A Real-World Analysis of Kerberos Password Security

Thomas Wu
Computer Science Department
Stanford University
tjw@cs.Stanford.EDU

Abstract

Kerberos is a distributed authentication system that many organizations use to handle domain-wide password security. Although it has been known for quite some time that Kerberos is vulnerable to brute-force password searches, there has so far been little analysis of the scope and extent of this vulnerability. This paper discusses the nature of this weakness in detail and attempts to quantify the severity of the danger it poses to existing Kerberized installations. The results of a controlled experiment, in which a large number of passwords from a Kerberos realm were broken off-line and subjected to analysis, will be presented. The author explores possible strategies for repairing this security hole, the most viable of which is the use of Kerberos V5 preauthentication coupled with a secure password authentication protocol such as SRP, SPEKE, or EKE.

1 Introduction

Kerberos [18], developed at MIT about ten years ago, was an authentication infrastructure designed to assure the security of user accounts and system services on potentially insecure networks. By distributing secret keys and using cryptographic protocols like Needham-Schroeder [14] to verify possession of these keys, Kerberos supposedly prevented unauthorized parties from compromising system security even if they had the means to subvert the network.

In 1989, Bellovin & Merritt outlined a number of security weaknesses in Kerberos [2]. While most of those problems have been addressed in some way since they were first brought to light, others still remain with us to this very day. One of the most serious of these is the susceptibility of Kerberos to off-line guessing attacks against its passwords; in a sense, the Kerberos Ticket-Granting-Ticket (TGT) protocol is its Achilles' Heel. Although both Kerberos V4 and V5 are affected, this vulnerability is especially problematic in Kerberos V4 because, as will be explained later, it allows a large-scale attack against it to proceed virtually undetected. Such an attack would require nothing more than Internet access, a dictionary, and spare CPU cycles. Section 2 discusses this attack in more detail.

Traditionally, sites have attempted to work around this problem by forcing their users to memorize longer and more complex passwords. In the spirit of earlier password security analyses [5, 13, 17], the author conducted an experiment to determine how effective such a policy has been in practice. The experiment involved a simulation of a distributed password cracking effort against the password database of a large Kerberos authentication environment, or *realm*. Section 3 presents the results of this experiment, which indicate that Kerberos does not protect passwords as well as previously thought, and which provide evidence that attempting to compel the use of "better" passwords has not, by itself, been successful in reducing the threat posed by a password-based attack against Kerberos.

Section 4 then discusses possible approaches to remedy this situation. The most practical approach, augmenting the TGT protocol with secure password authentication technology such as SRP [19], is discussed in greater depth, along with some implementation strategies.

Recent attacks against Kerberos [4] have underscored the need for continuing scrutiny of security protocols and the re-evaluation of past assumptions. This paper is intended for both network security experts, most of whom may already know about the security flaw in Kerberos but who may not have access to data on its severity, and for average users, who may not yet be aware of the existence of this problem or its implications for the security of their own accounts.

2 Anatomy of a Security Hole

We start with a very brief summary of how Kerberos is structured, followed by a closer look at the actual password authentication protocol. For more in-depth information on Kerberos, the original paper [18] is recommended, along with papers describing the underlying component protocols [14].

2.1 Kerberos Tickets

All entities in Kerberos, be they human users or non-human servers, such as those for E-mail or print services, have a secret key, which is shared only with the central authentication server. To obtain services from an application server in a "Kerberized" environment, a client must first obtain what is known as a Kerberos *ticket* from the authentication server. This ticket contains, among other things, a section of data encrypted with the secret key belonging to the requested service. The client presents this ticket to the application server, which can then verify the ticket for authenticity. Since the client does not know the server's secret key, it cannot forge a valid ticket, nor can it tamper with the contents of a ticket without being detected.

The actual procedure for obtaining, storing, and using tickets is divided into two steps:

- When the user first logs in and enters his password, the client software uses the password to obtain a special ticket known as a *Ticket-Granting Ticket* (TGT) from the central authentication server.

- When a user requires access to a Kerberized service, the client software presents the TGT to the Ticket-Granting Server (TGS), which then issues a ticket for that particular service. This service-specific ticket is then used to authenticate the actual requests for service.

