Ch. 2 - Access Control

Secure software development and web security

R. Absil

Haute École Bruxelles-Brabant
École supérieure d'Informatique

September 23, 2021

Table of contents

Introduction

Access control and vulnerabilities

- Many systems (computed related or not) have restricted access
- Some form of control is made on entry
- When a user wants to access a system, credentials are beeing presented
  - If the credential lies in some access control list, access is granted
  - If it does not, access is denied
  - Whatever happens, requests are logged to a database
- If access is forced, a detection system triggers an alarm

Two base principles

1. Record everything
2. Detect and deal with intrusions

Credentials

Question

- What shape can credentials take ?

- Always one of the following
  - Something you know
    - A password, a PIN
  - Something you have
    - A door key, a card
  - Something you are
    - Fingerprints, retinal imprint
  - Someone you know
    - Contact inside the restricted area

Computer related credentials

- Access control is often associated with authentication
- Only authenticated users are allowed
  - To grant access, the system must know who the user is
- Common authentication credentials
  - Passwords and passphrases
  - Security tokens and cryptographic keys

Base principle : least priviledge

- Always give the least privileges needed to
  - authenticated users
  - services executed

Other forms of access restriction

- Many file systems provide some form of inherent restriction on files
  - Unix file system permissions
- Datas that are not supposed to be read by anyone are "encrypted"
  - Sensitive files
  - Password files
- Object oriented programming languages provide access control on their functions
  - `private`, `protected`, etc.
- These systems alone are not enough

What this chapter is about

- Common vulnerabilities related to access control
  - How to bypass an arbitrary system without the codes
- Common different types of authentication systems
  - Security tokens two-factor authentication
  - Cryptographic key authentification
  - Password authentification
- Attack vectors on these systems
- Homework

Vulnerabilities

Common vector of attack

- Cryptographic systems are often hard to attack
  - Algorithmic complexity is against the hacker
- How to penetrate a system ?
  - Attack something else, somewhere else

Example

- You want to steal user data in order to sell it to marketing companies
- You can either
  - break encryption between some user and the server
  - try to log in as some user
  - try to log is as an administrator

- Authentication and permission managers are privileged targets

Two widespread vulnerabilities

Broken access control

- Bypass permission system
  - It doesn't check permissions after login
  - References can be guessed (and rewritten)

Bad session management

- Steal someone credentials
  - Unsafe storage of credentials
  - Unsafe session management

Table of contents

Illustration : an example

- Message : login, pwd, usrId, resrcId

```java
public Entry[] processRetrieval(SignedObject sm)
  throws SecurityException, AuthentificationException, IllegalAccessException
{
  if(sm.verify(s_publicKey, s_engine)) //check signature
  {
    SealedObject ciphered = (SealedObject)sm.getObject(); //recover ciphered object
    Message plain = (Message)ciphered.getObject(c_privateKey); //remove cipher

    boolean ok = auth_mgr.auth(plain.login(), hash.digest(plain.passwd().getBytes()));
    if(ok) //user is authenticated successfully
    {
      boolean access = access_mgr.check(plain.getUserId(), plain.getRessourceId());

      if(access)
        return db_mgr.retrieveEntries(plain.getUserId(), plain.getRessourceId());
      else
        throw new IllegalAccessException("Ressource access denied");
    }
    else
      throw new AuthentificationException("Invalid login/password");
  }
  else
    throw new SecurityException("Invalid message signature");
}
```

Principle

Idea

- Access a restricted resource of functionality by "guessing" a name

- For data, applications and API often use the real name or id of an object
  - For retrieval or page generation
- For functions, URLs and function names are often easily predictable
- Applications and API do not always check that a *registered* user has access to some resource
- Manipulate parameters and attack (injection)
- Careful code analysis often allows to detect and prevent such attacks

Risk analysis

1. An attacker is *always* a registered user
2. Such an attack can be easily exploited
3. These a vulnerabilities are *very* common
4. These vulnerabilities can be easily detected
5. Severity of the risk is moderated

Reference

- OWASP Top 10 - 2020

Prevention

- Verify that *all* data and function references have appropriate defenses
- After regular authentication,
  - for data reference, make sure the application (server) ensures the user is authorized for that data, e.g., with a reference map
  - for function reference, make sure the application ensures the user has required privileges for a call
- Each use of a reference from an untrusted source must include access control
- Use indirect references
  - Do not use DB keys
  - For user x, drop down a list of authorized resources / functions from 1 to k
  - This id denotes selection, *not a resource*
  - A user-depending table maps selection to resources / functions

