1996

Misplaced Trust: Kerberos 4 Session Keys

Bryn Dole

Steve Lodin

Eugene H. Spafford
Purdue University, spaf@cs.purdue.edu

Report Number:
96-078

MISPLACED TRUST: KERBEROS 4 SESSION KEYS

Bryn Dole
Steve Lodin
Eugene H. Spafford

CSD-TR 96-078
December 1996

Misplaced Trust: Kerberos 4 Session Keys *

Technical Report CSD–TR–96–078

Bryn Dole
Sun Microsystems
2550 Garcia Avenue
Mountain View, CA 94043-1100
bryn.dole@sun.com

Steve Lodin
Delco Electronics
One Corporate Center, MS CT200W
Kokomo, IN 46904-9005
swlodin@delcoelect.com

Eugene H. Spafford
COAST Laboratory
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907-1398
spaf@cs.purdue.edu

Abstract

One of the commonly-accepted principles of software design for security is that making the source code openly available leads to better security. The presumption is that the open publication of source code will lead others to review the code for errors. However, this openness is no guarantee of correctness.

One of the most widely-published and used pieces of security software in recent memory is the MIT implementation of the Kerberos authentication protocol. In the design of the protocol, random session keys are the basis for establishing the authenticity of service requests. Because of the way that the Kerberos Version 4 implementation selected its random keys, the secret keys could easily be guessed in a matter of seconds.

This paper discusses the difficulty of generating good random numbers, the mistakes that were made in implementing Kerberos Version 4, and the breakdown of software engineering that allowed this flaw to remain unfixed for ten years. We discuss this as a particularly notable example of the need to examine security-critical code carefully, even when it is made publicly available.

*A version of this paper is to appear in the proceedings of the 1997 Internet Society Symposium on Network and Distributed System Security.

1. Introduction

As we depend more on computing for critical tasks, the security of those systems becomes more significant. Obviously, a key component of the security for those systems is the underlying software that may run with privilege or regulate access. To develop a strong sense of trust in the underlying systems, we must establish trust in the supporting software.

One of the key principles of deploying trustable software is the concept of *open design*. (see the principles described in [21] as reprinted in [7], and the comments about "no security through obscurity" in [9].) This principle states that code — and especially security-critical code — should not depend on the secrecy of the code or algorithm. Code and algorithms can be accidentally disclosed or reverse-engineered, so any security implied by keeping that secret is transitory at best. Furthermore, the assumption is that by publishing security-critical code, others can examine it for flaws and gain confidence in the correctness of the code and the strength of the algorithm.

Although this principle suggest a necessary condition, it is not a sufficient one: It is also necessary that the code be examined carefully and critically by trained observers. Merely making source code available does not, by itself, add any confidence in the underlying correctness of the code. As noted in [6], even formal proofs of correctness require critical review over time to gain any validity.

Unfortunately, having code available for public scrutiny has often resulted in a false sense of security in that code by its users. What is more, as time goes on, this sense often increases — the belief is that if no problems have yet been discovered, that the passage of time increases the likelihood that no problems exist. Recent trends in software development, where flaws are often discovered in software within days (or hours!) of release on the Internet tend to exacerbate this overall problem with legacy software developed and released in earlier times.

In the remainder of this paper we examine a particularly notable example of this overall problem of misplaced trust: the generation of "random" numbers in Kerberos 4. We will begin with a review of the use of random numbers in secure applications, and then discuss the problem in the Kerberos 4 implementation.

2. The Importance of Random Numbers

Random means that, among other things, it should be impossible to guess the next value based on knowledge of past values. This can be achieved in a pseudo-random number generator (PRNG) only if it is based on cryptographic principals and is seeded with a sufficient amount of entropy, or truly random information. A pseudo-random number generator is a function that generates a predictable sequence of numbers that passes certain statistical tests for randomness. True random number generators (RNG) also satisfy tests for randomness but they are not predictable. For cryptography it is desirable to use a truly random number generator, such as one based on measurements of unpredictable physical phenomena, such as radioactive decay.

The following are characteristics of a cryptographically secure random number for secret keys 1. The sequence of random numbers should pass standard statistical tests for randomness. 2. The sequence should be unpredictable. Knowing the algorithm and the previous sequence items should not allow the next item in the sequence to be determined. 3. The sequence can not be reliably reproduced. If the generator is run twice with the same initial conditions, you will get two difference sequences.

The UNIX random number implementations `rand`, `random` and `lrand48` were designed to produce completely reproducible random number streams, violating properties two and three of cryptographically secure random numbers. These functions produce random number sequences for statistical purposes, like random events in computer simulations. They were never intended to generate cryptographically secure random numbers.

Examples of how to develop secure random-sequence numbers are available in RFC 1750 [8], Knuth [13], Schneier [22, pages 421-428] and Garfinkel and Spafford [9, pages 726-731].

2.1. Key Entropy

Entropy, in the cryptographic sense, is the amount of information that a message contains [22, page 233]. The entropy of a random key is the number of bits required to represent all possible keys. Ideally, the entropy of a key is equal to its length. In other words, every single bit is completely random and independent of every other bit in the key.

