# Programming Secure Applications

Ing. Francesco Giurlanda
francesco.giurlanda@iet.unipi.it

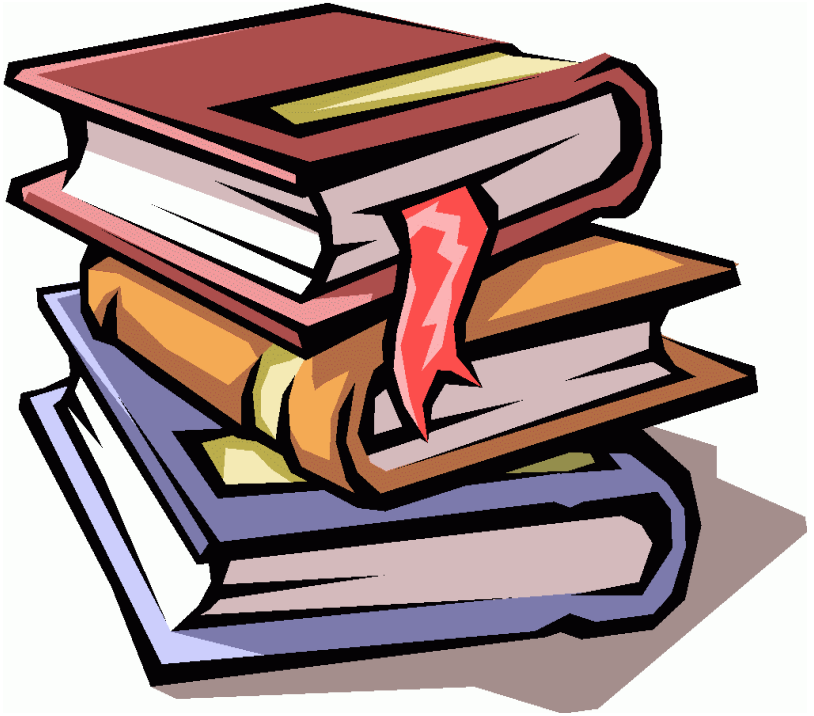
# Introduzione

- Cos'è la sicurezza nella programmazione?
  - È una pratica volta a risolvere le vulnerabilità di un programma tramite una collezione di regole e accorgimenti
- Come scrivere applicazioni sicure?
  - Basandoci su vulnerabilità già note.
- Non fare errori di programmazione non è una garanzia di sicurezza
- Applicazioni "sicure" elaborano "input non sicuri"
- Obiettivo: rendere un SW *immune* dagli attacchi

# Perché i programmatori scrivono programmi non sicuri?

Un programmatore non scrive volutamente programmi insicuri, ma lo fa comunque per svariate ragioni:

L'argomento è poco trattato nei vari programmi scolastici di informatica e la documentazione relativa è scarsa.



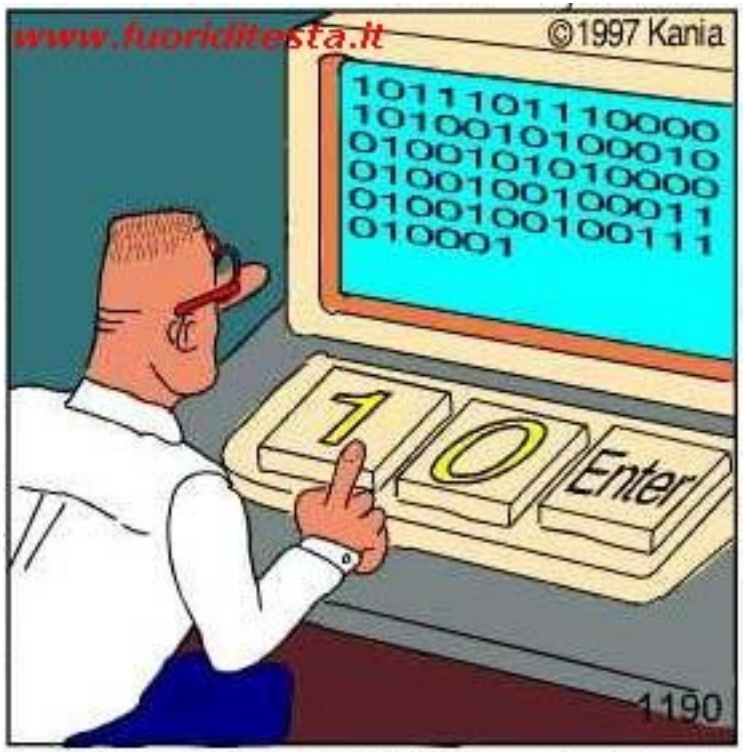
# Perché i programmatori scrivono programmi non sicuri?

**Nessuno utilizza metodi formali per la verifica della correttezza dei programmi.**

- *La **verifica formale** di programmi è un approccio sistematico alla dimostrazione della correttezza di un programma rispetto ad una specifica, basato su tecniche matematiche rigorose.*

# Perché i programmatori scrivono programmi non sicuri?

Il linguaggio C è un linguaggio a *basso livello*, ciò comporta che il programmatore deve essere consapevole di tutto ciò che accade durante l'esecuzione di un programma.

