takes place. This is acceptable as a user normally logs in using DECwindows from within their secure environment.

Interfacing to the `LOGINOUT` Callouts requires careful programming and consideration. The module is built as a shareable image, and is invoked by `LOGINOUT` if a number of conditions are met. The commands to compile and link the module can be found in Section A.3.

The callout module requires a universal symbol `LGI$LOGINOUT_CALLOUTS` which defines a vector of pointers to callout routines. If a callout routine is not being used, its entry in the vector is `0`. When a callout is executed, a number of standard parameters are passed to it. These are the argument vector and context. The argument vector is a vector of useful items including `LOGINOUT` callbacks (utility routines supplied by `LOGINOUT` to the callouts) plus `LOGINOUT` parameters (such as the username of the user trying to login), and a context which can be established at the start of processing.

The S/Key authentication callout requires a number of facilities from the argument vector. These include:

LGI$ICB_PASSWORD	do password prompting and reading;
LGI$ICB_VALIDATE	do OpenVMS password checking if the password is not authenticated by S/Key;
LGI$A_ICR_CREPRC_FLAGS	determine if the process is interactive, invoked from a logged in process, or is a subprocess;
LGI$A_ICR_PWDCOUNT	determine how many passwords this account has.

Once the module is linked as a shareable image, it must be placed in the SYS$SHARE: directory, and then installed into OpenVMS. Since LOGINOUT is a privileged program, OpenVMS dictates that it can only call installed images. This is done using the INSTALL command as demonstrated:

```
$ INSTALL REPLACE SYS$SHARE:LGI$CALLOUT_SKEY.EXE
```

A system-exec mode logical must then be defined that points to the filename. This can be done like:

```
$ DEFINE/SYSTEM/EXEC LGI$LOGINOUT_CALLOUTS LGI$CALLOUT_SKEY
```

Finally, to guard against subversion of the callouts by an intruder, a SYSGEN parameter is defined which dictates the number of callout modules that is expected. If this number differs from the definition, then the login fails. The sysgen parameter can be changed by:

```
$ RUN SYS$SYSTEM:SYSGEN
SYSGEN> SET LGI_CALLOUTS 1
SYSGEN> WRITE ACTIVE
SYSGEN> WRITE CURRENT
SYSGEN> EXIT
```

Note that these commands can only be executed by privileged users.

Testing the system showed a few more inconsistencies with S/Key and OpenVMS. When the one time password was returned from the LGI$ICB_PASSWORD call, it looked like:

```
FURYNASHNOBDEWPITBUG
```

instead of:

```
FURY NASH NOB DEW PIT BUG
```

Since it is almost impossible to determine where the spaces should have been placed, a design change was made to support OpenVMS. This design change was to place the "." character between S/Key words, rather than the space. The skey program was modified to reflect this change. Then internally to the callout, the "." characters are replaced by spaces prior to verifying the password. This has the effect of making the password valid to S/Key in the form it normally uses, even though it is entered in another form by the user. The resulting password now looks like:

```
FURY.NASH.NOB.DEW.PIT.BUG
```

A similar change was required to `skeyinit` when using the secure mode option. Using this option required the user to enter a one time password. Whilst it is possible to have the user enter the English words separated by spaces, it was deemed to be prudent to change `skeyinit` to accept the words with the "." character separating them for consistency with the rest of the package.

In addition to the above changes, the requirements of linking a shareable image meant that the normal VAXC run time library was not available. This dictated that the routine `strftime()` be developed, and linked in with the existing code.

Now when a user logs in to a system if the user is entered into the S/Key system, the interaction looks like:

```
Username: USER1
s/key 91 ko492804:        ! enter FURY.NASH.NOB.DEW.PIT.BUG
Welcome to the OpenVMS system.
$
```

If the user is entered into the S/Key system, but chooses to use their OpenVMS password (for example, they are logging in from within their secure environment):

```
Username: USER1
s/key 90 ko492804:        ! enter MYPASSWORD
Welcome to the OpenVMS system.
$
```

If a user is not entered into the S/Key system, then normal password prompting and validation will take place:

```
Username: USER3
Password:                 ! enter MYPASSWORD
Welcome to the OpenVMS system.
$
```

Any errors or failure to validate a password will cause the login to fail, and `breakin` detection to be performed by OpenVMS. This new design and implementation cannot be tampered with by an intruder. Using this new system, the passwords will remain secure provided the OpenVMS password is never transmitted across an open network, and only used internally on the secure network. If the user is required to login from an insecure network, they may optionally use a one time password to prevent replay attacks.

This greatly improves the security of OpenVMS as an intruder can no longer monitor the network and capture a valid username and password.

5.5. Future Work

More testing is required to determine how to protect the S/Key data file from modification by unauthorised means. Currently, all testing has been performed using a privileged account, which simplifies matters. Ultimately, the `skey` and `skeyinit` programs must be installed with privileges, so that access and modification of the S/Key data file is possible by non-privileged users, and the data file can then be better protected using standard OpenVMS file protections (such as permission bits [DEC89a], or ACLs [DEC88b]).

The date handling should be changed to conform to OpenVMS date formats, thereby removing the dependance on the routine `strftime()`.

6. Conclusion

The problem of computer and network security is an extremely complex one. It has received increased attention over the past five years, and many tools and techniques have been developed to combat a wide range of threats. This report analyses a wide range of threats, and describes and categorises a collection of available tools and techniques used to combat those threats. Through this analysis, the awareness of the issues relating to computer security will be heightened.

This report examines a number of authentication techniques, looking at their strengths and weaknesses, and draws comparisons between them. This will aid security personnel in deciding the most appropriate authentication technique to meet their requirements.

A list of known threats to computer systems is detailed, and these threats are classified using standard criteria. Heightened understanding of the types and classes of computer threats will help systems administrators and programmers to avoid common mistakes in the future.

An examination of tools and techniques is performed. For each of the listed threats, the tools and techniques that can help combat that threat is determined. This is a comprehensive list, allowing a wide range of choice as to which solution (or solutions) best serve the needs of the computer community.

Following the examination, an enhancement in the security of OpenVMS is provided through the provision of a one time password scheme based upon S/Key. Further development work is required in this area.

6.1 Acknowledgement

A project of this scale could not have been completed effectively without the support and guidance of the author's supervisor, Dr. Jadwiga Indulska. Her suggestions, comments, and gentle steering were greatly appreciated.

7. Bibliography

[Alv90] De Alvare A. M., *How Crackers Crack Passwords or What Passwords to Avoid*, Proceedings of the UNIX Security Workshop II, Portland, August 1990.

[Arn93] Arnold N., *UNIX Security: A Practical Tutorial*, McGraw-Hill Inc., 1993.

[Aus93] Austen J., Fifth Computer Incident Handling Workshop, St. Louis, MO., August 1993.

[BB91] den Boer B. and Bosselaers A., *An Attack on the Last Two Rounds of MD4*, Proceedings of the Crypto'91 conference, Santa Barbara, August 1991.

[BB93] den Boer B. and Bosselaers A., *Collisions for the compression function of MD5*, Pre-proceedings of the EUROCRYPT 93 conference, Lofthus, May 1993.

[Bis87] Bishop M., *How to Write a Setuid Program*, ;login, Volume 12, Number 1, January/February 1987.

[Bis92a] Bishop M., *Proactive Password Checking*, Proceedings of the 4th Workshop on Computer Security Incident Handling, Denver, August 1992.

[Bis92b] Bishop M., *README file for passwd+*, anonymous ftp from dartmouth.edu, June 1992.

[BKS90] Baran F., Kaye H., and Suarez M., *Security Breaches: Five Recent Incidents at Columbia University*, Proceedings of the UNIX Security Workshop II, Portland, August 1990.

[BM91] Bellovin S. and Merritt, M., *Limitations of the Kerberos Authentication System*, Proceedings of the USENIX Winter 1991.

[Bra90] Brand R., *Coping with the Threat of Computer Security Incidents: A Primer from Prevention through Recovery*. CERT 0.6, June 1990.

[Bro93] Brown L, *On Implementing Security Extensions to the TCP Transport Layer*, Proceedings of the 16th Australian Computer Science Conference (ASCS-16), Brisbane, February 1993.

[CER92] Computer Emergency Response Team, *Internet Security for UNIX System Administrators*, Presented at AARNet Networkshop, December 1992.

[CER93] Computer Emergency Response Team Advisory 93:14, *Internet Security Scanner (ISS)*, September 1993.

[Che92] Cheswick W.. *An evening with Berferd in which a Cracker is Lured, Endured, and Studied*, Proceedings of the Winter USENIX Conference, San Francisco, January 1992.

[CLS91] Caelli W., Longley D., and Shain M., *Information Security Handbook*, Stockton Press, 1991.

[Cly93] Clyde R., *DECnet Security (Not Necessarily an Oxymoron)*, Computers and Security, March 1993.

[Coh92] Cohen F., *A Formal Definition of Computer Worms and Some Related Results*, Computers and Security, Volume 11, Number 7, November 1992.

[Cov90] Covert J., *Functional Specification for Callouts for LOGINOUT & DECnet Session*, Version T1.0.0, Digital Equipment Corporation, July 1990.

[Cur90] Curry D., *Improving the Security of your UNIX System*, ITSTD-721-FR-90-21, SRI International, April 1990.

[DEC88a] *Guide to DECnet-VAX Networking Version 5.0*, Digital Equipment Corporation, April 1988.

[DEC88b] *VMS Access Control List Editor Manual Version 5.0*, Digital Equipment Corporation, April 1988.

[DEC89a] *Guide to VMS System Security Version 5.2*, Digital Equipment Corporation, June 1989.

[DEC89b] *VAX C Run-Time Library Reference Manual Version 3.1*, Digital Equipment Corporation, December 1989.

[DEC90] *VMS Authorize Utility Manual Version 5.4*, Digital Equipment Corporation, August 1990.

[Din90] Dinkel C., *Secure Data Network System (SDNS) Network, Transport and Message Security Protocols*, NIST, NISTIR-90/4250, March 1990.

[Edw90] Edwards B., *How to Survive a Computer Disaster*, Proceedings of the DECUS Symposium, August 1990.

[Ell92] Ellison C., *RESULTS: challenge login devices*, Usenet newsgroup sci.crypt, 6 October 1992.

[Far91a] Farmer D., *README.1 file from COPS system*, anonymous ftp from cert.org, November 1991.

[Far91b] Farrow R., *Unix System Security: How to Protect your Data and Prevent Intruders*, Addison-Wesley, April 1991.

[FIP77] Federal Information Processing Standards Publication 46, *Data Encryption Standard*, National Bureau of Standards, U.S. Department of Commerce, January 1977.

[Goa92] Goatley H., *Supervisor Reference Guide*, anonymous ftp from ftp.spc.edu, October 1992.

[Gro93] Grottola M., *The UNIX Audit: Using UNIX to Audit UNIX*, McGraw-Hill Inc., 1993.

[GS91] Garfinkel S. and Spafford G., *Practical UNIX Security*, O'Reilly and Associates, Inc., 1991.

[Hei90] Heirtzler J., *shadow.howto file from shadow system*, anonymous ftp from csc2.anu.edu.au, April 1990.

[Hey93a] Van Heyningen M., *RIPEM Frequently Asked Questions*, Usenet newsgroup alt.security.ripem, 31 March 1993.

[Hey93b] Van Heyningen M., *RIPEM Frequently Noted Vulnerabilities*, Usenet newsgroup alt.security.ripem, 31 March 1993.

[Hoo90] Hoover C., *README file from npasswd system*, anonymous ftp from ftp.cc.utexas.edu, March 1990.

[HY92] Harn L. and Yang S., *Group Oriented Undeniable Signature Schemes without the Assistance of a Mutually Trusted Party*, Proceedings AUSCRYPT '92, Gold Coast, December 1992.

[IBM89] *Virtual Machine/Directory Maintenance - Operation and Use, Release 4*, International Business Machines,1989.

[ISO92] International Standards Organisation ISO 9594-8: *The Directory: Authentication Framework*, 1992 (also known as CCITT Recommendation X.509).

[JM91] Janson P and Molva R., *Security in Open Networks and Distributed Systems*, Computer Networks and ISDN Systems, Volume 22, Number 5, October 1991.

[KC90] Kaplan R., and Clyde R., *Viruses, Worms, and Trojan Horses - Part VI: The War Continues*, Proceedings DECUS Fall 1990, Las Vegas, 1990.

[KCS90] Kohl J., Neuman B., and Steiner J., *The Kerberos Network Authentication Service*, MIT Project Athena, Version 5 Draft 3, October 1990.

[KHW93] Karn P., Haller N., and Walden J., *S/Key One Time Password System*, anonymous ftp from thumper.bellcore.com, July 1993.

[KK92] Koblas D. and Koblas M., *SOCKS*, Proceedings of the USENIX Security Symposium, 1992.

[Kle90] Klein D., *"Foiling the Cracker": A Survey of, and Improvements to, Password Security*, Proceedings of the UNIX Security Workshop II, Portland, August 1990.

[KS92] Kim G. and Spafford E., *README file from Tripwire system*, anonymous ftp from cert.org, November 1992.