Entropy is the deciding factor in how difficult keys are to guess. If the entropy is not equal to the size of the key, then some keys are more likely to be chosen than others. This statistical lack of randomness can be exploited to reduce the average number of keys that have to be tested before the actual key is found. For example, if there are 64 possible keys, but only the even numbers are used, then the entropy of any one key is only 5 bits (32 keys used is 2^5) even though the key is 6 bits long, because the lowest order bit is always zero. In this case, the number of keys that need to be tried to guess the key by brute force is halved. This might seem like an absurd example, but the UNIX `random` and `rand` functions both leave the highest-order bit zero so that the negative numbers are never returned, reducing the entropy.

2.2. Common Knowledge

While the computer security community as a whole considers the need for good random number generators common knowledge, the fact that this problem continues to appear indicates that this is still a major challenge confronting the cryptography community.

Little effort has been expended to make good random number generators available to implementors of cryptographic protocols. There is no standard mechanism in operating systems (OS), hardware, or applications for generating satisfactory, cryptographically strong random numbers. While RFC 1750 [8] provides excellent advice on where random information might reside on a computer, little practical information is given. In fact, nearly all of the suggested sources of good random entropy are impossible to collect on modern OSs. Everything that is a potential source of randomness is insulated by abstraction and high level interfaces. For some operating sustems this is more of a problem than others. For example, the Java OS [17], takes abstraction to the extreme. There is no way to access memory locations directly, all input/output is only done through drivers written in Java, and there is no way to get direct access to system statistics. The Java Platform [15] provides yet another difficulty, this defines the minimal Java API that can be safely assumed to exist on any Java machine. The existence of hard drives, or other hardware that could be a source of randomness cannot be presumed.

As suggested in RFC 1750 [8] the user is a good source of randomness. People rarely exhibit deterministic behavior and thus provide a good source of entropy. However, people sleep even if their computers do not. If computers can only get entropy from a user's input, what is the computer to do when the user leaves his terminal? Are random keys to be more secure when the user is present at the keyboard versus when the user goes on vacation for a week? What about servers that never have users directly logged into them? Or firewalls that implement virtual private networks? Most firewalls are designed to not support direct user connections. Or what about the Internet appliances of the future? These devices will have very limited user interfaces. Should users of the future be expected to open and close their refrigerator door 1000 times while the refrigerator collects entropy? User input is a limited resource that cannot be counted on.

Another method for producing secure random numbers is to seed a PRNG with a local secret key. This is the heart of the Kerberos Version 5 random number generator. This works for Kerberos Version 5, because to guess the seed (and thus the state) of the RNG one must first guess the local secret key of the Kerberos server. Of course, if you know the Kerberos ticket granting server (TGS) key, then you do not need to be able to guess session keys, because you can simply create your own tickets. Thus the RNG is equally, if not slightly more secure than the TGS secret key, which is as it should be. However, there is a bootstrapping problem for this technique in general. What happens if the master key, in this case the TGS key, is generated at random using the same RNG? How is the RNG seeded? If there is some other unsecure method for seeding the RNG, then the master key is not truly a secret, and the whole system is vulnerable.

2.3. Algorithm Analysis

While cryptographic protocols and encryption algorithms are subjected to great scrutiny, RNGs rarely are. Protocols are closely studied for weaknesses, and sometimes proven to be secure. The use of random nonces and randomly chosen keys are frequent component of such protocols but their correctness is taken for granted. There are no formal methods for analyzing the entropy of key data.

For example, Bellovin and Merritt examined several problems in the Kerberos protocol in *Limitations of the Kerberos Authentication System* [2]. These weaknesses in the protocol included replay attacks, secure time services, password-guessing attacks, and login spoofing. However, they did not address issues of protocol implementation, such as key randomness in their work. This excellent analysis of the protocol, demonstrates the community's focus on the theoretical vulnerability of systems.

3. Introduction to Kerberos

Kerberos is a secret key network authentication protocol [20] designed at MIT for Project Athena. It is based on the Needham-Schroeder [18] authentication protocol. The goals of Kerberos are authentication, authorization, and accounting. Some of the protocol requirements are that authentication be two-way, no cleartext passwords be transmitted over the net, no cleartext passwords be stored on servers, clients use cleartext passwords for the shortest time possible, authentication have a limited lifetime, and authentication be transparent to the user.

The basic "currency" of the Kerberos authentication protocol is a credential or capability known as a ticket. The possession of a ticket and its accompanying session key determines the ability to use a service. The protocol includes the use of time stamps and identification strings (known as authenticators) to prevent replay and man-in-the-middle attacks. It also includes the ability to use encryption for message authentication and message secrecy.

4. Kerberos Authentication Protocol

Kerberos is a trusted authentication protocol designed for TCP/IP networks. The protocol is described in *Kerberos: An Authentication Service for Open Network Systems* [23] and *Applied Cryptography* [22, pages 566-571]. In the descriptions below, the principal and the client denoted by C are the user. The Kerberos server is also called the Key Distribution Center and is denoted by KDC. The Ticket Granting Server which grants tickets to service principals is denoted by TGS. The service principals are services such as file systems, printers, remote command execution (rsh), remote login (rlogin) and e-mail gateways and are denoted by S. The exchange between the client and the Kerberos server is called the Authentication Service (AS). In the implementation, the KDC and the TGS are integrated in the Kerberos server process.

The following notation is used:

- K_a denotes a secret key owned by agent a.
- $K_{a,b}$ denotes a session key shared by agents a and b.
- $T_{a,s}$ is a ticket granting agent a access to service s.
- A_c is an authenticator containing the name of client c.
- $\{M\}K$ is message M enciphered with key K.

