Random # generation cleaned up in Linux 5.17 and 5.18

• /dev/urandom
  – Would block until there was sufficient entropy detected in the system
  – Danger of excessive (possibly indefinite) blocking

• /dev/random & /dev/urandom are now equivalent
  – Random driver actively adds entropy using the processor’s cycle counter (measuring the elapsed time after running the scheduler) if it doesn't have enough

https://www.zx2c4.com/projects/linux-rng-5.17-5.18/
Hundreds of GoDaddy-hosted sites backdoored in a single day

Bill Toulas • March 16, 2022

Internet security analysts have spotted a spike in backdoor infections on WordPress websites hosted on GoDaddy's Managed WordPress service, all featuring an identical backdoor payload. The case affects internet service resellers such as MediaTemple, tsoHost, 123Reg, Domain Factory, Heart Internet, and Host Europe Managed WordPress.

The discovery comes from Wordfence, whose team first observed the malicious activity on March 11, 2022, with 298 websites infected by the backdoor within 24 hours, 281 of which were hosted on GoDaddy.

Old template spammer

The backdoor infecting all sites is a 2015 Google search SEO-poisoning tool implanted on the wp-config.php to fetch spam link templates from the C2 that are used to inject malicious pages into search results.

The campaign uses predominately pharmaceutical spam templates, served to visitors of the compromised websites instead of the actual content.

The goal of these templates is likely to entice the victims to make purchases of fake products, losing money and payment details to the threat actors.

Weak RSA Public Keys

March 14, 2022 Update

• Older software generated RSA keys that can be broken instantly with commodity hardware

• SafeZone library doesn't randomize the prime numbers well
  – Used to generate RSA keys
  – After selecting one prime #, the second one is in close proximity to the first

• Keys generated with primes that are too close together can be broken with Fermat's factorization method, described in 1643

Weak RSA Public Keys

• Product of two large primes can be written as
  \[ N = (a-b)(a+b) \]
  where \( a \) is the middle between the two primes
  where \( b \) is the distance from the middle to each of the primes

• If the primes are close, then \( a \) is close to \( \sqrt{N} \)

• Attack: guess \( a \) by starting from \( \sqrt{N} \) and then incrementing the guess
  – Calculate \( b^2 = a^2 - N \)
  – If the result is a square then we guessed correctly
  – Calculate the factors \( p, q \) as \( p=a+b, q=a-b \)
Authentication

- **Identification:** who are you?
- **Authentication:** prove it
- **Authorization:** you can do this

Some protocols (or services) combine all three
Cryptographic Authentication
The concept: prove you have the key

Ask the other side to prove they can encrypt or decrypt a message with the key

Create a nonce, $n$ (random bunch of bits)

Encrypt the nonce with the shared key, $K$

Validate the result:

$D_K(E_K(n)) \rightleftharpoons n$

• This assumes a **pre-shared key** and symmetric cryptography.
• After that, Alice can encrypt & send a **session key**.
• Minimize the use of the pre-shared key.
Mutual authentication

• Alice had Bob prove he has the key

• Bob may want to validate Alice as well
  ⇒ mutual authentication
  – Bob will do the same thing: have Alice prove she has the key

• Pre-shared key: Alice encrypts the nonce with the key

• Public key: Alice encrypts the nonce with her private key
Basic idea with symmetric cryptography:

Use a trusted third party (Trent) that has all the keys

- Alice wants to talk to Bob: she asks Trent
  - Trent generates a session key encrypted for Alice
  - Trent encrypts the same key for Bob (ticket)

- Authentication is implicit:
  - If Alice can encrypt a message for Trent, she proved she knows her key
  - If Bob can decrypt the message from Trent, he proved he knows his key

- Trent can also perform authorization

- Weaknesses that we need to address:
  - Replay attacks
Combined authentication & key exchange algorithms
Security Protocol Notation

\( Z \parallel W \)
- \( Z \) concatenated with \( W \)

\( A \rightarrow B : \{ Z \parallel W \} k_{A,B} \)
- \( A \) sends a message to \( B \)
- The message is the concatenation of \( Z \) & \( W \) and is encrypted by key \( k_{A,B} \), which is shared by users \( A \) & \( B \)

\( A \rightarrow B : \{ Z \} k_A \parallel \{ W \} k_{A,B} \)
- \( A \) sends a message to \( B \)
- The message is a concatenation of \( Z \) encrypted using \( A \)'s key and \( W \) encrypted by a key shared by \( A \) and \( B \)

\( r_1, r_2 \)
- nonces – strings of random bits
Bootstrap problem

*How can Alice & Bob communicate securely?*

- **Alice cannot send a key to Bob in the clear**
  - We assume an unsecure network

- **We looked at two mechanisms:**
  - Diffie-Hellman key exchange
  - Public key cryptography

Let’s examine the problem some more … in the context of authentication & key exchange
Simple Protocol