[Kur90] Kuras J., *An Expert Systems Approach to Security Inspection of UNIX*, Proceedings of the UNIX Security Workshop II, Portland, August 1990.

[LAB92] Lampson B., Abadi M., Burrows M., and Wobber E., *Authentication in Distributed Systems: Theory and Practice*, acm Transactions on Computer Systems, November 1992.

[Lau92] Laun, R., *Asymmetric User Authentication*, Computers and Security, Volume 11, Number 2, April 1992.

[Law93] Lawrence L., *Digital Signatures - Explanation and Usage*, Computers and Security, Volume 12, Number 3, May 1993.

[LeF92] LeFebvre W., *Restricting Network Access to System Daemons under SunOS*, securelib system, anonymous ftp from eecs.nwu.edu, 1992.

[LS93] Longstaff T. and Schultz E., *Beyond Preliminary Analysis of the WANK and OILZ Worms: A Case Study of Malicious Code*, Computers and Security, Volume 12, Number 1, February 1993.

[Mad93] Madsen J., *World record in password checking*, Usenet newsgroup alt.security, 1 September 1993.

[MLJ92] McCanne S., Leres C., and Jacobson V., *README file from tcpdump system*, anonymous ftp from ftp.ee.lbl.gov, May 1992.

[Mor90] Moraes M., *YP is not secure*, Security Digest, Volume 3, Issue 12, May 1990.

[MP92] Mui L. and Pearce E., *X Window System Administrator's Guide*, O'Reilleys & Associates Inc., 1992.

[Muf92] Muffett A., "*Crack Version 4.1*" *A Sensible Password Checker for Unix*, anonymous ftp from cert.org, March 1992.

[Ney92] Ney S., *README file from TAP system*, anonymous ftp from ftp.cs.tu-berlin.de, March 1992.

[OSI92] *The OSI Security Package, OSISEC Users Manual V0.2*, July 1992.

[RFC783] Sollins K., *The TFTP Protocol (Revision 2)*, Network Working Group, RFC783, June 1981.

[RFC1094] Sun Microsystems Inc., Network File System Protocol Specification, Network Working Group, RFC1094, March 1989.

[RFC1319] Kaliski B., *The MD2 Message-Digest Algorithm*, Network Working Group, RFC1319, April 1992.

[RFC1320] Rivest R., *The MD4 Message-Digest Algorithm*, Network Working Group, RFC1320, April 1992.

[RFC1321] Rivest R., *The MD5 Message-Digest Algorithm*, Network Working Group, RFC1321, April 1992.

[RFC1421] Linn J., *Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures*, Network Working Group, RFC1421, February 1993.

[RFC1422] Kent S., *Privacy Enhancement for Internet Electronic Mail: Part II: Certificate-Based Key Management*, Network Working Group, RFC1422, February 1993.

[RFC1423] Balenson D., *Privacy Enhancement for Internet Electronic Mail: Part III: Algorithms, Modes, and Identifiers*, Network Working Group, RFC1423, February 1993.

[RFC1424] Kaliski B., *Privacy Enhancement for Internet Electronic Mail: Part IV: Key Certification and Related Services*, Network Working Group, RFC1424, February 1993.

[RSA78] Rivest R., Shamir A., and Adleman L., *A Method for Obtaining Digital Signatures and Public-key Cryptosystems*, Communications of the ACM, February 1978.

[SER93] Security Emergency Response Team Advisory 93.04, *Guidelines for Developing a Sensible Password Policy*, June 1993.

[Sho93] Shostack A., *NIS Issues*, Usenet Newsgroup comp.security.misc, 4 August 1993.

[Smi93] Smith D., *Public Key Cryptosystems, Certificates, and Certification Authorities*, Security Emergency Response Team, October 1993.

[Spa88] Spafford, E., *The Internet Work Program: An Analysis*, Technical Report CSD-TR-823, Department of Computer Science, Purdue University, November 1988.

[Spa92] Spafford E., *OPUS: Preventing Weak Password Choices*, Computers and Security, May 1992.

[Spa93] Spafford E., *NIS Issues*, Usenet Newsgroup comp.scurity.misc, 4 August 1993.

[SSH93] Safford D., Schales D., and Hess D., *Texas A & M Network Security Package Overview*, anonymous ftp from sc.tamu.edu, July 1993.

[Ste90] Stevens W., *UNIX Network Programming*, Prentice Hall, 1990.

[Sto89] Stoll C., *The Cuckoo's Egg*, Doubleday, 1989.

[Sun90a] System and Network Administration, SUN Microsystems, Revision A, March 1990.

[Sun90b] SunOS Reference Manual, Volume 1, SUN Microsystems, Revision A, March 1990.

[Sun90c] SunOS Reference Manual, Volume 2, SUN Microsystems, Revision A, March 1990.

[Tan89] Tanenbaum A., *Computer Networks*, Prentice-Hall International Inc.1989.

[TAP90] Tardo J., Alagappan K., and Pitkin R., *Public Key Authentication using Internet Certificates*, Proceedings of the UNIX Security Workshop II, Portland, August 1990.

[TIS93] *TIS/PEM FAQ (Frequently Asked Questions)*, anonymous ftp from ftp.tis.com, June 1993.

[UCL92] *Research Report*, Department of Computer Science, University College London, 1992.

[Ven92] Venema W., *BLURB file from TCP Wrapper system*, anonymous ftp from cert.org, June 1992.

[Zim92] Zimmermann P., *README file from PGP system*, anonymous ftp from ghost.dsi.unimi.it, November 1992.

A. Appendix A - Sample Source Code

This Appendix contains the source code used to implement the S/Key system under OpenVMS. It contains the original files from the S/Key system, plus additional files supplied by the author. Any changes or additions are marked with change bars in the left margin.

A.1. Header Files

A.1.1. challenge.h

```
/*
**  Define the log file location
*/
#define LOG_FILE      "sys$login:challenge.log"
#define VERBOSE_LOG   "sys$login:challenge_run.log"

/*
**  Define if debugging code is required
*/
/* #define DEBUG      /* if debugging the code */

/*
**  Define if verbose mode is required (for debugging)
*/
/* #define VERBOSE    /* if this is low level debugging */

/*
**  Define if you don't want the process really killed
**   (for debugging only!!!!)
*/
/* #define NO_KILL    /* don't kill the process */

/*
** Define some boolean types for our use
*/
#define BOOLEANenum{ TRUE, FALSE};
/*
**  Maximum length of input buffer from terminal
*/
#define ANSWER_LEN    128

/*
**  Size of buffer from log file
*/
#define BUFFER_LEN    132

/*
**  Size of buffer for time strings
*/
#define TIME_BUF_LEN  128

/*
**  Size of buffer to return DVI info in
*/
#define DVI_BUF_LEN   128

/*
**  Size of buffer to return JPI info in
*/
#define JPI_BUF_LEN   128
```

A.1.2. md4.h

```
#ifdef  __STDC__
#define __ARGS(X) X    /* For ANSI C */
#else
#define __ARGS(X) ()
#endif

/*
 *
 * md4.h -- Header file for implementation of MD4 Message Digest Algorithm
 * Updated: 2/13/90 by Ronald L. Rivest
 * (C) 1990 RSA Data Security, Inc.
 * Reformatted and de-linted - 2/12/91 Phil Karn
 */

/* MDstruct is the data structure for a message digest computation. */
typedef struct {
        unsigned long buffer[4];/* Holds 4-word result of MD computation */
        unsigned char count[8];       /* Number of bits processed so far */
        unsigned int done;    /* Nonzero means MD computation finished */
} MDstruct, *MDptr;

/* MDbegin(MD)
 * Input: MD -- an MDptr
 * Initialize the MDstruct prepatory to doing a message digest computation.
 */
extern void MDbegin __ARGS((MDptr MDp));

/* MDupdate(MD,X,count)
 * Input: MD -- an MDptr
 *        X -- a pointer to an array of unsigned characters.
 *        count -- the number of bits of X to use (an unsigned int).
 * Updates MD using the first ``count'' bits of X.
 * The array pointed to by X is not modified.
 * If count is not a multiple of 8, MDupdate uses high bits of last byte.
 * This is the basic input routine for a user.
 * The routine terminates the MD computation when count < 512, so
 * every MD computation should end with one call to MDupdate with a
 * count less than 512.  Zero is OK for a count.
 */
extern void MDupdate __ARGS((MDptr MDp,unsigned char *X,unsigned int count));

/* MDprint(MD)
 * Input: MD -- an MDptr
 * Prints message digest buffer MD as 32 hexadecimal digits.
 * Order is from low-order byte of buffer[0] to high-order byte of buffer[3].
 * Each byte is printed with high-order hexadecimal digit first.
 */
extern void MDprint __ARGS((MDptr MDp));
/* End of md4.h */
```

A.1.3. skey.h

```
#if     defined(__TURBOC__) || defined(__STDC__) || defined(LATTICE)
#define ANSIPROTO       1
#endif

#ifndef __ARGS
#ifdef  ANSIPROTO
#define __ARGS(x)       x
#else
#define __ARGS(x)       ()
#endif
#endif

/* Server-side data structure for reading keys file during login */
struct skey {
        FILE *keyfile;
        char buf[256];
        char *logname;
        int n;
        char *seed;
        char *val;
        long   recstart; /*needed so reread of buffer is efficient*/


};

/* Client-side structure for scanning data stream for challenge */
struct mc {
        char buf[256];
        int skip;
        int cnt;
};

void f __ARGS((char *x));
int keycrunch __ARGS((char *result,char *seed,char *passwd));
char *btoe __ARGS((char *engout,char *c));
char *put8 __ARGS((char *out,char *s));
int etob __ARGS((char *out,char *e));
void rip __ARGS((char *buf));
int skeychallenge __ARGS((struct skey *mp,char *name));
int skeylookup __ARGS((struct skey *mp,char *name));
int skeyverify __ARGS((struct skey *mp,char *response));
```

A.2. Source Code Files

A.2.1. gethostname.c

```
#include <stdio.h>
#include <ssdef.h>
#include <syidef.h>

#ifdef NOT_USED
main()
{
    char    name[ 8];

    gethostname( name, 8);
    printf( "hostname is %s\n", name);
}
#endif

int gethostname( name, namelen)
char    *name;
int     namelen;
{
    int                ret_length;
    struct
    {
        unsigned short len;
        unsigned short item;
        char           *buffer;
        int            *ret_len;
        int            terminator;
    } item_list = { namelen, SYI$_NODENAME, name, &ret_length, 0};
    unsigned int       status;
    struct
    {
        unsigned int   status;
        unsigned short ret_length;
        unsigned short padding;
    } iosb;
    char               *cp;

    status = sys$getsyi( 0, 0, 0, &item_list, &iosb, 0, 0);
    cp = name;
    while ( *cp != '\0')
    {
        *cp = ( char) tolower( ( int) *cp);
        cp++;
    }
    if ( status == SS$_NORMAL)
    {
        return( -1);
    }
    return( 0);
}
```

A.2.2. getskeyprompt.c

```
/*   get prompt code for S/KEY Authentication.  S/KEY is a trademark
 *   of Bellcore.
 *
 *   Mink is the former name of the S/KEY authentication system.
 *   Many references for mink  may still be found in this program.   */

#include <stdio.h>
#include <string.h>
#ifdef VAXC
#include <types.h>
#include <stat.h>
#else
#include <sys/types.h>
#include <sys/stat.h>
#endif
#include <time.h>
#include <errno.h>
#include "skey.h"

#ifdef VAXC
#define KEYFILE        "SYS$MANAGER:SKEYKEYS.DAT"
#else
#define KEYFILE        "/etc/skeykeys"
#endif

char *skipspace();
int skeylookup __ARGS((struct skey *mp,char *name));



/* Issue a skey challenge for user 'name'. If successful,
 * fill in the caller's skey structure and return 0. If unsuccessful
 * (e.g., if name is unknown) return -1.
 *
 * The file read/write pointer is left at the start of the
 * record.
 */
int
getskeyprompt(mp,name,prompt)
struct skey *mp;
char *name;
char *prompt;
{
        int rval;

        sevenbit(name);
        rval = skeylookup(mp,name);
        strcpy(prompt,"s/key 55 latour1\n");
        switch(rval){
        case -1:       /* File error */
               return -1;
        case 0:                /* Lookup succeeded, return challenge */
               sprintf(prompt,"s/key %d %s\n",mp->n - 1,mp->seed);
               return 0;
        case 1:                /* User not found */
               fclose(mp->keyfile);
               return -1;
        }
        return -1;     /* Can't happen */
}
```