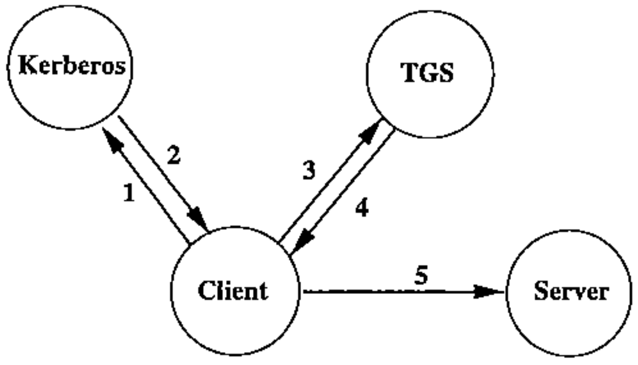

Figure 1. Kerberos Authentication Steps

4.1. Version 4 Protocol

The Kerberos Version 4 protocol consists of five steps in which the client, C, communicates to the Kerberos server, KDC, to get a Ticket Granting Ticket. It then communicates with the Ticket Granting Service, TGS, to get a Ticket to a service principal. Finally, it communicates with the service principal, S. Figure 1 is a pictorial view of the authentication protocol.

1. C $\rightarrow$ KDC: c, tgs
2. KDC $\rightarrow$ C: $\{K_{c,tgs}, \{T_{c,tgs}\}K_{tgs}\}K_c$
3. C $\rightarrow$ TGS: $s, \{T_{c,tgs}\}K_{tgs}, \{A_c\}K_{c,tgs}$
4. TGS $\rightarrow$ C: $\{\{T_{c,s}\}K_s, K_{c,s}\}K_{c,tgs}$
5. C $\rightarrow$ S: $\{A_c\}K_{c,s}, \{T_{c,s}\}K_s$

The random session keys are $K_{c,tgs}$ and $K_{c,s}$ and the ticket granting authority's secret key is K_{tgs}. The ticket is denoted by T and A is an authenticator.

5. Kerberos Session Key Generation

Kerberos makes use of random session keys for authenticating transactions. Knowledge of these secret keys is used as proof of identity and for verifying the authenticity of messages. The secrecy of these keys is paramount to the integrity of the Kerberos system. If one of the keys is compromised then anything authenticated by that key cannot be trusted.

The following sections describe how and when the various versions and implementations of Kerberos generate secret keys.

5.1. Kerberos Version 4 RNG

The session key generation code is implemented in `src/lib/des/random_key.c` in the Kerberos Version 4 source code hierarchy. The `random_key.c` code is fairly straightforward. A pseudo random number generator is seeded each time a session key is generated with the following information bitwise exclusive or'd (XOR):

1. time-of-day seconds since UTC 0:00 Jan. 1, 1970
2. process ID of the Kerberos server process
3. cumulative count of session keys generated
4. fractional part of time-of-day seconds since UTC 0:00 Jan. 1, 1970 in microseconds
5. hostid of the machine on which the Kerberos server is running

Kerberos Version 4 uses the UNIX `random` function to produce the random DES keys. Kerberos generates a random DES key by first seeding the random number generator with a seed chosen as outlined in above, then it makes two calls to the `random` function to get 64 pseudo-random bits. This 64-bit block has every eighth bit set as a parity bit, leaving a 56-bit DES key. The random function relies on a 32-bit seed value to determine the state of the linear feedback shift register used for generating the pseudo random numbers. The seed is composed as described above. Thus, any sequence of numbers created by the random function, no matter how long, relies solely on the 32-bit seed value. The entropy of any number sequence produced by `random()` has an entropy of only 32 bits. Likewise the Kerberos session keys have an entropy of only 32 bits.

The only component of the seed that significantly changes between successive key generations is the microseconds value. This yields a key entropy of about 20 bits (10^6). Other components such as the seconds, the cumulative count of keys generated, and the process

ID only affect the low-order bits. Because these values are XOR'd together any entropy that might have been gained from the other values is washed out by the microseconds value. Unlike the low-order 20 bits, the first twelve bits rarely change and are predictable. This can be seen graphically in Figure 2.

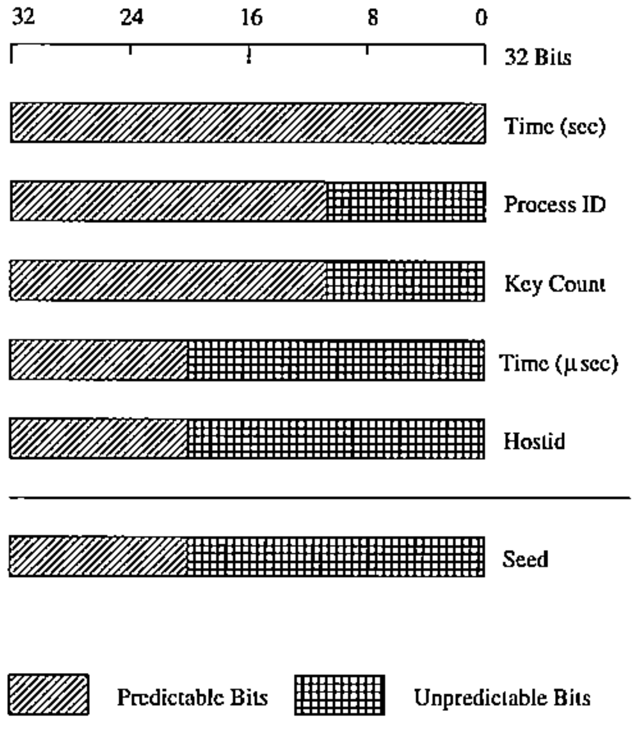

Figure 2. Random Number Generator Seed

As a result of this poor choice in seed values, given knowledge of the approximate time that a key was generated, there are only about 2^{20} (or approximately one million) possible keys.