This is done to minimize the number of times the user needs to enter his password. Once a user has a valid TGT, the application software can automatically obtain service-specific tickets without human intervention.

2.2 A Closer Look at the TGT

When a user logs into a Kerberized system and wishes to obtain a TGT, he sends Kerberos a request packet containing the fields listed in Table 1.

Field	Contents	Length
1	Protocol Version Number	1 byte
2	Message Type Identifier	1 byte
3	Username	string
4	Requested Ticket Instance	string
5	Kerberos Realm	string
6	Timestamp	4 bytes
7	Requested Ticket Lifetime	1 byte
8	Requested Service	string
9	Requested Service Instance	string

Table 1: TGT Request Format

All string data is variable-length and null-terminated, with no padding or alignment. Note that the server cannot authenticate this packet; an intruder can construct a valid-looking request packet that is indistinguishable from one sent by a legitimate user. Instead, Kerberos authenticates the client by sending back an encrypted packet formatted as described in Table 2.

Field	Contents	Length
1	Session Key	8 bytes
2	Service Name	string
3	Instance	string
4	Realm, or domain	string
5	Ticket Lifetime	1 byte
6	Version Number	1 byte
7	Encrypted Ticket Block length	1 byte
8	Encrypted Ticket Block	(field 7)
9	Timestamp	4 bytes

Table 2: TGT Return Packet Format

This entire packet is encrypted with a key derived from the user's password. Kerberos uses the Data Encryption Standard (DES) [15] as the encryption algorithm. Thus, if the user enters the correct password upon logging in, the client will be able to decrypt the return packet and obtain a valid TGT. An unauthorized user, without the correct password, only sees useless random bits.

2.3 In Enemy Hands

Normally, if the user enters an incorrect password, the initial decryption attempt produces a gibberish packet, which causes the the Kerberos client software to notify the user and discard the packet. But what if the client software, instead of throwing away the packet after each attempt, allowed the user to try decrypting the same packet again with different passwords? And what if, instead of having a human typing in different passwords, the software automated the procedure, pulling in passwords from a dictionary as fast as it could check them? Since the TGT has a fixed, publicly-known format, the software could determine if it had found the correct password by looking for one that decrypted the TGT properly.

This is an example of a *dictionary attack*, which has been used in the past, with great success, to compromise password security mechanisms [12]. A program that performs dictionary attacks against passwords is often known as a *password cracker*. Passwords crackers originally did nothing more than test words from a user-supplied word list, but they have evolved over the years into sophisticated engines that can expand dictionaries into massive lists of likely passwords based on transformation rules. The password cracker used in the experiment is an example of such a program, and it will be discussed in more detail in Section 3.3.

It is a fairly simple matter to construct a client program that saves tickets in a form suitable for repeated trial decryption. The **K-DUMP** program, written to gather TGTs for the experiment in Section 3, accepts a username and the network address of a Kerberos authentication server, constructs a valid request packet of the form described in Table 1, sends it to Kerberos, and saves the encrypted reply packet to a file.

2.4 Compromising the TGT

After the K-DUMP program saves a user's encrypted TGT to a file, the intruder can begin testing trial passwords. To verify a password guess P:

1. Convert the password to a DES key: K = STRING-TO-KEY(P)

2. Decrypt the ticket with the key K and see if it is a valid Kerberos ticket. If it is, then P is the user's password.

The STRING-TO-KEY function varies between Kerberos realms, but nearly all sites employ one of two different functions. This can be determined for any particular domain through trial-and-error[1]. Some STRING-TO-KEY functions, because of flaws in their design, also allow attacks that are not dependent on low password entropy; Section 3.5 explains this in more detail.