A.2.3. md4.c

```
/*
 * md4.c -- Implementation of MD4 Message Digest Algorithm
 * Updated: 2/16/90 by Ronald L. Rivest
 * (C) 1990 RSA Data Security, Inc.
 *
 * Portability nits fixed and reformatted - 2/12/91 Phil Karn
 */

/*
 * To use MD4:
 *   -- Include md4.h in your program
 *   -- Declare an MDstruct MD to hold the state of the digest computation.
 *   -- Initialize MD using MDbegin(&MD)
 *   -- For each full block (64 bytes) X you wish to process, call
 *          MDupdate(&MD,X,512)
 *      (512 is the number of bits in a full block.)
 *   -- For the last block (less than 64 bytes) you wish to process,
 *          MDupdate(&MD,X,n)
 *      where n is the number of bits in the partial block. A partial
 *      block terminates the computation, so every MD computation should
 *      terminate by processing a partial block, even if it has n = 0.
 *   -- The message digest is available in MD.buffer[0] ... MD.buffer[3].
 *      (Least-significant byte of each word should be output first.)
 *   -- You can print out the digest using MDprint(&MD)
 */

/* Implementation notes:
 * This implementation assumes that longs are 32-bit quantities.
 * If the machine stores the least-significant byte of an long in the
 * least-addressed byte (eg., VAX and 8086), then LOWBYTEFIRST should be
 * set to TRUE.  Otherwise (eg., SUNS), LOWBYTEFIRST should be set to
 * FALSE.  Note that on machines with LOWBYTEFIRST FALSE the routine
 * MDupdate modifies has a side-effect on its input array (the order of bytes
 * in each word are reversed).  If this is undesired a call to MDreverse(X) can
 * reverse the bytes of X back into order after each call to MDupdate.
 */
#define TRUE  1
#define FALSE 0

#ifdef VAXC           /* VAXC hates splitting lines in #if statements */
#define LOWBYTEFIRST TRUE
#else
#if (defined(__MSDOS__) || defined(MPU8086) || defined(MPU8080) \
 || defined(vax) || defined (MIPSEL))
#define LOWBYTEFIRST TRUE     /* Low order bytes are first in memory */
#else                 /* Almost all other machines are big-endian */
#define LOWBYTEFIRST FALSE
#endif
#endif


/* Compile-time includes */
#include <stdio.h>
#include "md4.h"

/* Compile-time declarations of MD4 ``magic constants'' */
#define I0  0x67452301       /* Initial values for MD buffer */
#define I1  0xefcdab89
#define I2  0x98badcfe
#define I3  0x10325476
#define C2  013240474631     /* round 2 constant = sqrt(2) in octal */
#define C3  015666365641     /* round 3 constant = sqrt(3) in octal */
/* C2 and C3 are from Knuth, The Art of Programming, Volume 2
 * (Seminumerical Algorithms), Second Edition (1981), Addison-Wesley.
 * Table 2, page 660.
 */
```

```
#define fs1  3              /* round 1 shift amounts */
#define fs2  7
#define fs3 11
#define fs4 19
#define gs1  3              /* round 2 shift amounts */
#define gs2  5
#define gs3  9
#define gs4 13
#define hs1  3              /* round 3 shift amounts */
#define hs2  9
#define hs3 11
#define hs4 15


/* Compile-time macro declarations for MD4.
 * Note: The ``rot'' operator uses the variable ``tmp''.
 * It assumes tmp is declared as unsigned long, so that the >>
 * operator will shift in zeros rather than extending the sign bit.
 */
#define f(X,Y,Z)             ((X&Y) | ((~X)&Z))
#define g(X,Y,Z)             ((X&Y) | (X&Z) | (Y&Z))
#define h(X,Y,Z)             (X^Y^Z)
#define rot(X,S)             (tmp=X,(tmp<<S) | (tmp>>(32-S)))
#define ff(A,B,C,D,i,s)      A = rot((A + f(B,C,D) + X[i]),s)
#define gg(A,B,C,D,i,s)      A = rot((A + g(B,C,D) + X[i] + C2),s)
#define hh(A,B,C,D,i,s)      A = rot((A + h(B,C,D) + X[i] + C3),s)

void MDreverse __ARGS((unsigned long *X));

/* MDprint(MDp)
 * Print message digest buffer MDp as 32 hexadecimal digits.
 * Order is from low-order byte of buffer[0] to high-order byte of buffer[3].
 * Each byte is printed with high-order hexadecimal digit first.
 * This is a user-callable routine.
 */
void
MDprint(MDp)
MDptr MDp;
{
        int i,j;

        for(i=0;i<4;i++)
               for(j=0;j<32;j=j+8)
                      printf("%02lx",(MDp->buffer[i]>>j) & 0xFF);
}

/* MDbegin(MDp)
 * Initialize message digest buffer MDp.
 * This is a user-callable routine.
 */
void
MDbegin(MDp)
MDptr MDp;
{
        int i;

        MDp->buffer[0] = I0;
        MDp->buffer[1] = I1;
        MDp->buffer[2] = I2;
        MDp->buffer[3] = I3;
        for(i=0;i<8;i++)
               MDp->count[i] = 0;
        MDp->done = 0;
}

/* MDreverse(X)
 * Reverse the byte-ordering of every long in X.
 * Assumes X is an array of 16 longs.
 * The macro revx reverses the byte-ordering of the next word of X.
```

```
 */
#define revx { t = (*X << 16) | (*X >> 16); \
               *X++ = ((t & 0xFF00FF00) >> 8) | ((t & 0x00FF00FF) << 8); }
void
MDreverse(X)
unsigned long *X;
{
        register unsigned long t;

        revx;
        revx;
        revx;
        revx;
        revx;
        revx;
        revx;
        revx;
        revx;
        revx;
        revx;
        revx;
        revx;
        revx;
        revx;
        revx;
}

/* MDblock(MDp,X)
 * Update message digest buffer MDp->buffer using 16-word data block X.
 * Assumes all 16 words of X are full of data.
 * Does not update MDp->count.
 * This routine is not user-callable.
 */
static void
MDblock(MDp,X)
MDptr MDp;
unsigned long *X;
{
        register unsigned long tmp, A, B, C, D;

#if LOWBYTEFIRST == FALSE
        MDreverse(X);
#endif
        A = MDp->buffer[0];
        B = MDp->buffer[1];
        C = MDp->buffer[2];
        D = MDp->buffer[3];
        /* Update the message digest buffer */
        ff(A,B,C,D,0,fs1); /* Round 1 */
        ff(D,A,B,C,1,fs2);
        ff(C,D,A,B,2,fs3);
        ff(B,C,D,A,3,fs4);
        ff(A,B,C,D,4,fs1);
        ff(D,A,B,C,5,fs2);
        ff(C,D,A,B,6,fs3);
        ff(B,C,D,A,7,fs4);
        ff(A,B,C,D,8,fs1);
        ff(D,A,B,C,9,fs2);
        ff(C,D,A,B,10,fs3);
        ff(B,C,D,A,11,fs4);
        ff(A,B,C,D,12,fs1);
        ff(D,A,B,C,13,fs2);
        ff(C,D,A,B,14,fs3);
        ff(B,C,D,A,15,fs4);
        gg(A,B,C,D,0,gs1); /* Round 2 */
        gg(D,A,B,C,4,gs2);
        gg(C,D,A,B,8,gs3);
        gg(B,C,D,A,12,gs4);
        gg(A,B,C,D,1,gs1);
```

```
        gg(D,A,B,C,5,gs2);
        gg(C,D,A,B,9,gs3);
        gg(B,C,D,A,13,gs4);
        gg(A,B,C,D,2,gs1);
        gg(D,A,B,C,6,gs2);
        gg(C,D,A,B,10,gs3);
        gg(B,C,D,A,14,gs4);
        gg(A,B,C,D,3,gs1);
        gg(D,A,B,C,7,gs2);
        gg(C,D,A,B,11,gs3);
        gg(B,C,D,A,15,gs4);
        hh(A,B,C,D,0,hs1); /* Round 3 */
        hh(D,A,B,C,8,hs2);
        hh(C,D,A,B,4,hs3);
        hh(B,C,D,A,12,hs4);
        hh(A,B,C,D,2,hs1);
        hh(D,A,B,C,10,hs2);
        hh(C,D,A,B,6,hs3);
        hh(B,C,D,A,14,hs4);
        hh(A,B,C,D,1,hs1);
        hh(D,A,B,C,9,hs2);
        hh(C,D,A,B,5,hs3);
        hh(B,C,D,A,13,hs4);
        hh(A,B,C,D,3,hs1);
        hh(D,A,B,C,11,hs2);
        hh(C,D,A,B,7,hs3);
        hh(B,C,D,A,15,hs4);
        MDp->buffer[0] += A;
        MDp->buffer[1] += B;
        MDp->buffer[2] += C;
        MDp->buffer[3] += D;
}

/* MDupdate(MDp,X,count)
 * Input: MDp -- an MDptr
 *        X -- a pointer to an array of unsigned characters.
 *        count -- the number of bits of X to use.
 *                  (if not a multiple of 8, uses high bits of last byte.)
 * Update MDp using the number of bits of X given by count.
 * This is the basic input routine for an MD4 user.
 * The routine completes the MD computation when count < 512, so
 * every MD computation should end with one call to MDupdate with a
 * count less than 512.  A call with count 0 will be ignored if the
 * MD has already been terminated (done != 0), so an extra call with count
 * 0 can be given as a ``courtesy close'' to force termination if desired.
 */
void
MDupdate(MDp,X,count)
MDptr MDp;
unsigned char *X;
unsigned int count;
{
        int i,bit,byte,mask;
        unsigned long tmp;
        unsigned char XX[64];
        unsigned char *p;

        /* return with no error if this is a courtesy close with count
         * zero and MDp->done is true.
         */
        if(count == 0 && MDp->done)
               return;
        /* check to see if MD is already done and report error */
        if(MDp->done){
               printf("\nError: MDupdate MD already done.");
               return;
        }
        /* Add count to MDp->count */
        tmp = count;
```

```
        p = MDp->count;
        while(tmp){
               tmp += *p;
               *p++ = tmp;
               tmp = tmp >> 8;
        }
        /* Process data */
        if(count == 512){
               /* Full block of data to handle */
               MDblock(MDp,(unsigned long *)X);
        } else if(count > 512){
               /* Check for count too large */
               printf("\nError: MDupdate called with illegal count value %ld.",count);
               return;
        } else {
               /* partial block -- must be last block so finish up
                * Find out how many bytes and residual bits there are
                */
               byte = count >> 3;
               bit =  count & 7;
               /* Copy X into XX since we need to modify it */
               for(i=0;i<=byte;i++)
                      XX[i] = X[i];
               for(i=byte+1;i<64;i++)
                      XX[i] = 0;
               /* Add padding '1' bit and low-order zeros in last byte */
               mask = 1 << (7 - bit);
               XX[byte] = (XX[byte] | mask) & ~( mask - 1);
               /* If room for bit count, finish up with this block */
               if(byte <= 55){
                      for(i=0;i<8;i++)
                             XX[56+i] = MDp->count[i];
                      MDblock(MDp,(unsigned long *)XX);
               } else {
                      /* need to do two blocks to finish up */
                      MDblock(MDp,(unsigned long *)XX);
                      for(i=0;i<56;i++)
                             XX[i] = 0;
                      for(i=0;i<8;i++)
                             XX[56+i] = MDp->count[i];
                      MDblock(MDp,(unsigned long *)XX);
               }
        /* Set flag saying we're done with MD computation */
        MDp->done = 1;
        }
}
/* End of md4.c */
```

A.2.4. put.c

```
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include <ctype.h>
#include "skey.h"

static unsigned long extract __ARGS((char *s,int start,int length));
static void standard __ARGS((char *word));
static void insert __ARGS((char *s, int x, int start, int length));
static int wsrch __ARGS((char *w,int low,int high));

/* Dictionary for integer-word translations */
char Wp[2048][4] = {
"A",
"ABE",
"ACE",
"ACT",
"AD",
"ADA",
"ADD",
"AGO",
"AID",
"AIM",
"AIR",
"ALL",
"ALP",
"AM",
"AMY",
```