Use a trusted third party – Trent – who has all the keys

Trent creates a session key for Alice and Bob

Alice → Trent

\{ k_{A,B} \} k_A \| \{ k_{A,B} \} k_B

Trent → Alice

\{ k_{A,B} \} k_B

Alice → Bob

\{ m \} k_{A,B}

Bob → Alice

\{ k_{A,B} \} k_B

Trent → Alice

{ k_{A,B} } k_A \| { k_{A,B} } k_B

Alice → Trent

Request session key to Bob

March 24, 2022

CS 419 © 2022 Paul Krzyzanowski
Problems

- How does Bob know he is talking to Alice?
  - Trusted third party, Trent, has all the keys
  - Trent knows the request came from Alice since only he and Alice can have the key
  - Trent can **authorize** Alice’s request
  - Bob gets a session key encrypted with Bob’s key, which only Trent could have created
    - But Bob doesn’t know who requested the session – *is the request really from Alice?*
    - *Trent would need to add sender information to the message encrypted for Bob*

- **Vulnerable to replay attacks**
  - Eve records the message from Alice to Bob and later replays it
  - Bob will think he’s talking to Alice and re-use the same session key

- Protocols should provide **authentication & defense against replay attacks**
Add *nonces* – random strings \((r_1, r_2)\) – to avoid replay attacks

1. Alice sends \{ Alice || Bob || r_1 \} to Trent.
2. Alice sends \{ Alice || Bob || r_1 || k_{A,B} || \{ Alice || k_{A,B} \} k_B \} k_A to Trent.
3. Alice sends \{ Alice || k_{A,B} \} k_B to Bob.
4. Alice sends \{ r_2 \} k_{A,B} to Bob.
5. Alice sends \{ r_2 - 1 \} k_{A,B} to Bob.
Add **nonces** – random strings – and get the session key.

- Alice knows only Bob & Trent can read this and get the session key.
- Bob knows it’s a request from Alice.

Message must have been created by Trent & is a response to the first message (contains $r_1$). Use of $r_1$ ensures it’s not a replay attack.

1. Alice sends: $\{ Alice \| Bob \| r_1 \}$ to Trent.

2. Alice sends: $\{ Alice \| Bob \| r_1 \| k_{A,B} \| \{ Alice \| k_{A,B} \} k_B \} k_A$ to Trent.

3. Alice sends: $\{ Alice \| k_{A,B} \} k_B$ to Bob.

4. Bob sends: $\{ r_2 \} k_{A,B}$ to Alice.

5. Bob sends: $\{ r_2 - 1 \} k_{A,B}$ to Alice.

This is an **authentication** step: Bob asks Alice to prove she knows $k_{A,B}$.

- Bob now tries to find out if this is a replay attack.
- If it is, Eve will not be able to decipher $r_2$.
Needham-Schroeder Protocol Vulnerability

- We assume all keys are secret
- But suppose Eve can obtain the session key from an old message (she worked hard, got lucky, and cracked an earlier message)

Needham-Schroeder is still vulnerable to a certain replay attack ... if an old session key is known!

Bob sees this as a legitimate request approved by Trent. It was ... but earlier!

Eve the eavesdropper. She decrypted an old session key and is trying to get Bob to use it to think he's talking to Alice.
• Problem: replay in the third step of the protocol
  – Eve replays the message: \{ Alice || k_{A,B} \} k_B

• Solution: use a timestamp $T$ to detect replay attacks
  – The trusted third party (Trent) places a timestamp in a message that is encrypted for Bob
  – The attacker has an old session key but not Alice’s, Bob’s or Trent’s keys
  – Eve cannot spoof a valid message that is encrypted for Bob
Add **nonces** – random strings – **AND** a timestamp

- Alice sends: \{ Alice || Bob || r_1 \} to Trent
- Trent replies: \{ Alice || Bob || r_1 || k_{A,B} || \{ Alice || T || k_{A,B} \} k_B \} k_A to Alice
- Alice replies: \{ Alice || T || k_{A,B} \} k_B to Bob
- Bob sends: \{ r_2 \} k_{A,B} to Alice
- Alice replies: \{ r_2 - 1 \} k_{A,B} to Bob
Problem with timestamps

- **Use of timestamps relies on synchronized clocks**
  - Messages may be falsely accepted or falsely rejected because of bad time

- **Time synchronization becomes an attack vector**
  - Create fake NTP responses
  - Generate fake GPS signals

---

**Impact of GPS Time Spoofing Attacks on Cyber Physical Systems**

Xiao Wei  
Department of Electrical and Computer Engineering  
National University of Singapore  
Singapore  
wexiao@nus.edu

Biplab Sikdar  
Department of Electrical and Computer Engineering  
National University of Singapore  
Singapore  
bsikdar@nus.edu.sg

Abstract—The development of software defined radio platforms and related open-source software has made it possible to generate and broadcast global positioning system (GPS) signals easily and at low cost. Since GPS time is widely used in time-sensitive systems for time reference, any attack on GPS can have serious consequences. This paper evaluates GPS time spoofing attacks in cyber-physical systems. We explore methods to spoof the GPS time by manipulating the GPS timestamp or the signal propagation time of GPS satellite signals. In our experiments, the impact of GPS time spoofing attacks on the pseudorange, receiver location, and time errors is investigated. Our results show that, although the time GPS is time is spoofed, it is not difficult to find a device which can receive and transmit signals in GPS civilian frequency. The attacks can also take advantage of the unencrypted GPS civilian signals to extract the GPS information. In addition, due to the long distance attenuation, ionospheric interference and other effects, the received carrier power is around -158.5 dBm [3] and the signal to noise ratio (SNR) is a small value. This makes the authentic GPS signal susceptible to interference from other signals with greater signal power. All of these factors make a GPS receiver vulnerable to GPS spoofing attacks.
Another way to correct the *third message replay* problem

