Care and Feeding of Passwords

Andreas Zeller/Stephan Neuhaus

Lehrstuhl Softwaretechnik
Universität des Saarlandes, Saarbrücken

The Menu

- Risks of Using Passwords

The Menu

- Risks of Using Passwords
- How to Steal ATM PINs

The Menu

- Risks of Using Passwords
- How to Steal ATM PINs
- Storing Passwords

The Menu

- Risks of Using Passwords
- How to Steal ATM PINs
- Storing Passwords
- Cracking Passwords, Brute-Force and Otherwise

The Menu

- Risks of Using Passwords
- How to Steal ATM PINs
- Storing Passwords
- Cracking Passwords, Brute-Force and Otherwise
- Back-of-Envelope Brute-Force Cost Calculation

The Menu

- Risks of Using Passwords
- How to Steal ATM PINs
- Storing Passwords
- Cracking Passwords, Brute-Force and Otherwise
- Back-of-Envelope Brute-Force Cost Calculation
- Defenses Against Atacks: Salting, Longer Passwords

The Menu

- Risks of Using Passwords
- How to Steal ATM PINs
- Storing Passwords
- Cracking Passwords, Brute-Force and Otherwise
- Back-of-Envelope Brute-Force Cost Calculation
- Defenses Against Atacks: Salting, Longer Passwords
- Passwords are Doomed

The Menu

- Risks of Using Passwords
- How to Steal ATM PINs
- Storing Passwords
- Cracking Passwords, Brute-Force and Otherwise
- Back-of-Envelope Brute-Force Cost Calculation
- Defenses Against Atacks: Salting, Longer Passwords
- Passwords are Doomed
- Eliminating Passwords

What is a Password?

A *password* is a piece of knowledge that you and another party share

What is a Password?

A *password* is a piece of knowledge that you and another party share

Whenever you want to communicate with that other party, you divulge that shared secret

What is a Password?

A *password* is a piece of knowledge that you and another party share

Whenever you want to communicate with that other party, you divulge that shared secret

You are thereby *authenticated*

What is a Password?

A *password* is a piece of knowledge that you and another party share

Whenever you want to communicate with that other party, you divulge that shared secret

You are thereby *authenticated*

A password is an *authenticator*

Risks of Using Passwords (1)

If anyone but you gets your authenticators, your account no longer belongs to you (this is true not only for passwords)

How to give away your password:

Risks of Using Passwords (1)

If anyone but you gets your authenticators, your account no longer belongs to you (this is true not only for passwords)

How to give away your password:

- it could be observed during entry

Risks of Using Passwords (1)

If anyone but you gets your authenticators, your account no longer belongs to you (this is true not only for passwords)

How to give away your password:

- it could be observed during entry
- you could give it away voluntarily

Risks of Using Passwords (1)

If anyone but you gets your authenticators, your account no longer belongs to you (this is true not only for passwords)

How to give away your password:

- it could be observed during entry
- you could give it away voluntarily
- you could give it away because someone puts a gun to your head

Risks of Using Passwords (1)

If anyone but you gets your authenticators, your account no longer belongs to you (this is true not only for passwords)

How to give away your password:

- it could be observed during entry
- you could give it away voluntarily
- you could give it away because someone puts a gun to your head
- you have written down the password somewhere and the piece of paper gets stolen (or copied)

Risks of Using Passwords (1)

If anyone but you gets your authenticators, your account no longer belongs to you (this is true not only for passwords)

How to give away your password:

- it could be observed during entry
- you could give it away voluntarily
- you could give it away because someone puts a gun to your head
- you have written down the password somewhere and the piece of paper gets stolen (or copied)
- it could be guessed if it is easily guessable

Risks of Using Passwords (2)

- it could be so short that an exhaustive search will quickly find it

Risks of Using Passwords (2)

- it could be so short that an exhaustive search will quickly find it
- the password could be stored somewhere in clear text and this clear text copied

Risks of Using Passwords (2)

- it could be so short that an exhaustive search will quickly find it
- the password could be stored somewhere in clear text and this clear text copied
- the password could be stored encrypted but the encryption might be breakable (or there might be other problems with the encryption)

ATM PIN Fraud

The PIN entry terminals for ATMs are highly secure. Still, some fraudsters managed to copy ATM cards and their associated PINs without the victim being aware of that:

ATM PIN Fraud

The PIN entry terminals for ATMs are highly secure. Still, some fraudsters managed to copy ATM cards and their associated PINs without the victim being aware of that:

First, the fraudsters attached a small device to the side of the ATM and observed the electromagnetic signals emanating from it when the card was swiped through the card reader. This was enough to create a copy of the card. Still, they needed the PIN in order to impersonate the customer.

How would you do it?

Methods to Get The PIN (1)

- Threaten the customer with a weapon (disadvantage: the customer then knows that his PIN is no longer secure)

Methods to Get The PIN (1)

- Threaten the customer with a weapon (disadvantage: the customer then knows that his PIN is no longer secure)
- Stand close behind the person when they're entering their PIN and observe the PIN directly.