... Many words deleted here for brevity. Consult the original source code for the entire contents ...

```
"YALE",
"YANG",
"YANK",
"YARD",
"YARN",
"YAWL",
"YAWN",
"YEAH",
"YEAR",
"YELL",
"YOGA",
"YOKE"
};

/* Encode 8 bytes in 'c' as a string of English words.
 * Returns a pointer to a static buffer
 */
char *
btoe(engout,c)
char *c, *engout;
{
        char cp[9];    /* add in room for the parity 2 bits*/
        int p,i ;
        char   seperator[ 2];

        engout[0] = '\0';
        memcpy(cp, c,8);
        /* compute parity */
        for(p = 0,i = 0; i < 64;i += 2)
```

```
                p += extract(cp,i,2);

#ifdef VAXC
        strncpy( seperator, ".", 2);
#else
        strncpy( seperator, " ", 2);
#endif
        cp[8] = (char)p << 6;
        strncat(engout,&Wp[extract(cp, 0,11)][0],4);
        strcat(engout, seperator);
        strncat(engout,&Wp[extract(cp,11,11)][0],4);
        strcat(engout, seperator);
        strncat(engout,&Wp[extract(cp,22,11)][0],4);
        strcat(engout, seperator);
        strncat(engout,&Wp[extract(cp,33,11)][0],4);
        strcat(engout, seperator);
        strncat(engout,&Wp[extract(cp,44,11)][0],4);
        strcat(engout, seperator);
        strncat(engout,&Wp[extract(cp,55,11)][0],4);
#ifdef  notdef
        printf("engout is %s\n\r",engout);
#endif
        return(engout);
}

/* convert English to binary
 * returns 1 OK - all good words and parity is OK
 *         0 word not in data base
 *        -1 badly formed in put ie > 4 char word
 *        -2 words OK but parity is wrong
 */
int
etob(out, e)
char *out;
char *e;
{
        char *word;
        int i, p, v,l, low,high;
        char b[9];
        char input[36];

        if(e == NULL)
                return -1;

        strncpy(input,e,sizeof(input));
        memset(b, 0, sizeof(b));
        memset(out, 0, 8);
        for(i=0,p=0;i<6;i++,p+=11){
                if((word = strtok(i == 0 ? input : NULL," ")) == NULL)
                        return -1;
                l = strlen(word);
                if(l > 4 || l < 1){
                        return -1;
                } else if(l < 4){
                        low = 0;
                        high = 570;
                } else {
                        low = 571;
                        high = 2047;
                }
                standard(word);
                if( (v = wsrch(word,low,high)) < 0 )
                        return 0;
                insert(b,v,p,11);
        }

        /* now check the parity of what we got */
        for(p = 0, i = 0; i < 64; i +=2)
                p += extract(b, i, 2);
```

```
        if( (p & 3) != extract(b, 64,2) )
                return -2;

        memcpy(out,b,8);

        return 1;
}
/* Display 8 bytes as a series of 16-bit hex digits */
char *
put8(out,s)
char *out;
char *s;
{
        sprintf(out,"%02X%02X %02X%02X %02X%02X %02X%02X",
                s[0] & 0xff,s[1] & 0xff,s[2] & 0xff,
                s[3] & 0xff,s[4] & 0xff,s[5] & 0xff,
                s[6] & 0xff,s[7] & 0xff);
        return out;
}
#ifdef  notdef
/* Encode 8 bytes in 'cp' as stream of ascii letters.
 * Provided as a possible alternative to btoe()
 */
char *
btoc(cp)
char *cp;
{
        int i;
        static char out[31];

        /* code out put by characters 6 bits each added to 0x21 (!)*/
        for(i=0;i <= 10;i++){
                /* last one is only 4 bits not 6*/
                out[i] = '!'+ extract(cp,6*i,i >= 10 ? 4:6);
        }
        out[i] = '\0';
        return(out);
}
#endif

/* Internal subroutines for word encoding/decoding */

/* Dictionary binary search */
static int
wsrch(w,low,high)
char *w;
int low, high;
{
        int i,j;

        for(;;){
                i = (low + high)/2;
                if((j = strncmp(w,Wp[i],4)) == 0)
                        return i;       /* Found it */
                if(high == low+1){
                        /* Avoid effects of integer truncation in /2 */
                        if(strncmp(w,Wp[high],4) == 0)
                                return high;
                        else
                                return -1;
                }
                if(low >= high)
                        return -1;      /* I don't *think* this can happen...*/
                if(j < 0)
                        high = i;       /* Search lower half */
                else
                        low = i;        /* Search upper half */
        }
```

```
}
static void
insert(s, x, start, length)
char *s;
int x;
int  start, length;
{
        unsigned char cl;
        unsigned char cc;
        unsigned char cr;
        unsigned long y;
        int shift;

        assert(length <= 11);
        assert(start >= 0);
        assert(length >= 0);
        assert(start +length <= 66);

        shift = ((8  -(( start + length) % 8))%8);
        y = (long) x << shift;
        cl = (y >> 16) & 0xff;
        cc = (y >> 8) & 0xff;
        cr = y & 0xff;
        if(shift + length > 16){
               s[start /8] |= cl;
               s[start/8 +1] |= cc;
               s[start/8 +2] |= cr;
        } else if(shift +length > 8){
               s[start/8] |= cc;
               s[start/8 + 1] |= cr;
        } else {
               s[start/8] |= cr;
        }
}

static void
standard(word)
register char *word;
{
        while(*word){
               if(!isascii(*word))
                      break;
               if(islower(*word))
                      *word = toupper(*word);
               if(*word == '1')
                      *word = 'L';
               if(*word == '0')
                      *word = 'O';
               if(*word == '5')
                      *word = 'S';
               word++;
        }
}

/* Extract 'length' bits from the char array 's' starting with bit 'start' */
static unsigned long
extract(s, start, length)
char *s;
int start, length;
{
        unsigned char cl;
        unsigned char cc;
        unsigned char cr;
        unsigned long x;

        assert(length <= 11);
        assert(start >= 0);
        assert(length >= 0);
        assert(start +length <= 66);
```

```
        cl = s[start/8];
        cc = s[start/8 +1];
        cr = s[start/8 +2];
        x = ((long)(cl<<8 | cc) <<8  | cr) ;
        x = x >> (24 - (length + (start %8)));
        x =( x & (0xffff >> (16-length) )   );
        return(x);
}
```

A.2.5. skey.c

```
/* Stand-alone program for computing responses to S/Key challenges.
 * Takes the iteration count and seed as command line args, prompts
 * for the user's key, and produces both word and hex format responses.
 *
 * Usage example:
 *      >skey 88 ka9q2
 *      Enter password:
 *      OMEN US HORN OMIT BACK AHOY
 *      C848 666B 6435 0A93
 *      >
 */

#ifdef VAXC
#ifndef HASSTDLIB
#define HASSTDLIB
#endif
#endif

#include <stdio.h>
#ifdef HASSTDLIB
#include <stdlib.h>
#else
#include <sys/types.h>
#endif
#include <string.h>
#ifdef  __MSDOS__
#include <dos.h>
#else   /* Assume BSD unix */
#ifdef VAXC
#include <ssdef.h>
#else
#include <fcntl.h>
#include <sgtty.h>
#endif
#endif
#include "md4.h"
#include "skey.h"

char *readpass();
void usage();
#ifdef VAXC
int     optind = 1;
char    *optarg;
#else
int getopt();
extern int optind;
extern char *optarg;
#endif

#ifdef VAXC
int getopt( argc, argv, optstring)
int     argc;
char    *argv[];
char    *optstring;
{
    int     i;

    for ( i = optind; i < argc; i++)
    {
        if ( strncmp( argv[ i], "-n", 2) == 0)
        {
            optind = i + 2;
            optarg = argv[ i + 1];
            return( argv[ i][ 1]);
        }
    }
```

```
    return( -1);
}
#endif

int
main(argc,argv)
int argc;
char *argv[];
{
        int n,cnt,i;
        char passwd[256],passwd2[256];
        char key[8];
        char *seed;
        char buf[30];
        char *slash;

        cnt = 1;
        while((i = getopt(argc,argv,"n:")) != EOF){
                switch(i){
                case 'n':
                        cnt = atoi(optarg);
                        break;
                }
        }
        /* could be in the form <number>/<seed> */
        if(argc <= optind + 1){
                /*look for / in it */
                if(argc <= optind){
                        usage(argv[0]);
                        return 1;
                }

                slash = strchr(argv[optind], '/');
                if(slash == NULL){
                        usage(argv[0]);
                        return 1;
                }
                *slash++ = '\0';
                seed = slash;

                if((n = atoi(argv[optind])) < 0){
                        fprintf(stderr,"%s not positive\n",argv[optind]);
                        usage(argv[0]);
                        return 1;
                }
        }
        else {

                if((n = atoi(argv[optind])) < 0){
                        fprintf(stderr,"%s not positive\n",argv[optind]);
                        usage(argv[0]);
                        return 1;
                }
                seed = argv[++optind];
        }
        fprintf(stderr,"Reminder - Do not use key while logged in via telnet or dial-
in.\n");

        /* Get user's secret password */
        for(;;){
                fprintf(stderr,"Enter secret password: ");
                readpass(passwd,sizeof(passwd));
#ifdef VAXC
                fprintf( stderr, "\n");
#endif
                break;
        /************
                fprintf(stderr,"Again secret password: ");
                readpass(passwd2,sizeof(passwd));
```

```
                if(strcmp(passwd,passwd2) == 0) break;
                fprintf(stderr, "Sorry no match\n");
        **************/

        }

        /* Crunch seed and password into starting key */
        if(keycrunch(key,seed,passwd) != 0){
                fprintf(stderr,"%s: key crunch failed\n",argv[0]);
                return 1;
        }
        if(cnt == 1){
                while(n-- != 0)
                        f(key);
                printf("%s\n",btoe(buf,key));
#ifdef  HEXIN
                printf("%s\n",put8(buf,key));
#endif
        } else {
                for(i=0;i<=n-cnt;i++)
                        f(key);
                for(;i<=n;i++){
#ifdef  HEXIN
                        printf("%d: %-29s  %s\n",i,btoe(buf,key),put8(buf,key));
#else
                        printf("%d: %-29s\n",i,btoe(buf,key));
#endif
                        f(key);
                }
        }
#ifdef VAXC
        return( SS$_NORMAL);
#else
        return 0;
#endif
}
void
usage(s)
char *s;
{
        fprintf(stderr,"Usage: %s [-n count] <sequence #>[/] <key> \n",s);
}
```