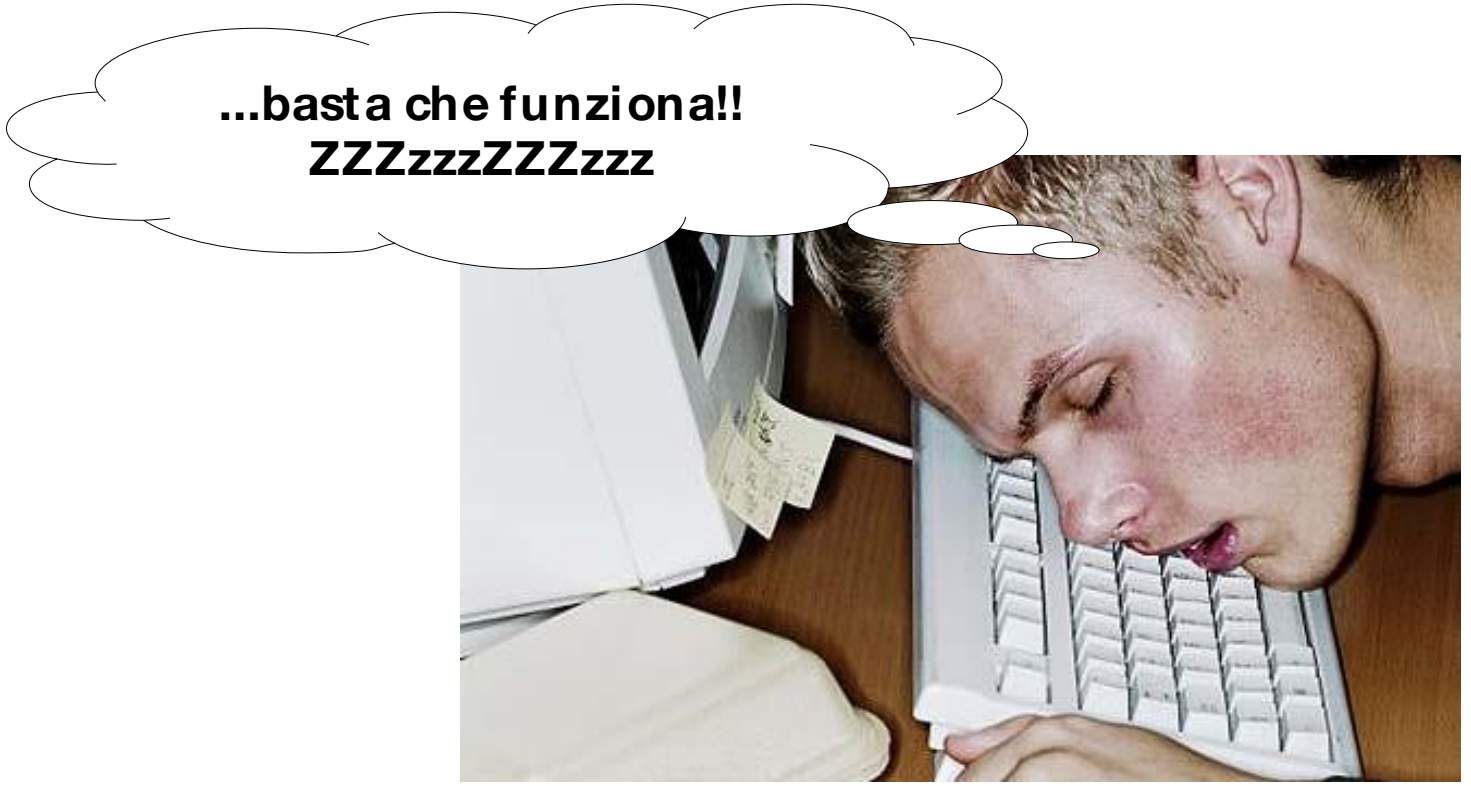



*I veri programmatori di codice binario*

# Perché i programmatori scrivono programmi non sicuri?

I programmatori sono esseri umani e molti di loro sono **pigri.**

*“L'approccio più semplice spesso è quello meno sicuro”*



# Perché i programmatori scrivono programmi non sicuri?

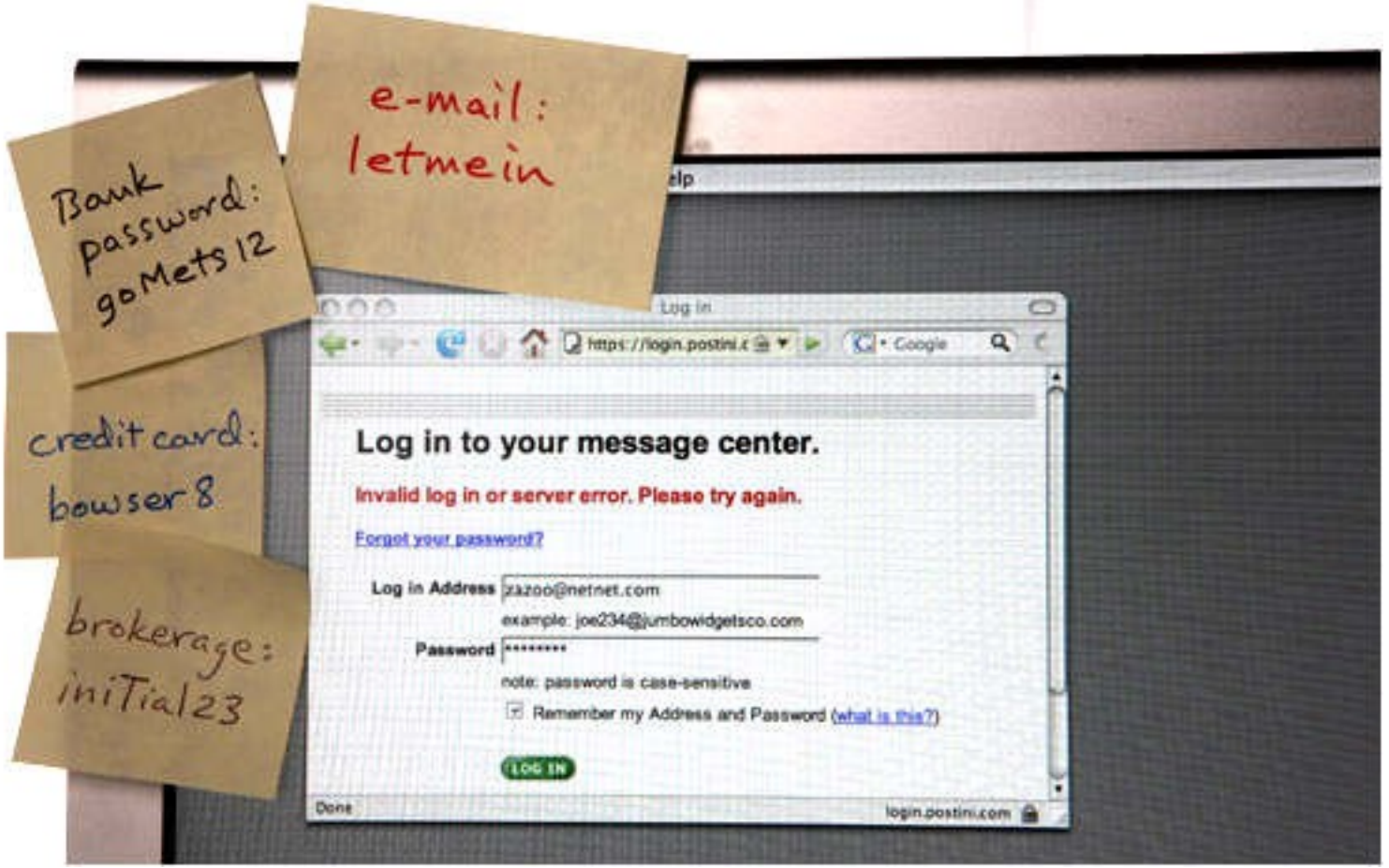
Molti programmatori non si occupano di sicurezza e sono ignari dei pericoli che corrono.

# Perché i programmatori scrivono programmi non sicuri?

- Ci sono molti software obsoleti e compromessi
- Correggere gli errore di questi software è un'operazione complessa

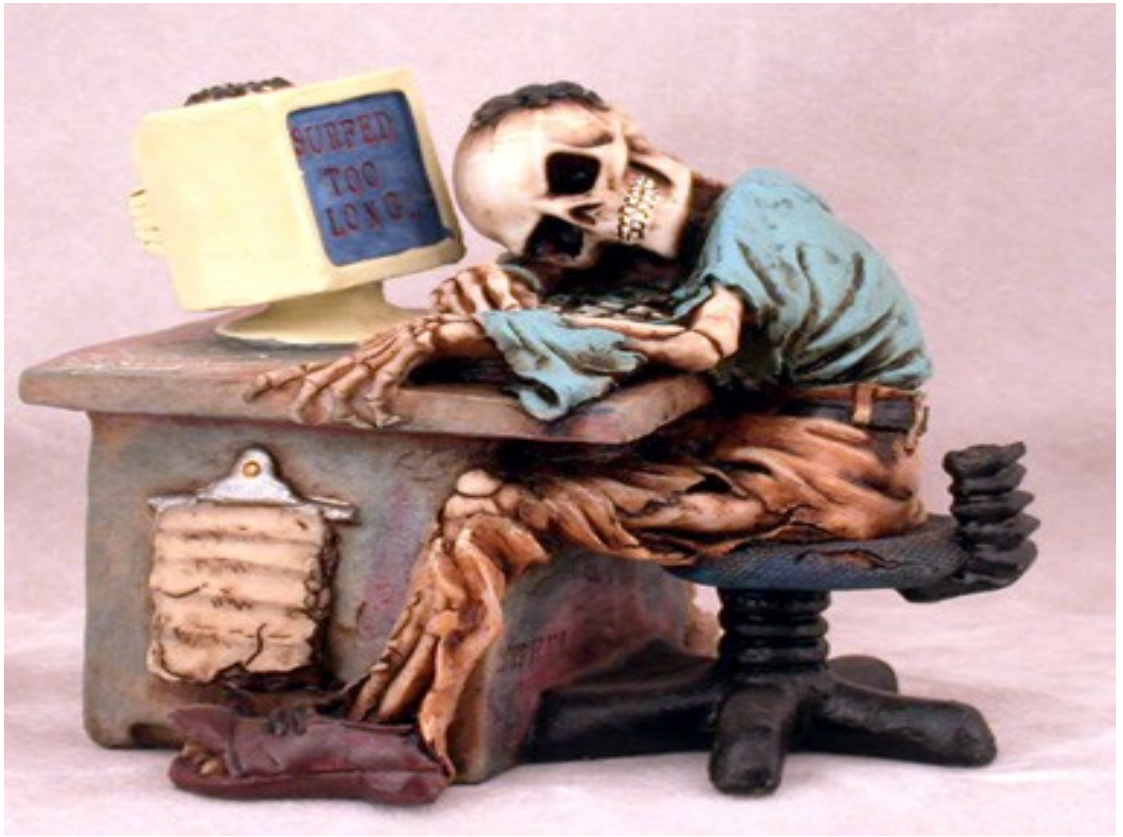
# Perché i programmatori scrivono programmi non sicuri?