Methods to Get The PIN (1)

- Threaten the customer with a weapon (disadvantage: the customer then knows that his PIN is no longer secure)
- Stand close behind the person when they're entering their PIN and observe the PIN directly.
- A slight variation: mount a small camera so that it can view the PIN entry terminal.

Methods to Get The PIN (1)

- Threaten the customer with a weapon (disadvantage: the customer then knows that his PIN is no longer secure)
- Stand close behind the person when they're entering their PIN and observe the PIN directly.
- A slight variation: mount a small camera so that it can view the PIN entry terminal.
- Wait for an elderly person to actually ask you to enter their PIN for them (it happens).

Methods to Get The PIN (2)

- Prepare the keyboard of the PIN entry terminal with a special dust that visibly changes its configuration when it's touched by something. (For example, try graphite.) That gives you the digits.

Methods to Get The PIN (2)

- Prepare the keyboard of the PIN entry terminal with a special dust that visibly changes its configuration when it's touched by something. (For example, try graphite.) That gives you the digits.

  In order to find out which of the $4! = 24$ permutations is the correct one, try two at a time, wait a month (why?), try another two etc. until you find the right one. (Failed attempts are logged, but the customer will be unaware of them.)

Methods to Get The PIN (2)

- Prepare the keyboard of the PIN entry terminal with a special dust that visibly changes its configuration when it's touched by something. (For example, try graphite.) That gives you the digits.

  In order to find out which of the $4! = 24$ permutations is the correct one, try two at a time, wait a month (why?), try another two etc. until you find the right one. (Failed attempts are logged, but the customer will be unaware of them.)

- The above can also be (and has been) tried with infrared cameras observing residual warmth on the keys. That will also give you the correct permutation.

Giving the Password Away

Giving the password away, either voluntarily or involuntarily, or having it stolen when it’s written down somewhere is really outside the scope of this lecture.

The only defense against that is to *educate* your users and having a good *security policy* in place that is *consistently enforced*.

Summary So Far

- A password is an authenticator.

Summary So Far

- A password is an authenticator.
- It is a shared secret.

Summary So Far

- A password is an authenticator.
- It is a shared secret.
- Therefore, *both* parties (the authenticating service and the authenticated person) must make sure to keep the password safe.

Summary So Far

- A password is an authenticator.
- It is a shared secret.
- Therefore, *both* parties (the authenticating service and the authenticated person) must make sure to keep the password safe.
- Educate your users to choose good passwords and keep them safe.

How to Keep Passwords Safe

- Choose good, unguessable passwords

How to Keep Passwords Safe

- Choose good, unguessable passwords
- Protect them during entry

How to Keep Passwords Safe

- Choose good, unguessable passwords
- Protect them during entry
- Store them in encrypted form (but do it right)

Password Storage: Turning Echo Off (1)

"Local Echo" is the name for the process of printing the character that you have just typed to the screen. For obvious reasons, local echo should be disabled when entering passwords.

Under Unix (Linux), using bash or in a boune-shell script:

```
stty -echo       # Turn echo off
stty echo        # Turn echo back on
read -s somevar # Read variable "somevar" without echo
```

Password Storage: Turning Echo Off (2)

Under Linux, in C, using *ioctl*(2):

```
#include <sys/ioctl.h>
#include <sys/types.h>

void echo_off(int fd) {
  struct sgttyb tdata;

  if (ioctl (fd, TIOCGETP, &tdata) == -1)
    error ("can't get terminal parameters");

  tdata.sg_flags &= ~ECHO; /* Use |= ECHO to turn echo back on. */
  /* tdata.sg_flags |= CBREAK; */

  if (ioctl (fd, TIOCSETP, &tdata) == -1)
    error ("can't set terminal parameters");
}
```

Password Storage: Turning Echo Off (3)

Under Linux, in C, using *tcgetattr*(3):

```c
#include <termios.h>
#include <unistd.h>

void echo_off(int fd) {
  struct termios ios;

  if (tcgetattr(fd, &ios) == -1)
    error ("can't get terminal parameters");

  ios.c_lflag &= ~ECHO;

  if (tcsetattr(fd, &ios) == -1)
    error ("can't set terminal parameters");
}
```

Password Storage: Cleartext

Storing passwords in clear text is *never* advised.

Passwords should be stored encrypted, but don’t do this:

```c
#include <string.h>

typedef unsigned char key_t[8];

extern key_t lookup_master_key(void);
extern char *decrypt(char *ciphertext, key_t key);

bool check_password(const char *given_password,
                    const char *encrypted_password) {
  key_t master_key = look_up_master_key();
  char *plaintext_password = decrypt(encrypted_password, master_key);

  return strcmp(plaintext_password, given_password) == 0;
}
```

Password Storage: Cleartext

Storing passwords in clear text is *never* advised.