A.2.6. skeyinit.c

```
/*   change password or add user to S/KEY authentication system.
 *   S/KEY is a tradmark of Bellcore  */

#include <stdio.h>
#include <string.h>
#ifdef VAXC
#include <ctype.h>
#include <ssdef.h>
#include <jpidef.h>
#include <descrip.h>
#include <prvdef.h>
#else
#include <pwd.h>
#endif
#include "skey.h"
#include <stdio.h>
#include <time.h>

extern int optind;
extern char *optarg;

char * readpass();

int skeylookup __ARGS((struct skey *mp,char *name));

#define NAMELEN 2

#ifdef VAXC
void uppercase( string)    /* uppercase a string in-situ */
char    *string;
{
    char    *cp;

    cp = string;
    while ( *cp != '\0')
    {
        *cp = toupper( *cp);
        cp++;
    }
}
#endif

int
main(argc,argv)
int argc;
char *argv[];
{
        struct skey skey;
        int rval,n,nn,i,defaultsetup;
        char seed[18],tmp[80],key[8];
        char defaultseed[17], passwd[256],passwd2[256] ;
#ifdef VAXC
        char   username[ 255];
        struct passwd
        {
            char    *pw_name;
        } vms_username = { username};
        struct passwd  *pp = &vms_username;
        int                 ret_length;
        struct
        {
            unsigned short    len;
            unsigned short    item;
            char              *buffer;
            int                       *ret_len;
            int                       terminator;
```

```
        } item_list = { sizeof( username), JPI$_USERNAME, username, &ret_length, 0};
        unsigned int        status;
        struct
        {
            unsigned int      status;
            unsigned short    ret_length;
            unsigned short    padding;
        } iosb;
        struct dsc$descriptor_s       name_desc =
            { 0, DSC$K_DTYPE_T, DSC$K_CLASS_S, username};
        unsigned long         auth_privs;
        char                  *cp;
#else
        struct passwd *ppuser,*pp;
#endif


        time_t now;
        struct tm *tm;
        char tbuf[27],buf[60];
        char lastc, me[80];
        int l;

        time(&now);
        tm = localtime(&now);
        strftime(tbuf, sizeof(tbuf), "%M%j", tm);
        gethostname(defaultseed,NAMELEN);
        strcpy(&defaultseed[NAMELEN],tbuf);

#ifdef VAXC
        status = sys$getjpi( 0, 0, 0, &item_list, &iosb, 0, 0);
        if ( status != SS$_NORMAL)
        {
            exit( status);
        }
        if ( iosb.status != SS$_NORMAL)
        {
            exit( iosb.status);
        }
        cp = strchr( username, ' ');
        if ( cp != NULL)
        {
            *cp = '\0';
        }
#else
        pp = ppuser = getpwuid(getuid());
#endif
        strcpy(me,pp->pw_name);
        defaultsetup = 1;
        if( argc > 1){
               if(strcmp("-s", argv[1]) == 0)
                       defaultsetup = 0;
               else
#ifdef VAXC
                       strcpy( pp->pw_name, argv[ 1]);
#else
                       pp = getpwnam(argv[1]);
#endif
               if(argc > 2)
#ifdef VAXC
                       strcpy( pp->pw_name, argv[ 2]);
#else
                       pp = getpwnam(argv[2]);
#endif

        }
#ifdef VAXC
        uppercase( username);
        name_desc.dsc$w_length = strlen( username);
```

```
        status = sys$asctoid( &name_desc, 0, 0);
        if ( status != SS$_NORMAL)
        {
            printf( "User unknown\n");
            return( 1);
        }
#else
        if(pp == NULL){
                printf("User unknown\n");
                return 1;
        }
#endif
        if(strcmp( pp->pw_name,me) != 0){
#ifdef VAXC
                item_list.buffer = ( char *) &auth_privs;
                item_list.len = sizeof( auth_privs);
                item_list.item = JPI$_AUTHPRIV;
                status = sys$getjpi( 0, 0, 0, &item_list, &iosb, 0, 0);
                if ( status != SS$_NORMAL)
                {
                    printf("Permission denied.\n");
                    return(1);
                }
                if ( ( auth_privs & ( PRV$M_SETPRV | PRV$M_SYSPRV | PRV$M_BYPASS)) == 0)
                {
                    printf("Permission denied.\n");
                    return(1);
                }
#else
                if(getuid() != 0){
                        /* Only root can change other's passwds */
                        printf("Permission denied.\n");
                        return(1);
                }
#endif
        }



        rval = skeylookup(&skey,pp->pw_name);
        switch(rval){
        case -1:
                perror("error in opening database");
                return 1;
        case 0:
                printf("Updating %s:\n",pp->pw_name);
                printf("Old key: %s\n",skey.seed);
                /* lets be nice if they have a skey.seed that ends in 0-8 just add one*/
                l = strlen(skey.seed);
                if( l > 0){
                        lastc = skey.seed[l-1];
                        if( isdigit(lastc) && lastc != '9' ){
                                strcpy(defaultseed, skey.seed);
                                defaultseed[l-1] = lastc + 1;
                        }
                        if( isdigit(lastc) && lastc == '9' && l < 16){
                                strcpy(defaultseed, skey.seed);
                                defaultseed[l-1] = '0';
                                defaultseed[l] = '0';
                                defaultseed[l+1] = '\0';
                        }
                }
                break;
        case 1:
                printf("Adding %s:\n",pp->pw_name);
                break;
        }
    n = 99;
    if( ! defaultsetup){
```

```
        printf("Reminder you need the 6 english words from the skey command.\n");
        for(i=0;;i++){
                if(i >= 2) exit(1);
                printf("Enter sequence count from 1 to 10000: ");
                fgets(tmp,sizeof(tmp),stdin);
                n = atoi(tmp);
                if(n > 0 && n < 10000)
                        break;  /* Valid range */
                printf("Count must be > 0 and < 10000\n");
        }
    }
    if( !defaultsetup){
        printf("Enter new key [default %s]: ", defaultseed);
        fflush(stdout);
        fgets(seed,sizeof(seed),stdin);
        rip(seed);
        if(strlen(seed) > 16){
                printf("Seed truncated to 16 chars\n");
                seed[16] = '\0';
        }
        if( seed[0] == '\0') strcpy(seed,defaultseed);
        for(i=0;;i++){
                if(i >= 2) exit(1);
                printf("s/key %d %s\ns/key access password: ",n,seed);
                fgets(tmp,sizeof(tmp),stdin);
                rip(tmp);
                backspace(tmp);
                if(tmp[0] == '?'){
                        printf("Enter 6 English words from secure S/Key calculation.\n");
                        continue;
                }
                if(tmp[0] == '\0'){
                        exit(1);
                }
#ifdef VAXC
                cp = tmp;
                while ( *cp != '\0')
                {
                    if ( *cp == '.')
                    {
                        *cp = ' ';
                    }
                    cp++;
                }
#endif
                if(etob(key,tmp) == 1 || atob8(key,tmp) == 0)
                        break;  /* Valid format */
                printf("Invalid format, try again with 6 English words.\n");
        }
    } else {
        /* Get user's secret password */
        fprintf(stderr,"Reminder - Only use this method if you are direct connected.\n");
        fprintf(stderr,"If you are using telnet or dial-in exit with no password and use
keyinit -s.\n");
        for(i=0;;i++){
                if(i >= 2) exit(1);
                fprintf(stderr,"Enter secret password: ");
                readpass(passwd,sizeof(passwd));
#ifdef VAXC
                fprintf( stderr, "\n");
#endif
                if(passwd[0] == '\0'){
                        exit(1);
                }
                fprintf(stderr,"Again secret password: ");
                readpass(passwd2,sizeof(passwd));
#ifdef VAXC
                fprintf( stderr, "\n");
#endif
```

```
                if(passwd2[0] == '\0'){
                        exit(1);
                }
                if(strlen(passwd) < 4 && strlen(passwd2) < 4) {
                        fprintf(stderr, "Sorry your password must be longer\n\r");
                        exit(1);
                }
                if(strcmp(passwd,passwd2) == 0) break;
                fprintf(stderr, "Sorry no match\n");


        }
        strcpy(seed,defaultseed);

        /* Crunch seed and password into starting key */
        if(keycrunch(key,seed,passwd) != 0){
                fprintf(stderr,"%s: key crunch failed\n",argv[0]);
                return 1;
        }
        nn = n;
        while(nn-- != 0)
                f(key);
    }
        time(&now);
        tm = localtime(&now);
        strftime(tbuf, sizeof(tbuf), " %b %d,%Y %T", tm);
         if (skey.val == NULL)
                    skey.val = (char *) malloc(16+1);


        btoa8(skey.val,key);
        fprintf(skey.keyfile,"%s %04d %-16s %s %-21s\n",pp->pw_name,n,
                seed,skey.val, tbuf);
        fclose(skey.keyfile);
        printf("\nID %s s/key is %d %s\n",pp->pw_name,n,seed);
        printf("%s\n",btoe(buf,key));
#ifdef HEXIN
        printf("%s\n",put8(buf,key));
#endif

#ifdef VAXC
        return( SS$_NORMAL);
#else
        return 0;
#endif
}
```

A.2.7. skeylogin.c

```
/*   Login code for S/KEY Authentication.  S/KEY is a trademark
 *   of Bellcore.
 *
 *   Mink is the former name of the S/KEY authentication system.
 *   Many references for mink  may still be found in this program.   */
#ifdef VAXC
#include <timeb.h>
#else
#include <sys/param.h>
#ifdef  QUOTA
#include <sys/quota.h>
#endif
#include <sys/stat.h>
#include <sys/time.h>
#include <sys/timeb.h>
#include <sys/resource.h>
#endif

#include <stdio.h>
#include <string.h>
#ifdef VAXC
#include <types.h>
#include <stat.h>
#else
#include <sys/types.h>
#include <sys/stat.h>
#endif
#include <time.h>
#include <errno.h>
#include "skey.h"

#ifdef VAXC
#define KEYFILE        "sys$manager:skeykeys.dat"
struct timeval
{
    long    tv_sec;
    long    tv_usec;
};

#else
#define KEYFILE        "/etc/skeykeys"
#endif

char *skipspace();
int skeylookup __ARGS((struct skey *mp,char *name));


/* Issue a skey challenge for user 'name'. If successful,
 * fill in the caller's skey structure and return 0. If unsuccessful
 * (e.g., if name is unknown) return -1.
 *
 * The file read/write pointer is left at the start of the
 * record.
 */
int
getskeyprompt(mp,name,prompt)
struct skey *mp;
char *name;
char *prompt;
{
        int rval;

        sevenbit(name);
        rval = skeylookup(mp,name);
        strcpy(prompt,"s/key 55 latour1\n");
        switch(rval){
```

```
        case -1:        /* File error */
                return -1;
        case 0:                 /* Lookup succeeded, return challenge */
                sprintf(prompt,"s/key %d %s\n",mp->n - 1,mp->seed);
                return 0;
        case 1:                 /* User not found */
                fclose(mp->keyfile);
                return -1;
        }
        return -1;      /* Can't happen */
}
/* Return  a skey challenge string for user 'name'. If successful,
 * fill in the caller's skey structure and return 0. If unsuccessful
 * (e.g., if name is unknown) return -1.
 *
 * The file read/write pointer is left at the start of the
 * record.
 */
int
skeychallenge(mp,name, ss)
struct skey *mp;
char *name;
char *ss;
{
        int rval;

        rval = skeylookup(mp,name);
        switch(rval){
        case -1:        /* File error */
                return -1;
        case 0:                 /* Lookup succeeded, issue challenge */
                 sprintf(ss, "s/key %d %s",mp->n - 1,mp->seed);
                return 0;
        case 1:                 /* User not found */
                fclose(mp->keyfile);
                return -1;
        }
        return -1;      /* Can't happen */
}

/* Find an entry in the One-time Password database.
 * Return codes:
 * -1: error in opening database
 *  0: entry found, file R/W pointer positioned at beginning of record
 *  1: entry not found, file R/W pointer positioned at EOF
 */
int
skeylookup(mp,name)
struct skey *mp;
char *name;
{
        int found;
        int len;
        long recstart;
        char *cp;
        struct stat statbuf;

        /* See if the KEYFILE exists, and create it if not */
        if(stat(KEYFILE,&statbuf) == -1 && errno == ENOENT){
                mp->keyfile = fopen(KEYFILE,"w+");
        } else {
                /* Otherwise open normally for update */
                mp->keyfile = fopen(KEYFILE,"r+");
        }
        if(mp->keyfile == NULL)
                return -1;

        /* Look up user name in database */
        len = strlen(name);
```

```
        if( len > 8 ) len = 8;         /*  Added 8/2/91  -  nmh */
        found = 0;
        while(!feof(mp->keyfile)){
               recstart = ftell(mp->keyfile);
               mp->recstart = recstart;
               if(fgets(mp->buf,sizeof(mp->buf),mp->keyfile) != mp->buf){
                      break;
               }
               rip(mp->buf);
               if(mp->buf[0] == '#')
                      continue;       /* Comment */
               if((mp->logname = strtok(mp->buf," \t")) == NULL)
                      continue;
               if((cp = strtok(NULL," \t")) == NULL)
                      continue;
               mp->n = atoi(cp);
               if((mp->seed = strtok(NULL," \t")) == NULL)
                      continue;
               if((mp->val = strtok(NULL," \t")) == NULL)
                      continue;
               if(strlen(mp->logname) == len
                && strncmp(mp->logname,name,len) == 0){
                      found = 1;
                      break;
               }
        }
        if(found){
               fseek(mp->keyfile,recstart,0);
               return 0;
        } else
               return 1;
}
/* Verify response to a s/key challenge.
 *
 * Return codes:
 * -1: Error of some sort; database unchanged
 *  0:  Verify successful, database updated
 *  1:  Verify failed, database unchanged
 *
 * The database file is always closed by this call.
 */
int
skeyverify(mp,response)
struct skey *mp;
char *response;
{
 struct timeval startval;
 struct timeval endval;
long microsec;
        char key[8];
        char fkey[8];
        char filekey[8];
        time_t now;
        struct tm *tm;
        char tbuf[27],buf[60];
        char me[80];
        int rval;
        char *cp;

        time(&now);
        tm = localtime(&now);
        strftime(tbuf, sizeof(tbuf), " %b %d,%Y %T", tm);

        if(response == NULL){
               fclose(mp->keyfile);
               return -1;
        }
        rip(response);
```

```
        /* Convert response to binary */
        if(etob(key,response) != 1 && atob8(key,response) != 0){
               /* Neither english words or ascii hex */
               fclose(mp->keyfile);
               return -1;
        }

        /* Compute fkey = f(key) */
        memcpy(fkey,key,sizeof(key));
        f(fkey);
        /* in order to make the window of update as short as possible
           we must do the comparison here and if OK write it back
           other wise the same password can be used twice to get in
          to the system
        */
#ifndef VAXC
        setpriority(PRIO_PROCESS, 0, -4);
#endif
/*
  gettimeofday(&startval, (char *)0 );
*/

        /* reread the file record NOW*/

        fseek(mp->keyfile,mp->recstart,0);
        if(fgets(mp->buf,sizeof(mp->buf),mp->keyfile) != mp->buf){
#ifndef VAXC
               setpriority(PRIO_PROCESS, 0, 0);
#endif
               fclose(mp->keyfile);
               return -1;
        }
        rip(mp->buf);
        mp->logname = strtok(mp->buf," \t");
        cp = strtok(NULL," \t") ;
        mp->seed = strtok(NULL," \t");
        mp->val = strtok(NULL," \t");
        /* And convert file value to hex for comparison */
        atob8(filekey,mp->val);

        /* Do actual comparison */
        if(memcmp(filekey,fkey,8) != 0){
               /* Wrong response */
#ifndef VAXC
               setpriority(PRIO_PROCESS, 0, 0);
#endif
               fclose(mp->keyfile);
               return 1;
        }

        /* Update key in database by overwriting entire record. Note
         * that we must write exactly the same number of bytes as in
         * the original record (note fixed width field for N)
         */
        btoa8(mp->val,key);
        mp->n--;
        fseek(mp->keyfile,mp->recstart,0);
        fprintf(mp->keyfile,"%s %04d %-16s %s %-21s\n",mp->logname,mp->n,mp->seed,
         mp->val, tbuf);
/*
gettimeofday(&endval, (char *)0 );
 microsec = (endval.tv_sec - startval.tv_sec) * 1000000 + (endval.tv_usec -
startval.tv_usec);
fprintf(stderr, "window= %d micro seconds \n"  , microsec);
*/


        fclose(mp->keyfile);
```

```
#ifndef VAXC
        setpriority(PRIO_PROCESS, 0, 0);
#endif
        return 0;
}


/* Convert 8-byte hex-ascii string to binary array
 * Returns 0 on success, -1 on error
 */
atob8(out,in)
register char *out,*in;
{
        register int i;
        register int val;

        if(in == NULL || out == NULL)
               return -1;

        for(i=0;i<8;i++){
               if((in = skipspace(in)) == NULL)
                      return -1;
               if((val = htoi(*in++)) == -1)
                      return -1;
               *out = val << 4;

               if((in = skipspace(in)) == NULL)
                      return -1;
               if((val = htoi(*in++)) == -1)
                      return -1;
               *out++ |= val;
        }
        return 0;
}

char *
skipspace(cp)
register char *cp;
{
        while(*cp == ' ' || *cp == '\t')
               cp++;

        if(*cp == '\0')
               return NULL;
        else
               return cp;
}

/* Convert 8-byte binary array to hex-ascii string */
int
btoa8(out,in)
register char *out,*in;
{
        register int i;

        if(in == NULL || out == NULL)
               return -1;

        for(i=0;i<8;i++){
               sprintf(out,"%02x",*in++ & 0xff);
               out += 2;
        }
        return 0;
}


/* Convert hex digit to binary integer */
int
htoi(c)
```

```
register char c;
{
        if('0' <= c && c <= '9')
                return c - '0';
        if('a' <= c && c <= 'f')
                return 10 + c - 'a';
        if('A' <= c && c <= 'F')
                return 10 + c - 'A';
        return -1;
}
```