6. Guessing Random Keys

This section describes the process of guessing Kerberos random keys and discusses how to exploit these weak keys.

6.1. The Naive Brute Force Method

A brute force attack on a key is an exhaustive search of all possible key values until the correct one is found. Brute force attacks typically require a known plaintext and ciphertext pair enciphered with the desired key. The known plaintext is used to identify when the correct key is found. Ideally, the size and entropy of the key is such that it takes a very long time, on the order of 10^{10} years or more, to try all possible keys.

Finding DES keys by brute force has been a hot topic of discussion in the cryptography world. DES keys are 56 bits long and since the adoption of the Data Encryption Standard in 1981, the time and expense required to discover a DES key by brute force has been reduced to alarmingly easy levels. For example, it is estimated that 56-bit keys could be broken in an average of 3.5 hours with $1 million of special hardware [3].

Kerberos uses DES for authentication and encrypting its tickets and session traffic. Someone with enough computing resources, as stated above, could brute force session keys in about 3.5 hours, which is within the lifetime of these keys, and use them. This assumes that the 56-bit DES session keys are chosen completely at random and that the entropy of the keys is 56 bits. In practice, Version 4 falls short of this goal. Each session key has an entropy much smaller than 56 bits. The result is that Version 4 keys are relatively easy to guess using an intelligent brute force attack.

6.2. Brute Forcing Seed Values

The brute force method can be scaled down from 2^{56} possible DES keys to the 2^{32} possible seed values used to generate the keys. This reduces the search by a factor of 2^{24} (or roughly 17 million times). The weakness lies in the fact that the entropy of the DES key cannot excede the entropy of the seed value used to create it. Because the seed is only 32 bits long, the entropy of the resulting keys cannot be more than 32 bits.

Estimated average time to search the 2^{32} possible seed values is about 28 hours on a SPARC 5 and 6 hours on a DEC Alpha. Although 28 hours is longer than the 21 hour lifespan of Kerberos tickets and the 5 minute lifespan of the authenticators, it is still well within the lifetime of other secret keys used by Kerberos services. If the seed itself is not truly random, then the number of seeds that must be tried can reduce the time to search the entire key space.

6.3. Educated Guessing of Seed Values

Trying to guess all possible 2^{32} seed values produces a tremendous savings in time over guessing a full DES key. However, without resorting to using parallelization or supercomputers, session keys and encrypted tickets will expire before they can be successfully decrypted. The solution is to exploit the lack of entropy in the seed values.

Table 1 gives the amount of entropy, in bits, that exists in each component of the random seed. Because all the nonpredictable bits are located in the low-order position of each seed component and the values are XOR'ed together, the total entropy of the seed is equal to that of the component with the largest entropy. Figure 2 shows how the seed value is formed.

Seed Component	Insider	Outsider
time (seconds)	0	0
process ID	0	≤ 10
number of keys	0	≤ 10
time (microseconds)	20	20
hostid	0	20
Total Unknown bits	20	20

Table 1. Entropy of seed components.

It is startling to observe that an inside attacker who has access to the machine on which the Kerberos server is running does not gain any information about the seed value. This is because the values of the process id, hostid, number of keys generated, and current time are all obfuscated by the time in microseconds value. Thus an outside attacker that knows nothing about the machine running the Kerberos server generating the keys is at no disadvantage when it comes to guessing the final seed values.

The hostid is credited with 20 or fewer bits of entropy. This is because the top 12 bits can easily be determined. If the hostid value is not known, then the whole 32-bit seed value must be brute forced once. With this seed value the top 12 bits of the hostid can be determined by XOR'ing the seed with the time in seconds of when the key was generated.

This method takes advantage of the the low entropy of the seed values used to generate the Kerberos keys. With only 20 bits of entropy in each seed, there are only 2^{20} possible seed values and therefore only 2^{20} possible Kerberos keys used. Searching this key space can be accomplished in seconds on common workstations or PCs. Using this method keys can be guessed sufficiently fast enough so that Kerberos session keys will not have expired and can, therefore, be exploited.

6.4. Precomputation Attacks

A precomputation attack can be used to reduce the time for finding individual keys at the expense of some initial computation and storage.

To launch a precomputation attack on Kerberos Version 4, all possible seed values are determined given the set of conditions under which the targeted key will be generated. With knowledge of the approximate time and the hostid of the Kerberos server, only the low-order 20 bits of the seed are unknown. This means that there are 2^{20} possible seed values that could be used to generate a key. Because changes in the low-order 20 bits of the seconds value are obscured by changes in the milliseconds value, this set of seed values is valid for 2^{20} seconds, or approximately 12 days.

All 2^{20} possible keys are generated and used to encipher a known plaintext. The known plaintext is the userid of the targeted victim. The resultant encrypted blocks are stored with their respective keys and sorted by the encrypted blocks. Once a Kerberos transaction occurs on the network that contains one of these encrypted blocks, we can discover the key used by searching the table of preencrypted plaintext blocks and reading out the respective key.