Passwords should be stored encrypted, but don't do this:

```c
#include <string.h>

typedef unsigned char key_t[8];

extern key_t lookup_master_key(void);
extern char *decrypt(char *ciphertext, key_t key);

bool check_password(const char *given_password,
                    const char *encrypted_password) {
  key_t master_key = look_up_master_key();
  char *plaintext_password = decrypt(encrypted_password, master_key);

  return strcmp(plaintext_password, given_password) == 0;
}
```

This needs the master key in plaintext stored somewhere

Storing Encrypted Passwords (1)

Don't use the password as the *data* to en/decrypt, use it as the *key* to encrypt a known plaintext block:

```
#include <string.h>

typedef unsigned char key_t[8];

extern key_t make_key(const char* key_material);
extern char *encrypt(char *plaintext, key_t key);

static const char *block = "AAAAAAAA";

bool check_password(const char *given_password,
                    const char *encrypted_password) {
  key_t key = make_key(given_password);
  char *encrypted_given_password = encrypt(block, key);

  return strcmp(encrypted_given_password, encrypted_password) == 0;
}
```

Storing Encrypted Passwords (2)

The following encryption algorithm appears in Microsoft SQL Server:

1. Convert the password to UTF-16, an encoding of Unicode. Because of some peculiarities of UTF-16 and Unicode, the effect is the same as as putting each ASCII character right-justified into a 16-bit field. The must significant 8 bits will be zero.

Storing Encrypted Passwords (2)

The following encryption algorithm appears in Microsoft SQL Server:

1. Convert the password to UTF-16, an encoding of Unicode. Because of some peculiarities of UTF-16 and Unicode, the effect is the same as as putting each ASCII character right-justified into a 16-bit field. The must significant 8 bits will be zero.
2. For each byte b of the password, swap the most significant nibble (four bits) with the least significant nibble.

Storing Encrypted Passwords (2)

The following encryption algorithm appears in Microsoft SQL Server:

1. Convert the password to UTF-16, an encoding of Unicode. Because of some peculiarities of UTF-16 and Unicode, the effect is the same as as putting each ASCII character right-justified into a 16-bit field. The must significant 8 bits will be zero.
2. For each byte b of the password, swap the most significant nibble (four bits) with the least significant nibble.
3. Set $b \leftarrow b \oplus$ 0xa5.

Storing Encrypted Passwords (2)

The following encryption algorithm appears in Microsoft SQL Server:

1. Convert the password to UTF-16, an encoding of Unicode. Because of some peculiarities of UTF-16 and Unicode, the effect is the same as as putting each ASCII character right-justified into a 16-bit field. The must significant 8 bits will be zero.
2. For each byte b of the password, swap the most significant nibble (four bits) with the least significant nibble.
3. Set $b \leftarrow b \oplus 0\mathrm{xa5}$.

Now, since there is no *secret* involved, this is at most an *encoding*, but not an *encryption*. It is totally reversible without knowing any secrets.

Storing Encrypted Passwords (3)

If the encryption method is good (Unix used 25 rounds of the DES which is about as good as they come), the password file can be stolen without ill effects

Storing Encrypted Passwords (3)

If the encryption method is good (Unix used 25 rounds of the DES which is about as good as they come), the password file can be stolen without ill effects

Therefore, this is a secure method of storing passwords

Storing Encrypted Passwords (3)

If the encryption method is good (Unix used 25 rounds of the DES which is about as good as they come), the password file can be stolen without ill effects

Therefore, this is a secure method of storing passwords

Right?

Storing Encrypted Passwords (3)

If the encryption method is good (Unix used 25 rounds of the DES which is about as good as they come), the password file can be stolen without ill effects

Therefore, this is a secure method of storing passwords

Right? Right?!

Passwords: Some Simple Theory

Let Σ be an alphabet. For example, Σ could be the set of printable seven-bit ASCII characters, or the set of lower-case alphabetic ASCII characters. Let the maximum length of a password be n.

7-bit ASCII subsection	Range	$\|\Sigma\|$
All printable	32 (' ') to 126 ('~')	95
Letters and digits	48 ('0') to 57 ('9'); 65 ('A') to 90 ('Z'); 97 ('a') to 122 ('z');	62
Letters	65 ('A') to 90 ('Z'); 97 ('a') to 122 ('z');	52
Lowercase letters	97 ('a') to 122 ('z');	26

Number of Passwords

There are $|\Sigma|^k$ possible strings of length k made out of characters of Σ ($0 \le k \le n$).

There are $\sum_{k=0}^{n} |\Sigma|^k = \left(|\Sigma|^{n+1} - 1\right) / \left(|\Sigma| - 1\right)$ possible passwords of length at most n. Unix used to have $n = 8$:

ASCII subsection	Number of Passwords	Fraction
All printable	6704780954517121	1.0
Letters and digits	221919451578091	0.033
Letters	54507958502661	0.00813
Lowercase letters	217180147159	0.0000324
Lowercase words(*)	96099	0.0000000000143

(*) Number of words in a word list made from Webster's that are eight characters or less. Webster's has 311141 words total.