A.2.8. skeysubr.c

```
#include <stdio.h>
#ifdef VAXC
#ifndef HASSTDLIB
#define HASSTDLIB
#endif
#include <descrip.h>
#include <ssdef.h>
#include <iodef.h>
#endif

#ifdef HASSTDLIB
#include <stdlib.h>
#else
#include <sys/types.h>
#endif
#include <string.h>
#ifdef  __MSDOS__
#include <dos.h>
#else   /* Assume BSD unix */
#ifndef VAXC
#include <fcntl.h>
#include <sgtty.h>
#endif
#endif
#include "md4.h"
#include "skey.h"

#ifdef VAXC        /* VAXC hates splitting the #if line */
#define LITTLE_ENDIAN
#else
#if (defined(__MSDOS__) || defined(MPU8086) || defined(MPU8080) \
 || defined(vax) || defined (MIPSEL))
#define LITTLE_ENDIAN  /* Low order bytes are first in memory */
#endif                 /* Almost all other machines are big-endian */
#endif

/* Crunch a key:
 * concatenate the seed and the password, run through MD4 and
 * collapse to 64 bits. This is defined as the user's starting key.
 */
int
keycrunch(result,seed,passwd)
char *result;  /* 8-byte result */
char *seed;    /* Seed, any length */
char *passwd;  /* Password, any length */
{
        char *buf;
        MDstruct md;
        unsigned int buflen;
#ifndef LITTLE_ENDIAN
        int i;
        register long tmp;
#endif

        buflen = strlen(seed) + strlen(passwd);
        if((buf = malloc(buflen+1)) == NULL)
               return -1;
        strcpy(buf,seed);
        strcat(buf,passwd);

        /* Crunch the key through MD4 */
        sevenbit(buf);
        MDbegin(&md);
        MDupdate(&md,(unsigned char *)buf,8*buflen);

        free(buf);
```

```
        /* Fold result from 128 to 64 bits */
        md.buffer[0] ^= md.buffer[2];
        md.buffer[1] ^= md.buffer[3];

#ifdef  LITTLE_ENDIAN
        /* Only works on byte-addressed little-endian machines!! */
        memcpy(result,(char *)md.buffer,8);
#else
        /* Default (but slow) code that will convert to
         * little-endian byte ordering on any machine
         */
        for(i=0;i<2;i++){
               tmp = md.buffer[i];
               *result++ = tmp;
               tmp >>= 8;
               *result++ = tmp;
               tmp >>= 8;
               *result++ = tmp;
               tmp >>= 8;
               *result++ = tmp;
        }
#endif

        return 0;
}

/* The one-way function f(). Takes 8 bytes and returns 8 bytes in place */
void
f(x)
char *x;
{
        MDstruct md;
#ifndef LITTLE_ENDIAN
        register long tmp;
#endif

        MDbegin(&md);
        MDupdate(&md,(unsigned char *)x,64);

        /* Fold 128 to 64 bits */
        md.buffer[0] ^= md.buffer[2];
        md.buffer[1] ^= md.buffer[3];

#ifdef  LITTLE_ENDIAN
        /* Only works on byte-addressed little-endian machines!! */
        memcpy(x,(char *)md.buffer,8);

#else
        /* Default (but slow) code that will convert to
         * little-endian byte ordering on any machine
         */
        tmp = md.buffer[0];
        *x++ = tmp;
        tmp >>= 8;
        *x++ = tmp;
        tmp >>= 8;
        *x++ = tmp;
        tmp >>= 8;
        *x++ = tmp;

        tmp = md.buffer[1];
        *x++ = tmp;
        tmp >>= 8;
        *x++ = tmp;
        tmp >>= 8;
        *x++ = tmp;
        tmp >>= 8;
        *x = tmp;
```

```
#endif
}

/* Strip trailing cr/lf from a line of text */
void
rip(buf)
char *buf;
{
        char *cp;

        if((cp = strchr(buf,'\r')) != NULL)
               *cp = '\0';

        if((cp = strchr(buf,'\n')) != NULL)
               *cp = '\0';
}
/************************/
#ifdef  __MSDOS__
char *
readpass(buf,n)
char *buf;
int n;
{
        int i;
        char *cp;

        for(cp=buf,i = 0; i < n ; i++)
               if ((*cp++ = bdos(7,0,0)) == '\r')
                      break;
        *cp = '\0';
        printf("\n");
        rip(buf);
        return buf;
}
#else
#ifdef VAXC

char *readpass( buf, n)
char    *buf;
int     n;
{
        $DESCRIPTOR( terminal, "TT");
        unsigned int   status;
        struct
        {
            unsigned int    status;
            unsigned short  ret_len;
            unsigned short  filler;
        } iosb;
        unsigned short     channel;

        status = sys$assign( &terminal, &channel, 0, 0);
        if ( status != SS$_NORMAL)
        {
            return( NULL);
        }

        status = sys$qiow( 0, channel, IO$_READVBLK | IO$M_NOECHO, &iosb, 0, 0,
                           buf, n, 0, 0, 0, 0);
        if ( ( status != SS$_NORMAL) || ( iosb.status != SS$_NORMAL))
        {
            return( NULL);
        }

        fprintf(stderr, "\n");
        fflush(stderr);
        sevenbit(buf);

        return( buf);
```

```
}

#else
char *
readpass(buf,n)
char *buf;
int n;
{
        int fflags,lword,lwordsav;
        struct sgttyb ttyf,ttysave;

        /* Set normal line editing */
        fflags = fcntl(fileno(stdin),F_GETFL,0);
        fcntl(fileno(stdin),F_SETFL,fflags & ~FNDELAY);
        ioctl(fileno(stdin),TIOCLGET,&lword);
        ioctl(fileno(stdin),TIOCLGET,&lwordsav);
        lword |= LCRTERA|LCRTKIL;
        ioctl(fileno(stdin),TIOCLSET,&lword);

        /* Turn off echoing */
        ioctl(fileno(stdin), TIOCGETP, &ttyf);
        ioctl(fileno(stdin), TIOCGETP, &ttysave);
        ttyf.sg_flags &= ~(ECHO|RAW|CBREAK);
        ttyf.sg_flags |= CRMOD;
        ioctl(fileno(stdin),TIOCSETP,&ttyf);
        fgets(buf,n,stdin);
        rip(buf);

        /* Restore previous tty modes */
        fcntl(fileno(stdin),F_SETFL,fflags);
        ioctl(fileno(stdin),TIOCSETP,&ttysave);
        ioctl(fileno(stdin),TIOCLSET,&lwordsav);

        /*
        after the secret key is taken from the keyboard, the line feed is
        written to standard error instead of standard output.  That means that
        anyone using the program from a terminal won't notice, but capturing
        standard output will get the key words without a newline in front of
        them.
        */
         fprintf(stderr, "\n");
         fflush(stderr);
        sevenbit(buf);

        return buf;
}

#endif
#endif

/* removebackspaced over charaters from the string*/
backspace(buf)
char *buf;
{
        char bs = 0x8;
        char *cp = buf;
        char *out = buf;

        while(*cp){
                if( *cp == bs ) {
                        if(out == buf){
                                cp++;
                                continue;
                        }
                        else {
                          cp++;
                          out--;
                        }
                }
```

```
                else {
                        *out++ = *cp++;
                }

        }
        *out = '\0';

}
sevenbit(s)
char *s;
{
        /* make sure there are only 7 bit code in the line*/
        while(*s){
                *s = 0x7f & ( *s);
                s++;
        }
}
```

A.2.9. skey_challenge.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <ssdef.h>
#include <iodef.h>
#include <descrip.h>
#include <dvidef.h>
#include <jpidef.h>

#include "challenge.h"
#include "skey.h"

/*
**  External definitions
*/
extern unsigned int   sys$qiow();
extern unsigned int   sys$delprc();
extern long int       sys$gettim();
extern unsigned int   sys$getdviw();
extern unsigned int   sys$dassgn();
extern void           lib$date_time();

/*
**  Function prototypes
*/
void    kill_process();

/*
**  Global variables
*/
FILE        *verbose_log_file;
FILE        *log_file;
BOOLEAN     exit_ok = FALSE;
char        dvi_buffer[ DVI_BUF_LEN];
int         dvi_buffer_len;


char *get_time( timestamp)
char    *timestamp;
{
    time_t                      time_str;
    char                        *cp;
    struct dsc$descriptor_s     time_desc;

    time_str = time( NULL);
/*
**  asctime() puts a '\n' on the end that we don't want - smash it to '\0'
*/
    strcpy( timestamp, asctime( localtime( &time_str)));
    cp = timestamp;
    while ( ( *cp != '\0') && ( *cp != '\n'))
    {
        cp++;
    }
    if ( *cp == '\n')
    {
        *cp = '\0';
    }

    time_desc.dsc$a_pointer = timestamp;
    time_desc.dsc$w_length = TIME_BUF_LEN;
    lib$date_time( &time_desc);
    cp = timestamp;
    while ( ( *cp != '\0') && ( *cp != '.'))
    {
```

```
        cp++;
    }
    *cp = '\0';
    return( timestamp);
} /* end get_time */


/*
**  log_message
**
**  Logger if running verbose mode (implies debugging code).
**
*/


void log_message (message)
char * message;
{

#ifdef VERBOSE

        char timestamp[TIME_BUF_LEN];

        verbose_log_file = fopen( VERBOSE_LOG, "a");
        if ( verbose_log_file != NULL)
        {
            fprintf( verbose_log_file, "%s\t%s\n", get_time(timestamp), message);
            fclose( verbose_log_file);
        }

#endif /* VERBOSE */

} /* end log_message */


/*
**  exit_handler
**
**  Exit handler to catch inadvertent exiting.  If exit_ok
**  has been set, we will exit.  Else, we will kill the
**  process.
**
*/
void exit_handler()
{
    log_message( "exit_handler: Beginning exit handler...");
    if ( exit_ok)
    {
        log_message( "exit_handler: ...returned");
        return;
    }
    else
    {
        log_message( "exit_handler: ...failed");
        log_file = fopen( LOG_FILE, "a");
        if ( log_file != NULL)
        {
            fprintf( log_file, "\tProcess caught in the exit handler\n");
            fclose( log_file);
        }
        kill_process();
    }
}


/*
**  kill_process
**
**  Kills the current process (on Unix, the parent process)
```

```
**
*/

void kill_process()
{
    unsigned int    status;

#ifndef NO_KILL
    status = sys$delprc( 0, 0);
#endif

    log_message( "kill_process: Killed!");
    printf( "Killed!\n");
    exit_ok = TRUE;
    exit( SS$_NORMAL);
}


int get_response( buffer, buf_len)
char    *buffer;
int     buf_len;
{
    unsigned int      status;
    int               response = 0;
    static          $DESCRIPTOR( input_dev_desc, "TT");
    unsigned int    term_chan;
    struct
    {
        short int   buf_len;
        short int   item_code;
        char       *buffer;
        int        *ret_len;
        int        terminator;
    } dvi_item_list = { DVI_BUF_LEN, DVI$_TT_ACCPORNAM, dvi_buffer, &dvi_buffer_len, 0};

    log_message( "Beginning get_response");

    log_message( "get_response: Issuing sys$assign...");
    status = sys$assign( &input_dev_desc,
                         &term_chan,
                         0,
                         0);
    log_message( "...Done");
    if ( status != SS$_NORMAL)
    {
        kill_process();
    }

    log_message( "get_response: Issuing sys$getdviw...");
    status = sys$getdviw( 0,
                         term_chan,
                         0,
                         &dvi_item_list,
                         0,
                         0,
                         0,
                         0);
    log_message( "...Done");
    if ( status != SS$_NORMAL)
    {
        kill_process();
    }

    log_message( "get_response: Issuing sys$qiow...");
    status = sys$qiow( 0,
                     term_chan,
                     IO$_READVBLK | IO$M_TIMED,
                     0,
                     0,
```