Encryption algorithms such as DES operate on fixed-sized chunks of data, known as *blocks*; DES uses 64-bit (8-byte) blocks. If the input to the encryption algorithm is more than one block long, each block can be encrypted individually, or they can be *chained* together to improve security. In Kerberos, the TGT encryption uses a chaining method known as Propagating Cipher Block Chaining (PCBC), and it uses K as the initialization vector (IV). What this means is that the encrypted ticket blocks $T_0, T_1, \ldots$ are generated from the plaintext blocks $B_0, B_1, \ldots$, as follows:

$$\begin{aligned} T_0 &= E_K(B_0 \oplus IV) \\ T_1 &= E_K(B_1 \oplus B_0 \oplus T_0) \\ T_2 &= E_K(B_2 \oplus B_1 \oplus T_1) \\ &\vdots \end{aligned}$$

This process can be reversed to generate the plaintext blocks B_n from the encrypted ticket blocks T_n:

$$\begin{aligned} B_0 &= D_K(T_0) \oplus IV \\ B_1 &= D_K(T_1) \oplus B_0 \oplus T_0 \\ B_2 &= D_K(T_2) \oplus B_1 \oplus T_1 \\ &\vdots \end{aligned}$$

Reference [16] has more information on PCBC, other cipher chaining modes, and initialization vectors.

2.4.1 Verifiable Plaintext

To determine whether or not a given key produces a valid ticket, the Kerberos client software examines the decrypted ticket fields to see if they make sense. If the wrong key was used, and if the software attempts to find a null terminator for each string, it may find too many or not enough of them for the number of strings in a ticket. It may also discover that some of the other fields, like the timestamp or length fields, are internally inconsistent.

[1] If the wrong STRING-TO-KEY function is used, the chance of turning up *any* successful guesses is infinitesimal.

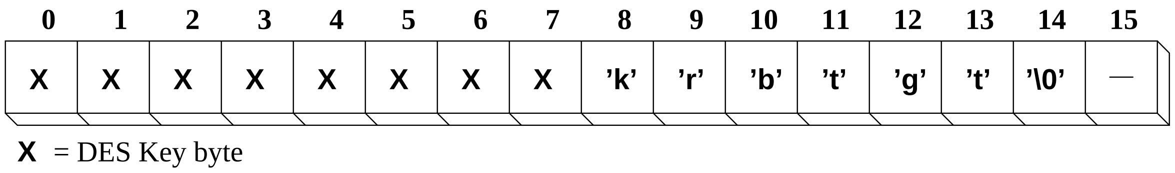

Figure 1: Correctly Decrypted TGT (First Two Blocks)

Kerberos gives the attacker an important piece of information that greatly simplifies this check: The "service name" field, which is the second field of the decrypted ticket, is always the string `krbtgt` for TGT packets. Since those seven bytes (positions 8-14 in the TGT) are always within the second block B_1, only the first two blocks of the TGT need to be decrypted before the correctness of the guess can be verified (see Figure 1). The probability of this string occurring at random in the right location despite having the wrong key is only 2^{-56}, so finding this string in the second block is a highly reliable indicator of success.

For each user, then, only the first two blocks (T_0, T_1) of his encrypted TGT need to be stored, and each trial password requires only one call to STRING-TO-KEY and two DES decryptions.

2.4.2 DES Parity Optimization

The format of the Kerberos TGT packet permits a further refinement of the password verification procedure. The DES algorithm requires its keys to have odd parity, i.e. the number of "1" bits in each byte of a DES key must be odd [15]. Since the first block B_0 of the plaintext TGT is a DES session key (see Table 2), we know each of its bytes must have odd parity.

Instead of computing both B_0 and B_1 for each guess, the attacker can compute just B_0 and check the parity of all 8 bytes. If any of them are not odd-parity bytes, he can conclude that the guess was incorrect without having to decrypt the second TGT block. 255 out of every 256 tries will, on average, fail the parity check right away, so most guesses now require only one DES decryption instead of two.

This optimization nearly doubles the speed of dictionary searches on password files, especially large ones. Because a large-scale attack against Kerberos involves testing a large number of DES keys against a large number of plaintexts, strategies for optimizing parallelized DES operations, such as [3], can also be applied for additional performance gains. Section 3.3 discusses performance and benchmark measurements in more detail.