The storage requirements for some precomputation attacks can be very large. For example storing all possible DES encryptions of a fixed block requires over 1 billion gigabytes of storage. However, in the case of Kerberos Version 4 there are only about 1 million possible keys per 12 day period. Each entry in the lookup table needs only 12 bytes, 8 for the encrypted block and 4 for the seed value. The entire lookup table can be stored in 12 Megabytes.

7. Results

This section describes the results from our work. Table 2 shows the results (in seconds) for guessing Kerberos Version 4 randomly generated keys. These results apply to non-parallelized C code running on a single workstation. The times and statistics presented here are the product of successfully gathering Kerberos ciphertext and guessing the DES session key of 400 different Kerberos authentication sessions.

	SPARCStation 5 (seconds)	DEC Alpha (seconds)
minimum	0.2	0.1
maximum	48.7	10.9
mean	24.8	5.5
median	24.8	5.5
standard deviation	13.9	3.1

Table 2. Statistics for guessing a single Kerberos key.

The majority of the time needed to guess each key was spent performing DES decryptions of the ciphertext. All DES operations were done in software, using the `libdes` library. The tests on the DEC machine did not take advantage of the Alpha's 64-bit architecture, suggesting that faster times are possible with 64-bit optimized DES code. Using hardware implementations of DES is another source of potential performance improvements.

Regardless of what other optimizations that might be made, these times for guessing keys are sufficiently fast to be a threat to the security of any system depending on Kerberos Version 4. However, a precomputation attack stands out as an easy, low-cost method for improving performance.

7.1. Precompute Attack

Using the same database of about 400 randomly generated Kerberos keys used for testing the brute force method, our average time to find one key using the precomputation method was about 710 microseconds on a SPARCStation 5. The time to generate the table of ciphertext and seed values was approximately 2.5 minutes and sorting the table took about 5 minutes (both values are for a SPARCStation 5.) The time required for table generation and sorting is a cost that only needs to be repeated once every 12 days when the top 12 bits of the seed value changes.

8. Vulnerable Keys

Random numbers are used in Kerberos in many places. They are used as randomly generated session keys to encrypt traffic between the client and a Kerberos server. In addition, the random numbers are used for random keys for principals and servers in the Kerberos database. In particular, the *krbtgt* and *changepw* principals have random keys. These two principals play important roles in the administration of the Kerberos authentication system. Compromising these keys subverts the security of the whole system.

Any key that is created using the Version 4 `random_key` code is vulnerable to guessing. These keys include all session keys generated by the KDC server and keys generated for the initial Kerberos database. Weak database keys include the Ticket Granting Server's secret key. This server key is used to generate the Kerberos Ticket Granting Tickets (TGT). Once the TGS key is guessed, that key can be used to generate TGT to obtain access to any service that is regulated by that TGS, including administrative commands to the Kerberos server itself.

9. Exploiting Weak Keys

The discovery of the weakness in the implementation of the Kerberos Version 4 random number generation undermines the security of the Kerberos authentication protocol.

9.1. Snooping Encrypted Traffic

When the session key between the user and the service ($K_{c,s}$) is guessed, encrypted traffic between the user and the server is vulnerable to snooping. Encrypted traffic that is "sniffed" and saved is especially vulnerable to disclosure at a later time, once the session key is guessed.

9.2. Masquerading as Another User

When the Ticket Granting Server secret key (K_{TGS}) is guessed, then valid Ticket Granting Tickets can be fabricated without the step of client authentication to the Kerberos server. The Ticket Granting Ticket also contains a session key ($K_{c,TGS}$) that is used to encrypt an authenticator. These Ticket Granting Tickets can then be presented to the TGS requesting access to a service that the user would not normally be granted by the legitimate KDC.

10. Related Work

There have been other incidents of attacks on security systems by exploiting weaknesses in the random number generation routines. In addition, the available patches and fixes from Kerberos vendors are listed.

10.1. Netscape SSL Random Number Guessing Attack

The Netscape SSL random number generation vulnerability received a great deal of coverage in the media because of Netscape's high visibility. The publicity of this bug was one of the motivations that prompted the authors to examine the Kerberos RNG. The Netscape vulnerability was very similar in nature to the weakness in Kerberos Version 4. Netscape used known, deterministic components to seed the random number generator. The Netscape SSL random number guessing attack as described by Goldberg and Wagner [10]. More information is available on the World Wide Web [5].

10.2. X11 MIT-MAGIC-COOKIE-1 Random Number Attack

The MIT-MAGIC-COOKIE-1 random number generation vulnerability was originally discovered by Chris Hall [11] and was discussed in the Best-of-Security mailing list [25]. This is another instance of a poorly implemented random number generation routine used for security purposes. In this case, the random number generator is seeded with the time of day and the process id of the xdm client. Hall also noted that if the DES routines were not compiled into the X11 code, then for some operating systems only 256 possible random magic cookies are generated.

When the magic cookie is guessed, it can be added to the attacker's .Xauthority file (or wherever the attacker keeps his magic cookies). This gives the attacker the ability to connect to the display of the victim. At this point, any of the standard X11 attacks can occur. These include killing X server processes, faking input on behalf of the victim, and even capturing the victim's keystrokes (which might include passwords).