```
                      0,
                      buffer,
                      buf_len,
                      30,
                      0, 0, 0);
    log_message( "...Done");

    if ( status != SS$_NORMAL)
    {
        log_message( "get_response: Problem waiting for response");
        if ( status == SS$_TIMEOUT)
        {
            log_message( "get_response: Timeout waiting for response");
            log_file = fopen( LOG_FILE, "a");
            if ( log_file != NULL)
            {
                fprintf( log_file, "\tTimeout waiting for response\n");
                fclose( log_file);
            }
        }
        kill_process();
    }

    log_message( "get_response: Got response okay");
    log_message( "get_response: Issuing sys$dassgn...");
    status = sys$dassgn( term_chan);
    log_message( "...Done");

    log_message( "get_response: Now leaving get_response successfully");
    return;
}


int main()
{
    int             challenge;
    int             response = 0;
    char            buffer[ BUFFER_LEN];
    unsigned int     status;
    char            time_buf[ TIME_BUF_LEN];
    char            *connection_source = dvi_buffer;
    unsigned int     exit_status = SS$_NORMAL;
    struct
    {
        int             flink;
        unsigned int    addr;
        int             arg_count;
        int             reason_addr;
    } desblk = { 0, &exit_handler, 0, &exit_status};
    union
    {
        long int        time;
        struct
        {
            int             top;
            int             bottom;
        } t;
    } time_blk;
    char            jpi_buffer[ JPI_BUF_LEN];
    int             jpi_buffer_len;
    char            jpi_buffer1[ JPI_BUF_LEN];
    int             jpi_buffer_len1;
    struct
    {
        short int    buf_len;
        short int    function_code;
        char        *jpi_buffer;
        int         *ret_buffer_len;
        short int    buf_len1;
```

```
        short int   function_code1;
        char        *jpi_buffer1;
        int         *ret_buffer_len1;
        int         terminator;
    } jpi_item_list = { JPI_BUF_LEN, JPI$_MODE, jpi_buffer, &jpi_buffer_len,
                        JPI_BUF_LEN, JPI$_USERNAME, jpi_buffer1, &jpi_buffer_len1, 0};
    $DESCRIPTOR( time_desc, time_buf);

    char            dev_buffer[ DVI_BUF_LEN];
    int             dev_buffer_len;
    static          $DESCRIPTOR( input_dev_desc, "TT");
    unsigned int     term_chan;
    struct
    {
        short int   buf_len;
        short int   item_code;
        char        *buffer;
        int         *ret_len;
        int         terminator;
    } dvi_item_list = { DVI_BUF_LEN, DVI$_TT_PHYDEVNAM, dev_buffer, &dev_buffer_len, 0};
    struct skey     skey;
    char            skeyprompt[ 80];
    int             found;
    char            pbuf[ ANSWER_LEN];
    char            *cp;

    log_message( "main: Issuing sys$dclexh...");
    status = sys$dclexh( &desblk);
    log_message( "...Done");
    if ( status != SS$_NORMAL)
    {
        kill_process();
    }

    log_message( "main: Issuing sys$gettim...");
    status = sys$gettim( &time_blk.time);
    log_message( "...Done");
    if ( status != SS$_NORMAL)
    {
        kill_process();
    }
    time_desc.dsc$w_length = TIME_BUF_LEN;
    log_message( "main: Issuing sys$asctim...");
    status = sys$asctim( 0, &time_desc, 0, 0);
    log_message( "...Done");
    if ( status != SS$_NORMAL)
    {
        kill_process();
    }

    log_message( "main: Issuing sys$assign...");
    status = sys$assign( &input_dev_desc,
                         &term_chan,
                         0,
                         0);
    log_message( "...Done");
    if ( status != SS$_NORMAL)
    {
        kill_process();
    }

    log_message( "main: Issuing sys$getdviw...");
    status = sys$getdviw( 0,
                         term_chan,
                         0,
                         &dvi_item_list,
                         0,
                         0,
                         0,
```

```
                         0);
    log_message( "...Done");
    if ( status != SS$_NORMAL)
    {
        kill_process();
    }

    log_message( "main: Issuing sys$dassgn...");
    status = sys$dassgn( term_chan);
    log_message( "...Done");
    log_message( dev_buffer);

    if ( ( strlen( dev_buffer) == 0))
/* || ( strncmp( dev_buffer, "_TWA", 4) == 0)) */
    {
        exit_ok = TRUE;
        exit( SS$_NORMAL);
    }

    log_message( "main: Issuing sys$getjpi...");
    status = sys$getjpi( 0,
                         0,
                         0,
                         &jpi_item_list,
                         0,
                         0,
                         0);
    log_message( "...Done");
    if ( status != SS$_NORMAL)
    {
        kill_process();
    }
    if ( ( int) jpi_buffer[ 0] != JPI$K_INTERACTIVE)
    {
        log_file = fopen( LOG_FILE, "r");
        if ( log_file != NULL)
        {
            if ( fgets( buffer, BUFFER_LEN, log_file) != NULL)
            {
                kill_process();
            }
        }
        exit_ok = TRUE;
        exit( SS$_NORMAL);
    }
    cp = strchr( jpi_buffer1, ' ');
    if ( cp != NULL)
    {
        *cp = '\0';
    }
    found = skeychallenge( &skey, jpi_buffer1, skeyprompt);
    if ( found == 1)
    {
        exit_ok = TRUE;
        exit( SS$_NORMAL);
    }
    if ( found == -1)
    {
        kill_process();
    }
    printf( "%s: ", skeyprompt);
    get_response( pbuf, ANSWER_LEN);

    if ( skeyverify( &skey, pbuf) != 0)
    {
        log_file = fopen( LOG_FILE, "a");
        if ( log_file != NULL)
        {
```

```
            fprintf( log_file, "%s: Connect from '%s' failed S/Key\n", time_buf,
connection_source);
            fclose( log_file);
        }
        kill_process();
    }
    else
    {
        exit_ok = TRUE;
        printf( "\n");
        log_file = fopen( LOG_FILE, "r");
        if ( log_file != NULL)
        {
            while ( fgets( buffer, BUFFER_LEN, log_file) != NULL)
            {
               printf( "%s", buffer);
            }
            fclose( log_file);
        }

        exit( SS$_NORMAL);
    }
}
```

A.2.10. lgi$callout_skey.c

```
/*
**  LOGINOUT Callout LGI$ICR_AUTHENTICATE which implements S/Key
**  one time password checks.
**
**  If the VMS account as more than one password, then we cannot use
**  one time passwords (who holds which list?).  If the user is
**  initialised to use S/Key, but fails their password test, we
**  attempt to see if the password supplied is their VMS password.
**
**  Author:  Danny Smith
**  Date:  7-Nov-1993.
**
**  This program can be linked using the following example commands:
**
**      $ CC LGI$CALLOUT_SKEY.C
**      $ LINK/SHARE=LGI$CALLOUT_SKEY SYS$INPUT/OPT
**      LGI$CALLOUT_SKEY.OBJ, SYS$SHARE:VAXCRTL.EXE/SHARE
**      UNIVERSAL=LGI$LOGINOUT_CALLOUTS
**
**  Then the following steps are required to install it:
**
**      $ DEFINE/SYSTEM/EXEC LGI$LOGINOUT_CALLOUTS LGI$CALLOUT_SKEY
**
**  If the file is not located is SYS$SHARE, also define:
**
**      $ DEFINE/SYSTEM/EXEC LGI$CALLOUT_SKEY <filespec>
**
**  Then the image must be installed
**
**      $ INSTALL REPLACE SYS$SHARE:LGI$CALLOUT_SKEY.EXE
**
**  Finally, the SYSGEN parameter LGI_CALLOUTS needs to be set to the number
**  of callout modules (at this stage, 1 only)
**
**      $ RUN SYS$SYSTEM:SYSGEN
**      SYSGEN> SET LGI_CALLOUTS 1
**      SYSGEN> WRITE CURRENT
**      SYSGEN> WRITE ACTIVE
**      SYSGEN> EXIT
**
**  Edit History:
**
**  0   DFS    Initial Version (7-Nov-1993).
**
**  [End of Revision History]
*/

/*
**  Header files
*/
#include <stdio.h>
#include <descrip.h>
#include <rms.h>
#include <stsdef.h>
#include <ssdef.h>
#include <prcdef.h>
#include "skey.h"

#define SKEYPROMPT_LEN 80
#define PASSWORD_LEN   32
#define USERNAME_LEN   32

/*
**  Structures for callout vector and callout arguments vector
*/
struct LGI$CALLOUT_VECTOR
```

```
{
    long int                  LGI$L_ICR_ENTRY_COUNT;
    int                       ( *LGI$ICR_INIT) ();
    int                       ( *LGI$ICR_IACT_START) ();
    int                       ( *LGI$ICR_DECWINIT) ();
    int                       ( *LGI$ICR_IDENTIFY) ();
    int                       ( *LGI$ICR_AUTHENTICATE) ();
    int                       ( *LGI$ICR_CHKRESTRICT) ();
    int                       ( *LGI$ICR_FINISH) ();
    int                       ( *LGI$ICR_LOGOUT) ();
    int                       ( *LGI$ICR_JOBSTEP) ();
};

struct LGI$ARG_VECTOR
{
    int                       ( *LGI$ICB_GET_INPUT) ();
    int                       ( *LGI$ICB_DECW_IDENT) ();
    int                       ( *LGI$ICB_DECW_AUTH) ();
    void                      ( *LGI$ICB_GET_SYSPWD) ();
    int                       ( *LGI$ICB_USERPROMPT) ();
    int                       ( *LGI$ICB_USERPARSE) ();
    int                       ( *LGI$ICB_AUTOLOGIN) ();
    int                       ( *LGI$ICB_PASSWORD) ();
    int                       ( *LGI$ICB_CHECK_PASS) ();
    int                       ( *LGI$ICB_VALIDATE) ();
    void                      ( *LGI$ICB_ACCTEXPIRED) ();
    void                      ( *LGI$ICB_PWDEXPIRED) ();
    int                       ( *LGI$ICB_DISUSER) ();
    void                      ( *LGI$ICB_MODALHOURS) ();
    short int                 *LGI$A_ICR_CREPRC_FLAGS;
    char                      *LGI$A_ICR_JOB_TYPE;
    char                      *LGI$A_ICR_SUBPROCESS;
    char                      *LGI$A_ICR_TERMINAL_DEV;
    struct dsc$descriptor_s   *LGI$A_ICR_TT_PHYDEVNAM;
    struct dsc$descriptor_s   *LGI$A_ICR_TT_ACCPORNAM;
    struct dsc$descriptor_s   *LGI$A_ICR_CLINAME;
    struct dsc$descriptor_s   *LGI$A_ICR_CLITABLES;
    struct dsc$descriptor_s   *LGI$A_ICR_NCB;
    int                       *LGI$A_ICR_LOGLINK;
    struct dsc$descriptor_s   *LGI$A_ICR_REM_NODE_NAME;
    struct dsc$descriptor_s   *LGI$A_ICR_REM_ID;
    unsigned int              *LGI$A_ICR_UAF_RECORD;
    struct RAB                *LGI$A_ICR_INPUT_RAB;
    char                      *LGI$A_ICR_AUTOLOGIN;
    struct dsc$descriptor_s   *LGI$A_ICR_USERNAME;
    struct dsc$descriptor_s   *LGI$A_ICR_PWD1;
    struct dsc$descriptor_s   *LGI$A_ICR_PWD2;
    int                       *LGI$A_ICR_PWDCOUNT;
    short int                 *LGI$A_ICR_NETFLAGS;
};

globalvalue int               LGI$_SKIPRELATED;
globalvalue int               LGI$_DISUSER;
globalvalue int               LGI$_INVPWD;
globalvalue int               LGI$_NOSUCHUSER;
globalvalue int               LGI$_NOTVALID;
globalvalue int               LGI$_INVINPUT;
globalvalue int               LGI$_CMDINPUT;
globalvalue int               LGI$_FILEACC;

static int                    callout_decwinit();
static int                    callout_authenticate();

globaldef struct LGI$CALLOUT_VECTOR          LGI$LOGINOUT_CALLOUTS =
{
    9,                        /* entry count */
    0,                        /* init */
    0,                        /* iact_start */
    callout_decwinit,         /* decwinit */
```

```
    0,                          /* identify */
    callout_authenticate,       /* authenticate */
    0,                          /* chkrestrict */
    0,                          /* finish */
    0,                          /* logout */
    0                           /* jobstep */
};

/*
**  DECwindows initialisation
*/
static int callout_decwinit( arg_vector, context)
struct LGI$ARG_VECTOR *arg_vector;
int                    *context;
{
    /* disable any further callouts */
    LGI$LOGINOUT_CALLOUTS.LGI$L_ICR_ENTRY_COUNT = 0;

    /* return and do normal DECwindows processing */
    return( SS$_NORMAL);
}

/*
**  Authenticate
**
**  If the account has two passwords, we cannot be using S/Key as this
**  system currently only supports a single one time password for the
**  account.  Therefore, do normal password processing.
**
**  If the user is found in S/Key, prompt for the S/Key password.  If it
**  is not valid, check to see if the password is the standard VMS
**  password.  Validate if it is.  Any errors in the S/Key system will
**  cause normal VMS authentication to take place.
*/
static int callout_authenticate( arg_vector, context)
struct LGI$ARG_VECTOR *arg_vector;
int                    *context;
{
    int                         status;
    char                        skeyprompt[ SKEYPROMPT_LEN + 1];
    struct dsc$descriptor_s     skeyprompt_desc =
        { SKEYPROMPT_LEN, DSC$K_DTYPE_T, DSC$K_CLASS_S, skeyprompt};
    char                        password[ PASSWORD_LEN + 1];
    char                        skeypassword[ PASSWORD_LEN + 1];
    struct dsc$descriptor_s     password_desc =
        { PASSWORD_LEN, DSC$K_DTYPE_T, DSC$K_CLASS_S, password};
    struct skey                 skey;
    int                         found;
    char                        username[ USERNAME_LEN + 1];
    char                        *cp;