3 The Experiment

In 1979, Morris & Thompson studied the security of the UNIX `crypt()` function against brute-force dictionary attack [13]. Ten years later, Feldmeier and Karn published a followup paper [5] that charted the progress of password security and cracking over the previous decade. We now attempt to bring the analysis of password cracking methods to the networked world, nearly twenty years after the original Morris & Thompson paper.

Usually, when one discusses password-guessing attacks against network protocols, one assumes that an adversary can at least eavesdrop on legitimate sessions and use that information to verify passwords. In practice, this requires access to some part of the network between client and server. The attack against Kerberos described in the previous section, on the other hand, requires no such access to carry out. Instead, Kerberos allows an attacker to initiate requests for the encrypted TGTs of any or all users in a Kerberos realm, as long as he knows the name or IP address of an authentication server.

The data for this experiment were gathered in this manner from the authentication server of a large Kerberos realm, serving over twenty-five thousand users. A small cluster of three Sun UltraSPARC-2s (200MHz) and five UltraSPARC-1s (167MHz) participated in the off-line password cracking effort using spare CPU cycles. The run was limited to two weeks, at which time the results were collected and tabulated.

3.1 Overall Statistics

From a Kerberos realm containing slightly over twenty-five thousand users, a grand total of 2045 passwords were successfully guessed by the end of the two-week experiment. Table 3 shows the distribution of passwords by length.

Length	Frequency	
2	2	(0.1%)
3	12	(0.6%)
4	77	(3.8%)
5	146	(7%)
6	227	(11%)
7	164	(8%)
8	1108	(54%)
9	159	(8%)
10	92	(4.5%)
> 10	58	(3%)

Table 3: Passwords by Length

The large number of eight-character passwords is a consequence of having some client programs that do not support longer passwords and a password checker that frowns upon shorter ones. Some interesting observations on users' password choices:

- 527 passwords (26%) used at least one digit.
- 84 passwords (4%) used at least one non-alphanumeric symbol.
- 24 passwords (1.2%) were calendar dates. Only one of them was a well-known holiday.
- Users preferred the `mm/dd/yy` notation (21 times) to the `mm-dd-yy` notation (3 times).
- One person used a telephone number as a password.
- Shared passwords abounded: 67 passwords were common to more than one account.
- The most "promiscuous" password was shared by an astounding 53 accounts! It is believed that this statistical anomaly was not the result of voluntary, coincidental user choice.
- Users generally avoided the shift key: 86% of the passwords could be typed without it.

The analysis done here may appear similar to [17], although that study was based on passwords collected directly from users. This experiment, on the other hand, deliberately analyzed only those passwords that were successfully cracked because it sought to evaluate the efficacy of password checking. It should not be surprising that the length distribution exhibited by this study is skewed more strongly towards 8-character and longer passwords, since the password checker would filter out short ones at a disproportionate rate.

3.2 Analysis by Rules

The password cracker combines user-specific information, like the username and full name, with a series of precompiled word lists to obtain a list of password candidates. To each word in this list, it applies a set of transformation rules to generate even more potential passwords. Of the successfully guessed passwords in the experiment, 283 passwords were based on either the corresponding username or some derivation of the person's full name. The remainder originated from one of the word lists.

Only about half of the cracked passwords came *directly* from a word list, however. The remaining "hits" were the result of applying one of the transformation rules to a password candidate. Table 4 shows the effectiveness of various word transformations.

Transformation	Hits	
None	1010	(49%)
Prefix	100	(5%)
Suffix	390	(19%)
Simple	511	(25%)
Other	34	(2%)

Table 4: Transformation Rule Statistics

The "simple" rules include capitalizing, doubling, or reversing a word.

- The most effective single transform simply converted all the letters in a word to lowercase, accounting for 225 hits.
- 36% of the hits from user-specific information came from *other* users' information; clearly it pays to cross-check all dictionaries.
- The digit "1" was by far the most frequently used suffix, occurring 206 times. All other digit suffixes combined totaled only 130 hits.
- "1" was also the most frequent prefix, with 37 hits.
- "2" was a distant second, while "0" and "6" were the least popular.

