Authentication methodologies

- can be based on different factors (1/2/3-factors authentication):
  - something I know (e.g. a password)
  - something I have (e.g. magnetic card)
  - something I am (e.g. my fingerprint)
- multiple different mechanisms can be combined to achieve identification

User authentication

server

authentication request

UID

proof request

proof = F (SUID)

user (UID)

storage of the secret?

insertion of the secret?

transmitting the proof?

storage of the secrets?

verification of the proof?
Password-based authentication

- secret = the user password
- (client) create and transmit proof
  - \( F = I \) (the identity function)
  - i.e. proof = password (cleartext!)
- (server) verify the proof:
  - case #1: \( f = I \) (the identity function)
    - server knows all passwords in cleartext (!)
    - access control: proof = password ?
  - case #2: \( f = \) one-way hash (that is a digest)
    - server knows the passwords' digests, \( H_{UID} \)
    - access control: \( f(proof) = H_{UID} ? \)

Password-based authentication

pro:
- simple for the user
- … only if she has to remember just one passwor!

cons:
- user-side password storage (post-it!)
- password guessable (my son’s name!)
- password readable during transmission
- server-side password storage – the server must know in cleartext the password or its unprotected digest (dictionary attack)
Password

- suggestions to reduce the associated risks:
  - alphabetic characters (uppercase + lowercase) + digits + special characters
  - long (at least 8 characters)
  - never use dictionary words
  - frequently changed (but not too frequently!)
  - don't use them :-)
  - use of at least one password (or PIN or access code di or ...) unavoidable unless we use biometric techniques

Storing the password

- NEVER in cleartext!
- encrypted password? then the server must know the key in cleartext …
- store a digest of the password
- … but beware of the “dictionary” attack
- … that can be made faster by a “rainbow table”
- we must therefore insert an unexpected variation, usually named “salt”

The “dictionary” attack

- hypothesis:
  - known hash algorithm
  - known password hash values
- pre-computation:
  - for (each Word in Dictionary) do
    store ( DB, Word, hash(Word) )
- attack:
  - let HP be the hash value of a (unknown) password
  - w = lookup ( DB, HP )
  - if ( success ) then write ( “pwd = ”, w )
    else write ( “pwd not in my dictionary” )
Rainbow table (I)
- A space-time trade-off technique to store and lookup an exhaustive hash table
- E.g. table for a 12 digits password
  - Exhaustive = $10^{12}$ rows \{ P; HP \}
  - Rainbow = $10^6$ rows, each representing 1000 pwd
- Uses the reduction function $r : h \Rightarrow p$ (NOT $h^{-1}$)
- Pre-computation:
  - For $10^9$ distinct P
  - For (p=P, n=0; n<1000; n++)
    - $k = h(p); \quad p = r(k);
  - Store (DB, P, p) // chain head and tail

Rainbow table (II)
- Attack:
  - Let HP be the hash of a password
  - For (k=HP; n=0; n<1000; n++)
    - $p = r(k)
  - If lookup(DB, x, p) then exit ("chain found!")
  - $k = h(p)
  - Exit ("HP is not in any chain of mine")
- To avoid "fusion" of chains $r_1(\ldots r_n(\ldots)$ are used
- On sale pre-computed rainbow tables for various hash functions password sets (e.g. alphanum)
- This technique is used by various attack programs

Using the salt in storing passwords
- For each user UID:
  - Create / ask the pwd
  - Generate a salt (different for each user)
    - Random (unpredictable) and long (increased dictionary complexity)
    - Should contain rarely used or control characters
  - Compute HP = hash (pwd || salt)
  - Store the triples \{ UID, HP_{UID}, salt_{UID} \}
  - Additionally we have different HP for users having the same pwd
  - Makes the dictionary attack nearly impossible (included those based on rainbow tables)
The Linkedin attack

- June 2012, copied 6.5 M password from Linkedin
  - ... unsalted, plain SHA-1 hash!!!
- Crowdsourcing used for cooperative password cracking
  - At least 236,578 passwords found (before ban of the site publishing the password hashes)
- Note: nearly simultaneous problem with the discovery that the Linkedin app for iPad/iPhone was sending in clear sensible data (not relevant to Linkedin!)