Attacking the Encrypted Scheme

We are somehow in posession of a file containing user names and encrypted passwords. The cryptographic algorithm (that we know) is so strong that we cannot break the encryption. How can we still try to get the plaintext passwords?

Attacking the Encrypted Scheme

We are somehow in posession of a file containing user names and encrypted passwords. The cryptographic algorithm (that we know) is so strong that we cannot break the encryption. How can we still try to get the plaintext passwords?

By trying all possible plaintext passwords, encrypting them and seeing if any of our encrypted passwords match.

Attacking the Encrypted Scheme

We are somehow in posession of a file containing user names and encrypted passwords. The cryptographic algorithm (that we know) is so strong that we cannot break the encryption. How can we still try to get the plaintext passwords?

By trying all possible plaintext passwords, encrypting them and seeing if any of our encrypted passwords match.

If no special precautions are taken, we can do this *offline* and thus precompute a dictionary of encrypted passwords (Dictionary Attack).

Storage Estimate for Dictionary Attack

Assume that both the plaintext password and the corresponding encrypted password each need eight bytes to store. With 6704780954517121 plain/ciphertext pairs, that would be 107276495272273936 bytes, or about 2^{53} bytes (about 8000 Terabytes).

This would be just about feasible for a really large organization or government today.

But for everone else, this would be out of reach.

Storage Estimate for Dictionary Attack

Assume that both the plaintext password and the corresponding encrypted password each need eight bytes to store. With 670478095451 7121 plain/ciphertext pairs, that would be 107276495272273936 bytes, or about 2^{53} bytes (about 8000 Terabytes).

This would be just about feasible for a really large organization or government today.

But for everone else, this would be out of reach.

So this attack is not feasible and passwords are secure from this attack.

Storage Estimate for Dictionary Attack

Assume that both the plaintext password and the corresponding encrypted password each need eight bytes to store. With 670478095451 7121 plain/ciphertext pairs, that would be 107276495272273936 bytes, or about 2^{53} bytes (about 8000 Terabytes).

This would be just about feasible for a really large organization or government today.

But for everone else, this would be out of reach.

So this attack is not feasible and passwords are secure from this attack.

Right?

Storage Estimate for Dictionary Attack

Assume that both the plaintext password and the corresponding encrypted password each need eight bytes to store. With 6704780954517121 plain/ciphertext pairs, that would be 107276495272273936 bytes, or about 2^{53} bytes (about 8000 Terabytes).

This would be just about feasible for a really large organization or government today.

But for everone else, this would be out of reach.

So this attack is not feasible and passwords are secure from this attack.

Right? Right?!

Password Quality (1)

First Study by Robert T. Morris (once head scientist at the NSA) and Thompson (co-inventor of Unix) in *Communications of the ACM* **22**(11), November 1979, pp. 594–597.

They tested 3289 passwords and here is what they found:

Password Quality (1)

First Study by Robert T. Morris (once head scientist at the NSA) and Thompson (co-inventor of Unix) in *Communications of the ACM* **22**(11), November 1979, pp. 594–597.

They tested 3289 passwords and here is what they found:

15	(0.5%)	were a single ASCII character

Password Quality (1)

First Study by Robert T. Morris (once head scientist at the NSA) and Thompson (co-inventor of Unix) in *Communications of the ACM* **22**(11), November 1979, pp. 594–597.

They tested 3289 passwords and here is what they found:

15	(0.5%)	were a single ASCII character
72	(2.2%)	were strings of two ASCII characters

Password Quality (1)

First Study by Robert T. Morris (once head scientist at the NSA) and Thompson (co-inventor of Unix) in *Communications of the ACM* **22**(11), November 1979, pp. 594–597.

They tested 3289 passwords and here is what they found:

15	(0.5%)	were a single ASCII character
72	(2.2%)	were strings of two ASCII characters
464	(14.1%)	were strings of three ASCII characters

Password Quality (1)

First Study by Robert T. Morris (once head scientist at the NSA) and Thompson (co-inventor of Unix) in *Communications of the ACM* **22**(11), November 1979, pp. 594–597.

They tested 3289 passwords and here is what they found:

15	(0.5%)	were a single ASCII character
72	(2.2%)	were strings of two ASCII characters
464	(14.1%)	were strings of three ASCII characters
477	(14.5%)	were strings of four alphanumerics

Password Quality (1)

First Study by Robert T. Morris (once head scientist at the NSA) and Thompson (co-inventor of Unix) in *Communications of the ACM* **22**(11), November 1979, pp. 594–597.