- Instead of using timestamps
  - Use a random integer, $n$, that is associated with all messages in the key exchange

- The protocol is altered slightly
  - Alice first sends a message to Bob
    - The message contains the session ID & nonce encrypted with Alice’s secret key
  - Bob forwards the message to Trent
    - And creates a message containing a nonce & the same session ID encrypted with Bob’s secret key
  - Trent creates a session key & encrypts it for both Alice and for Bob
Use nonces \((r_1, r_2)\) & session IDs \((n)\)

Alice sends the communication request to Bob – with the session ID

Bob authenticates himself & forwards request to Trent

---

**Otway-Rees Protocol**

March 24, 2022
Kerberos
Kerberos

• Authentication service developed by MIT
  – project Athena 1983-1988

• Uses a trusted third party & symmetric cryptography

• Based on Needham Schroeder with the Denning Sacco modification

• Passwords not sent in clear text
  – assumes only the network can be compromised

• Supported in most all popular operating systems
  – Default network authentication used in Microsoft Windows
  – Supported in macOS, Linux, FreeBSD, z/OS, …
  – Used by Rutgers to store NetIDs via the Central Authentication Service (CAS)
Users and services authenticate themselves to each other

To access a service:
- User presents a ticket issued by the Kerberos authentication server
- Service uses the ticket to verify the identity of the user

Kerberos is a trusted third party
- Knows all (users and services) passwords
- Responsible for
  - Authentication: validating an identity
  - Authorization: deciding whether someone can access a service
  - Key distribution: giving both parties an encryption key (securely)
Kerberos – General Flow

User Alice wants to communicate with a service Bob

Both Alice and Bob have keys – Kerberos has copies
  – key = hash(password)

Step 1:
  – Alice authenticates with Kerberos server
    • Gets session key and ticket

Step 2:
  – Alice gives Bob the ticket, which contains the session key
  – Convinces Bob that she got the session key from Kerberos
Kerberos (1): Authorize, Authenticate

“I’m Alice and want to talk to Bob”

\{ "Alice" || "Bob" \}  \rightarrow

If Alice is allowed to talk to Bob, generate session key, $k_{A,B}$

\{ "Bob’s server", T, $k_{A,B}$ \} $k_A$  \leftarrow

Alice decrypts this:
- Gets ID of “Bob’s server”
- Gets session key & timestamp
- Knows message came from AS

\{ "Alice", T, $k_{A,B}$ \} $k_B$  \leftarrow

“TICKET”

eh? (Alice can’t read this!)

Alice

Kerberos Authentication Service (AS)
Kerberos (2): Send key

Alice encrypts a timestamp with session key:

\[ \{ "Alice", k_{A,B} \} k_B \parallel \{ T' \} k_{A,B} \]

Bob decrypts the ticket:
- Ticket was created by Kerberos on request from Alice
- Gets session key
- Decrypts timestamp
- Validates time window
- Prevents replay attacks
Kerberos (3): Authenticate recipient of message

Alice validates timestamp

Alice

{ T'+1 }_{k_{A,B}}

Bob

Encrypt Alice’s timestamp in return message

{Messages}_{k_{A,B}}

Alice & Bob communicate by encrypting data with $k_{A,B}$
Kerberos key usage

• Every time a user wants to access a service
  – User’s password (key) must be used to decode the message from Kerberos

• We can avoid this by caching the password in a file
  – Not a good idea

• Another way: create a temporary password
  – We can cache this temporary password
  – It's just a session key to access Kerberos – to get access to other services
  – Split Kerberos server into Authentication Service + Ticket Granting Service
Ticket Granting Server (TGS)

- **TGS works like a temporary ID**
- **User first requests access to the TGS**
  - Contact Kerberos Authentication Service (AS knows all users & their keys)
    - Gets back a ticket & session key to the TGS – these can be cached
- **To access any service**
  - Send a request to the TGS – encrypted with the TGS session key along with the ticket for the TGS
  - The ticket tells the TGS what your session key is
  - It responds with a session key & ticket for that service
Kerberos AS + TGS: Step 1

1. Request access to TGS

2. Here's a session key & ticket for the TGS
   Enter password to decrypt \( k_{TGS,A} \times k_A \)
   Cache the TGS session key, \( k_{TGS,A} \)
Kerberos AS + TGS

Authentication Service (AS)

Ticket Granting Service (TGS)

users & user keys

Authorization

Kerberos Key Distribution Center (KDC)

(3) TGS-ticket, \{ T \} k_{TGS,A}
\{ Bob, please \} k_{TGS,A}

(4) Here's a session key & ticket for the Bob
session key: \{ "Bob's server", T, k_{A,B} \} k_{TGS,A}
ticket: \{ "Alice", T, k_{A,B} \} k_{B}
Kerberos AS + TGS

Kerberos Key Distribution Center (KDC)

Authentication Service (AS)
users & user keys

Ticket Granting Service (TGS)
Authorization

(5) \{ "Alice", k_{A,B} \} k_B \parallel \{ T' \} k_{A,B}

(6) \{ T'+1 \} k_{A,B}

\{ messages \}k_{A,B}
Using Kerberos

$ kinit

Password: enter password

ask AS for permission (session key) to access TGS

Alice gets:

\{“TGS”, T, k_{A,TGS} \} k_A \quad \leftarrow \text{Session key & encrypted timestamp}

\{“Alice”, k_{A,TGS} \} k_{TGS} \quad \leftarrow \text{TGS Ticket}

Compute key (A) from password to decrypt session key $k_{A,TGS}$ and get TGS ID.

