Sicurezza nei web-service

In ASP.NET Core

Presentazione di Fabio Iotti

Argomenti

- Tipi di API esposte in web-services
- Possibili attacchi verso i web-services
- Come proteggere le password degli utenti
- Meccanismi standard di autenticazione
- Autenticazione in ASP.NET Core

Web-services

- Servizi web espongono API
- API pubbliche (usufruibili liberamente da servizi esterni)
- API private (solo per uso interno del servizio stesso)
- API interne (usufruibili solo da servizi autorizzati)

Attacchi

• Denial of Service (DOS)	• Load-balancing
• Distributed Denial of Service (DDOS)	• CDN, accertamento user-agent autentico
• Man in the Middle (MITM)	• HTTPS-Only Standard (HSTS)
• Cross-site Request Forgery (CSRF)	• Token anti-CSRF
• Abuso di API pubbliche	• Autenticazione, usage-quota
• Abuso di API private	• Autenticazione, Cross-Origin Resource Sharing (CORS)
• Abuso di API interne	• Autenticazione, Cross-Origin Resource Sharing (CORS)
• SQL-Injection	• Fare attenzione, separare dati dal codice
• Cross-site Scripting (XSS)	• Fare attenzione, separare dati dal codice
• Accesso fisico al server	• Accessi controllati, crittografia hardware, no porte USB
• Furto di credenziali (social engineering)	• Two-factor authentication
• Altro	• Altro

Denial of Service (DOS)

- Un hacker chiama ripetutamente le API, migliaia di richieste al secondo
- Il server non è in grado di gestire il carico di lavoro
- Gli utenti onesti non riescono più a comunicare con il server

- Load-balancing su più server
- L'hacker riesce a bloccare uno solo dei server
- La maggior parte degli utenti onesti continuano ad usufruire del servizio

Distributed Denial of Service (DDOS)

- Un hacker fa un DOS con più client, milioni di richieste al secondo
- I server in load-balancing non riescono a gestire il carico di lavoro
- Gli utenti onesti non riescono più a comunicare con i server

- CDN distribuita su più regioni
- L'hacker è in grado di bloccare il servizio solo sulla regione nella quale si trovano i client usati per fare l'attacco DDOS
- Gli altri utenti continuano ad usufruire del servizio

- Il server si accerta che lo user-agent sia autentico, captcha
- L'hacker non è più in grado di fare DDOS tramite bot

Man in the Middle (MITM)

- Utente visita http://www.banca.it da WiFi pubblica in un bar
- Proprietario del bar non sa nulla di sicurezza informatica
- Hacker collegato alla stessa WiFi, intercetta la richiesta ad http://www.banca.it e la reindirizza ad http://localhost con una copia identica della home-page del sito originale
- Utente fa login con le proprie credenziali
- L'hacker entra in possesso delle credenziali dell'utente

- Il sito è configurato con HTTPS-Only Standard
- Se l'utente aveva visitato almeno una volta https://www.banca.it in passato, il browser si rifiuta di aprire la versione non autentica del sito, in quanto non HTTPS

- L'hacker tenta di reindirizzare l'utente su https://localhost

- Il browser rileva che il certificato non è valido ed avvisa l'utente del tentativo di hacking

Cross-site Request Forgery (CSRF)

- Utente visita il sito www.banca.it e fa login
- API www.banca.it/trasferisci?iban=CONTO&qta=EURO
- Utente chiude il sito dimenticandosi di fare logout
- Utente visita www.streaming.it per guardare film pirata
- Di nascosto il sito fa una richiesta all'API per trasferire 10000 euro su un conto sulle Isole Cayman
- L'utente non si accorge di nulla, e sono stati prelevati 10000 euro dal suo conto

- API www.banca.it/trasferisci?iban=CONTO&qta=EURO&csrf=SECRET
- Token segreto anti-CSRF generato dal server quando l'utente apre la pagina su www.banca.it per fare trasferimenti
- Quando il server riceve una richiesta, verifica che il token sia autentico
- Se il token non coincide, la richiesta viene ignorata

- In ASP.NET Core questo comportamento è abilitato automaticamente ed è trasparente

https://learn.microsoft.com/aspnet/core/security/anti-request-forgery

Abuso di API pubbliche

- DOS/DDOS, l'hacker fa migliaia di richieste al secondo
- Server sovraccarico

- Autenticazione, per fare una richiesta alle API pubbliche è necessario autenticarsi con un token
- Usage-quota, max 10 richieste al secondo per ciascun token
- Quando il server rileva un abuso ignora le richieste

Abuso di API private

- Hacker chiama API private per uso interno del servizio stesso tramite uno script
- L'hacker è in grado di automatizzare operazioni che normalmente prevedono la presenza di un umano
- È in grado anche di utilizzare le API private del servizio dal proprio sito pubblico, facendo chiamate REST esterne