They tested 3289 passwords and here is what they found:

15	(0.5%)	were a single ASCII character
72	(2.2%)	were strings of two ASCII characters
464	(14.1%)	were strings of three ASCII characters
477	(14.5%)	were strings of four alphanumerics
706	(21.4%)	were five letters, all upper-case or all lower-case

Password Quality (1)

First Study by Robert T. Morris (once head scientist at the NSA) and Thompson (co-inventor of Unix) in *Communications of the ACM* **22**(11), November 1979, pp. 594–597.

They tested 3289 passwords and here is what they found:

15	(0.5%)	were a single ASCII character
72	(2.2%)	were strings of two ASCII characters
464	(14.1%)	were strings of three ASCII characters
477	(14.5%)	were strings of four alphanumerics
706	(21.4%)	were five letters, all upper-case or all lower-case
605	(18.3%)	were six letters, all lower-case

Password Quality (1)

First Study by Robert T. Morris (once head scientist at the NSA) and Thompson (co-inventor of Unix) in *Communications of the ACM* **22**(11), November 1979, pp. 594–597.

They tested 3289 passwords and here is what they found:

15	(0.5%)	were a single ASCII character
72	(2.2%)	were strings of two ASCII characters
464	(14.1%)	were strings of three ASCII characters
477	(14.5%)	were strings of four alphanumerics
706	(21.4%)	were five letters, all upper-case or all lower-case
605	(18.3%)	were six letters, all lower-case
2339	(71.0%)	were easily guessable passwords

Password Quality (2)

Additionally, 492 passwords appeared in various dictionaries and word lists. All in all, **2831 or 86% of all passwords fell into these classes!** (There was overlap between the word lists and the exhaustive tests)

How bad is that?

Cost of Dictionary Attack (1)

There are

95	single ASCII characters
9025	strings of two ASCII characters
100000	English words
857375	strings of three ASCII characters
14776336	strings of four alphanumerics
23762752	were five letters, all upper-case or all lower-case
308915776	were six letters, all lower-case
348421359	Total

A *conservative* estimate is that you can do 1,000,000 crypto operations per second on a current machine. Compared with the time to write that to a disk, the time to encrypt is negligible.

Cost of Dictionary Attack (2)

Assume disk write speeds of just 1MB/s (*much* too low!). We assume that we can encrypt data faster than we can write it.

Cost of Dictionary Attack (2)

Assume disk write speeds of just 1MB/s (*much* too low!). We assume that we can encrypt data faster than we can write it.

We need to write 5574741744 bytes or about 5 Gigabytes of storage (easily available these days). This takes about 5000 seconds, or about 1.5 hours.

Cost of Dictionary Attack (2)

Assume disk write speeds of just 1MB/s (*much* too low!). We assume that we can encrypt data faster than we can write it.

We need to write 5574741744 bytes or about 5 Gigabytes of storage (easily available these days). This takes about 5000 seconds, or about 1.5 hours.

With a hour and a half of precomputation and five Gigs of hard disk space, you could crack about 85% of all Unix passwords in 1979 just by an $O(1)$ lookup operation for each.

Cost of Dictionary Attack (2)

Assume disk write speeds of just 1MB/s (*much* too low!). We assume that we can encrypt data faster than we can write it.

We need to write 5574741744 bytes or about 5 Gigabytes of storage (easily available these days). This takes about 5000 seconds, or about 1.5 hours.

With a hour and a half of precomputation and five Gigs of hard disk space, you could crack about 85% of all Unix passwords in 1979 just by an $O(1)$ lookup operation for each.

If that's not bad, I don't know what is!

Cost of Dictionary Attack (2)

Assume disk write speeds of just 1MB/s (*much* too low!). We assume that we can encrypt data faster than we can write it.

We need to write 5574741744 bytes or about 5 Gigabytes of storage (easily available these days). This takes about 5000 seconds, or about 1.5 hours.

With a hour and a half of precomputation and five Gigs of hard disk space, you could crack about 85% of all Unix passwords in 1979 just by an $O(1)$ lookup operation for each.

If that's not bad, I don't know what is!

But that was 1979; the situation has markedly improved since then, of course.

Cost of Dictionary Attack (2)

Assume disk write speeds of just 1MB/s (*much* too low!). We assume that we can encrypt data faster than we can write it.

We need to write 5574741744 bytes or about 5 Gigabytes of storage (easily available these days). This takes about 5000 seconds, or about 1.5 hours.

With a hour and a half of precomputation and five Gigs of hard disk space, you could crack about 85% of all Unix passwords in 1979 just by an $O(1)$ lookup operation for each.

If that's not bad, I don't know what is!

But that was 1979; the situation has markedly improved since then, of course.

Really?

Cost of Dictionary Attack (2)

Assume disk write speeds of just 1MB/s (*much* too low!). We assume that we can encrypt data faster than we can write it.

We need to write 5574741744 bytes or about 5 Gigabytes of storage (easily available these days). This takes about 5000 seconds, or about 1.5 hours.

With a hour and a half of precomputation and five Gigs of hard disk space, you could crack about 85% of all Unix passwords in 1979 just by an $O(1)$ lookup operation for each.

If that's not bad, I don't know what is!