You now have a ticket to access the Ticket Granting Service
Using Kerberos

$ \texttt{rlogin somehost}

\textit{rlogin} uses the TGS Ticket to request a ticket for the \textit{rlogin} service on \textit{somehost}

Alice sends session key, S, to TGS

\begin{align*}
\text{rlogin} & \rightarrow \text{TGS} \\
\{"Alice", k_{A,TGS}\} k_{TGS}, \{T\} & \rightarrow k_{A,TGS}
\end{align*}

Alice receives session key for \textit{rlogin} service & ticket to pass to rlogin service

\begin{align*}
\text{rlogin} & \leftarrow \text{TGS} \\
\{"rlogin@somehost", k_{A,R}\} & \rightarrow k_{A,TGS}
\end{align*}

\begin{align*}
\text{rlogin} & \leftarrow \text{TGS} \\
\{"Alice", k_{A,R}\} & \rightarrow k_{R}
\end{align*}

\begin{align*}
\text{rlogin} & \leftarrow \text{TGS} \\
\{"rlogin@somehost", k_{A,R}\} & \rightarrow k_{A,TGS}
\end{align*}

k_{A,R} = \text{session key for \textit{rlogin}}

ticket for \textit{rlogin} server on \textit{somehost}
Basic idea with symmetric cryptography:

*Use a trusted third party (Trent) that has all the keys*

- Alice wants to talk to Bob: she asks Trent
  - Trent generates a session key encrypted for Alice
  - Trent encrypts the same key for Bob (ticket)

- Authentication is implicit:
  - If Alice can decrypt the session key, she proved she knows her key
  - If Alice can decrypt the session key, he proved he knows his key

- Weaknesses that we had to fix:
  - Replay attacks – add nonces – Needham-Schroeder protocol
  - Replay attacks re-using a cracked old session key
    - Add timestamps: Denning-Sacco protocol, Kerberos
    - Add session IDs at each step: Otway-Rees protocol
Public Key Based Key Exchange

We saw how this works…

• Alice’s & Bob’s public keys known to all: $e_A, e_B$
• Alice & Bob’s private keys are known only to the owner: $d_a, d_b$
• Simple protocol to send symmetric session key, $k_S$:

\[
\{ k_S \} e_B
\]
Adding authentication

- Have Bob prove that he has the private key
  - Same way as with symmetric cryptography – prove he can encrypt or decrypt

Create nonce, $r_1$

\[ \{ r_1 \} e_B \]

\[ r_1 \]
Adding mutual authentication

- Bob asks Alice to prove that she has her private key

Create nonce, $r_1$

$\{r_1\} e_B$

$r_1$

$\{r_2\} e_A$

$r_2$
Adding identity binding

• How do we know we have the right public keys?
• Get the public key from a trusted certificate
  – Validate the signature on the certificate before trusting the public key within
Cryptographic toolbox

- Symmetric encryption
- Public key encryption
- Hash functions
- Random number generators
User Authentication
Three Factors of Authentication

1. Ownership
   - Key, card
   - Can be stolen
   - Something you have

2. Knowledge
   - Passwords, PINs
   - Can be guessed, shared, stolen
   - Something you know

3. Inherence
   - Biometrics (face, fingerprints)
   - Requires hardware
   - May be copied
   - Not replaceable if lost or stolen
   - Something you are
Factors may be combined

- **ATM machine:** 2-factor authentication (2FA)
  - ATM card: something you have
  - PIN: something you know

- **Password + code delivered via SMS:** 2-factor authentication
  - Password: something you know
  - Code: something you have: your phone

**Two passwords ≠ Two-factor authentication**

The factors must be different
Password Authentication Protocol

- Unencrypted, reusable passwords
- Insecure on an open network
- Also, the password file must be protected from open access
  - But administrators can still see everyone’s passwords

*What if you use the same password on Facebook as on Amazon?*
Passwords are bad

• Human readable & easy to guess
  – People usually pick really bad passwords

• Easy to forget

• Usually short

• Static ... reused over & over
  – Security is as strong as the weakest link
  – If a username (or email) & password is stolen from one server, it might be usable on others