- Gli utenti non curano affatto la sicurezza



# Perché i programmatori scrivono programmi non sicuri?

- La sicurezza comporta un tempo di sviluppo maggiore e una fase di testing estesa e accurata



# Perché i programmatori scrivono programmi non sicuri?

# Open vs. Closed Source

- **Open Source**
  - **Contro:** maggiori informazioni per un attaccante che ha a disposizione il codice del programma.
  - **Pro:** il software è analizzato da un maggior numero di persone che rendono l'identificazione di vulnerabilità più efficiente
- **Closed Source**
  - **Contro:** non fornire il codice sorgente di un programma non implica maggiore sicurezza
  - **Pro:** *dovrebbe* prevedere una fase di testing più accurata in fase di progettazione

# Conosci il tuo avversario: Hacker, Cracker e Attacker

- **Hacker:** è una persona che si impegna nell'affrontare sfide intellettuali per aggirare o superare creativamente le limitazioni che gli vengono imposte.
- **Cracker:** qualcuno che infrange la sicurezza di un sistema.
- **Attacker:** qualcuno che attacca un sistema
- **Distinsioni:**
  - **Non tutti gli hacker sono cracker** (white hats)
  - Non tutti i cracker sono hacker (script kiddies)
  - Non tutti gli attacker sono cracker (DoS attack)

# Fase di progettazione

# Quali sono i tuoi requisiti di sicurezza?

- ***Common Criteria*** (CC) è un documento molto utile per l'identificazione dei requisiti di sicurezza (ISO/IEC 15408)
  - *Protection Profile* (PP)
  - *Security Target* (ST)
- **PP:** documento creato da un gruppo di utenti che identificano le proprietà di sicurezza desiderate da un prodotto
- **ST:** documento che identifica cosa fa realmente un prodotto, che sia rilevante dal punto di vista della sicurezza
- PP & ST sono sottoposti a verifiche formali
  - La verifica sul PP coinvolge il documento verificando le dipendenze tra i requisiti specificati dagli utenti
  - La verifica del ST non coinvolge il documento ma effettua la valutazione su un sistema reale chiamato "*target of evaluation*" (TOE). Si vuole che il TOE rispetti i requisiti specificati nel ST

# Quali sono i tuoi requisiti di sicurezza?

- Qual è il tuo **contesto**?
  - Quali sono le minacce e la loro gravità?
  - Qual è il contesto di lavoro (singolo host, network)?
  - Quali sono le politiche di sicurezza?
- Quali sono gli **obiettivi di sicurezza** del tuo prodotto?
  - Confidenzialità (accesso al servizio)
  - Integrità (modifica dei dati forniti da un servizio)
  - Disponibilità (continuità del servizio)
  - Privacy (segretezza dei dati)
  - Audit (tenere traccia di ciò che succede)

# Garanzie di sicurezza

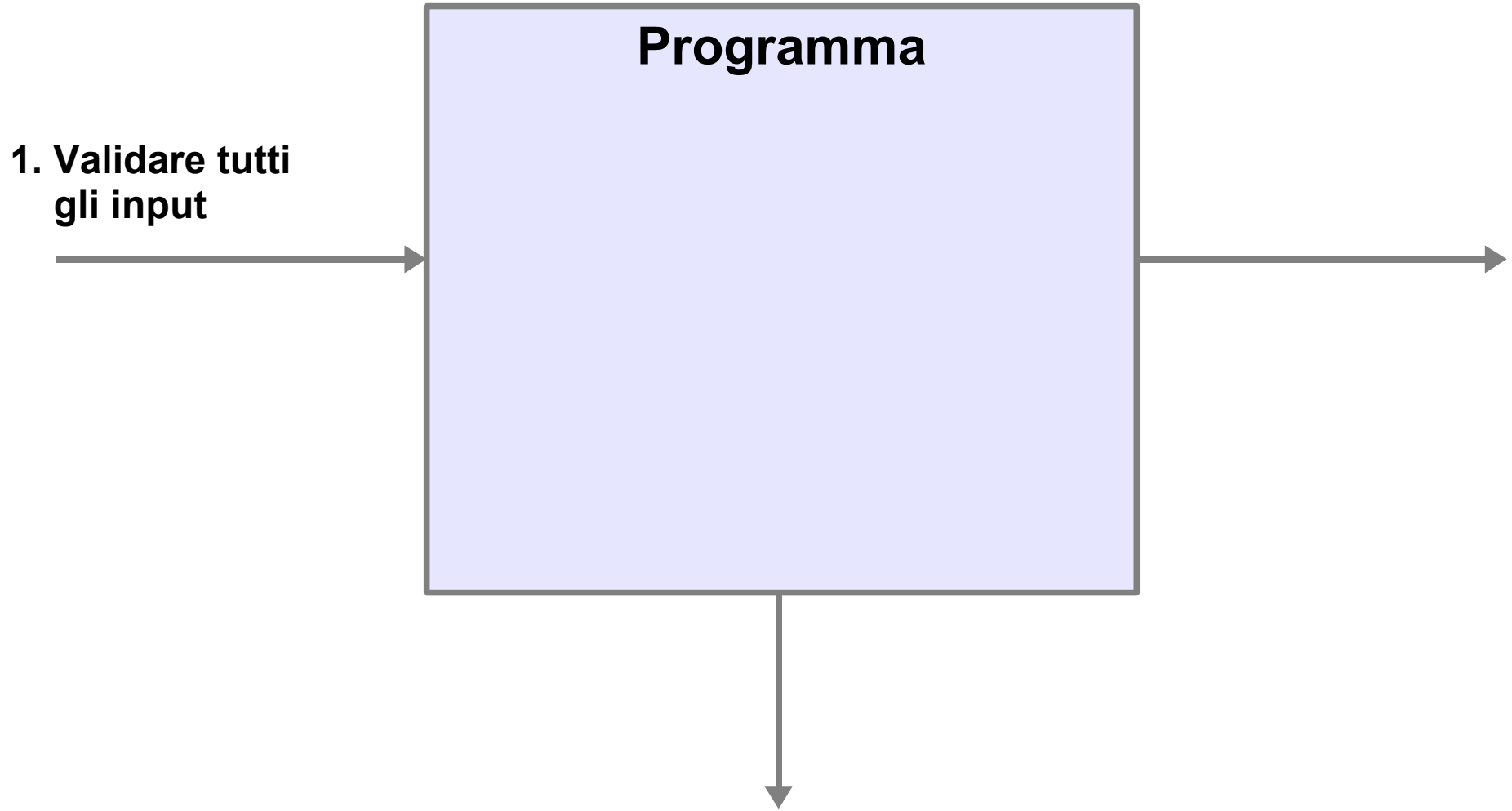
- Alcune categorie di garanzia di sicurezza presenti in CC:
  - Configuration management (ACM)
  - Delivery and operation (ADO)
  - Development (ADV)
  - Guidance documents (AGD)
  - Life-cicle support (ALC)
  - Test (ATE)
  - Vulnerability Assessment (AVA)
  - Maintenance of assurance (AMA)