But that was 1979; the situation has markedly improved since then, of course.

Really? Really?!

Defense Against Precomputation: Salting

Store, together with each password, a small integer (up to a few thousand) in the clear. This integer is called the *salt*. For Unix, the salt is a 12-bit value.

In order to check a password, the salt is used to perturb the encryption algorithm (for example, by prepending the salt to the given password prior to encryption). $\Longrightarrow$ the same password with two different salts will encrypt to different strings.

Same passwords no longer appear related in the password file and a precomputed dictionary attack is made more difficult. For Unix, a dictionary would have to be 2^{12} or 4096 times as large.

Unix crypt() (1)

The Unix password file contains seven fields:

```
neuhaus:abf/31kl&1@fe:7006:100:Stephan Neuhaus:/home/neuhaus:/bin/bash
```

The second field is the password field. The password contains the salt in the first two characters, and the encrypted password in the following characters.

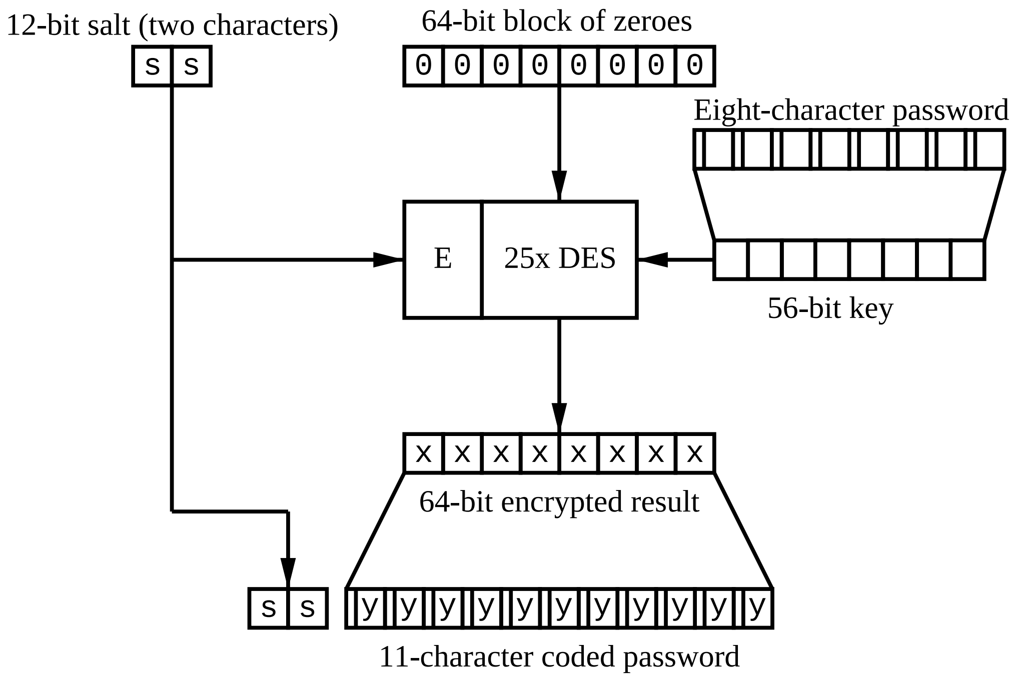
The eight seven-bit characters of the password are used to form a 56-bit DES key.

The salt is used to change the E table in DES that expands a 32-bit intermediate result to 48 bits

The DES is called 25 times on a 64-bit block of zeroes.

The result is expanded into 11 7-bit ASCII characters and prepended with the salt.

Unix crypt() Function (2)



Unix crypt() Function (3)

Is this a good password encryption function?

- Not reversible (that we know)
- Salted

Unix crypt() Function (3)

Is this a good password encryption function?

- Not reversible (that we know)
- Salted

But. . .

Unix crypt() Function (3)

Is this a good password encryption function?

- Not reversible (that we know)
- Salted

But. . .

Still can't make good passwords from bad ones!

State of the Passwords in 1989/1993

David C. Feldmeier, Philip R. Karn, *Unix Password Security - Ten Years Later*, Proceedings of CRYPTO ’89, pp. 44–63.

Walter Belgers, *Unix Password Security*, Technical Report, Technische Universiteit Eindhoven, December 1993.

Very fast *crypt*(3) functions tuned especially for cracking many passwords

Cracked 11% of all accounts (10.7% of accounts with shell) in about 25 hours of wall clock (not CPU) time using 11 Sun ELCs (25 MHz Sparc processors, 16–24 MB main memory).

State of the Passwords in 2004

- Hash functions instead of *crypt*(3) allow passwords longer than eight characters
- New password algorithm allows longer salt

State of the Passwords in 2004

- Hash functions instead of *crypt*(3) allow passwords longer than eight characters
- New password algorithm allows longer salt

But. . .

State of the Passwords in 2004

- Hash functions instead of *crypt*(3) allow passwords longer than eight characters
- New password algorithm allows longer salt

But... *still* can't make good passwords from bad ones!