- Il server viene configurato per mandare direttive Cross-Origin Resource Sharing (CORS) al browser
- Il browser si rifiuta di chiamare le API a meno che la chiamata non venga fatta dal sito stesso che espone le API private

- L'hacker usa/scrive un client che ignora le direttive CORS
- L'hacker non è più in grado di fare chiamate dal proprio sito pubblico, ma può comunque farle da un client non ufficiale

- Autenticazione, per fare richieste il server richiede un token, il token identifica l'utente che fa richieste
- L'hacker non riesce più a fare richieste senza prima ottenere un token, ma può comunque farle usare un client non ufficiale

- Anche se l'utente è un hacker, per fare le richieste dovrà usare un token personale che lo identifica
- Dai log risulterà un utilizzo anomalo associato all'hacker, i log dovranno essere analizzati per identificare le anomalie
- L'hacker può essere bannato

Abuso di API interne

- Le API interne si hanno gli stessi problemi delle API private

- Il server può essere configurato con direttive CORS più avanzate
- Il browser permette di fare richieste solo se il dominio dal quale parte la richiesta è tra quelli autorizzati dal server

SQL-injection

- L'hacker identifica la pagina www.banca.it/login?usr=USR&pwd=PWD
- Quando un utente fa login, il server fa una richieda al database per validare le credenziali
```
SELECT u.Role FROM Users u WHERE u.Username = 'USR' AND u.Password = 'PWD';
```
- L'hacker fa una richiesta a www.banca.it/login?usr=admin';--&pwd=x
- Il server fa una richiesta al database per validare le credenziali
```
SELECT u.Role FROM Users u WHERE u.Username = 'admin';--' AND u.Password = 'x';
```
- L'hacker si autentica con ruolo di admin

- Il programmatore separa codice e dati, utilizzando variabili per passare i dati al database
```
SELECT u.Role FROM Users u WHERE u.Username = @usr AND u.Password = @pwd;
```
- SQL-injection evitabile usando le variabili, il database non le converte mai in codice

- Entity Framework separa automaticamente codice e dati, ma è importante ricordarsi di usare le variabili ogni volta che si fa una query senza passare da un ORM come Entity Framework

Cross-site Scripting (XSS)

- L'hacker crea un profilo sul sito www.gattini.it/user/h4ck3r
- Il sito www.gattini.it permette di caricare una biografia Bio… visibile in una sezione `<p>Bio...</p>` sulla pagina di profilo
- L'hacker carica una biografia contenente codice HTML
  Ciao <script>fetch("/api/regalaGattino?id=1234&user=h4ck3r")</script>
- Tutti gli utenti che visitano la pagina di profilo dell'hacker eseguono il codice iniettato
  `<p>Ciao</p><script>fetch("/api/regalaGattino?id=1234&user=h4ck3r")</script><p></p>`
- Se uno degli utenti che visitano la pagina è il proprietario del gattino 1234, il gattino cambia proprietario

- Il programmatore fa attenzione a separare i dati dal codice HTML
  `<p>@biografia</p>`

- ASP.NET Core fa automaticamente l'escape di tutte le variabili nei template
  `<p>&lt;/p&gt;&lt;script&gt;fetch("/api/regalaGattino?id=1234&amp;user=h4ck3r")&lt;/script&gt;&lt;p&gt;</p>`

https://learn.microsoft.com/aspnet/core/security/cross-site-scripting

Password

Come salvarle in modo sicuro su un database

Sicurezza

- Il database potrebbe essere rubato…
  - …furto di credenziali di uno degli amministratori del database
  - …tramite SQL-injection
  - …sfruttando uno zero-day sul server
  - …errore di configurazione
  - …accesso fisico al server del database

Password su database

- Clear-text
  - Se il database viene rubato, tutte le password sono leggibili
- Hash della password
  - Se il database viene rubato, le password non sono leggibili
  - Le password non sicure sono recuperabili con un dictionary-attack
  - Password recuperabili tramite rainbow-tables
- Hash con salt
  - Se il database viene rubato, le password non sono leggibili
  - Le password non sicure sono recuperabili con un dictionary-attack
  - Password non recuperabili tramite rainbow-tables
- Hash con salt e key-derivation
  - Se il database viene rubato, le password non sono leggibili
  - Non è possibile fare un dictionary-attack (almeno non in un tempo ragionevole)
  - Password non recuperabili tramite rainbow-tables

Autenticazione

Cookie, bearer token, JWT, e protocollo OpenID Connect

Cookie

- È possibile memorizzare l'identità di un utente usando un cookie
- L'utente si autentica inserendo le proprie credenziali
- Viene generato un cookie firmato contenente la sua identità
- Ad ogni richiesta viene allegato il cookie che identifica l'utente