# Requisiti funzionali

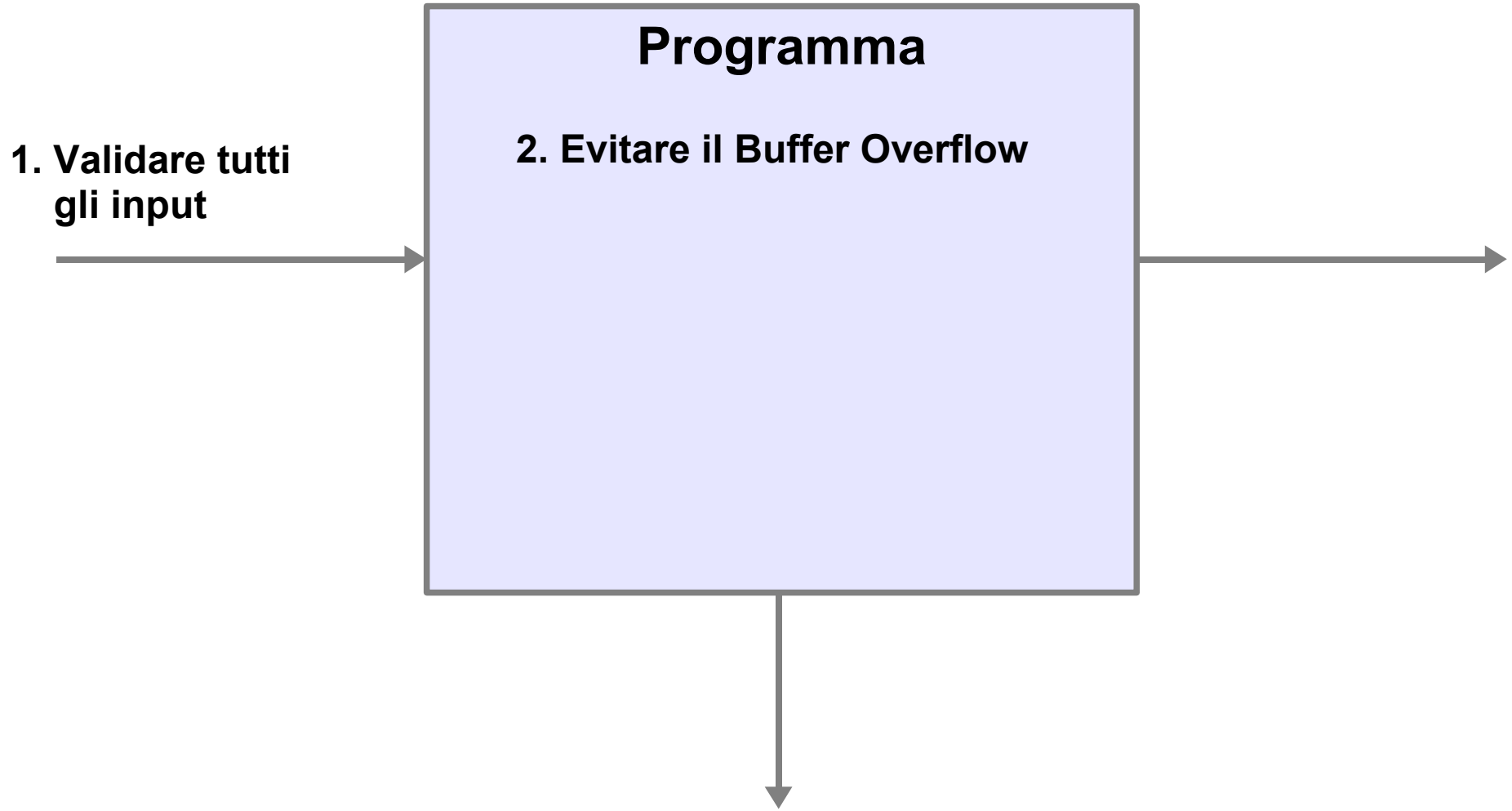
- Alcune classi di requisiti funzionali descritte nel CC:
  - Security Audit (FAU)
  - Communication/non-repudiation (FCO)
  - Cryptographic support (FCS)
  - User data protection (FDP)
  - Identification and authentication (FIA)
  - Security management (FMT)
  - Privacy (FPR)
  - Resource utilization (FRU)
  - TOE access (FTA)
  - Trusted path/channels (FTP)