3.3 Software and Hardware

The preceding experiment used a well-known Internet password cracking package, modified to decrypt and verify Kerberos V4 tickets. This package generates password candidates by reading words from dictionaries and applying the same transformations that users often use to generate their passwords. The list of rules can be extended to accommodate nearly any type of password-generating transformation, including common ones like adding digits to the end of words or substituting digits for letters that resemble them.

The running time of the password cracker is divided into two components. Let k denote the amount of time needed to convert each password guess into a key. Most of this time is taken by the STRING-TO-KEY function of Section 2.4. Let c denote the amount of time needed to apply this key to a password entry and check if the key results in a valid decryption. This time is mostly determined by the speed of the DES decryption code. Table 5 shows the relative performance figures for some well-known hardware platforms.

Platform	k	c
Sun UltraSPARC-1 (167 MHz)	110 μs	4.0 μs
Sun UltraSPARC-2 (200 MHz)	100 μs	3.3 μs

Table 5: Password Cracking Benchmarks

Since Kerberos V4 does not "salt"[2] its password entries, each new password guess requires only a single call to STRING-TO-KEY; the resulting key can be used to verify an arbitrary number of password entries. The amount of time needed to search a Kerberos domain with n users against a dictionary of w words is

$$t = w(k + nc)$$

For twenty-five thousand users and a million-word dictionary, this works out to just under 23 hours on an UltraSPARC-2. Each trial password only requires 0.8 seconds to verify against the entire database.

The lack of salt is just one factor that allows rapid and exhaustive dictionary searches against Kerberos V4 passwords. The format of the TGT, which leads to the optimizations outlined in Section 2.4, also contributes to the fast cracking times. In this two-week experiment, it is estimated that our password cracker verified slightly over 100 million candidate passwords, distributed over the eight different workstations. Although that figure might seem high, it was only a small fraction of the total number of candidate passwords available to the password cracker. Had the experiment been allowed to continue for a greater length of time, a larger portion of the total password space could have been searched, with a correspondingly larger number of cracked passwords.

[2] Salt is a random input to the password-to-key function that makes any password map to a potentially large number of keys. This is done to frustrate attempts at building precompiled password dictionaries.

3.4 Dictionary Construction

An important part of the success of this password-cracking effort was the input dictionary used to construct the password guesses. Because this experiment was conducted at a site that already implemented password-checking, it would have been pointless to try passwords that were already in the password checker's dictionary. It thus improves our chances to have more words in the cracker's dictionary than in the dictionary used to screen passwords. At the same time, it is desirable to ensure that any such words added to the dictionary are still likely password choices.

Most password crackers are bundled with standard dictionaries, which generally contain a modest list of English words, plus a few categories of words that commonly appear in passwords (e.g. science fiction vocabulary, female names, technical terms and jargon). To expand this word list to include words that a password checker might miss, the experiment exploited domain-specific characteristics of the user population.

Users within a particular domain often have commonalities that are reflected in their password choices. Including words that are familiar to such users but not necessarily to those outside the field enhances the effectiveness of password searches, especially if the list of words is comprehensive and up-to-date. For example, a list of company names and stock symbols might be useful against a financial-services system, while a list of popular music album and band names might be useful at a college site.

Today's Internet search engines makes it a simple matter to compile current dictionaries based on nearly any possible category. A quick session using any of the popular search engines, such as Yahoo, Excite, or InfoSeek, can yield a great deal of raw subject-specific text, which can then be filtered and distilled, and the results merged into the existing dictionary. Some of the more productive categories in the experiment contributed over 100 successful password guesses (5%) each. These word lists were compiled directly from the pages of the search engines themselves; a more aggressive strategy (which was not pursued in the experiment) might involve spidering (i.e. recursively traversing) the sites returned by the search engines.

The techniques employed here merely scratch the surface of possible strategies available to password crackers today. The Web alone provides a nearly limitless source of material, which can be screened selectively to suit any user population, or even specific users. Section 4.1 explains some of the implications this has on Kerberos password security.