State of the Passwords in 2004

- Hash functions instead of *crypt*(3) allow passwords longer than eight characters
- New password algorithm allows longer salt

But. . . *still* can’t make good passwords from bad ones!

- Computers have gotten so fast that you can easily do 2,000,000 crypts/second on a machine that’s not top of the line
- People just don’t choose longer passwords:

State of the Passwords in 2004

- Hash functions instead of *crypt*(3) allow passwords longer than eight characters
- New password algorithm allows longer salt

But. . . *still* can't make good passwords from bad ones!

- Computers have gotten so fast that you can easily do 2,000,000 crypts/second on a machine that's not top of the line
- People just don't choose longer passwords: older people remember the eight-character limit

State of the Passwords in 2004

- Hash functions instead of *crypt*(3) allow passwords longer than eight characters
- New password algorithm allows longer salt

But... *still* can't make good passwords from bad ones!

- Computers have gotten so fast that you can easily do 2,000,000 crypts/second on a machine that's not top of the line
- People just don't choose longer passwords: older people remember the eight-character limit and the younger ones are just too lazy :-)

$\Longrightarrow$ Passwords are probably not more difficult to crack now than they were in 1993.

How to Choose Good Passwords? (1)

Generate passwords by machine (and live with the fact that they'll be written down)

Attempt to crack a password when it's set and dismiss those that can be cracked (there are libraries that plug into the password program that do precisely this). A good password should be at least seven characters, have uppercase and lowercase letters and some punctuation.

How to Choose Good Passwords? (2)

Here are some bad passwords for a user ‘neuhaus’. They should be rejected by a password entry program.

Bad password	Reason
neuhaus	Same as user name
neuhaus!	Derived from user name
NeUhAuS	Also derived from user name
suahuen	Also derived from user name
tree	In word list
eert	Derived from word in word list
qwerty	Simple keyboard pattern
Bessie	Name of user’s cat(*)

(*) I don’t know how the password program could know this, but it should reject the password anyway!

How to Choose Good Passwords? (3)

Here are some more bad passwords.

Bad password	Reason
`fuck`	In wordlist
`Frodo`	Well-known password
`Joshua`	Also a well-known password(*)
`agnitfom`	Also a well-known password(*)
`redrum`	Also a well-known password(*)

(*) Name the source and win a prize!

How to Choose Good Passwords? (3)

Take a sentence that you can easily remember. Use the first letter of each word, preserving case, and including punctuation. For example, “My name is Ozymandias, king of kings!” becomes “MniO,kok!”.

Take care that you don’t misremember the sentence as, e.g., “I am Ozymandias. . . ” I have lost several passwords and passphrases this way!

“One Ring to rule them, one Ring to find them” ⇒ “1R2rt,1R2ft”; “If this be error and upon me proved, I never writ and no man ever loved” ⇒ “Itbeaump,Inwanmel.”; “Call me Ishmael. Some years ago - never mind how long precisely - having little or no money in my purse,” ⇒ “CmI. Sya-nmhlp-hlonmimp,” (Name the sources and win prizes).

How to Choose Good Passwords? (4)

For a variation, take the last letter or the second letter of each word. For example, “My name is Ozymandias, king of kings!” becomes “yess,gfs!”.

Or: “One Ring to rule them, one Ring to find them” ⇒ “niouh,nioih”; “If this be error and upon me proved, I never writ and no man ever loved” ⇒ “Iibonome,Ieinnaee.”; “Call me Ishmael. Some years ago - never mind how long precisely - having little or no money in my purse,” ⇒ “CeI.oyg-nihop-alrnoiyp,”

Don’t choose *any* of these examples any more!

Thease passwords can still be brute-forced (see exercises).

More Bad News: Replay Attacks

Passwords are often used to authenticate yourself over some network.

More Bad News: Replay Attacks

Passwords are often used to authenticate yourself over some network.

If that network is packet-switched (e.g., IP), the information travels through many hosts on the way from source to destination

More Bad News: Replay Attacks

Passwords are often used to authenticate yourself over some network.

If that network is packet-switched (e.g., IP), the information travels through many hosts on the way from source to destination

If the password is (a) reusable and (b) sent in the clear, an eavesdropper can sniff the password and impersonate you.

This is known as a replay attack.

More Bad News: Replay Attacks

Passwords are often used to authenticate yourself over some network.

If that network is packet-switched (e.g., IP), the information travels through many hosts on the way from source to destination

If the password is (a) reusable and (b) sent in the clear, an eavesdropper can sniff the password and impersonate you.

This is known as a replay attack.

Normal passwords are very much vulnerable to this attack.

More Bad News: Replay Attacks

Passwords are often used to authenticate yourself over some network.

If that network is packet-switched (e.g., IP), the information travels through many hosts on the way from source to destination

If the password is (a) reusable and (b) sent in the clear, an eavesdropper can sniff the password and impersonate you.