• Replayable
  – If someone can grab it or see it, they can play it back
Recent large-scale leaks of password from servers have shown that people **DO NOT pick good passwords**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>123456</td>
<td>123456</td>
<td>123456</td>
<td>123456</td>
<td>123456</td>
<td>123456</td>
<td>123456</td>
<td>123456</td>
<td>123456</td>
</tr>
<tr>
<td>2</td>
<td>password</td>
<td>password</td>
<td>password</td>
<td>password</td>
<td>password</td>
<td>password</td>
<td>123456789</td>
<td>123456789</td>
<td>123456789</td>
</tr>
<tr>
<td>3</td>
<td>12345678</td>
<td>12345</td>
<td>12345678</td>
<td>12345</td>
<td>12345678</td>
<td>123456789</td>
<td>qwerty</td>
<td>picture1</td>
<td>12345</td>
</tr>
<tr>
<td>4</td>
<td>qwerty</td>
<td>12345678</td>
<td>qwerty</td>
<td>12345678</td>
<td>qwerty</td>
<td>12345678</td>
<td>password</td>
<td>password</td>
<td>qwerty</td>
</tr>
<tr>
<td>5</td>
<td>abc123</td>
<td>qwerty</td>
<td>12345</td>
<td>football</td>
<td>12345</td>
<td>12345</td>
<td>1234567</td>
<td>12345678</td>
<td>password</td>
</tr>
<tr>
<td>6</td>
<td>123456789</td>
<td>123456789</td>
<td>123456789</td>
<td>qwerty</td>
<td>123456789</td>
<td>111111</td>
<td>12345678</td>
<td>111111</td>
<td>12345678</td>
</tr>
<tr>
<td>7</td>
<td>111111</td>
<td>1234</td>
<td>football</td>
<td>1234567890</td>
<td>letmein</td>
<td>1234567</td>
<td>12345</td>
<td>123123</td>
<td>111111</td>
</tr>
<tr>
<td>8</td>
<td>1234567</td>
<td>baseball</td>
<td>1234</td>
<td>1234567</td>
<td>1234567</td>
<td>sunshine</td>
<td>iloveyou</td>
<td>12345</td>
<td>123123</td>
</tr>
</tbody>
</table>

*Top passwords by year 2013-2019: SplashData; 2020-2021: NordPass*

https://nordpass.com/most-common-passwords-list/
https://en.wikipedia.org/wiki/List_of_the_most_common_passwords
Policies to the rescue?

Password rules
“Everyone knows that an exclamation point is a 1, or an I, or the last character of a password. $ is an S or a 5. If we use these well-known tricks, we aren’t fooling any adversary. We are simply fooling the database that stores passwords into thinking the user did something good”
— Paul Grassi, NIST

Periodic password change requirement problems
– People tend to change passwords rapidly to exhaust the history list and get back to their favorite password
– Forbidding changes for several days enables a denial of service attack
– People pick worse passwords, incorporating numbers, months, or years

https://fortune.com/2017/05/11/password-rules/
NIST recommendations

• Remove periodic password change requirements

• Drop complexity requirements (numbers, letters, symbols)

• Choose long passwords

• Avoid
  – Passwords obtained from databases of previous breaches
  – Dictionary words
  – Repetitive or sequential characters (e.g. ‘aaaaa’, ‘1234abcd’)
  – Context-specific words, such as the name of the service, the username, and derivatives thereof

Problem #1: Open access to the password file

What if the password file isn’t sufficiently protected and an intruder gets hold of it? All passwords are now compromised!

Even if an admin sees your password, this might also be your password on other systems.

How about encrypting the passwords?

• Where would you store the key?

• Adobe did that
  – 2013 Adobe security breach leaked 152 million Adobe customer records
  – Adobe used encrypted passwords
    • But the passwords were all encrypted with the same key
    • If the attackers steal the key, they get the passwords
Poor Password Management

Adobe security breach (November 2013)

- 152 million Adobe customer records … with encrypted passwords
- Adobe encrypted passwords with a symmetric key algorithm
  … and used the same key to encrypt every password!

<table>
<thead>
<tr>
<th>Frequency</th>
<th>Password</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1,911,938</td>
</tr>
<tr>
<td>2</td>
<td>446,162</td>
</tr>
<tr>
<td>3</td>
<td>345,834</td>
</tr>
<tr>
<td>4</td>
<td>211,659</td>
</tr>
<tr>
<td>5</td>
<td>201,580</td>
</tr>
<tr>
<td>6</td>
<td>130,832</td>
</tr>
<tr>
<td>7</td>
<td>124,253</td>
</tr>
<tr>
<td>8</td>
<td>113,884</td>
</tr>
<tr>
<td>9</td>
<td>83,411</td>
</tr>
<tr>
<td>10</td>
<td>82,694</td>
</tr>
<tr>
<td>11</td>
<td>76,910</td>
</tr>
<tr>
<td>12</td>
<td>76,186</td>
</tr>
<tr>
<td>13</td>
<td>70,791</td>
</tr>
<tr>
<td>14</td>
<td>61,453</td>
</tr>
<tr>
<td>15</td>
<td>56,744</td>
</tr>
<tr>
<td>16</td>
<td>54,651</td>
</tr>
<tr>
<td>17</td>
<td>48,850</td>
</tr>
<tr>
<td>18</td>
<td>47,142</td>
</tr>
<tr>
<td>19</td>
<td>44,281</td>
</tr>
<tr>
<td>20</td>
<td>43,670</td>
</tr>
<tr>
<td>21</td>
<td>43,497</td>
</tr>
<tr>
<td>22</td>
<td>37,407</td>
</tr>
<tr>
<td>23</td>
<td>35,325</td>
</tr>
<tr>
<td>24</td>
<td>34,963</td>
</tr>
</tbody>
</table>
Solution:

Store a **hash** of the password in a file

- Given a file, you don’t get the passwords, only their hashes
  - Hashes are one-way functions
  - Example, Linux passwords hashed with a SHA-512 hash (SHA-2)
- Have to resort to a **dictionary** or **brute-force attack**
Brute force password attacks