11. Other Implementations of Kerberos

Kerberos has been widely successful and imitated. There is a risk in copying the design of a system too closely, in that the bugs can also be duplicated. For such an "obvious" flaw, the RNG bug found its way into a surprising number of Kerberos Version 4's descendants.

11.1. Kerberos Version 5

A notable exception to the inheritance of the RNG bug is Kerberos Version 5. The new version of Kerberos uses the random number generator that was meant to be adopted by Version 4 years ago. Instead of using the UNIX random function, Version 5 uses the DES encryption algorithm as a mixing function. The state from previous key generations is kept as a seed for generating the next key. In addition, some local information and the time of day is mixed in to add to the entropy. What makes this random number generator strong is that it is initially seeded with the TGS's secret key.

One potential vulnerability of Kerberos Version 5 is the Version 4 backward compatibility mode. This is not a problem though because the new RNG is used.

11.2. SESAME

SESAME is a European Community security project that implements authentication and key exchange. It uses the Needham-Schroeder protocol like Kerberos along with public-key cryptography. According to Schneier [22, page 572], the SESAME key generation algorithm consists of two calls to the UNIX **rand** function, similar to the Kerberos Version 4 algorithm. This yields the same upper bound of 2^{32} possible values for the seed and resulting key. However, SESAME documentation available on the Web indicates that some of the SESAME components are accessible through the Kerberos V5 protocol (as specified in RFC 1510), and would use Kerberos data structures [14]. The architects of SESAME explained that Schneier's criticisms were based on exportable versions of the code using XOR, not the real code which uses DES encryption [16].

The random number generation code in SESAME V4, recently released to the public, is more secure than as described in Schneier's book. The algorithm takes the time in seconds, the process id, the number of clock ticks since last reboot, some constants, and mixes them up using shifting and exclusive or functions. This is then diffused through MD5.

11.3. Cygnus Network Security

Cygnus Kerberos, also known as Cygnus Network Security (CNS), is based on MIT Kerberos Version 4 and exhibits the same vulnerabilities. Further exacerbating the problem in CNS, for many platforms, the hostid command is not used as a component of the seed. These platforms include Linux, SCO, HP-UX, and Sun Solaris. When the hostid is not included as a component of the seed, the only component that determines the first 12 bits of the seed is the time in seconds since 1970. Without the usage of the hostid, guessing the key by an outsider is made easier by eliminating the need to determine the hostid either by brute forcing the entire seed value once to determine the top 12 bits or by social engineering.

11.4. Other Commercial Versions

OSF DCE Version 1.1 contains Kerberos Version 5. Vendors include HP, IBM, DEC, Open*Vision Technologies and Sun. Transarc claims their AFS product, which is based on Kerberos Version 4, was fixed in 1990. They claim their DCE and Encina products are not vulnerable to this key guessing attack. CyberSafe sells primarily Kerberos Version 5, but they also sell products based on Kerberos Version 4. CyberSafe claims they fixed the problem with Version 4 more than a year ago.

Another product built on the Kerberos Version 5 authentication protocol is NetCheque [19]. It is being developed at the University of Southern California Information Sciences Institute. One of the primary developers is B. Clifford Neuman who was also one of the primary developers of Kerberos. Because it is based on Kerberos Version 5, it does not suffer from a poor random number generator.

12. Patches and Fixes

Since the public release of information pertaining to the Kerberos Version 4 random number generation vulnerability, many vendors have released patches. MIT has released a patch that replaces all calls to the weak random number generation routine with calls to the Kerberos Version 5 random number generation routine. Instructions to download the patch are available via anonymous ftp from `athena-dist.mit.edu` in the `/pub/kerberos` directory. This will install changes to various Kerberos modules to upgrade them to use `des_new_random_key()`. It also will install a new program, `fix_kdb_keys`. The `fix_kdb_keys` program, which runs on the KDC server, will update the *krbtgt* and *changpw* keys to new values using the new random number generator. The only client program modified is `ksrvutil` which is used to generate new server keys. All other client/server programs are unaffected.

Cygnus has fixed their version of Kerberos Version 4 (Cygnus Network Security) and is releasing an updated distribution to their customers. The patched version uses the secure random number generator previously distributed, but never properly used. All randomly generated keys (including session keys, and the *krbtgt* and *changepw* service principal keys) now have more than 20 bits of entropy. Because CNS is based on MIT Kerberos Version 4, the CNS patch does the same thing as the MIT patch, except without the `fix_kdb_keys` program. More information is available from their Web site at `http://www.cygnus.com`.

HP has released a patch for their DCE product based on Kerberos Version 5. This information is available in the Hewlett-Packard Security Bulletin HPSBUX9602-030 [12].

There are several solutions to the vulnerability we have described. The most basic solution is to install the patch for Version 4 and to make sure that all random keys generated with the flawed random number generator are destroyed. The next recommended solution is to migrate to Kerberos Version 5, which does not suffer from the vulnerability outlined in this paper.

13. What Went Wrong and Why

It is surprising that the RNG bug in Kerberos Version 4 existed for so long. The cause is purely a social engineering failure. The bug was known to the developers but somehow never got fixed in the end product.

13.1. What Went Wrong

In 1988 Ted Anderson noticed that there was an alarmingly high collision rate with keys generated by the Kerberos RNG [1]. What he noticed was that after generating a table of 170 thousand unique keys, every key generated after that had approximately a one in five chance of already being in the table. If one extrapolates these data one gets a key space of only about 850 thousand (or $2^{19.7}$) keys. This implies that there are only about 19.7 bits of entropy in the each key, very close to the 20 bits of entropy that were determined by analyzing the RNG code.

