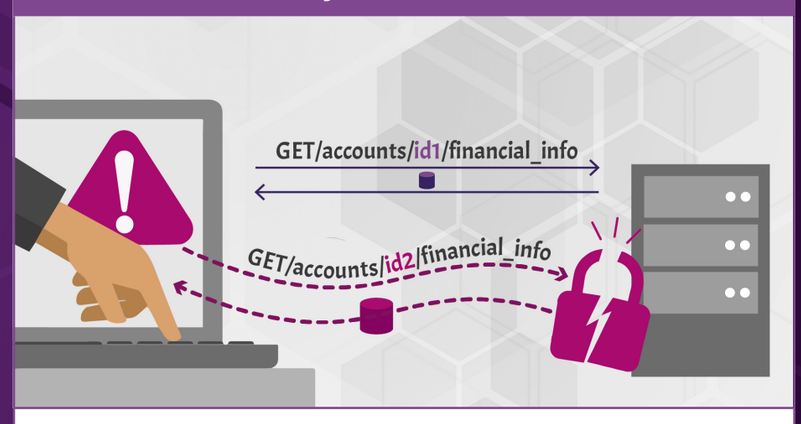
CHEAT SHEET

OWASP API Security Top 10

42crunch 42CRUNCH.COM

A1: BROKEN OBJECT LEVEL AUTHORIZATION



Attacker substitutes ID of their resource in API call with an ID of a resource belonging to another user. Lack of proper authorization checks allows access. This attack is also known as IDOR (Insecure Direct Object Reference).

USE CASES
- API call parameters use IDs of resourced accessed by the API
- /api/shop1/financial_details
- Attackers replace the IDs of their resources with a different, which they guessed
- /api/shop2/financial_details
- The API does not check permissions and lets the call through
- Problem is aggravated if IDs can be enumerated
- /api/123/financial_details

HOW TO PREVENT
- Implement authorization checks with user policies and hierarchy
- Don't rely on IDs sent from client. Use IDs stored in the session object instead.
- Check authorization each time there is a client request to access database
- Use random non-guessable IDs (UUIDs)

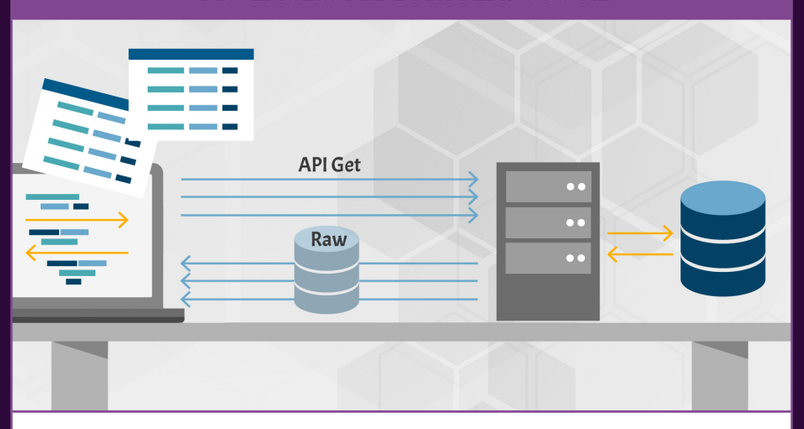
A2: BROKEN AUTHENTICATION

Poorly implemented API authentication allowing attackers to assume other users' identities.

USE CASES
- Unprotected APIs that are considered "internal"
- Weak authentication not following industry best practices
- Weak, not rotating API keys
- Weak, plain text, encrypted, poorly hashed, shared/default passwords
- Susceptible to brute force attacks and credential stuffing
- Credentials and keys in URL
- Lack of access token validation (including JWT validation)
- Unsigned, weakly signed, non-expiring JWTs

HOW TO PREVENT
- Check all possible ways to authenticate to all APIs
- Password reset APIs and one-time links also allow users to get authenticated and should be protected just as seriously
- Use standard authentication, token generation, password storage, MFA
- Use short-lived access tokens
- Authenticate your apps (so you know who is talking to you)
- Use stricter rate-limiting for authentication, implement lockout policies and weak password checks

A3: EXCESSIVE DATA EXPOSURE



API exposing a lot more data than the client legitimately needs, relying on the client to do the filtering. Attacker goes directly to the API and has it all.