# Implementazione

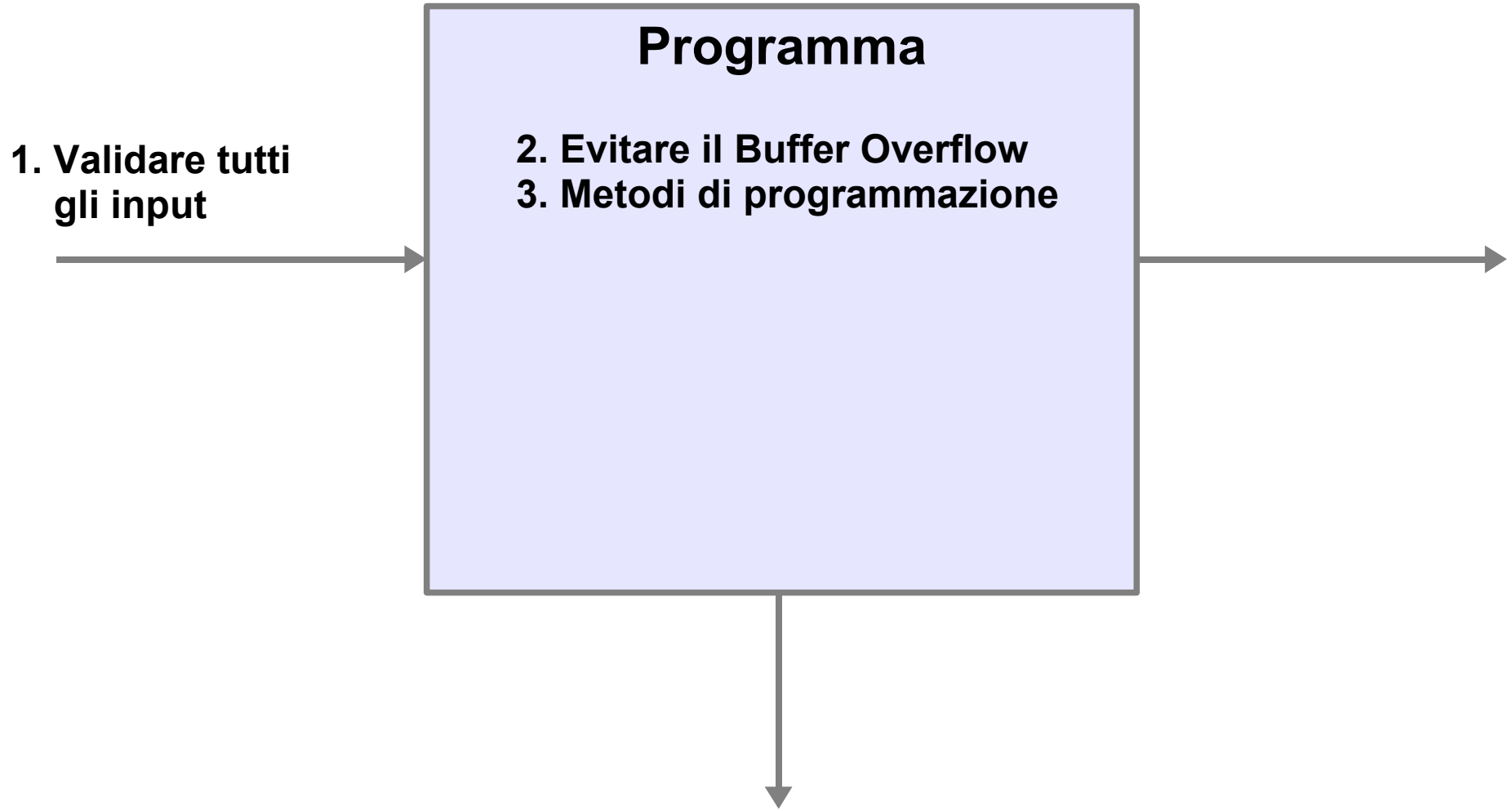
# Linee guida per l'implementazione



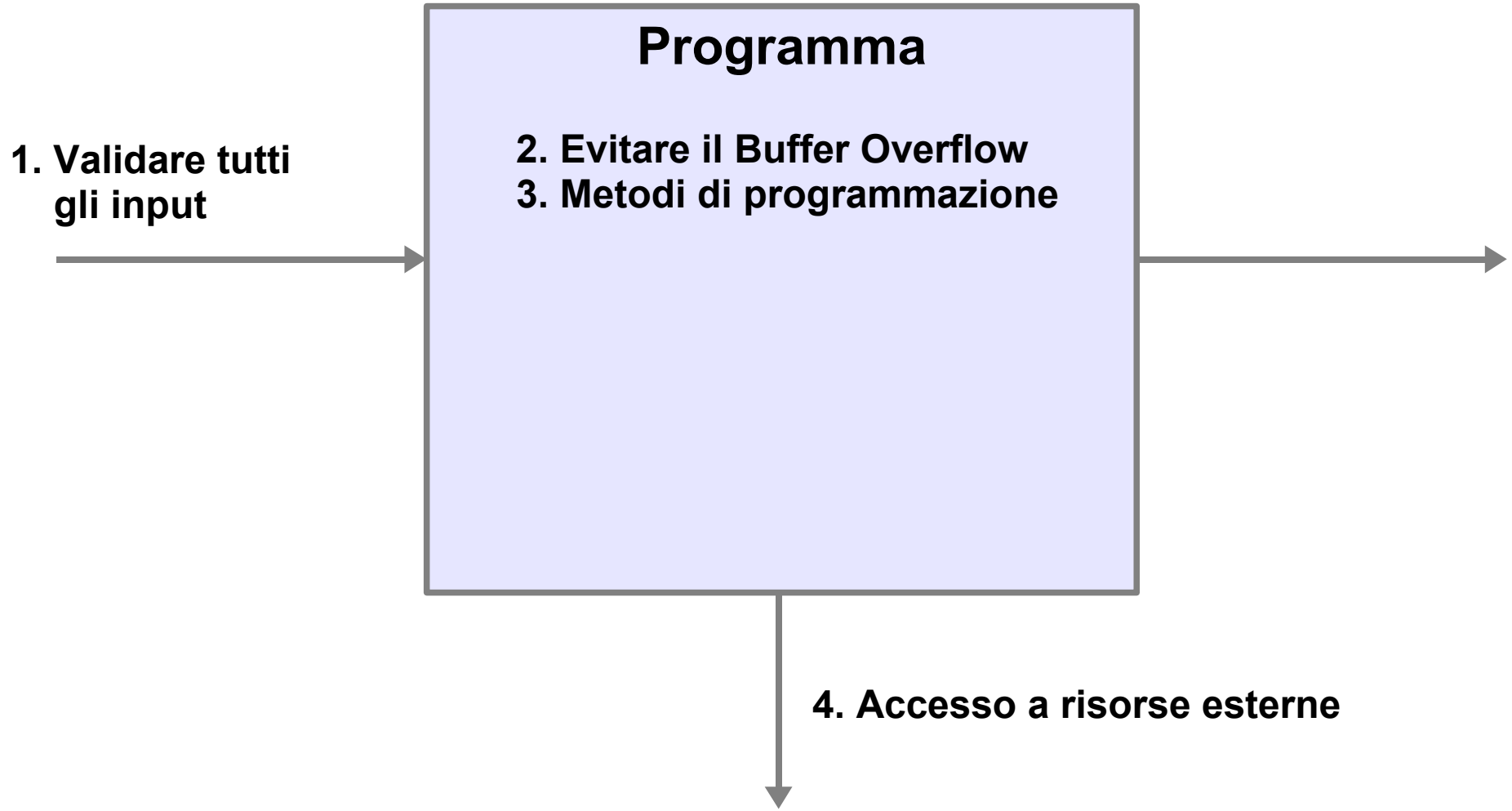
# Linee guida per l'implementazione



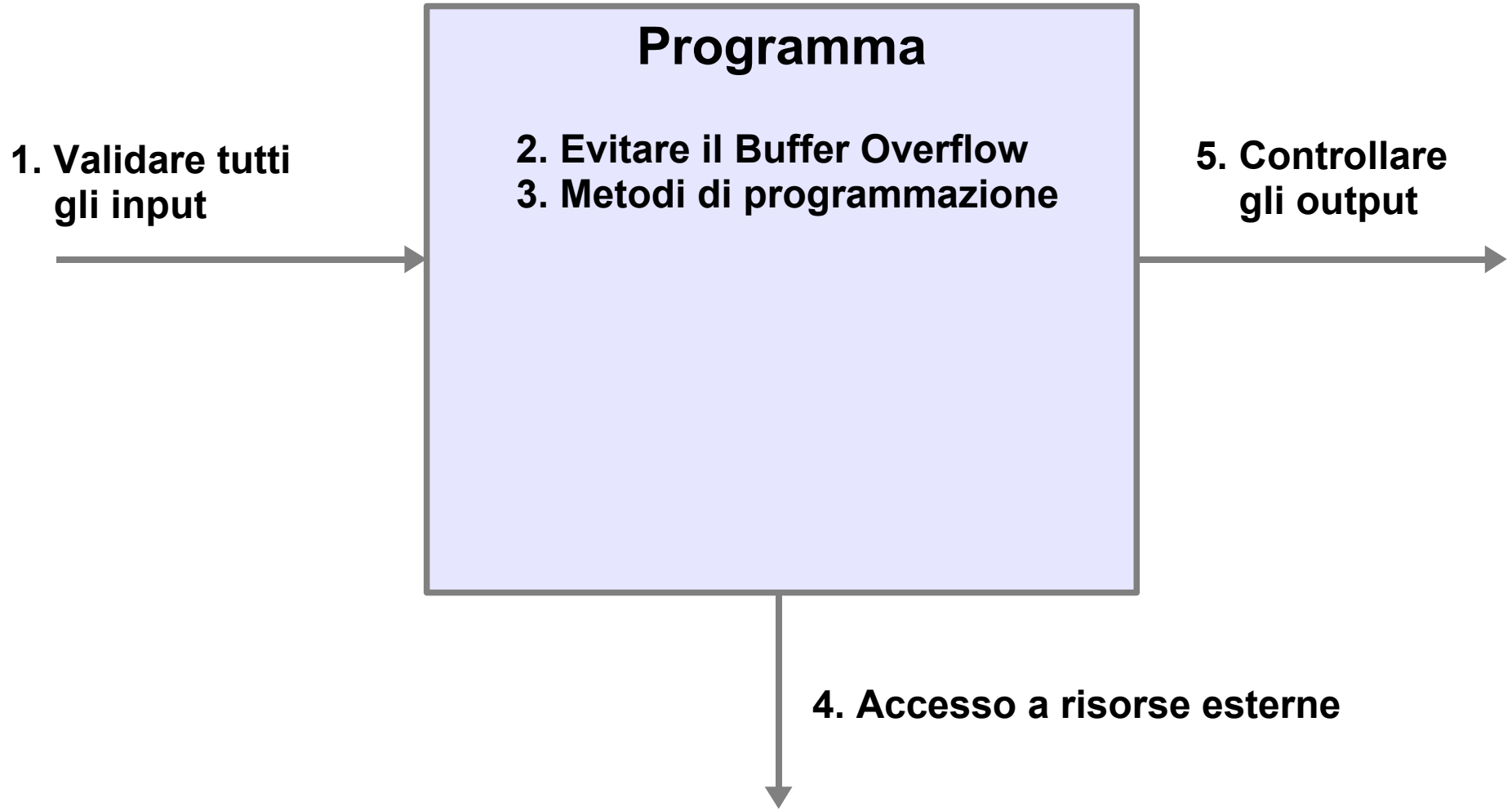
# Linee guida per l'implementazione



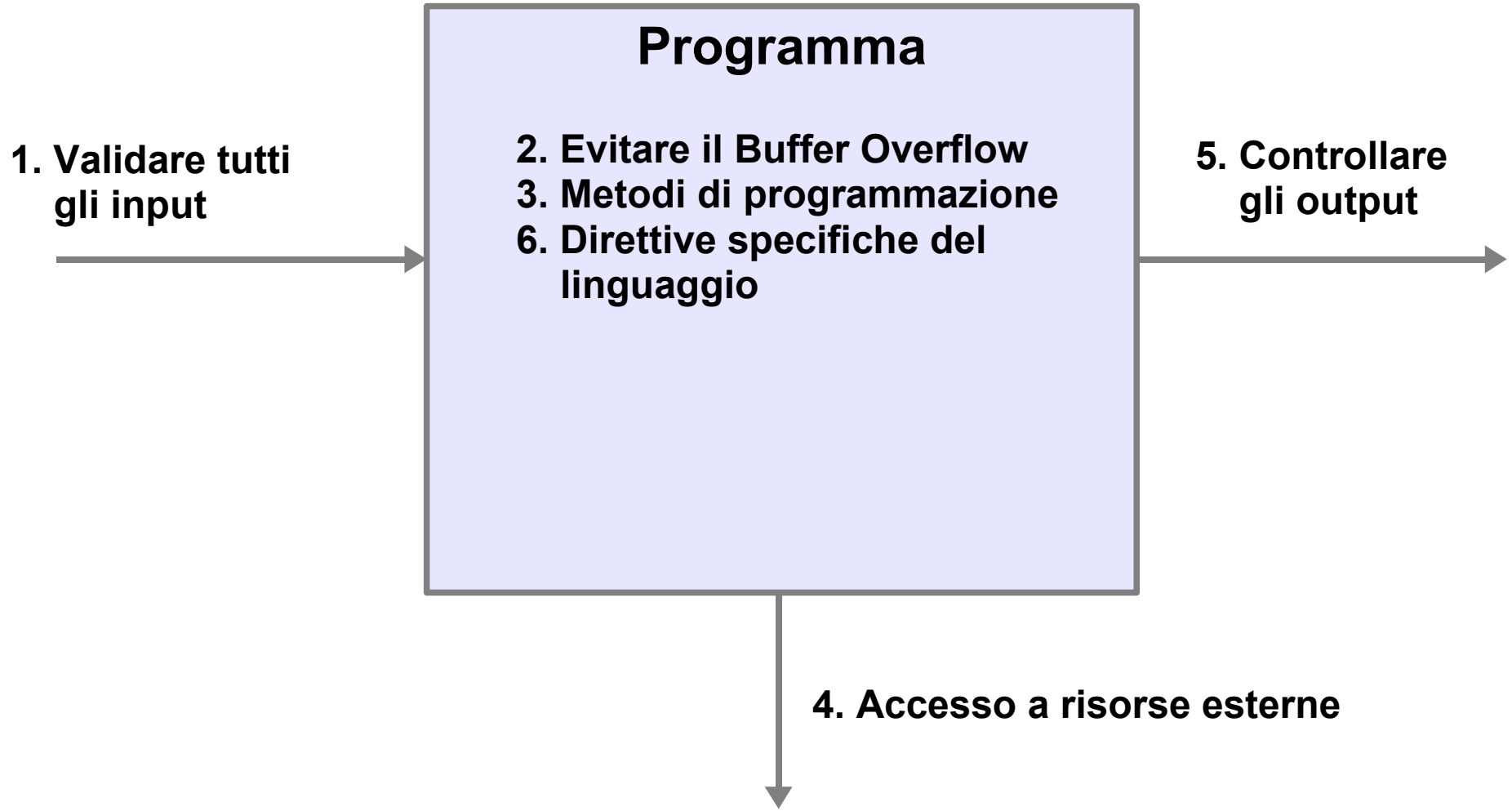
# Linee guida per l'implementazione



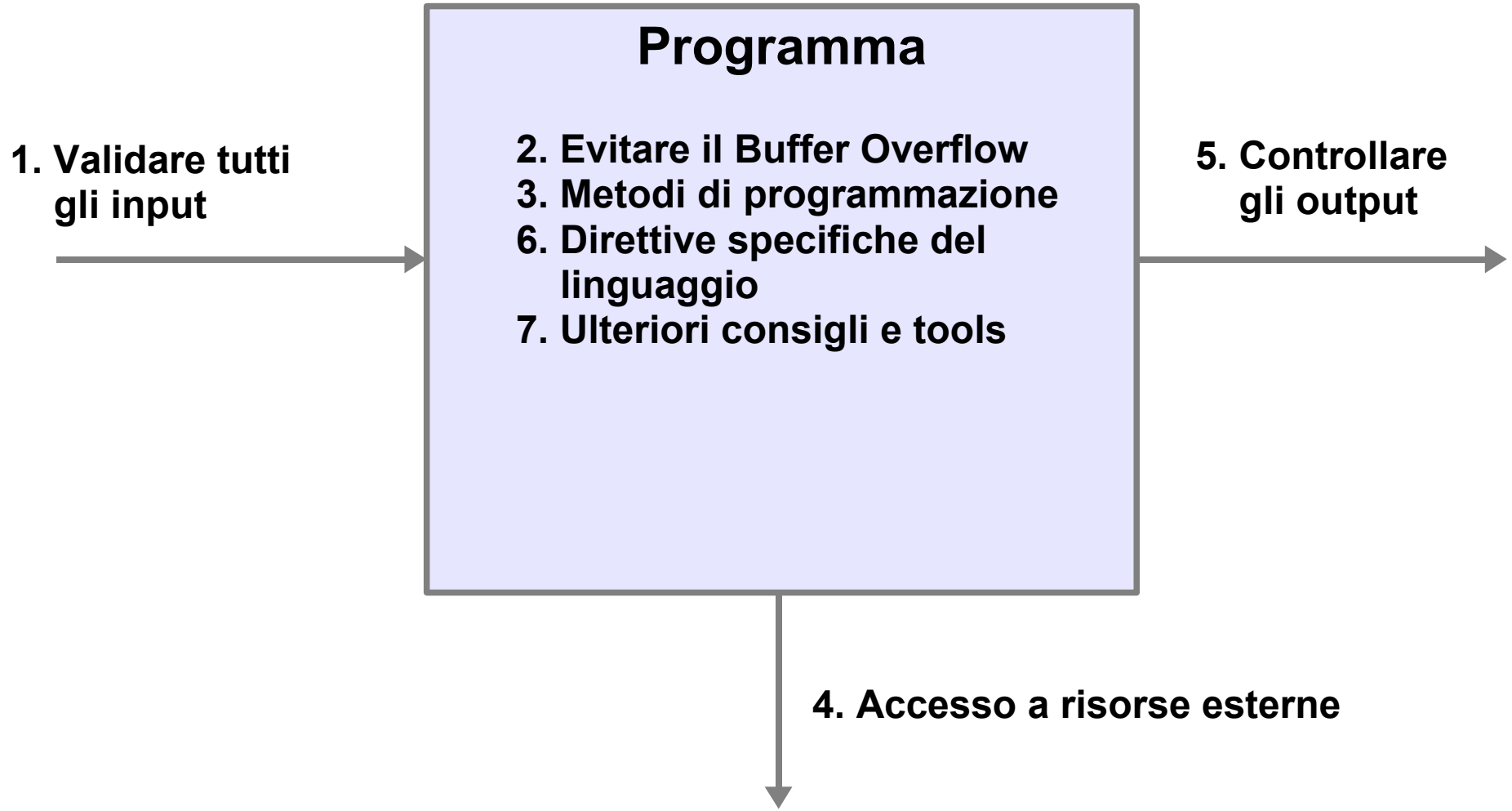
# Linee guida per l'implementazione



# Linee guida per l'implementazione



# Linee guida per l'implementazione



# Validazione degli Input

# Validazione degli Input

- Validare **tutti** gli input da sorgenti non fidate
  - *“Gli input di programmi **sicuri** vengono inseriti da utenti **inaffidabili**”*
- Identificare tutti gli input leciti e scartare tutti i rimanenti
  - **Mai il contrario**: identificare input illeciti e scrivere codice per scartarli; *in questo modo è molto probabile dimenticare o non accorgersi di casi particolari*
  - *Utilizzare gli input illeciti conosciuti come test di validazione*