One suggestion that Anderson made to fix this problem was to initialize the RNG only once, instead of with every call to the RNG, increasing the number of possible keys generated to 2^{31}. This would force an attacker attempting to guess Kerberos keys to keep track of how many keys had been generated. This would actually decrease the entropy of of the keys because the method for choosing the initial seed was not changed. If implemented, guessing the entire key stream would be only slightly harder than guessing just one key from the old RNG. Thankfully, there is no sign that this "fix" was ever introduced to the Kerberos source release.

It was not until a year later that new code was added to improve the RNG. Even then, the problem was not solved. While the new code was technically superior and much more secure, it was never utilized. This new RNG was eventually carried over to the new Kerberos Version 5 project and correctly used, however in Version 4 the new RNG went unused until 1996 when a security patch was released.

13.2. Why

Because code had been checked in that was suppose to have fixed the RNG, everyone assumed that it worked. The documentation supported this idea; there was a new module called `new_rnd_key.c` and e-mail stating that the bug had been fixed was posted to the Kerberos developers' mailing list. However, in actuality, the bug was still there.

The circumstances that allowed this to happen were a breakdown in the social and software engineering processes. In 1989, the Kerberos Version 4 project was drawing to a close. People who formerly worked on Kerberos Version 4 development were migrating to the Version 5 project. The code review and quality control processes for Version 4 simple broke down for lack of attention and staffing. Clearly, no one attempted to verify that the fix worked, because if they had they would have discovered that the old RNG was still being used. To make matters worse, owners of checked in code were responsible for getting their own code reviewed. The owner of the RNG bug fix code was, for some reason, unsuccessful in getting his new code properly reviewed.

The perseverance of this bug can be attributed to several factors including poor code legibility, obfuscated function calls, and leftover dead code. Anyone who has looked at the Kerberos RNG code, new or old, can testify that the code is difficult to follow, despite the straight-forward algorithms. To make matters worse the function names of the RNG are renamed with a `#define` in one of the included header files. Thus anyone searching for the name of the RNG function would never find where it is called, unless they knew the function had been renamed. Finally, the existence of old legacy code made it possible for the incorrect function calls to continue to work and go undiscovered.

13.3. Lessons Learned

The important lesson to be learned here is that software engineering is important. If code reviews had been more strictly enforced or regression testing had been performed, this bug would have died back in 1989. Communication and social skills are also important to the success of projects [4]. Perhaps if the communication between the writer of the new random number generator and the rest of the development team had been better, his code would have been reviewed more carefully and installed properly. It is not sufficient to have brilliant programmers that can write excellent code; they must also possess the social skills to see that their code gets properly integrated.

13.4. Timeline of Events

The following is a reconstruction of the timeline of events surrounding the Kerberos Version 4 RNG problems.

- Sept 1988, Ted Anderson points out the problems in the `random_key()` routine [1]
- Jan 1989, John Kohl checks in the new RNG code to the Kerberos source tree, but is never used.
- Mar 1991, Ted Ts'o encourages users of Kerberos to review the source code [24]:

  > We encourage people to at least look over the source code of what they FTP over; and if they want to, they're perfectly welcome to perform a security audit over the code.

- Jun 1991, Kerberos Version 5 public beta is announced.
- 1992, Ted Ts'o makes the first reference to the new RNG code in the get_srvtab administration utility.
- Oct 1995, Steve Lodin rediscovers the usage of the bad RNG in Kerberos Version 4.
- Jan 1996, Bryn Dole joins the team to write the code exploiting the bad RNG.
- Jan 1996, Gene Spafford sends an e-mail to the COAST sponsors about the potential vulnerability in Kerberos Version 4.
- Feb 1996, Gene Spafford sends a similar e-mail to the FIRST members.
- Feb 1996, various vendors start distributing patches to fix the Version 4 RNG.
- Feb 1996, the Wall Street Journal prints an article about the Kerberos Version 4 RNG weakness, followed by many additional articles in trade magazines.

14. Conclusion

The method that Kerberos Version 4 used to generate random keys for services and session keys was flawed. The mistake was that the random number generator used was not cryptographically secure. Pseudorandom number generators are not sufficiently random for secure keys because they are too deterministic and predictable. This predictability makes guessing keys from a pseudo-random number generator much easier than attempting to naively brute force the entire key space. In the case of Kerberos Version 4, keys can be guessed in seconds, allowing an attacker to make use of the key and subvert the Kerberos authentication system.

Many security experts are proponents of forcing computer security developers to open their algorithms, designs and source code for review. Programs like PGP are available in source form and some vendors have their code reviewed by outside experts in the field.

Still, many developers do not in an attempt to protect their intellectual property or in an attempt to seek security through obscurity. Peer review might have caught some past implementation flaws, such as the flaw in the Netscape SSL random number generator. However, Kerberos serves as a testament that public peer review is not a perfect solution.

During its lifetime Kerberos has been studied, modified, cleaned up, and ported to a number of operating system systems. Public scrutiny is no substitute for structured code reviews, good software engineering practices and quality testing. More significantly, users should bear in mind that *public availability* of source code does not imply *public scrutiny*. Code that is badly structured, poorly written, frequently modified, and insufficiently documented may not be scrutinized at all.