- Questo meccanismo è valido se si lavora su un singolo dominio
- Non permette di identificare l'utente tra domini diversi
- Non permette di dimostrare l'identità dell'utente a servizi esterni

https://learn.microsoft.com/aspnet/core/security/authentication/cookie

Bearer token

- È un token semplice di autenticazione
- Un utente ottiene un “bearer token” (token al portatore) al login
- L’utente si identifica allegando il token alle richieste successive
  `Authorization = Bearer SOME_TOKEN_HERE`
- È parte dello standard web OAuth 2.0

https://developer.mozilla.org/en-US/docs/Web/HTTP/Authentication#authentication_schemes

JSON Web Token (JWT)

- È un token strutturato, può essere usato come Bearer token
- Composto da tre sezioni codificate in base64
  - Intestazione: definisce il formato delle altre sezioni
  - Payload: contiene dati arbitrari in JSON
  - Firma: firma di intestazione e payload, usando un secret noto al server

token

```
eyJhbGciOiJIUzI1NiIsIn
R5cCI6IkpXVCJ9.eyJzd
WIiOiIxMjM0NTY3ODk
wIiwibmFtZSI6IkpvaG4
gRG9lIiwiaWF0IjoxNTE
2MjM5MDIyfQ.XbPfbIH
MI6arZ3Y922BhjWgQz
WXcXNrz0ogtVhfEd2o
```

intestazione

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

payload

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

firma

```
HMACSHA256(
  base64UrlEncode(intestazione) + "." +
  base64UrlEncode(payload),
  "secret"
)
```

https://jwt.io/

OAuth 2.0

- È lo standard di autenticazione inter-servizio più utilizzato
- Prevede che autenticazione ed autorizzazione siano distinte
  - Autenticazione consiste nell'identificare l'utente
  - Autorizzazione definisce per quali risorse è consentito l'accesso
- È ricco di funzionalità, versatile
- Abbastanza complesso

https://oauth.net/2/

OpenID Connect (OIDC)

- Si basa su OAuth 2.0
- Prevede (almeno) tre servizi distinti:
  - Client: è colui che desidera accedere ad una risorsa a nome di un utente
  - Authorization service: permette di ottenere un token di autorizzazione che identifica l'utente e garantisce accesso ad un set di risorse (scope)
  - Token service: permette di scambiare un token di autorizzazione con un token
  - Resource server: permette di accedere ad una risorsa, tramite un token che garantisce accesso a quella risorsa
- Definisce un meccanismo per fare in modo che sia lo user-agent (il browser) dell'utente a gestire tutti i token, i vari servizi/server non comunicano mai direttamente tra di loro, se non per effettuare la validazione del token ricevuto con la richiesta

Flusso OIDC

- Un client desidera accedere ad una o più risorse a nome di un utente
- Il client contatta il server OIDC per leggerne la configurazione https://example.com/.well-known/openid-configuration
- Nella configurazione è indicato un indirizzo di autorizzazione https://example.com/authorize
- Il client apre la pagina del server di autorizzazione su un browser web (se il client è una pagina web, fa un redirect), aggiungendo un parametro con l'indirizzo di ritorno e le risorse (scope) alle quali desidera accedere
- A questo punto il controllo passa al server di autorizzazione
- Se l'utente non è autenticato, il server chiede all'utente di autenticarsi
- Eventualmente viene chiesto all'utente se è sicuro di voler autorizzare il client ad accedere alle risorse richieste a proprio nome
- Una volta che l'utente è autenticato, l'utente viene reindirizzato all'indirizzo di ritorno, aggiungendo un parametro con il token di autorizzazione
- Il client scambia il token di autorizzazione con un token, inviandolo all'indirizzo indicato nella configurazione https://example.com/token
- Il servizio genera un token che restituisce al client, ed invalida il token di autorizzazione utilizzato, per evitare che possa essere riciclato
- Il token può essere utilizzato per accedere alle risorse, allegandolo ad una richiesta indirizzata al server delle risorse
- Il server delle risorse verifica che il token sia autorizzato ad accedere alla risorsa richiesta
- Il token viene validato tramite una richiesta al servizio relativo
- Se il token risulta valido, viene restituita al client la risorsa richiesta

Flusso OIDC

- Isola i vari attori coinvolti nel processo, evitando la necessità di un'interconnessione
- Lo user-agent (browser) ha la responsabilità di gestire i token
- È totalmente sicuro e mantiene la sicurezza tra i vari attori
- È relativamente complesso
- Sono previsti altri meccanismi non descritti in questo corso
- OpenIddict, implementazione open-source in .NET

Autenticazione

Come implementarla in ASP.NET Core

https://learn.microsoft.com/aspnet/core/security/authentication/

Risorse esterne

OpenIddict https://github.com/openiddict/openiddict-core

Grazie!

Presentazione di Fabio Iotti