USE CASES
- APIs return full data objects as they are stored by the database
- Client application shows only the data that user needs to see
- Attacker calls the API directly and gets sensitive data

HOW TO PREVENT
- Never rely on client to filter data
- Review all responses and adapt responses to what the API consumers really need
- Define schemas of all the API responses
- Don't forget about error responses
- Identify all the sensitive or PII info and justify its use
- Enforce response checks to prevent accidental data and exception leaks

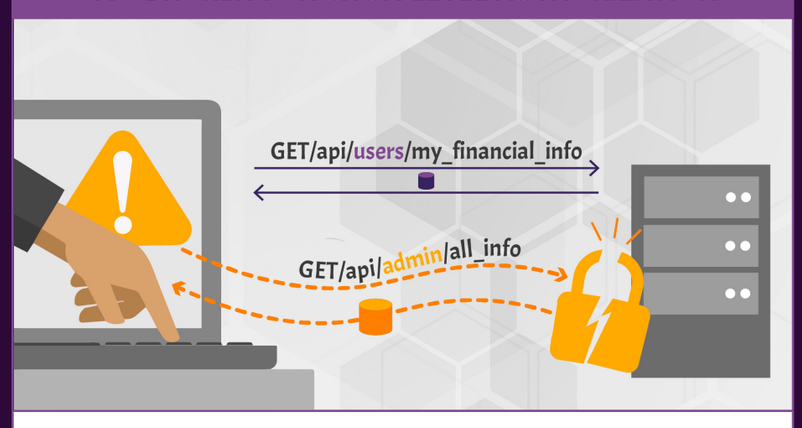
A4: LACK OF RESOURCES & RATE LIMITING

API is not protected against an excessive amount of calls or payload sizes. Attackers use that for DoS and brute force attacks.

USE CASES
- Attacker overloading the API
- Excessive rate of requests
- Request or field sizes
- "Zip bombs"

HOW TO PREVENT
- Rate limiting
- Payload size limits
- Rate limits specific to API methods, clients, addresses
- Checks on compression ratios
- Limits on container resources

A5: BROKEN FUNCTION LEVEL AUTHORIZATION



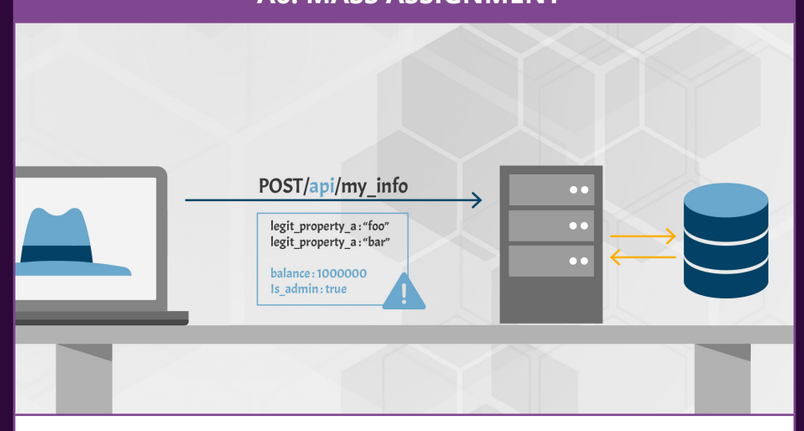
API relies on client to use user level or admin level APIs. Attacker figures out the "hidden" admin API methods and invokes them directly.

USE CASES
- Some administrative functions are exposed as APIs
- Non-privileged users can access these functions if they know how
- Can be a matter of knowing the URL, using a different verb or parameter

```
/api/users/v1/user/myinfo
/api/admins/v1/users/all
```

HOW TO PREVENT
- Don't rely on app to enforce admin access
- Deny all access by default
- Grant access based on specific roles
- Properly design and test authorization

A6: MASS ASSIGNMENT



USE CASES
- API working with the data structures
- Received payload is blindly transformed into an object and stored

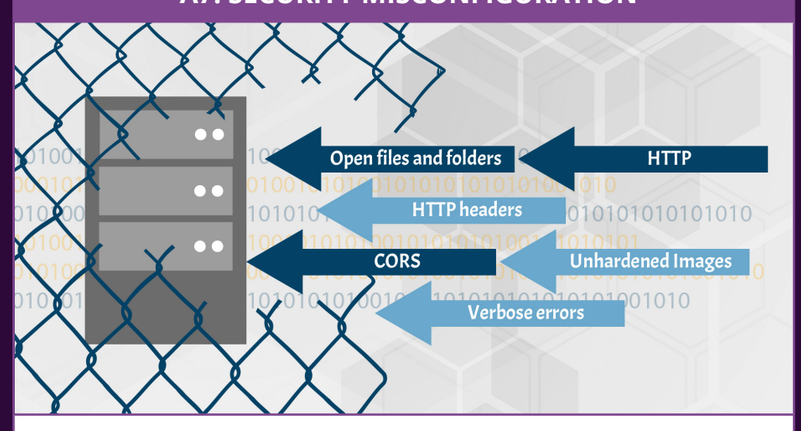
```
NodeJS:
var user = new User(req.body);
user.save();

Rails:
@user = User.new(params[:user])
```

- Attackers can guess the fields by looking at the GET request data

HOW TO PREVENT
- Don't automatically bind incoming data and internal objects
- Explicitly define all the parameters and payloads you are expecting
- For object schemas, use the readOnly set to true for all properties that can be retrieved via APIs but should never be modified
- Precisely define at design time the schemas, types, patterns you will accept in requests and enforce them at runtime