Example: passwords in MySQL

- Username and password stored in the "user" table
- MySQL (from v 4.1) uses a double hash (but no salt!) to store the password
  - sha1( sha1( password ) )
- The hex encoding of the result is stored, preceded by * (to distinguish this case from MySQL < 4.1)
- Example (for the password "Superman!!!"):
  - Field user.password = *868E8E4F0E782EA610A67B01E63EF04817F60005
- Verification

```
$ echo -n 'Superman!!!'| openssl sha1 -binary | openssl sha1 -hex
(stdin)= 868e8e4f0e782ea610a67b01e63ef04817f60005
```

Iphone ransomware

- May 2014
- iCloud accounts (with 1-factor authN) violated
- Then "remote lock" used with "find my device"
- Also a message is sent to the device (iphone, ipad):
  - "Device hacked by Oleg Pliss!"
  - To regain control send 100 USD/EUR via Paypal to lock404(at)hotmail.com
- Don't pay? then use "recovery mode" (but all the device data and app are lost...)
- Paying doesn't help either! (fake Paypal account)

http://thehackernews.com/2014/05/apple-devices-hacked-by-oleg-pliss-held.html
**(Symmetric) challenge-response systems**

- a challenge (typically a random nonce) is sent to the user...
- ... who replies with the solution after a computation involving the shared secret and the challenge
- the server must know the secret in clear
- often R is a hash function (can't be encryption)

\[ \text{response} = R(\text{challenge}, SUID) \]

*Mutual authentication with symmetric challenge (v1)*

- this is the base exchange
- only the initiator provides explicitly its (claimed) identity

\[
\begin{align*}
\text{Alice} & \quad \text{Bob} \\
A & \quad C_B \\
\text{enc}(K_{AB}, C_B) & \quad \text{enc}(K_{AB}, C_B) \\
C_A & \quad \text{enc}(K_{AB}, C_A) \\
\end{align*}
\]

*Mutual authentication with symmetric challenge (v2)*

- reduction in the number of messages (better performance but no impact on security)
- used by the IBM SNA

\[
\begin{align*}
\text{Alice} & \quad \text{Bob} \\
A, C_A & \quad C_B, \text{enc}(K_{AB}, C_B) \\
\text{enc}(K_{AB}, C_B) & \quad \text{enc}(K_{AB}, C_A) \\
\end{align*}
\]
Authentication systems