This is known as a replay attack.

Normal passwords are very much vulnerable to this attack.

Defense: Use One-time passwords (later in this lecture) and/or encrypt passwords (e.g., ssh, later lecture), or eliminate passwords altogether (below).

Eliminate Passwords Altogether

A password is of the type *you are what you know*.

If others know the shared secret, they can impersonate you. There is no way around that.

Therefore:

Eliminate Passwords Altogether

A password is of the type *you are what you know*.

If others know the shared secret, they can impersonate you. There is no way around that.

Therefore: Eliminate reusable passwords altogether

Eliminate Passwords Altogether

A password is of the type *you are what you know*.

If others know the shared secret, they can impersonate you. There is no way around that.

Therefore: Eliminate reusable passwords altogether

Augment them with *you are what you have*-type (security tokens) or biometric authenticators.

Biometric authenticators are problematic:

Eliminate Passwords Altogether

A password is of the type *you are what you know*.

If others know the shared secret, they can impersonate you. There is no way around that.

Therefore: Eliminate reusable passwords altogether

Augment them with *you are what you have*-type (security tokens) or biometric authenticators.

Biometric authenticators are problematic:

- lawful storage of biometric features

Eliminate Passwords Altogether

A password is of the type *you are what you know*.

If others know the shared secret, they can impersonate you. There is no way around that.

Therefore: Eliminate reusable passwords altogether

Augment them with *you are what you have*-type (security tokens) or biometric authenticators.

Biometric authenticators are problematic:

- lawful storage of biometric features
- liability: what if the retina-scan laser makes some people blind?

Eliminate Passwords Altogether

A password is of the type *you are what you know*.

If others know the shared secret, they can impersonate you. There is no way around that.

Therefore: Eliminate reusable passwords altogether

Augment them with *you are what you have*-type (security tokens) or biometric authenticators.

Biometric authenticators are problematic:

- lawful storage of biometric features
- liability: what if the retina-scan laser makes some people blind?
- acceptance: “Please deposit urine sample here”

Cryptographic Tokens (1)

The RSA SecurID token contains a secret cryptographic key and a clock that changes the display every minute.

The corresponding server software also contains the key.

The key and the time-of-day together make a unique code with a lifetime of 60 seconds.

When you authenticate, you enter your user ID and the code on your SecurID token. The server also generates a token from the time-of-day and its copy of the key. If they match, you're in.

Cryptographic Tokens (1)

This can be combined with a PIN pad.

Therefore the user is identified by a combination of what they have (the token) and what they know (the PIN)

Advantages

Authentication is not only by *what you know*

Advantages

Authentication is not only by *what you know*

Authentication token changes quickly $\Longrightarrow$ no replay atack possible

Advantages

Authentication is not only by *what you know*

Authentication token changes quickly $\Longrightarrow$ no replay atack possible

With additional PIN, stealing only the device is useless

Problems

What happens when the clock inside the token and the clock inside the server don't agree?

Recommended solution:

- Keep server clocks in check using the Network Time Protocol (NTP)
- Periodically re-synchronize the tokens on the server

What if the token goes out of sync on a Friday evening?

Summary (1)

The problem with passwords is typical of security design problems:

How to ensure Confidentiality, Integrity, and Authenticity?

Summary (2)

- Risks of Using Passwords

Summary (2)

- Risks of Using Passwords
- How to Steal ATM PINs

Summary (2)

- Risks of Using Passwords
- How to Steal ATM PINs
- Storing Passwords

Summary (2)

- Risks of Using Passwords
- How to Steal ATM PINs
- Storing Passwords
- Cracking Passwords, Brute-Force and Otherwise

Summary (2)

- Risks of Using Passwords
- How to Steal ATM PINs
- Storing Passwords
- Cracking Passwords, Brute-Force and Otherwise
- Back-of-Envelope Brute-Force Cost Calculation

Summary (2)

- Risks of Using Passwords
- How to Steal ATM PINs
- Storing Passwords
- Cracking Passwords, Brute-Force and Otherwise
- Back-of-Envelope Brute-Force Cost Calculation
- Defenses Against Atacks: Salting, Longer Passwords

Summary (2)

- Risks of Using Passwords
- How to Steal ATM PINs
- Storing Passwords
- Cracking Passwords, Brute-Force and Otherwise
- Back-of-Envelope Brute-Force Cost Calculation
- Defenses Against Atacks: Salting, Longer Passwords
- Passwords are Doomed

Summary (2)

- Risks of Using Passwords
- How to Steal ATM PINs
- Storing Passwords
- Cracking Passwords, Brute-Force and Otherwise
- Back-of-Envelope Brute-Force Cost Calculation
- Defenses Against Atacks: Salting, Longer Passwords
- Passwords are Doomed
- Eliminating Passwords

References

Ross Anderson, *Security Engineering*, Wiley

Viega, McGraw, *Building Secure Software*, Addison-Wesley