Hacking JWTs: When a Token Turns Traitor

A Practical Guide to Common JSON Web Token Attacks

The Modern "Key Card" of the Web

JSON Web Token (JWT) is a compact, URL-safe standard (RFC 7519) for creating access tokens that enable stateless authentication.

Why stateless matters: The server doesn't need to store session data. It simply trusts the data inside the token after verifying its cryptographic signature.

This makes JWTs incredibly efficient for modern APIs and microservices architectures, but introduces unique security challenges when implemented incorrectly.

The Three Parts of a Token

Every JWT follows this structure: **xxxxx**.**yyyyy**.**zzzzz**

Header

Base64-encoded JSON specifying token type (`typ`) and signing algorithm (`alg`)

```
{"alg":"HS256","typ":"JWT"}
```

Payload

Base64-encoded JSON containing claims (user data like `sub`, `name`, `role`, `exp`)

```
{"sub":"1234","name":"user"}
```

Signature

Cryptographic signature verifying token integrity and authenticity

```
HMACSHA256(...)
```

The "Tamper-Proof" Seal

The signature is the **only component** that ensures token integrity. Without proper signature verification, an attacker can forge any token they want.

How HS256 Signature Works

```
# Symmetric (HS256) signature creation
signature = HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  your-256-bit-secret
)
```

The signature is the critical defense preventing attackers from forging their own tokens. Everything depends on it.

Attack #1: Weak Secret (Brute-Forcing)

When Your "Secret" Isn't a Secret

The Problem: Symmetric algorithms like HS256 rely on a single shared secret. If this secret is weak ("secret", "12345", "password"), attackers can discover it through brute-force.

Attack Flow

01

Attacker captures a valid JWT from network traffic

02

Runs brute-force tool against signature using wordlist

03

Once secret is found, signs malicious payloads

Practical Example

```
# Using hashcat to crack JWT
hashcat -m 16500 token.txt \
  /usr/share/wordlists/rockyou.txt
```

Result: Attacker becomes admin by creating tokens with `{"role": "admin"}` and signing them with the discovered secret.

Attack #2: The alg: none Attack

"I Swear This Token is Valid. Just... Trust Me."

The Vulnerability: The JWT spec includes an "unsecured" option where `alg` is set to `none`. Misconfigured libraries may honor this, completely bypassing signature checks.

Original Token

Header:
{"alg":"HS256","typ":"JWT"}

Payload:
{"user":"normal"}

Malicious Token

Header:
{"alg":"none","typ":"JWT"}

Payload:
{"user":"admin"}

The attacker Base64-encodes the new header and payload, adds an empty signature (`xxxxx.yyyyy.`), and sends it. Vulnerable servers accept it without verification.

Attack #3: Algorithm Confusion

The Most Infamous JWT Attack

Normal Server Configuration

Server uses RS256 with a private key to sign tokens and a public key to verify them. The public key is openly available.

Attacker Obtains Public Key

Attacker downloads the server's public key (this is completely normal and expected behavior).

Token Manipulation

Attacker crafts a malicious token with admin payload, changes `alg` from RS256 to HS256.

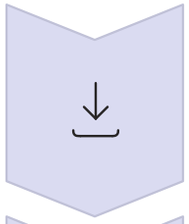
The Trick

Attacker signs the token using HS256 with the server's *public key* as the secret.

Server Accepts It

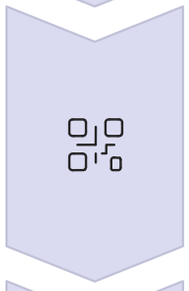
Vulnerable server sees `alg: HS256`, loads the public key file as the "secret", validates the signature successfully.

Exploiting alg Confusion



Get Server's Public Key

Download from `/jwks.json` or `/public.pem` endpoint

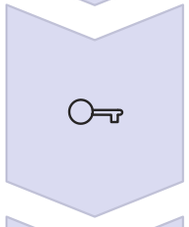


Craft Malicious Payload

```
{"user":"admin","iat":1678886400}
```

Modify Header

```
{"alg":"HS256","typ":"JWT"}
```



Sign with Public Key

Use HS256 algorithm with the entire public key text as the secret

Send Token

Server's `jwt.verify(token, public_key)` mistakenly uses public key as HMAC secret—token accepted!

More Ways a Token Can Fail

Expired Token Not Checked

Server doesn't validate the `exp` claim, allowing attackers to reuse old stolen tokens indefinitely.

Sensitive Data in Payload

Developers forget payload is just Base64-encoded (readable by anyone) and include secrets like passwords or API keys.

kid (Key ID) Injection

The `kid` parameter in header tells server which key to use. Attackers manipulate it to load predictable keys (e.g., `kid: "/dev/null"`).

Token Replay Attacks

Lack of `jti` (JWT ID) or nonce tracking allows attackers to reuse valid tokens captured from legitimate sessions.

Building an Impenetrable Defense

Use Strong Secrets

Generate cryptographically random 256-bit secrets for symmetric keys. Never use dictionary words or predictable patterns.

Whitelist Algorithms

Never trust the alg from the token. Hard-code the expected algorithm in your server code. This single fix stops `none` and confusion attacks.

Always Validate Claims

Check `exp` (expiration), `nbf` (not before), and `aud` (audience) on every request.

Use Updated Libraries

Modern JWT libraries have built-in protections. Keep dependencies current and review security advisories regularly.

No Sensitive Payload Data

Remember: payloads are Base64-encoded, not encrypted. Never include passwords, secrets, or PII in token claims.