3.5 Dedicated Key Search

In 1998, the Electronic Frontier Foundation (EFF) constructed a machine that brute-forced DES keys with dedicated hardware at a cost of under $250,000 [6]. According to their most recent results, this machine could search the entire DES keyspace in at most 228 hours, for an average of 114 hours per key. Although the attack of Section 2.4 could be conducted directly by this machine, a design flaw in some STRING-TO-KEY functions permits an optimization that makes this attack much more feasible.

Specifically, for Kerberos V4 implementations that interoperate with the Andrew File System (AFS), the STRING-TO-KEY function performs this following step as part of its password processing: For passwords of 8 characters or less, it passes the string through the UNIX `crypt()` function with a constant salt and uses the first 8 bytes of the output as a DES key. Unfortunately, while the output of `crypt()` represents a 64-bit DES block, this output is encoded as 11 ASCII bytes in a base-64 encoding, not as a pure binary block. By saving only the first 8 bytes of this data, the resulting DES key only has 48 bits of entropy.

Because the 64 possible values of each key byte are fixed, a dedicated DES cracker can be easily be configured to search this reduced key space. The EFF cracker, in its present form, would require at most 54 minutes (average 27 minutes) to compromise these keys. Table 3 shows that 84.5% of the passwords in our cracked sample were 8 characters or less and thus vulnerable to this attack. It is clear from this example that once a weak protocol exposes passwords to any form of brute-force search, the inevitability of highly optimized attacks against the protocol is assured.

4 Devising a Cure

To protect an authentication system from dictionary attacks, one can attempt to make the passwords more difficult to guess, or one can use cryptographic techniques to prevent dictionary attacks from occurring at all. The first approach has been used in the past, but with very limited success. Our experimental data provide some useful insights into the habits of users, and they demonstrate why the prevention of dictionary attacks is best done through careful use of cryptography instead of administrative policy.

4.1 Stronger Passwords?

The experiment targeted a Kerberos realm that already had password strength-checking in place. Despite this, over two thousand passwords were compromised in only two weeks. Compared to previous password case studies [13], the percentage of crackable passwords was in fact lower than it would have been if no such checking were used, yet a password security system that permits this many passwords to fall into the hands of an attacker is unacceptably weak by almost any criterion.

Password checkers generally reject only those passwords that match some well-defined criteria, like being entirely numeric, or containing a dictionary word. They often fail to detect passwords that pass simple algorithmic tests but still have low entropy, like `a1b2c3d4` or `wwxxyyzz` (both these passwords were considered to be "good" by the password-checking software used by the Kerberos realm in Section 3). On the other hand, users are very good at selecting such passwords, because their low entropy makes them easy to remember. The "Simple" collection of transformations in Table 4 included many such examples of "ordered nonsense"; many of the 511 so compromised passwords fell into this category.

Users are also very good at selecting passwords that are just "good enough" to pass whatever checking is in place. This often takes the form of appending or prepending digits and symbols to a dictionary word until it "passes", leading again to a password that is only slightly harder to crack than the original. The number of passwords broken by such rules (Section 3.2) points out the frequency with which this practice occurs.

One might legitimately ask why these dictionary and transformation rules can't be used in the password checking software itself to prevent a cracker from succeeding with them. But this question points out exactly the futility of trying to match dictionaries with attackers. Even if system administrators knew exactly what dictionaries their adversaries were using (which they don't), it is far more difficult to upgrade widely-distributed security databases than it is to add words to a cracker's wordlist. In addition, while the password checker must constantly be upgrading his wordlists, an attacker only needs to build a dictionary when she is about to mount an attack. Thus the attacker has more up-to-date dictionaries, may have access to greater computing power (an unfortunate consequence of Moore's Law - the attacker can choose what computers to employ in the attack), and can thus overwhelm most password-strength measures with less effort than it took to enforce those measures. This form of reverse leverage works in favor of the attacker and all but guarantees that relying on password checks alone will produce unsatisfactory results.