# Validazione di stringhe e numeri

- Far attenzione ai caratteri speciali
  - Caratteri di controllo (es. ASCII Null)
  - Metacaratteri in shell, SQL … (es. *, ?, \, “, …)
  - Delimitatori (es. tab, virgola, <, :)
  - Assicurarsi che le codifiche siano corrette (es. UTF-8, URL …) ed altrettanto la loro decodifica
- Limitare sempre il valore dei dati numerici da immettere: massimo e se necessario minimo
- Controllo sul tipo di dato

# Validazione dei nomi di file

- Controllare l'utilizzo di caratteri speciali nella specifica di nomi di file
  - Per modificare il percorso: “../”, “/”,
  - Espansione dei nomi (*globbing*): “*”, “?”,
  - Caratteri che identificano opzioni: “-”, “--”
  - Caratteri di controllo: ESCAPE, NL, CR, SPACE

# Validare tutti gli input

- **Linea di comando**:
  - Non fidarsi di ogni valore della linea di comando se un attaccante può averne accesso
- **Variabili d'ambiente**:
  - Variabili d'ambiente ereditate: potrebbero derivare da un attaccante, anche indirettamente?
  - Un attaccante locale può configurare **ogni** cosa, anche variabili non documentate con effetti sulla Shell o altri programmi
  - Alcune variabili potrebbero essere assegnate più volte per aggirare i controlli

# Validare tutti gli input: Ulteriori consigli

- **Applicazioni Web**: Limitare i comandi GET
  - Ignorare/controllare i comandi GET se non sono semplici query (es. modifica di dati, trasferimento di fondi, firme e consensi a qualcosa)
- **Contenuto dei file**:
  - Non fidarsi di file che potrebbero essere sotto il controllo di utenti non fidati (es. file di config.)
- **Dati HTML & Cookies**:
  - Un utente può settarli a piacimento; per sicurezza introdurre autenticatori e controlli sui dati

# Espressioni regolari

- Una **espressione regolare** è una sequenza di simboli (quindi una stringa) che identifica un insieme di stringhe.
- Quando possibile, uitilizzare le espressioni regolari per validare gli input

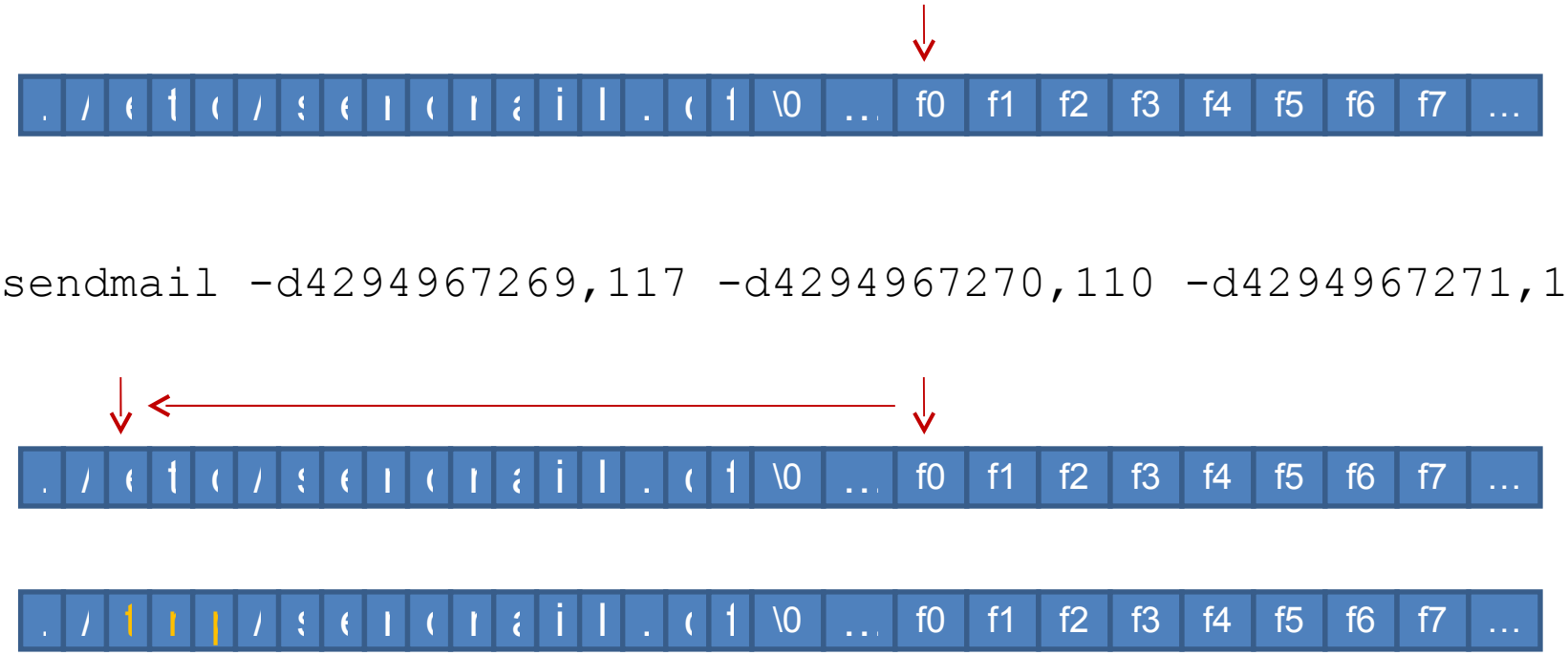
# Caso Sendmail

`sendmail -d`*flag,value*

`sendmail -d`*8,100*

- Configura il flag #8 con il valore 100
- Il nome del file di configurazione (`/etc/sendmail.cf`) si trova nel segmento precedente a quello dell'array dei flags, questo file contiene il percorso del programma mail da avviare (`/bin/mail`).
- Sendmail controlla il valore massimo del numero di flag, ma non il valore minimo, input negativi non dovrebbero essere ammessi.
- Valori interi superiori a $2^{31}$ sono considerati negativi in C su host a 32-bit
- `sendmail -d4294967269,117 -d4294967270,110 -d4294967271,113`
  modifica "etc" in "tmp"
- Un attaccante può creare un file /tmp/sendmail.cf che dichiara come programma mail da adottare /bin/sh, dando l'accesso come root all'attaccante