While the Kerberos protocol, which is based on Needham-Schroeder authentication, is fundamentally sound, the Kerberos Version 4 implementation of the protocol was faulty. This suggests that the computer security community may spend too much time validating algorithms and too little time verifying implementations. This should serve as a warning to security developers everywhere.

Also, contrary to popular belief, RNG vulnerabilities are not old news. This problem is still with us and the more we attempt to trivialize it, the more it will be overlooked and come back to haunt us. Hardware and operating system designers do not understand the need for good random number generators and do not implement them. Knowledgeable cryptographers are then forced to make concessions to support portability and reliability of their code. Meanwhile, designers oblivious to the nuances of the field of cryptography are left to invent more poor RNGs or use the woefully inadequate system calls that are provided them. Either way, we are likely to see the past repeat itself until we have some real random numbers provided at the operating system level. The only way to do that is to add specialized hardware to every motherboard or CPU.

We hope that by documenting this vulnerability, the flawed design and the errors made when fixing the bug, that this can serve as an example and warning. We hope to raise the awareness of the importance that random numbers and software engineering play in creating secure programs.

15. Acknowledgments

We thank Dr. Samuel Wagstaff for the instruction he has given us in the cryptography classes while at Purdue University. All the members of the COAST Lab, including Mark Crosbie, Christoph Schuba, and Ivan Krsul, receive our praise for providing support, endless reviews of this paper and hours of riveting discussion. Kudos are also in order for Susan Baldry, Martyn Perry and the many others that took the time to help proofread this paper. Special thanks to Sam Hartman, Clifford Neuman and Theodore Ts'o for helping to provide the MIT side of the story, and for their many technical comments on this paper.

References

[1] T. Anderson. random_key(). `http://www.mit.edu:8008/menelaus.mit.edu/kerberos/487`, Sept. 1988.

[2] S. M. Bellovin and M. Merritt. Limitations of the Kerberos authentication system. In *USENIX Conference Proceedings*, pages 253–267, Dallas, TX, Winter 1991. USENIX.

[3] M. Blaze, W. Diffie, R. L. Rivest, B. Schneier, T. Shimomura, E. Thompson, and M. Wiener. Minimal key lengths for symmetric ciphers to provide adequate commercial security. 1996.

[4] F. P. Brooks Jr. *The Mythical Man-Month*, chapter Why Did the Tower of Babel Fail? Addison-Wesley, Menlo Park, CA, anniversary edition edition, 1995.

[5] L. Demailly. Netscape security (problems). `http://hplyot.obspm.fr/~dl/netscapesec/`, 1995.

[6] R. A. DeMillo, R. J. Lipton, and A. J. Perlis. Social processes and the proofs of theorems and programs. *Commun. ACM*, 22(5):271–280, May 1979.

[7] D. E. R. Denning. *Cryptography and Data Security*. Addison Wesley, 1982.

[8] D. Eastlake, S. Crocker, and J. Schiller. Randomness recommendations for security. Request for Comments (Informational) RFC 1750, Internet Engineering Task Force, Dec. 1994.

[9] S. Garfinkel and G. Spafford. *Practical UNIX & Internet Security*. O'Reilly & Associates, Inc, Sebastopol, CA, USA, 2nd edition, 1996.

[10] I. Goldberg and D. Wagner. Randomness and the netscape browser. *Dr. Dobb's Journal*, Jan. 1995.

[11] C. Hall. MIT-MAGIC-COOKIE-1 random number generator problems. E-mail correspondence, Feb. 1996.

[12] Hewlett-Packard. Security Bulletin: HPSBUX9602-030. `http://us.external.hp.com/search/bin/wwwsdoc.pl?DOCID=HPSBUX9602-030`, Feb. 1996. Security Vulnerability DCE Security Service session key generationn.

[13] D. Knuth. *The Art of Computer Programming, Vol. II: Seminumerical Algorithms*. Addison-Wesley, Reading, MA, 1973.

[14] J. Kohl and B. C. Neuman. The Kerberos Network Authentication Service (V5). Request for Comments (Proposed Standard) RFC 1510, Internet Engineering Task Force, Sept. 1993.

[15] D. Kramer. The Java Platform. White Paper, Sun Microsystems, Mountain View, CA, May 1996.

[16] J. Lebastard. Sesame security issues. E-mail correspondence, Jan. 1996.

[17] P. W. Madany. JavaOS: A Standalone Java Environment. White Paper, Sun Microsystems, Mountain View, CA, May 1996.

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

[19] B. C. Neuman and G. Medvinsky. Requirements for Network Payment: The Netcheque Perspective. In *Proceedings of IEEE COMPCON'95*. IEEE, Mar. 1995.

[20] B. C. Neuman and T. Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications Magazine*, 32(9):33–38, Sept. 1994.

[21] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sept. 1975.

[22] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C.* John Wiley & Sons, Inc, New York, NY, USA, 2nd edition, 1996.

[23] J. G. Steiner, B. C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Winter 1988 USENIX Conference*, pages 191–201, Dallas, TX, 1988. USENIX Association.

[24] T. Ts'o. Re: Integrity of MIT source. `http://www.mit.edu:8008/menelaus.mit.edu/kerberos/1293`, Mar. 1991.

[25] Unknown. X11 mit-magic-cookie-1 random number weakness. E-mail correspondence to the Best-of-Security mailing list, 1995. Documents the X11 MIT-MAGIC-COOKIE-1 random number generator weakness.