Other methods of password strengthening, including system-generated passwords, periodic password cracking by administrators, and asking users to pick stronger passwords voluntarily, appear to be even less successful [17] in practice. Although individual, security-conscious users can select passwords that are virtually uncrackable, it is perhaps a bit unrealistic to assume that all users possess that level of commitment to system security [7]. Indeed, any form of password checks stringent enough to offer long-term security would tend to pose a significant inconvenience to most casual users, frequently rejecting even reasonable password choices. To maintain the same level of security over any length of time would require the constant review and upgrade of acceptable password standards, with a corresponding reduction in user convenience. While password checking can certainly reduce the number of crackable passwords in a system, it is evident that even a dramatic reduction from, say, ten thousand to two thousand broken passwords still leaves the overall system vulnerable to attack, while gradually eroding usability to unacceptable levels.

4.2 Kerberos V5?

Kerberos V5 introduces *preauthentication*, which requires the user to provide some evidence that she knows the shared key K before the authentication server will issue a TGT. This evidence comes in the form of an encrypted timestamp t:

$$\begin{array}{ccc} C & \xrightarrow{R, E_K(t)} & S \\ C & \xleftarrow{E_K(\text{TGT})} & S \end{array}$$

The server S sends its reply to the client C only if t decrypts to the correct time within some predefined tolerance. Although this prevents an attacker from requesting TGTs, it does not protect against an eavesdropper who captures either $E_K(t)$ or $E_K(\text{TGT})$. Either of those quantities constitutes verifiable plaintext that can be used to mount a dictionary attack. While this is an improvement relative to Kerberos V4, an attacker with a network sniffer can still carry out the same off-line dictionary attack against any authentication requests captured over the network [10]. Kerberos V5 by itself is thus an incomplete solution.

4.3 Stronger Cryptography

Rather than incrementally increasing the difficulty of password cracking, we instead apply a variant of public-key cryptography to eliminate the possibility of dictionary attacks altogether. This approach takes advantage of a password authentication technique known as the Secure Remote Password (SRP) protocol [19]. The SRP protocol authenticates a client to a server without exposing the password to off-line dictionary attack, which preserves security even if the password has low entropy. Such a protocol can be incorporated into Kerberos V5 as a preauthentication mechanism, an idea originally proposed by Jaspan [9].

Instead of using the password itself to encrypt and decrypt the initial TGT, the client and server use the first round of the protocol to negotiate a secure session key:

$$\begin{array}{ccc} C & \xrightarrow{R, A} & S \\ C & \xleftarrow{B} & S \end{array}$$

The quantities A and B are exchanged as part of the authenticated key agreement phase; each side uses the other side's quantity to construct the session key K' [19]. R is the original TGT request packet (see Table 1). After the initial round, the client then proves its knowledge of the session key before receiving the encrypted TGT:

$$\begin{array}{ccc} C & \xrightarrow{M} & S \\ C & \xleftarrow{E_{K'}(\text{TGT})} & S \end{array}$$

M is a function of the shared key K', effectively making it a preauthenticator. Since K' is not derivable from the user's password and other publicly available information, verifiable plaintext in the TGT no longer poses a security risk. A variant of SRP performs the entire secure ticket exchange in a single round, which fits even better with Kerberos V5's preauthentication model. Other strong authentication protocols like EKE [1] and SPEKE [8] can also be used to augment Kerberos. The main cryptographic advantage of SRP over similar past proposals is that SRP stores password verifiers in a form that is not *plaintext-equivalent* to the password. Traditionally, the Kerberos KDC (Key Distribution Center) stores a quantity P for each user that, if revealed publicly, would allow an attacker to compromise that user's account. While EKE and SPEKE would also require the KDC to store P, SRP only requires that the KDC store $V = g^P \pmod{n}$. For well-chosen values of g and n, it is computationally infeasible to extract P from V [11].

By resisting both passive and active dictionary attacks, SRP protects the Kerberos TGT protocol from dictionary attacks launched from anywhere on the network. The advantages of this approach are obvious: Users can use even fairly simple passwords without exposing the system to dictionary attacks, and such a solution can be deployed without modifying or adding a public-key infrastructure. This long-term solution requires no significant change to the Kerberos authentication model and consequently introduces the least inconvenience for both users and administrators.

5 Conclusion

Vulnerability to dictionary attacks has long been an acknowledged weakness in Kerberos [2], yet at the beginning of this experiment, there existed little hard data on its severity. Could simple password-checking prevent a password cracker from revealing any passwords, as some have claimed in the past?