<table>
<thead>
<tr>
<th>Number of Characters</th>
<th>Numbers Only</th>
<th>Lowercase Letters</th>
<th>Upper and Lowercase Letters</th>
<th>Numbers, Upper and Lowercase Letters</th>
<th>Numbers, Upper and Lowercase Letters, Symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Instantly</td>
<td>Instantly</td>
<td>Instantly</td>
<td>Instantly</td>
<td>Instantly</td>
</tr>
<tr>
<td>5</td>
<td>Instantly</td>
<td>Instantly</td>
<td>Instantly</td>
<td>Instantly</td>
<td>Instantly</td>
</tr>
<tr>
<td>6</td>
<td>Instantly</td>
<td>Instantly</td>
<td>Instantly</td>
<td>1 sec</td>
<td>5 secs</td>
</tr>
<tr>
<td>7</td>
<td>Instantly</td>
<td>Instantly</td>
<td>25 secs</td>
<td>1 min</td>
<td>6 mins</td>
</tr>
<tr>
<td>8</td>
<td>Instantly</td>
<td>5 secs</td>
<td>22 mins</td>
<td>1 hour</td>
<td>8 hours</td>
</tr>
<tr>
<td>9</td>
<td>Instantly</td>
<td>2 mins</td>
<td>19 hours</td>
<td>3 days</td>
<td>3 weeks</td>
</tr>
<tr>
<td>10</td>
<td>Instantly</td>
<td>58 mins</td>
<td>1 month</td>
<td>7 months</td>
<td>5 years</td>
</tr>
<tr>
<td>11</td>
<td>2 secs</td>
<td>1 day</td>
<td>5 years</td>
<td>41 years</td>
<td>400 years</td>
</tr>
<tr>
<td>12</td>
<td>25 secs</td>
<td>3 weeks</td>
<td>300 years</td>
<td>2k years</td>
<td>34k years</td>
</tr>
<tr>
<td>13</td>
<td>4 mins</td>
<td>1 year</td>
<td>16k years</td>
<td>100k years</td>
<td>2m years</td>
</tr>
<tr>
<td>14</td>
<td>41 mins</td>
<td>51 years</td>
<td>800k years</td>
<td>9m years</td>
<td>200m years</td>
</tr>
<tr>
<td>15</td>
<td>6 hours</td>
<td>1k years</td>
<td>43m years</td>
<td>600m years</td>
<td>15bn years</td>
</tr>
<tr>
<td>16</td>
<td>2 days</td>
<td>34k years</td>
<td>2bn years</td>
<td>37bn years</td>
<td>1tn years</td>
</tr>
<tr>
<td>17</td>
<td>4 weeks</td>
<td>800k years</td>
<td>100bn years</td>
<td>2tn years</td>
<td>93tn years</td>
</tr>
<tr>
<td>18</td>
<td>9 months</td>
<td>23m years</td>
<td>61tm years</td>
<td>100tn years</td>
<td>7qd years</td>
</tr>
</tbody>
</table>

MD5 hashed passwords cracked by a GeForce RTX 2080 GPU

https://www.hivesystems.io/blog/are-your-passwords-in-the-green
What is a dictionary attack?

• Suppose you got access to a list of hashed passwords

• Brute-force, exhaustive search: try every combination
  – Letters (A-Z, a-z), numbers (0-9), symbols (!@#$%...)  
  – Assume 30 symbols + 52 letters + 10 digits = 92 characters  
  – Test all passwords up to length 8  
  – Combinations = 92^8 + 92^7 + 92^6 + 92^5 + 92^4 + 92^3 + 92^2 + 92^1 = 5.189 \times 10^{15}  
  – If we test 1 billion passwords per second: \approx 60 days

• But some passwords are more likely than others
  – 1,991,938 Adobe customers used a password = “123456”  
  – 345,834 users used a password = “password”

• Dictionary attack
  – Test lists of common passwords, dictionary words, names  
  – Add common substitutions, prefixes, and suffixes

Easiest to do if the attacker steals a hashed password file – so we read-protect the hashed passwords to make it harder to get them
How to speed up a dictionary attack

Create a table of **precomputed hashes**

Now we just search a table for the hash to find the password

<table>
<thead>
<tr>
<th>SHA-256 Hash</th>
<th>password</th>
</tr>
</thead>
<tbody>
<tr>
<td>8d969eef6ecad3c29a3a629280e686cf0c3f5d5a86aff3ca12020c923adc6c92</td>
<td>123456</td>
</tr>
<tr>
<td>5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef721d1542d8</td>
<td>password</td>
</tr>
<tr>
<td>ef797c8118f02dfb649607dd5d3f8c7623048c9c063d532cc95c5ed7a898a64f</td>
<td>12345678</td>
</tr>
<tr>
<td>1c8bfe8f801d79745c4631d09fff36c82aa37fc4cce4fc946683d7b336b63032</td>
<td>letmein</td>
</tr>
</tbody>
</table>

...
Salt: defeating dictionary attacks

**Salt** = random string (typically up to 16 characters)
- Concatenated with the password
- Stored with the password file (it’s not secret)

\[ "VhsRrsFA" + "password" \]

Even if you know the salt, you cannot use precomputed hashes to search for a password (because the salt is prefixed to the password string and becomes part of the hash)

**Example:**
SHA-256 hash of "password", salt = "VhsRrsFA" = hash("VhsRrsFApasword") = b791b1b572c0025ef30ecc5fc5ecc5c623f52fca66250560fce8d22623b166c8

*You will not have a precomputed hash("VhsRrsFApasword")*
Longer passwords

English text has an entropy of about 1.2-1.5 bits per character

Random text has an entropy $\approx \log_2(1/95) \approx 6.6$ bits/character

Assume 95 printable characters
Defenses

• **Use longer passwords**
  – But can you trust users to pick ones with enough entropy?