Attack to the symmetric challenge protocol

(Assymmetric) challenge-response systems

Risks with asymmetric challenges

© Antonio Lioy - Politecnico di Torino (1995-2014)
**Password (one-time)**

```
server

richiesta di autenticazione

UID

utente (UID)

richiesta della password n. 48

P48UID

P47UID

P46UID

P45UID

P44UID

P43UID

P42UID

P41UID

P40UID

...```

**One-Time Passwords (OTP)**

- **original idea:**
  - Bell Labs
  - the S/KEY system
  - public-domain implementation
- **commercial implementations with automatic hardware generators (authenticator)**

**OTP provisioning to the users**

- **on "stupid" or insecure workstation:**
  - paper sheet of pre-computed passwords
  - hardware authenticator (crypto token)
- **on intelligent and secure workstation:**
  - automatically computed by an ad-hoc application
  - eventual integration into the communication sw (e.g. telnet client) or hw (e.g. modem)
The S/KEY system (I)

- RFC-1760
- the user generates a secret S
- the user computes N one-time passwords:
  - P₁ = h(S)
  - P₂ = h(P₁) = h(h(S))
  - ... 
- the user initializes the authentication server with the last generated password (e.g. P₁₀₀)
  - this password will never be used directly for authentication, but only indirectly

The S/KEY system (II)

- the server prompts for the passwords in reverse order:
  - S: P₉₉?
  - C: X
  - S: if h(X) = P₁₀₀ then access allowed + X is stored
  - in this way the server doesn’t need to know the client’s secret
- RFC-1760 uses MD4 (other choices possible)
- public-domain implementation for Unix, MS-DOS, Windows, MacOs

Il sistema S/KEY (III)

user password generation: S/KEY authentication

user has ... server has ...

\[ P_n = h^n(s) \]
\[ P_{n-1} = h^{n-1}(s) \]
\[ P_2 = h(h(S)) = h^2(S) \]
\[ P_1 = h(S) \]

\[ \text{initial secret } S \]

\[ h \]

\[ h \]

\[ h \]
S/KEY – generation of the password list
- the user inserts a pass phrase (PP):
  - minimum 8 char long
  - secret! (if disclosed then the security of S/KEY is compromised)
- PP is concatenated with a server-provided seed
  - the seed is not secret (sent in cleartext from S to C)
  - allows to use the same PP for multiple servers
    (using different seeds) and to safely reuse the same
    PP by changing the seed
- a 64 bit quantity is extracted from the MD4 hash
  (by XORing the first / third 32 bit groups and the
  second / fourth groups)

S/KEY – passwords
- 64 bit passwords are a compromise
- neither too long nor too short
- possible typing as a sequence of 6 short english
  words chosen from a dictionary of 2048
  (e.g. “A”, “ABE”, “ACE”, “ACT”, “AD”, “ADA”)
- client and server must share the same dictionary

OTP problems
- generally uncomfortable
- uncomfortable when used to access multiple
  password-based services (e.g. POP with periodic
  check of the mailbox)
- expensive when based on hw authenticators
- paper-based passwords cannot be used by a
  process but only by a human operator
- generation of good random passwords
- password provisioning (generator? SMS?)
- when time is used to generate the OTP, time
  synchronization between user and system
Problems of hw authenticators

- denial-of-service:
  - deliberately wrong attempts to trigger account blocking
- social engineering:
  - phone call to simulate loss of the authenticator and remotely initialize a new one

Password (one-time token-based)

- server authentication request → UID
- user (UID) request of the 11:07 password → UID : S_{UID}
- P1107_{UID} = p ( 11:07, S_{UID} ) ?

RSA SecurID

- invented and patented by Security Dynamics
- time-based synchronous OTP technique:
  - P_{UID} ( t ) = h ( S_{UID}, t )
- access code (token-code):
  - 8 digits
  - random, never repeats itself
  - changes every 60 s
  - maximum drift 15 s / year
  - expires in 4 years
  - based on proprietary and secret (!) hash algorithm
SecurID: architecture

- the client sends in clear user, PIN, token-code (seed, time)
- based on user and PIN the server verifies against three possible token-codes: $T_{C,T_1}, T_{C_0}, T_{C+1}$
- duress code: PIN to generate an alarm (useful for authentication under threat)
- wrong authentication attempts limited (default: 10)
- may have three different keys per device

SecurID: hardware

- SecurID Card (e.g. SD200): classic device (credit-card size)
- SecurID PinPad (e.g. SD520): local PIN keying to generate token-code*, sent to the server with user
- SecurID Key Fob (e.g. SD600, SID700): key fob
- SecurID Key Fob with smart-card (e.g. SID800)

SecurID: software authenticators

- SoftID
  - works as a SecurID PinPad but is a sw application
  - automatic or manual transmission of the token-code
  - problem: clock synchronization
RSA SecurID: prodotti recenti

- STD700
- SDE600
- SDE800
- SDE200
- SecurID Token

SecurID: architecture

- ACE server
- ACE client
- TELNET server
- TELNET client
- SecurID (normal)
- SecurID (pinpad)
- DBMS server
- DBMS client
- user, PIN, TC
- user, TC*
- token OK?
- OK!
- KO!

SecurID: client

- ACE/client
  - installed at the application that wishes to use SecurID for authenticating its users
  - manages the dialogue with the ACE/server
  - encrypted channel
**SecurID: server**

- **ACE/server:**
  - installed at the authentication server
  - talks with the ACE/client for responding to authentication requests that use the SecurID tokens