That question was answered exactly nine seconds into the experiment, when the first cracked password appeared on the screen. Two thousand passwords later, it became obvious that another approach was needed to cope with the threat of off-line dictionary attack. Distributed systems need to contend with several long-term trends that magnify this threat:

- Computers are becoming faster and cheaper.
- As cracking ability increases, the minimum entropy needed for a "safe" password rises.
- Human memory is *not* improving to match.
- The average number of users served by each distributed system is increasing (i.e. bigger password databases). This reduces the average amount of CPU time needed to attack each password.
- Attackers can take advantage of both software and hardware improvements more quickly than defenders. This also applies to newer dictionaries and wordlists.

A system that exposes its passwords to dictionary attacks from the network is inherently insecure in the face of these trends. It is not particularly surprising that attempts to compel the use of harder passwords have yielded only modest gains in overall system security, along with some disgruntled users.

Secure password technologies like SRP, on the other hand, prevent dictionary attacks from occurring in the first place. This ensures long-term network security with a one-time deployment effort and little user-visible change. Ultimately, the best solution would involve a combination of SRP and light password strength-checking, instead of relying entirely on a single approach.

References

[1] S.M. Bellovin and M. Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *Proceedings of the 1992 IEEE Computer Society Conference on Research in Security and Privacy*, pages 72–84, 1992.

[2] Steven M. Bellovin and Michael Merritt. Limitations of the kerberos authentication system. In *Proceedings of the 1991 Winter USENIX Conference*, pages 253–267, 1991.

[3] Eli Biham. A fast new des implementation in software. In *Fast Software Encryption 4*, 1997.

[4] Bryn Dole, Steve Lodin, and Eugene Spafford. Misplaced trust: Kerberos 4 session keys. In *Proceedings of the Internet Society Network and Distributed System Security Symposium*, pages 60–70, March 1997.

[5] David C. Feldmeier and Philip R. Karn. Unix password security - ten years later. In *CRYPTO Proceedings*, 1989.

[6] The Electronic Frontier Foundation. *Cracking DES: Secrets of Encryption Research, Wiretap Politics & Chip Design*. O'Reilly, July 1998.

[7] Israel Herschberg. The hackers' comfort. *Computers and Security*, 6(2):133–138, April 1987.

[8] D. Jablon. Strong password-only authenticated key exchange. *Computer Communication Review*, 26(5):5–26, October 1996.

[9] B. Jaspan. Dual-workfactor encrypted key exchange: Efficiently preventing password chaining and dictionary attacks. In *Proceedings of the Sixth Annual USENIX Security Conference*, pages 43–50, July 1996.

[10] Charlie Kaufman, Radia Perlman, and Mike Speciner. *Network Security: Private Communication in a Public World*, pages 310–311. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1995.

[11] B.A. LaMacchia and A.M. Odlyzko. Computation of discrete logarithms in prime fields. *Designs, Codes, and Cryptography*, 1:46–62, 1991.

[12] Philip Leong and Chris Tham. Unix password encryption considered insecure. In *Proceedings of the Winter USENIX Conference*, 1991.

[13] R.H. Morris and K. Thompson. Unix password security. *Communications of the ACM*, 22(11):594, November 1979.

[14] R.M. Needham and M.D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, December 1978.

[15] National Bureau of Standards. Data encryption standard. NBS FIPS PUB 46, January 1977.

[16] Bruce Schneier. *Applied Cryptography*. John Wiley & Sons, Inc., New York, 1996.

[17] Eugene H. Spafford. Observations on reusable password choices. In *Proceedings of the Third Usenix Unix Security Symposium*, pages 299–312, Baltimore, MD, September 1992.

[18] J.G. Steiner, B.C. Newman, and J.I. Schiller. Kerberos: An authentication service for open network systems. In *USENIX Conference Proceedings*, pages 191–202, February 1988.

[19] Thomas Wu. The secure remote password protocol. In *Proceedings of the Internet Society Network and Distributed System Security Symposium*, pages 97–111, March 1998.