Illustration

- Injection vulnerable code

```
pstmt.setString( 1, request.getParameter("acct")); //prepare statement
ResultSet results = pstmt.executeQuery( );
```

- URL injecton attack

```
http://example.com/app/accountInfo?acct=notmyacct
```

Table of contents

Principle

Idea

- Steal (temporarily) a user's credentials, through bad session management or unsafe password storage

- Use flaws and leaks in session or authentication functions in order to impersonate someone
- Made possible since developers frequently design custom authentications and session schemes
  - Hard to do properly, though
  - Frequent flaws
- Privileges accounts are often targeted
  - But success can compromise every account

Risk analysis

1. Attacker : any extern, or malicious intern user
2. Average exploitability
   - Often, control is temporary, or doesn't give that much priviledges
3. Common vulnerability
   - Often custom made authentification and session schemes
4. Average detectability
   - Again, custom made systems
5. Severe impact
   - Non repudiation can be compromised

Reference

- OWASP Top 10 - 2020

Weaknesses

- User access codes are not properly stored using cryptogrphic hash functions and/or cipher algorithms
- Credentials can be guessed or overwritten through weak policies
  - Account creation, password change, password recovery, session ids, etc.
- Session ids exposed through URL
- Unproper token and session ids invalidation
  - When times out or on logout
- Unproper token and session ids rotation
  - After successful login

Example

- A website allows URL rewriting
  - http://stuff.be/discounts/list;jsessid=2P0OC2JSNDLPSKHCJUN2JV?cat=electro
  - Some user shares the URL on social networks to make his friends know about the discount
    - The user also gives away his session id
  - Using the link leaks the user's session, along with its credit card information, etc.
- A website does not implement timeouts correctly
  - On a public or chared computer, a user does not logs out properly, but just closes its web browser
  - An attacker using the same browser later is still logged in
- Some user or attacker have access to password database, not properly hashed and / or salted
  - Every account can be compromised

Detection and prevention

Detection

- Analyse
  - storage and transmission of access codes
  - login and logout procedures

Prevention

- Unique access control and session management
  1. Respect standards and norms in authentication and session management[1]
  2. Simple interface for developers
- Force timeouts, invalidations and rotations of session ids and tokens

[1] https://www.owasp.org/index.php/ASVS

Authentication

Several types of authentication

- There are several types of authentication protocols

Example

- Passwords (popular)
- Security tokens (required hardware)
- Cryptographic keys

- We will see these three

What authentication is about

Goal

- Establish a level of trust

- Make sure the identification is authentic
- Requires you to provide one of the following
  - Something you know
  - Something you have
  - Something you are
- Drawbacks
  - Something you know : cultural, social engineering
  - Something you have : can be stolen / lost
  - Something you are : requires initial measurement

Example

- Something you know
  - Passwords
  - PIN codes
- Something you have
  - Token
  - Piece of hardware issued for the user
- Something you are
  - Biometrics
  - Fingerprints, retinal imprint, etc.

Multi-factor authentication

- Idea : force the user to present more than one piece of evidence
  - Two-factor authentication : present two of the three something you know/have/are
- Motivation : increase trust during the authentication process
  - Unauthorized users are less likely to provide every factor

Example

- Cash withdrawal : you need *both* the card, and the PIN
- Gmail/Steam/FB : logging in from unknown devices requires a one-time password sent by email/phone

- Most of the time : something you know (password) along with something you have (token)

Table of contents

Different types of tokens

- We can categorise tokens into three sets
  1. Paper tokens : one time passwords, sets of codes
  2. Software tokens : relies on a software installed on the client computer/device
  3. Hardware tokens : requires physical device
- Tokens can be physically connected (auto-transmits data) or not

Primary constraint

- The token *has* to implement the fundamental principal of "something you have"
  - And *only you* have

- The token has to be *unique*

Security considerations

- Some form of protection against copy must be provided
- Rules out paper tokens
- Case of software tokens
  - Must be securely linked to the hardware on which it is installed
  - Makes the computer a hardware token
  - Reduces portability
- Case of hardware tokens
  - Needs to prevent tampering
  - Needs to be protected against reverse engineering
- For two-factor authentication, it is acknowledged that hard tokens are unmatched
- Basic idea : pack up some seed

Software tokens

- Designed to be flexible
  - Difficult to provide a hard token to someone far away
  - Easy to revoke should an employee be terminated
- The seed must be transmitted securely
- A software token *can* be copied
  - If passphrase-protected, an eavesdropper can copy it. If the destinatory changes the password, the pirate still has his copy with the original passphrase
  - If on an infected computer (malware, keylogger), other opportunities to copy