  - monitor, audit and report
  - GUI management interface
  - authentication API (for new applications)
  - SQL interface to access a DBMS (already) storing the user data

**Out-of-band authentication**

1. authentication request
2. UID + PUID
3. authN data generation
4. OOB transmission
5. OOB data

**Authentication of human beings**

- how can we be sure of interacting with a human being rather than with a program (e.g. sensing a password stored in a file)?
- two solutions:
  - CAPTCHA techniques (Completely Automated Public Turing test to tell Computers and Humans Apart)
    - e.g. picture with images of distorted characters
  - biometric techniques
    - e.g. fingerprint
### Biometric systems

- measure of one biologic characteristics of the user
- main characteristics being used:
  - fingerprint
  - voice
  - retinal scan
  - iris scan
  - hands’ blood vein pattern
- useful to "locally" replace a PIN or a password

### Problems of biometric systems

- **FAR** = False Acceptance Rate
- **FRR** = False Rejection Rate
- FAR and FRR may be partly tuned but they heavily depend on the cost of the device
- variable biological characteristics:
  - finger wound
  - voice altered due to emotion
  - retinal blood pattern altered due to alcohol or drug

---

![FAR / FRR Graph](image)
Problems of biometric systems

- psychological acceptance:
  - “Big Brother” syndrome (=personal data collection)
  - some technologies are intrusive and could harm
- lack of a standard API / SPI:
  - high development costs
  - heavy dependence on single/few vendors
- API / SPI being developed:
  - standard and unified
  - based on CDSA

API? SPI? middleware!

- API (Application Programming Interface)
- SPI (Service Programming Interface)
  - device / service no. 1
  - device / service no. 2
  - device / service no. 3

Kerberos

- authentication system based on a TTP (Trusted Third Party)
- invented as part of the MIT project Athena
- user password never transmitted but only used locally as cryptographic (symmetric) key
- realm = Kerberos domain, that is the set of systems that use Kerberos as authentication system
- credential = user.instance@realm
**Kerberos**

- **ticket**
  - data structure to authenticate a client to a server
  - variable lifetime
    - (V4: max 21 hours = 5’ x 255)
    - (V5: unlimited)
  - encrypted with the DES key of the target server
  - bound to the IP address of the client
  - bound to just one credential
  - simple or mutual authentication

---

**Kerberos high-level view**

- AS (Authentication Server)
  - request
- TGS (Ticket Granting Server)
- client
- server

---

**Kerberos: data formats (v4)**

**TICKET**

- server-id
- client-id
- client-address
- timestamp
- life
  - $K_{SC}$
  - $K_S$

**AUTHENTICATOR**

- client-id
- client-address
- timestamp
  - $K_{SC}$
  - $K_S$
TGT request

\[ \{ K_{C,TGS}, \{ T_{C,TGS} \} K_{TGS} \} K_{C} \]

Ticket request

\[ s, \{ T_{C,TGS} \} K_{TGS}, \{ A_{C} \} K_{C,TGS} \]

\[ \{ \{ T_{C,S} \} K_{S}, K_{C,S} \} K_{C,TGS} \]

Ticket use

\[ \{ T_{C,S} \} K_{S}, \{ A_{C} \} K_{C,S} \]

\[ \{ \text{timestamp}(A_{C}) + 1 \} K_{C,S} \]
Kerberos versions

- MIT V4 (the original public one)
- MIT V5 (RFC-1510)
  - not only DES
  - extended ticket lifetime (begin-end)
  - inter-realm authentication
  - forwardable ticket
  - extendable ticket
- OSF-DCE
  - based on MIT V5
  - implemented as RPC rather than a message exchange protocol

Kerberos: problems

- clock synchronization required:
  - within a LAN it's useful anyway
  - in WAN may originate problems
  - Kryptoknight (alias IBM NetSP) doesn't require clock synchronization
- remote access needs cleartext password:
  - encrypted channel or integration with OTP, symmetric or asymmetric challenge
  - Kerberized dial-up modems

Kerberos: advantages

- single login to all Kerberized services
  - K-POP, K-NFS, K-LPD
  - K-telnet, K-ftp
  - K-dbms
- the ticket mechanism is ideally for intermittent connections
  - mobile computers
  - ISDN, WiFi, UMTS
- increasing commercial support
  (MS has adopted Kerberos* since Windows-2000)
SSO (Single Sign-On)

- the user has a single “credential” to authenticate himself and access any service in the system
- fictitious SSO:
  - client for automatic password synchronization / management (alias “password wallet”)
  - specific for some applications only
- integral SSO:
  - multiapplication authentication techniques (e.g. asymmetric challenge, Kerberos)
  - likely requires a change in the applications
  - multi-domain SSO (e.g. with SAML tokens, that generalize Kerberos tickets)

Authentication interoperability

- OATH (www.openauthentication.org)
- interoperability of authentication systems based on OTP, symmetric or asymmetric challenge
- development of standards for the client-server protocol and the data format on the client

OATH specifications

- http://www.openauthentication.org/specifications
- HOTP (HMAC OTP, RFC-4226)
- TOTP (Time-based OTP, RFC-6238)
- OATH challenge-response protocol (OCRA, RFC-6287)
- Portable Symmetric Key Container (PSKC, RFC-6030)
  - XML-based key container for transporting symmetric keys and key-related meta-data
- Dynamic Symmetric Key Provisioning Protocol (DSKPP, RFC-6063)
  - client-server protocol for provisioning symmetric keys to a crypto-engine by a key-provisioning server