/*
**  This system can only deal with interactive jobs
*/
    if ( !( *arg_vector->LGI$A_ICR_CREPRC_FLAGS & PRC$M_INTER))
    {
        return( SS$_NORMAL);   /* not interactive - do normal processing
*/
    }

    if ( *arg_vector->LGI$A_ICR_CREPRC_FLAGS & PRC$M_NOPASSWORD)
    {
        return( SS$_NORMAL);   /* invoked as logged in - don't prompt */
    }

    if ( *arg_vector->LGI$A_ICR_SUBPROCESS != 0)
    {
        return( SS$_NORMAL);   /* don't prompt on subprocesses */
    }
```

```
/*
**  If the account does not have a password, then don't prompt
*/
    if ( *arg_vector->LGI$A_ICR_PWDCOUNT == 0)
    {
        return( LGI$_SKIPRELATED);    /* continue processing */
    }

/*
**  Can only check accounts with a single password
*/
    if ( *arg_vector->LGI$A_ICR_PWDCOUNT == 1)
    {
        strncpy( username, arg_vector->LGI$A_ICR_USERNAME->dsc$a_pointer, USERNAME_LEN);

/*
**  Ensure there are no trailing spaces
*/
        cp = strchr( username, ' ');
        if ( cp != NULL)
        {
            *cp = '\0';
        }

/*
**  Is this user in the S/Key system?
**  prompt *ABSOLUTELY MUST* start with <CR><LF>
*/
        strcpy( skeyprompt, "\r\n");
        cp = skeyprompt;
        cp += strlen( skeyprompt);
        found = skeychallenge( &skey, username, cp);
        strcat( skeyprompt, ": ");
        if ( found == 0)              /* found a user */
        {
            skeyprompt_desc.dsc$w_length = strlen( skeyprompt);

/*
**  Get the password, but don't validate it - we'll do that!
*/
            status = arg_vector->LGI$ICB_PASSWORD( -3, &skeyprompt_desc, &password_desc);
            if ( $VMS_STATUS_SUCCESS( status))
            {
/*
**  Now check the password with S/Key
*/
                strncpy( skeypassword, password, password_desc.dsc$w_length);
                cp = skeypassword;
                cp += password_desc.dsc$w_length;
                *cp = '\0';
                cp = skeypassword;
                while ( *cp != '\0')
                {
                    if ( *cp == '.')
                    {
                        *cp = ' ';
                    }
                    cp++;
                }
                status = skeyverify( &skey, skeypassword);
                if ( status == 0)
                {
/*
**  Victory!!  Let them in!
*/
                    return( LGI$_SKIPRELATED);
                }
                else
```

```
                {
/*
**  It may be their VMS password!!
*/
                    if ( arg_vector->LGI$ICB_VALIDATE != NULL)
                    {
                        status = arg_vector->LGI$ICB_VALIDATE(
                                arg_vector->LGI$A_ICR_USERNAME, &password_desc, 0);
                        if ( $VMS_STATUS_SUCCESS( status))
                        {
                            return( LGI$_SKIPRELATED);
                        }
                        else
                        {
                            return( status);
                        }
                    }
                    else
                    {
                        return( -4);
                    }
                }
            }
            else
            {
/*
**  couldn't get any password - just return the status
*/
                return( status);
            }
        }
    }

/*
**  Account has at least one password, and is not in S/Key system.
**  Do normal VMS processing
*/
    strcpy( skeyprompt, "\r\nPassword: ");
    skeyprompt_desc.dsc$w_length = strlen( skeyprompt);
    status = arg_vector->LGI$ICB_PASSWORD( 0, &skeyprompt_desc, 0);
    if ( !$VMS_STATUS_SUCCESS( status))
    {
        return( status);        /* on error, return status here */
    }

/*
**  Does this account have a secondary password?
*/
    if ( *arg_vector->LGI$A_ICR_PWDCOUNT == 2)
    {
        status = arg_vector->LGI$ICB_PASSWORD( 1, &skeyprompt_desc, 0);
        if ( !$VMS_STATUS_SUCCESS( status))
        {
            return( status);  /* on error, return status here */
        }
    }

/*
**  Successful prompt and password validation, skip VMS policy
*/
    return( LGI$_SKIPRELATED);
}
```

A.2.11. strftime.c

```
#include <stdio.h>
#include <time.h>

/*
 * strftime --- produce formatted time
 */

int strftime(s, maxsize, format, timeptr)
char *s;
int maxsize;
char *format;
struct tm *timeptr;
{
        char *endp = s + maxsize;
        char *start = s;
        char tbuf[100];
        int i;

        /*
         * various tables, useful in North America
         */
        static char *days_a[] = {
                "Sun", "Mon", "Tue", "Wed",
                "Thu", "Fri", "Sat",
        };
        static char *days_l[] = {
                "Sunday", "Monday", "Tuesday", "Wednesday",
                "Thursday", "Friday", "Saturday",
        };
        static char *months_a[] = {
                "Jan", "Feb", "Mar", "Apr", "May", "Jun",
                "Jul", "Aug", "Sep", "Oct", "Nov", "Dec",
        };
        static char *months_l[] = {
                "January", "February", "March", "April",
                "May", "June", "July", "August", "September",
                "October", "November", "December",
        };
        static char *ampm[] = { "AM", "PM", };

        if (s == NULL || format == NULL || timeptr == NULL || maxsize == 0)
                return 0;

        if (strchr(format, '%') == NULL && strlen(format) + 1 >= maxsize)
                return 0;

        for (; *format && s < endp - 1; format++) {
                tbuf[0] = '\0';
                if (*format != '%') {
                        *s++ = *format;
                        continue;
                }
                switch (*++format) {
                case '\0':
                        *s++ = '%';
                        goto out;

                case '%':
                        *s++ = '%';
                        continue;

                case 'a':       /* abbreviated weekday name */
                        strcpy(tbuf, days_a[timeptr->tm_wday]);
                        break;

                case 'A':       /* full weekday name */
```

```
                strcpy(tbuf, days_l[timeptr->tm_wday]);
                break;

        case 'b':       /* abbreviated month name */
                strcpy(tbuf, months_a[timeptr->tm_mon]);
                break;

        case 'B':       /* full month name */
                strcpy(tbuf, months_l[timeptr->tm_mon]);
                break;

        case 'c':       /* appropriate date and time representation */
                sprintf(tbuf, "%s %s %2d %02d:%02d:%02d %d",
                        days_a[timeptr->tm_wday],
                        months_a[timeptr->tm_mon],
                        timeptr->tm_mday,
                        timeptr->tm_hour,
                        timeptr->tm_min,
                        timeptr->tm_sec,
                        timeptr->tm_year + 1900);
                break;

        case 'd':       /* day of the month, 01 - 31 */
                sprintf(tbuf, "%02d", timeptr->tm_mday);
                break;

        case 'H':       /* hour, 24-hour clock, 00 - 23 */
                sprintf(tbuf, "%02d", timeptr->tm_hour);
                break;

        case 'I':       /* hour, 12-hour clock, 01 - 12 */
                i = timeptr->tm_hour;
                if (i == 0)
                        i = 12;
                else if (i > 12)
                        i -= 12;
                sprintf(tbuf, "%02d", i);
                break;

        case 'j':       /* day of the year, 001 - 366 */
                sprintf(tbuf, "%03d", timeptr->tm_yday + 1);
                break;

        case 'm':       /* month, 01 - 12 */
                sprintf(tbuf, "%02d", timeptr->tm_mon + 1);
                break;

        case 'M':       /* minute, 00 - 59 */
                sprintf(tbuf, "%02d", timeptr->tm_min);
                break;

        case 'p':       /* am or pm based on 12-hour clock */
                if (timeptr->tm_hour < 12)
                        strcpy(tbuf, ampm[0]);
                else
                        strcpy(tbuf, ampm[1]);
                break;

        case 'S':       /* second, 00 - 61 */
                sprintf(tbuf, "%02d", timeptr->tm_sec);
                break;

        case 'T':       /* time as %H:%M:%S */
                strftime(tbuf, sizeof tbuf, "%H:%M:%S", timeptr);
                break;

        case 'w':       /* weekday, Sunday == 0, 0 - 6 */
                sprintf(tbuf, "%d", timeptr->tm_wday);
                break;
```

```
                case 'x':       /* appropriate date representation */
                        sprintf(tbuf, "%s %s %2d %d",
                                days_a[timeptr->tm_wday],
                                months_a[timeptr->tm_mon],
                                timeptr->tm_mday,
                                timeptr->tm_year + 1900);
                        break;

                case 'X':       /* appropriate time representation */
                        sprintf(tbuf, "%02d:%02d:%02d",
                                timeptr->tm_hour,
                                timeptr->tm_min,
                                timeptr->tm_sec);
                        break;

                case 'y':       /* year without a century, 00 - 99 */
                        i = timeptr->tm_year % 100;
                        sprintf(tbuf, "%d", i);
                        break;

                case 'Y':       /* year with century */
                        sprintf(tbuf, "%d", 1900 + timeptr->tm_year);
                        break;

                default:
                        tbuf[0] = '%';
                        tbuf[1] = *format;
                        tbuf[2] = '\0';
                        break;
                }
                i = strlen(tbuf);
                if (i)
                        if (s + i < endp - 1) {
                                strcpy(s, tbuf);
                                s += i;
                        } else
                                return 0;
        }
out:
        if (s < endp && *format == '\0') {
                *s = '\0';
                return (s - start);
        } else
                return 0;

}
```

A.3. Utility Command Files

A.3.1. compile.com

```
$ if P1 .eqs. "CALLOUT_SKEY" then goto callout_skey
$ if P1 .eqs. "SKEYINIT" then goto skeyinit
$ if P1 .eqs. "SKEYSUBR" then goto skeysubr
$ if P1 .eqs. "SKEYLOGIN" then goto skeylogin
$ if P1 .eqs. "MD4" then goto md4
$ if P1 .eqs. "PUT" then goto put
$ if P1 .eqs. "GETHOSTNAME" then goto gethostname
$ if P1 .eqs. "GETSKEYPROMPT" then goto getskeyprompt
$ if P1 .eqs. "SKEY" then goto skey
$ if P1 .eqs. "TEST_SKEY" then goto test_skey
$!
$ if P1 .eqs. "ALL" then goto all
$ write sys$output "Unknown module"
$ exit
$!
$ callout_skey:
$ cc/debug=notraceback lgi$callout_skey
$ exit
$!
$ skeyinit:
$ cc skeyinit
$ exit
$!
$ skeysubr:
$ cc skeysubr
$ exit
$!
$ skeylogin:
$ cc skeylogin
$ exit
$!
$ md4:
$ cc md4
$ exit
$!
$ put:
$ cc put
$ exit
$!
$ gethostname:
$ cc gethostname
$ exit
$!
$ getskeyprompt:
$ cc getskeyprompt
$ exit
$!
$!
$ skey:
$ cc skey
$ exit
$!
$ test_skey:
$ cc test_skey
$ exit
$!
$ all:
$ @compile CALLOUT_SKEY
$ @compile SKEYINIT
$ @compile SKEYSUBR
$ @compile SKEYLOGIN
$ @compile MD4
$ @compile PUT
$ @compile GETHOSTNAME
```

```
$ @compile GETSKEYPROMPT
$ @compile SKEY
$ @compile TEST_SKEY
$ exit
$!
```

A.3.2. link.com

```
$ if P1 .eqs. "CALLOUT_SKEY" then goto callout_skey
$ if P1 .eqs. "SKEYINIT" then goto skeyinit
$ if P1 .eqs. "SKEY" then goto skey
$ if P1 .eqs. "TEST_SKEY" then goto test_skey
$ if P1 .eqs. "ALL" then goto all
$!
$ write sys$output "unknown module"
$ exit
$!
$ callout_skey:
$ link/share=LGI$CALLOUT_SKEY/notraceback sys$input/opt
lgi$callout_skey.obj, skeylogin, skeysubr, md4, put, strftime, sys$share:vaxcrtl.exe/sha
universal=lgi$loginout_callouts
$ exit
$!
$ skeyinit:
$ link skeyinit, skeysubr, skeylogin, md4, put, gethostname, sys$library:vaxcrtl/lib
$ exit
$!
$ skey:
$ link skey, skeysubr, skeylogin, md4, put, sys$library:vaxcrtl/lib
$ exit
$!
$ test_skey:
$ link test_skey, skeysubr, skeylogin, md4, put, sys$library:vaxcrtl/lib
$ exit
$!
$!
$ all:
$ @link CALLOUT_SKEY
$ @link SKEYINIT
$ @link SKEY
$ @link TEST_SKEY
$ exit
```

A.3.3. define.com

```
$ skeyinit :== "$dskb:[ccdanny.skey]skeyinit.exe"
$ skey :== "$dskb:[ccdanny.skey]skey.exe"
```

A.4. Data Files

A.4.1. SYS$MANAGER:SKEYKEYS.DAT;1

```
USER1 0099 ko492802         eb8fa6cded94a3f3  Oct 24,1993
USER2 0099 ko15288          aa2fb780f2383cc4  Oct 15,1993
```