Consequence

- Proves (not only) the user "has" the token

Conclusion

- Software tokens are considered instances of "something you know"
  - Passphrase to unlock the seed record
- A password + a software token are two instances of "something you know"
- Some experts state soft tokens are not tokens at all

Rhetorical question

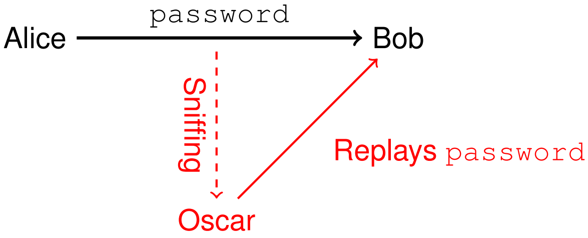
- "Does the complexity of properly distributing seed records in a secure channel as well as the expense of managing and supporting the software token application code really provide sufficient benefit over a simple, yet strong, password only method?"[2]

[2] Securology, 2007

Password tokens

- Idea : hide some secret information inside of a token
  - A "password"
- Types of password tokens
  1. Static : directly and securely store the password
     - Vulnerable to replay attacks
  2. One-time password token
     - Asynchronous password token
     - Synchronous password token
     - Challenge answer

Replay attack



- Counter : include randomness in the transaction
  - Session id, etc.

One-time asynchronous password

- Idea : a password is only valid for one login / transaction
  - Avoids cultural problems with password reuse, non randomness, post-its, etc.
- Not vulnerable to replay attacks
  - If an attacker recovers a password already used, he cannot misuse it
  - That password is invalid now
- Generation is often based either on
  - time synchronisation between the server and the client
  - the previously generated password
  - a challenge

Lamport scheme

- Each generated password relies on
  1. some cryptographic hash function
  2. some (not secret) seed s
  3. the previously generated password

Algorithm

1. Initially, compute $p_0 = H^k(s) = H(H(H(\dots H(s)\dots)))$, and k "large"
2. Store p_0 on the server
3. The first password required is $p_1 = H^{k-1}(s)$, store p_1
4. The second password required is $p_2 = H^{k-2}(s)$, store p_2
5. Repeat

Advantages and drawbacks

Advantages

- Safe against replay attacks

Drawbacks

- If a password is compromised, every previously generated password is

- There other one-time password policies
  - OTPW (unix-like systems) : passwords are generated from pseudorandom number generators (combines RIPEMD-160 hashes of several shell output commands)

One-time synchronous password tokens

Principle

- Internally, the token generates a one-time password every unit of time
- The server stores the seed of each token
- The password generated by the client must be the same as the one generated by the server
- If clocks become out of synch, the drift of the tokens are usually stored as well

- Example : RSA SecurID, Gemalto's IDaaS

Challenge-response tokens

- Principle : present the client with a question (challenge), who must provide a valid answer (response
- Implemented with the help of cryptographic hash functions
- Challenge a nounce, expect something dependent from "something you have"
  - Generate nounces with secure random generators and a hash function

Example

- Kerberos : send encrypted integer n, expect $n + 1$

- Mutual authentication : perform challenge-response handshake in both directions

Table of contents

Overview

Drawbacks of other systems

- Passwords
  - Password reuse, weak randomness, etc.
- Tokens
  - Cost (hard)
  - Deployment (soft)

Cryptographic key authentication

- Alice sends data signed with her private key to Bob
- Bob authenticates her

- Prerequisite : Bob must have the public key of Alice

Storing private keys

- A private key is usually password-encrypted and stored *locally* on a *trusted* computer
- To access the private key, the user must provide the password
- Decrypted private keys are often cached and released on demand
  - Linux : `ssh-agent` (`ptrace`-proof)
  - Critics over `sudo`
- When the session ends or after a timeout, the private key is *physically* cleared from memory

OpenSSH use (client)

- Generate a 4096 bytes RSA key : `ssh-keygen`
  - A few configurations options are asked (storage location, passphrase, etc.)

```
ssh-keygen -t rsa -b 4096 -C "my␣rsa␣key"
```

- Send your public key to some server
- Possible warning if the server is unknow
  - Safely removed with certification

```
ssh-copy-id login@host.com
```

- Login

```
ssh login@host.com
```

- Administrators should disable password logins for these users
  - Change the flag PasswordAuthentication in `/etc/ssh/sshd_config`
  - Restart the SSH service

Possible attacks

- Since the key is stored locally and on a trusted computer, a remote attack cannot be performed
- Should the computer already be infected at storage, data remanence attacks can be performed
  - If an unsafe use of private keys is made
    - No `ssh-agent`
    - Copy the private key on flash driver, then `rm` it locally
- If the computer is stolen, cleverly bruteforcing the key is possible
  - If you're clever, and if it is possible
  - The user has time to change keys

Table of contents

Introduction

- Many systems are password protected
  - Mail servers, social networks, online accounts, etc.