• **Rate-limit guesses**
  – Add timeouts after an incorrect password
    • Linux waits about 3 secs – and terminates the `login` program after 5 tries

• **Lock out the account after $N$ bad guesses**
  – But this makes you vulnerable to denial-of-service attacks

• **Use a slow algorithm to make guessing slow**
  – OpenBSD `bcrypt` Blowfish password hashing algorithm
People forget passwords

• Especially seldom-used ones – how do we handle that?

• Email them?
  – Common solution
  – Requires that the server be able to get the password (can’t store a hash)
  – What if someone reads your email?

• Reset them?
  – How do you authenticate the requester?
  – Usually send reset link to email address created at registration
  – But – what if someone reads your mail? …or you no longer have that address?

• Provide hints?

• Write them down?
  – OK if the threat model is electronic only
Reusable passwords in multiple places

• People often use the same password in different places

• If one site is compromised, the password can be used elsewhere
  – People often try to use the same email address and/or username

• This is the root of phishing attacks
Password Managers

Software that stores passwords in an encrypted file

• Do you trust the protection?
  – The reputation of the company & its security policies
  – The synchronization capabilities?

• Can malware get to the database?

• In general, these are good
  – Way better than storing passwords in a file
  – Encourages having unique passwords per site
  – Generates strong passwords
  – Password managers may have the ability to recognize web sites & defend against phishing while providing auto-complete convenience for legitimate sites
LastPass fixes bug that could let malicious websites extract your last used password

*Even password managers have security bugs*

By Jon Porter • Sep 16, 2019

LastPass has patched a bug that would have allowed a malicious website to extract a previous password entered by the service’s browser extension. ZDNet reports that the bug was discovered by Tavis Ormandy, a researcher in Google’s Project Zero team, and was disclosed in a bug report dated August 29th. LastPass fixed the issue on September 13th, and deployed the update to all browsers where it should be applied automatically, something LastPass users would be smart to verify.
Key management risks

Password managers are a form of key storage
If attackers get your credentials, they can get all your passwords

South African bank to replace 12m cards after employees stole master key

Postbank says employees printed its master key at one of its data centers and then used it to steal $3.2 million.

Catalin Cimpanu • June 15 2020

Postbank, the banking division of South Africa's Post Office, has lost more than $3.2 million from fraudulent transactions and will now have to replace more than 12 million cards for its customers after employees printed and then stole its master key.

The bank suspects that employees are behind the breach, the news publication said, citing an internal security audit they obtained from a source in the bank.

The master key is a 36-digit code (encryption key) that allows its holder to decrypt the bank's operations and even access and modify banking systems. It is also used to generate keys for customer cards.

https://www.zdnet.com/article/south-african-bank-to-replace-12m-cards-after-employees-stole-master-key/
Problem #2: Network sniffing or shoulder surfing

Passwords can be stolen by observing a user’s session in person or over a network:

- Snoop on http, telnet, ftp, rlogin, rsh sessions
- Trojan horse
- Social engineering
- Key logger, camera, physical proximity
- Brute-force or dictionary attacks

Solutions:

(1) Use an encrypted communication channel
(2) Use multi-factor authentication, so a password alone is not sufficient
(3) Use one-time passwords
One-time passwords

Use a different password each time
- If an intruder captures the transaction, it won’t work next time

Three forms

1. **Sequence-based**: $\text{password} = f(\text{previous password})$ or $f(\text{secret}, \text{sequence#})$
2. **Challenge-based**: $f(\text{challenge}, \text{secret})$
3. **Time-based**: $\text{password} = f(\text{time}, \text{secret})$
S/key authentication

- One-time password scheme
- Produces a limited number of authentication sessions
- Relies on one-way functions
Authenticate Alice for 100 logins

• Pick a random number, R

• Using a one-way function (e.g., a hash function), $f(x)$:

$$x_1 = f(R)$$
$$x_2 = f(x_1) = f(f(R))$$
$$x_3 = f(x_2) = f(f(f(R)))$$

... ... ...

$$x_{100} = f(x_{99}) = f(...f(f(f(R)))...$$

• Then compute:

$$x_{101} = f(x_{100}) = f(...f(f(f(R)))...$$

Give this list to Alice
Authenticate Alice for 100 logins

Store $x_{101}$ in a password file or database record associated with Alice

$\text{alice: } x_{101}$
Alice presents the last number on her list:

\[ \text{Alice to host: \{ “alice”, } x_{100} \} \]

Host computes \( f(x_{100}) \) and compares it with the value in the database

\[
\text{if } f(x_{100 \text{ provided by alice}}) = \text{passwd(“alice”)} \\
\text{replace } x_{101} \text{ in db with } x_{100} \text{ provided by alice} \\
\text{return success} \\
\text{else} \\
\text{fail}
\]