A7: SECURITY MISCONFIGURATION

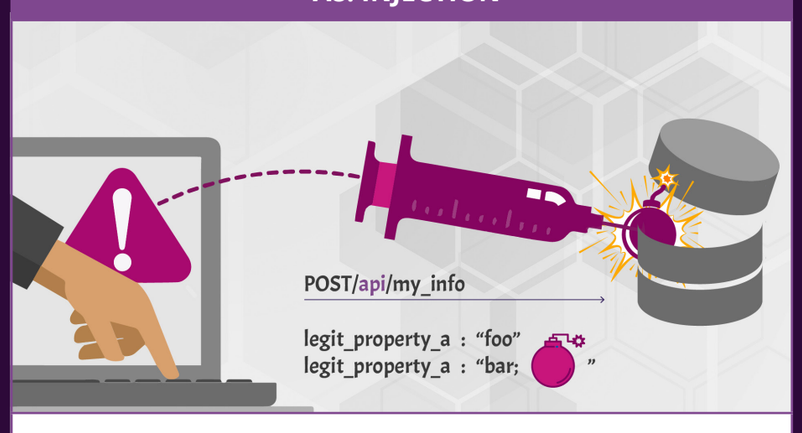


Poor configuration of the API servers allows attackers to exploit them.

USE CASES
- Unpatched systems
- Unprotected files and directories
- Unhardened images
- Missing, outdated, misconfigured TLS
- Exposed storage or server management panels
- Missing CORS policy or security headers
- Error messages with stack traces
- Unnecessary features enabled

HOW TO PREVENT
- Repeatable hardening and patching processes
- Automated process to locate configuration flaws
- Disable unnecessary features
- Restrict administrative access
- Define and enforce all outputs including errors

A8: INJECTION

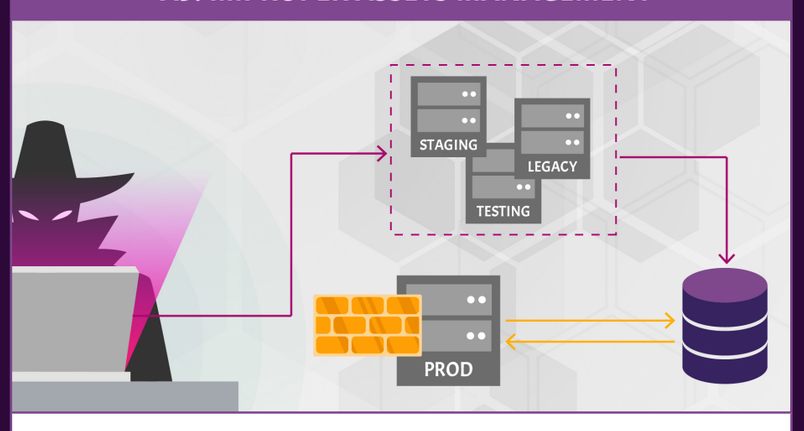


Attacker constructs API calls that include SQL-, NoSQL-, LDAP-, OS- and other commands that the API or backend behind it blindly executes.

USE CASES
Attackers send malicious input to be forwarded to an internal interpreter:
- SQL
- NoSQL
- LDAP
- OS commands
- XML parsers
- Object-Relational Mapping (ORM)

HOW TO PREVENT
- Never trust your API consumers, even if internal
- Strictly define all input data: schemas, types, string patterns - and enforce them at runtime
- Validate, filter, sanitize all incoming data
- Define, limit, and enforce API outputs to prevent data leaks

A9: IMPROPER ASSETS MANAGEMENT



Attacker finds non-production versions of the API: such as staging, testing, beta or earlier versions - that are not as well protected, and uses those to launch the attack.

USE CASES
- DevOps, cloud, containers, K8S make having multiple deployments easy (Dev, Test, Branches, Staging, Old versions)
- Desire to maintain backward compatibility forces to leave old APIs running
- Old or non-production versions are not properly maintained
- These endpoints still have access to production data
- Once authenticated with one endpoint, attacker may switch to the other

HOW TO PREVENT
- Inventory all API hosts
- Limit access to anything that should not be public
- Limit access to production data. Segregate access to production and non-production data.
- Implement additional external controls such as API firewalls
- Properly retire old versions or backport security fixes
- Implement strict authentication, redirects, CORS, etc.

A10: INSUFFICIENT LOGGING & MONITORING

Lack of proper logging, monitoring, and alerting let attacks go unnoticed.

USE CASES
- Lack of logging, monitoring, alerting allow attackers to go unnoticed
- Logs are not protected for integrity
- Logs are not integrated into Security Information and Event Management (SIEM) systems
- Logs and alerts are poorly designed
- Companies rely on manual rather than automated systems

HOW TO PREVENT
- Log failed attempts, denied access, input validation failures, any failures in security policy checks
- Ensure that logs are formatted to be consumable by other tools
- Protect logs as highly sensitive
- Include enough detail to identify attackers
- Avoid having sensitive data in logs - If you need the information for debugging purposes, redact it partially.
- Integrate with SIEMs and other dashboards, monitoring, alerting tools

VS Code OpenAPI Extension
http://bit.ly/42vscode

API Security Info & News
APIsecurity.io

42Crunch API Security Platform
42Crunch.com