Question

- How to authenticate "securely"?
  - Transmission
  - Storage
  - Common attacks

A problem of culture

- Often, security systems work fine "in theory"
- In practice, authorised users often compromise the system

Password case

- Low entropy : dictionary of common passwords
- Too hard to remember : "post-it on the screen"
- Password reuse
- Key logging

- Some experts think we need another system

Transmission

- Obviously, a password cannot transmitted as plain text
- An authentication request if often *not* signed
  1. A signature already proves identity
  2. Would require a second set of keys
     - And it would also cause "cultural problems"
- In practice, passwords are transmitted ciphered
  - Either with an asymmetric cipher,
  - Or with session key wrapping
- When received by the server, the password is unciphered and submitted to the system

Storage

- Pairs `login,password` are stored in (huge) databases
- Considering their size and the frequency of authentication requests, ciphering the tables is inadvisable
  - Beside, auto-ciphering would not solve most vulnerabilities

Question

- How to store passwords so that an attacker able to see the tables cannot do anything with this information?

Answer

- Use cryptographic hash functions
- Store password hashes, not passwords

Principle

1. Reception of a ciphered pair `login,password`
2. Unciphering this pair with the server private key, or with a session key
3. Compute hash *h* of `password`
4. Check if pair `login,`*h* is in the database
   - If yes, access granted
   - If not, authentication failure

Advantages

- Fast : hashes are easy to compute
- Safe : resistant to pre-image
- Unlikely collisions

Password salting

Drawback

- Two identical passwords have the same hashes
  - Password reuse
  - Same password for different users

Solution : salt

- Affix a string to plain text passwords : the salt
- Compute hash h' of this
- Salt is different for each users
- Salt is not secret
- Store the triple `login,salt,`h'

Dictionary attack

- Very common attack against authentication
- Takes advantage that some passwords are very common

Principle

1. Build a list of common passwords
   - With their hashes out of several cryptographic hash functions
2. Look for a known hash in the table
3. If there is a match, the attack is successful

- Protection
  1. Bound the number of request in time
  2. Bound the number of failures
  3. Make password policy more severe

Striving for more

Observation

- If the dictionary is *large enough*, likely success
  - Idea : try to invert the hash function on enough entries

- But it is impossible to simply brute-force the password space
- How to compromise?

Preprocessing compromise

- Assume we have a cryptographic hash function H with output of size n
- Let P be the set of all passwords accepted by the policy

Objective

- Build up a data structure such that given an output h of H, it is possible to
  - either find $p \in P$ such that $H(p) = h$,
  - or state there is no such p in the data structure

- Brute force : compute $H(p)$ for all $p \in P$
  - Impossible in practice
  - Birthday attack

Precomputed hash chain

- Define a *reduction function* R mapping a hash to some $p \in P$
  - It is *not* required that R is the inverse of H (pre-image)
  - *No* constraint on collisions
  - R *should* uniformly distribute its output

Principle

- Alternate H and R to create "chains"

$$\text{password} \xrightarrow{H} \text{AC42D321} \xrightarrow{R} \text{xfcgvhjb} \xrightarrow{H} \text{DC4256FF} \xrightarrow{R} \text{vgfohiyu} \xrightarrow{H} \text{23DEE32F} \xrightarrow{R} \text{cvbhjklk}$$

- We only store the head and the tail of a chain in the data structure

Password recovery

- Given some hash h, we apply R, then H, then R, then H, etc.
- Goal : at any point after computing R, discover the tail of some chain
- Build up the password from the head of the matching chain
  1. Starting from the head, we apply H, then R, then H, etc.
  2. If we get h, the previous element is the password we are looking for
  3. It is possible not to find h, since R can have collisions

Data structure

- Generate *a lot* of random passwords
- Compute *lengthy* chains
- For each chain, store the head and tail