Next time: Alice presents \( x_{99} \)

If someone sees \( x_{100} \) there is no way to generate \( x_{99} \).
S/Key → OPIE

S/Key slightly refined by the U.S. Naval Research Laboratory (NRL)

• OPIE = One time Passwords In Everything
  – Comes with FreeBSD, OpenBSD; available on Linux & other POSIX platforms
  – Use `/usr/sbin/opielogin` instead of standard `/bin/login` program

• Same iterative generation as S/Key
  \[ \text{starting\_password} = \text{Hash}(\text{seed, secret\_pass\_phrase}) \]
  The \text{seed} can differ among applications and enables a user to use the same passphrase securely for different applications

• Operates in two modes
  – Sequence-based: pre-generate a sequence of one-time passwords
    • A password is represented as 6 short words
  – Challenge-based: user is presented with a sequence number
    • Computes the proper password from a stored seed value

Challenge-Handshake Authentication Protocol

The challenge is a *nonce* (random bits).
We create a hash of the nonce and the secret.
An intruder does not have the secret and cannot do this!
CHAP authentication

Alice

“alice”

network

“alice”

host

look up alice’s key, $K$

generate random challenge number $C$

$R' = f(K, C)$

an eavesdropper does not see $K$

$R' = f(K, C)$

$R = f(K, C)$

$R = R' ?$

“welcome”
Time-Based Authentication

Time-based One-time Password (TOTP) algorithm

• Both sides share a secret key
  – Sometimes sent via a QR code so the user can scan it into the TOTP app

• User runs TOTP function to generate a one-time password
  \[
  \text{one_time_password} = \text{hash}(\text{secret_key}, \text{time})
  \]

• User logs in with: \text{name}, \text{password}, and \text{one_time_password}

• Service generates the same password
  \[
  \text{one_time_password} = \text{hash}(\text{secret_key}, \text{time})
  \]

• Typically 30-second granularity for time
Time-based One-time Passwords

Popular authenticators:
- Microsoft Two-step Verification
- Google Authenticator
- Facebook Code Generator
- Okta
- Duo

Used by
- Microsoft Azure, 365
- Amazon Web Services
- Bitbucket
- Dropbox
- Evernote
- Zoho
- Wordpress
- 1Password
- Many others…
Username: paul

Password: 1234032848

PIN + passcode from card

Something you know
Something you have

Passcode changes every 60 seconds

1. Enter PIN
2. Press ◊
3. Card computes password
4. Read password & enter

Password: 354982
SecurID card

**Same principle as Time-based One-Time Passwords**

- **Proprietary device from RSA**
  - SASL mechanism: RFC 2808

- **Two-factor authentication** based on:
  - **Shared secret key** (seed)
    - stored on authentication card
  - **Shared personal ID** – PIN
    - known by user

- Something you have
- Something you know
SecurID (SASL) authentication: server side

• Look up user’s PIN and seed associated with the token

• Get the time of day
  – Server stores relative accuracy of clock in that SecurID card
  – historic pattern of drift
  – adds or subtracts offset to determine what the clock chip on the SecurID card believes is its current time

• Passcode is a cryptographic hash of seed, PIN, and time
  – server computes $f(\text{seed, PIN, time})$

• Server compares results with data sent by client
HOTP = Hash-based One-Time Password

OTP = hash(hardware_id, passcode, counter)

Passcode generated on the device from session counters, previous values, other sources
SMS/Email Authentication

- Second factor = your possession of a phone (or computer)
- After login, sever sends you a code via SMS (or email)
- Entering it is proof that you could receive the message
- Dangers
  - **SIM swapping** attacks
    (social engineering on the phone company)
    - Targeted but viable for high-value targets
  - Social engineering to get email credentials

Password systems are vulnerable to man-in-the-middle attacks
– Attacker acts as the server
Man-in-the-Middle Attacks (MitM)

Password systems are vulnerable to **man-in-the-middle attacks**

- Attacker acts as the server

![Diagram showing man-in-the-middle attack between Alice and Bob through Mike]
Man-in-the-Middle Attacks (MitM)

Password systems are vulnerable to **man-in-the-middle attacks**
- Attacker acts as the server

Alice  
Mike  
Bob

What's your password?  
What's your password?
Man-in-the-Middle Attacks (MitM)

Password systems are vulnerable to man-in-the-middle attacks

– Attacker acts as the server
Man-in-the-Middle Attacks (MitM)

Password systems are vulnerable to man-in-the-middle attacks
- Attacker acts as the server
Man-in-the-Middle Attacks (MitM)

Password systems are vulnerable to **man-in-the-middle attacks**

- Attacker acts as the server

![Diagram showing man-in-the-middle attack]

- Alice
- Mike
- Bob

Huh?

Download my files
Guarding against man-in-the-middle attacks

• **Use a covert communication channel**
  – The intruder won’t have the key
  – Can’t see the contents of any messages

• **Use signed messages for all communication**
  – **Signed message** = \{ message, private-key-encrypted hash of message \}
  – Both parties can reject unauthenticated messages
  – The intruder cannot modify the messages
    - Signatures will fail (they will need to know how to encrypt the hash)

• **But watch out for replay attacks!**
  – May need to use session numbers or timestamps
The End