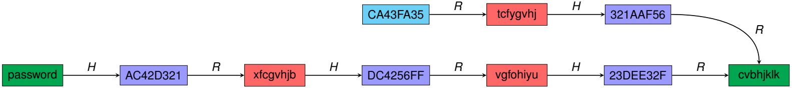
Illustration : success

password $\xrightarrow{H}$ AC42D321 $\xrightarrow{R}$ xfcgvhjb $\xrightarrow{H}$ DC4256FF $\xrightarrow{R}$ vgfohiyu $\xrightarrow{H}$ 23DEE32F $\xrightarrow{R}$ cvbhjklk

- We are looking for the password matching the hash "DC4256FF"
- We apply *R*, and we get "vgfohiyu" : it is not a tail
- We apply *H* then *R*, we get "cvbhjklk" : it is a tail
- From "password" (the matching head), we apply *H* then *R* twice to get "DC4256FF"
- The password we were looking for is "xfcgvhjb"

Illustration : collision failure

- We are looking for the password with hash "CA43FA35"
- We apply *R*, we get "tcfygvhj"
- We apply *H*, we get "321AAF56"
- We apply *R*, we get "cvbhjklk"



- Building the password back from `password`, we *never* get "CA43FA35"
  - Failure
- Since *R* is far from being collision-free, chains can "merge"

Avoid collisions

Question

- How to avoid collisions in precomputed hash chains?

Solution

- Use several reduction functions
  - In practice, we use *a lot* of them

- Consequently, if two chains are to merge, they need to merge on *the same* value on *the same* spot.
  - Unlikely

Rainbow table

- Build k reduction functions R_i
  - In practice $\simeq 50\,000$
- From a (sufficiently large) set of random passwords, build a set of precomputed hash chains
  - Use H, then R_1, then H, then R_2, ..., then H, then R_k
- Only store the head and tails of the chains (as pairs)
- Optimisations
  - Sort entries by tails
  - Delete duplicated tails

Advantage

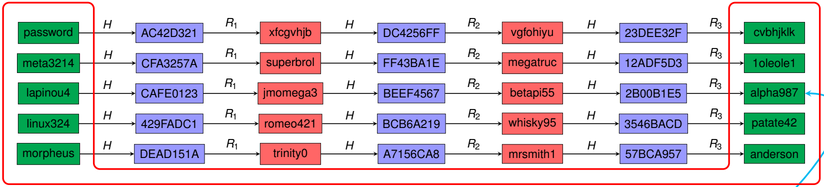
1. Little space used (compared to $|P|$), since we only store heads and tails
2. Low probability of collision on reduction functions

Rainbow attack

- Objective : crack some hash h
- How to
  1. Check whether $R_k(h)$ is the end of some precomputed hash chain
     - If it is, build up p back
  2. Check whether $R_k(H(R_{k-1}(h)))$ is the end of some precomputed hash chain
     - If it is, build up p back
  3. Iterate this technique until we check the complete chain $R_1 - H - R_2 - \cdots - H - R_k$
     - If it fails (again), the attack is a failure
- The bigger the table, the higher the chances of success
- Requires *a lot* of reduction functions to avoid collisions
- Requires a fitting implementation (BDD + SGBD, efficient hash tables, etc.)
- Salting prevents "mass cracking" of passwords
  - Along with usual advantages

Illustration

1. Build up the rainbow table
2. Check the last reduction
3. After i failures, check $R_{k-i} - H - R_{k-i+1} - \cdots - H - R_k$
4. Build the password back if a match is found



RAINBOW TABLE

Crack "BEEF4567"

1. Check R_k : "BEEF4567" $\xrightarrow{R_3}$ "rienavwr" $\longrightarrow$ Failure
2. Check $R_{k-1} - H - R_k$: "BEEF4567" $\xrightarrow{R_2}$ "betapi55" $\xrightarrow{H}$ "2B00B1E5" $\xrightarrow{R_3}$ "alpha987" $\longrightarrow$ Success

Reduction functions

- Objective : map a hash to a password
  - No no-collision requirement
  - Has to uniformly distribute the output

Construction example

- Build up the i$^{\text{th}}$ function
  1. Parse the hash as some integer x
  2. Return $(x + i) \bmod |P|$
  3. Map the output to a string, e.g., with a character table

- Considering lower case alphanumeric passwords of length 8, $|P| = 36^8$

Homework

Implementation

- Implement a rainbow attack

Instructions

- Minimum : lower case alphanumeric passwords of length 8
  - The bigger / more complex, the better
- Passwords are stored as "one time SHA-256 hashs"
- Unsalted passwords

- Two sessions, groups of max. 2 students
- Submit scripts (generation and cracking)
  - *Not* the table
  - Dependencies must be in updated debian packages
  - No proprietary code
- Choose whatever language you think is appropriate