# Caso Sendmail



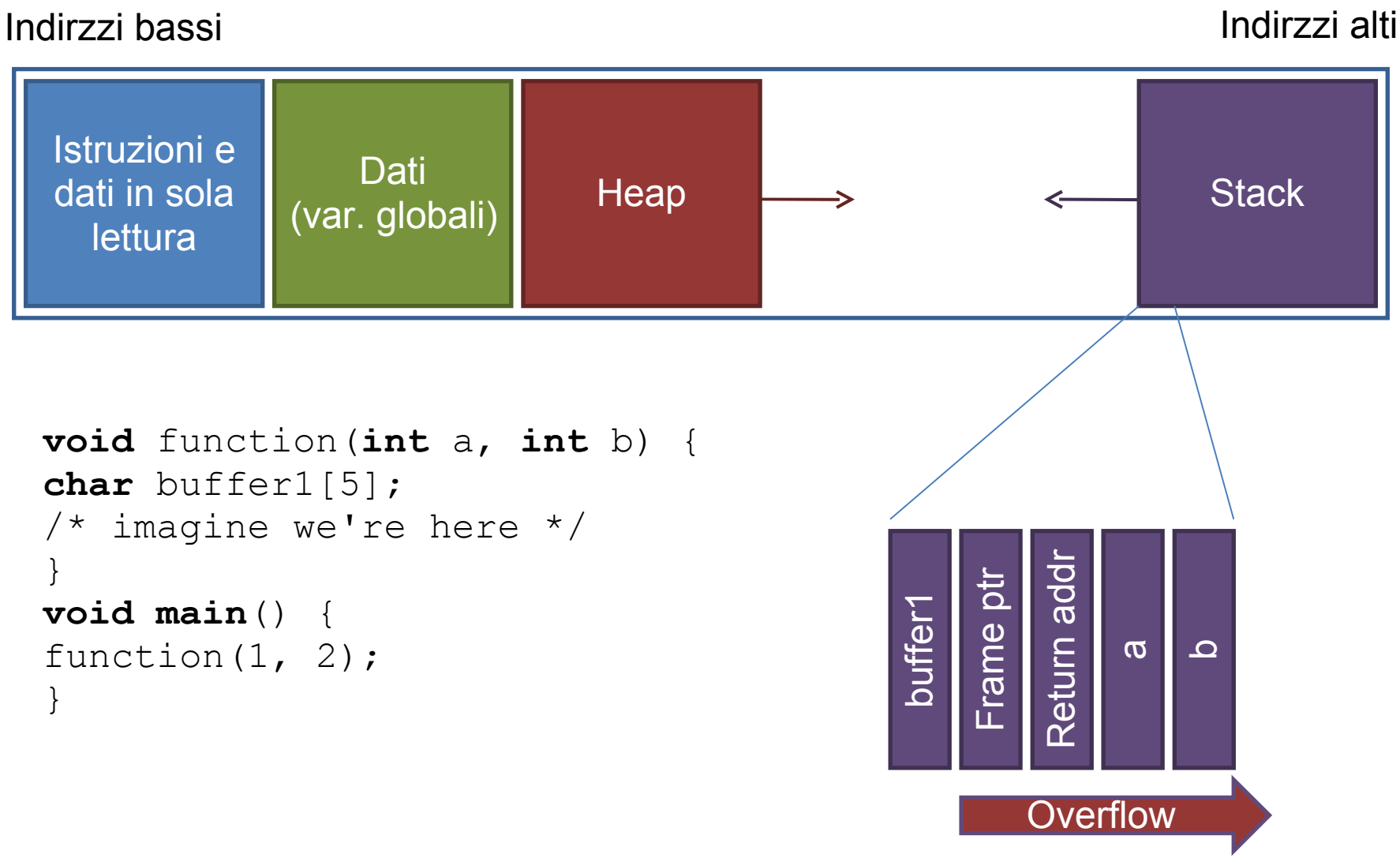

```
sendmail -d4294967269,117 -d4294967270,110 -d4294967271,113
```

# Evitare il buffer overflow

# Evitare il buffer overflow

- **Buffer Overflow**
  - Si verifica quando un attaccante può effettuare la scrittura di dati fuori dal limite del buffer, sovrascrivendo altre variabili valide.
  - Se il buffer è nello stack si chiama anche “stack overflow” o “smashing the stack”; può modificare un indirizzo di ritorno e fornire il codice  a cui restituire l’esecuzione
  - È possibile perché C/C++ non controllano automaticamente i limiti di un buffer
  - Permette anche di modificare dei dati del programma altrimenti inaccessibili

# Evitare il buffer overflow




```
void function(int a, int b) {
char buffer1[5];
/* imagine we're here */
}
void main() {
function(1, 2);
}
```

# Caso Wu-ftpd

- Wu-ftpd vulnerabilità *realpath* (<2.4.2)
  - Realpath() opera sui pathname eliminando “/”, “..”,”.”
  - Realpath() è implementata internamente con un buffer di dimensione fissa senza controllo della dimensione
  - Un attaccante con accesso in scrittura può creare percorsi arbitrariamente lunghi (es. mkdir AAA…; cd AAA…; quindi ripete mkdir … )
  - Alla fine, l’attaccante crea un file con nome uguale all’indirizzo di ritorno più il codice macchina da eseguire (es. “run shell”)
  - Quando ftpd chiama realpath() per trovare il percorso reale, invece di ritornare al programma principale la funzione avvia il codice scelto dall’attaccante (es. root shell)

# Soluzione

- Evitare o usare attentamente funzioni a rischio (dove non specifico la lunghezza del buffer) come: *gets(), strcpy(), strcat(),* …
- Alternativa: dimensione fissa vs dimensione dinamica
- Scegliere uno dei seguenti approcci:
  - **Std C fixed-length:** strncpy(), strncat(), snprintf()
  - **Std C dynamic-length:** malloc()
  - **Strlcpy & strlcat:** più semplici di strncpy
  - **Libmib** (lib. efficiente per lavorare con stringhe di dimensione variabile)

# Metodi di programmazione

# Metodi di programmazione 1

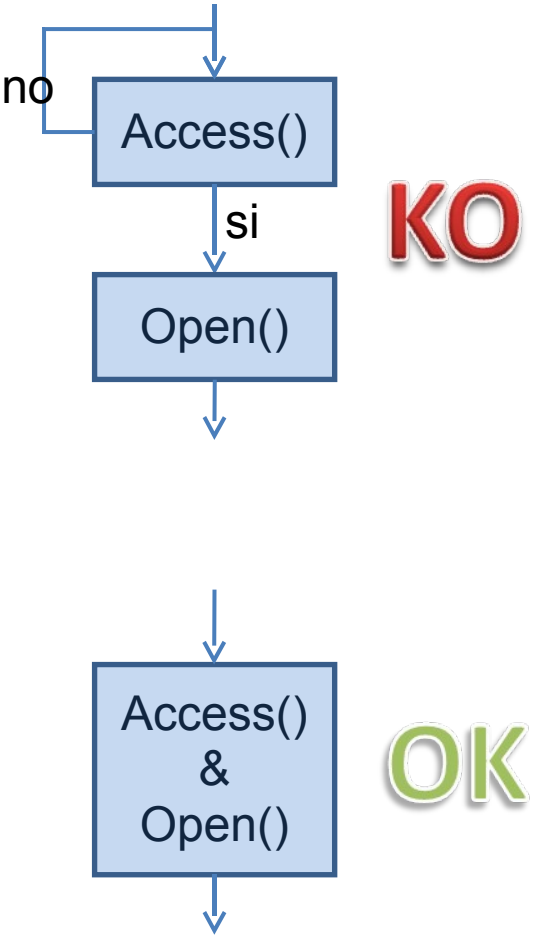
- **Fissare le Interfacce**
  - Semplici, dimensioni ridotte; evitare le macro
- **Ridurre i privilegi**
  - Minimizzare i privilegi a quelli strettamente necessari per lavorare: eseguire come utente/gruppo speciale non root, minimizzare i permessi sui file, limitare/rimuovere le richieste di debug
  - Ridurre i punti di accesso a quelli strettamente necessari (es. eliminare porte TCP/IP aperte inutilmente)
  - Minimizzare il tempo in cui i privilegi sono attivi
  - Minimizzare i moduli che richiedono maggiori privilegi

# Metodi di programmazione 2

- Attenzione alle impostazioni di **default**
  - Installa le applicazioni con tutte le *impostazioni di sicurezza attive*; in seguito sarà l'utente a disattivarne alche in base alle proprie necessità
  - **Mai installare password di "default"**, imporre all'utente l'immissione di una password di lavoro
  - **Installare applicazioni da utente** (solo se necessario da root) e rendi i file non scrivibili da altri utenti (blocca la diffusione di virus)
- Caricare i valori di inizializzazione in maniera sicura (es. /etc)
- Proteggere dai guasti: fermare l'esecuzione di una richiesta se si verificano errori o problemi con gli input

# Metodi di programmazione 3

- **Multiprogrammazione**: evitare le **corse critiche** (cc)
  - Si verifica quando più processi interferiscono l'uno con l'altro
  - Le cc si possono manifestare tra programmi sicuri interagenti, o tra programmi sicuri che interagiscono con un processo di un attaccante
  - Non separare le operazioni di access() per verificare se una risorsa è disponibile e open() per accedervi; tra le due operazioni lo stato della risorsa potrebbe cambiare



# Metodi di programmazione 4

- Far attenzione ai **file temporanei** nelle directory condivise
  - /tmp e /var/tmp condivise da tutti; l'attaccante può sfruttarle per i propri file
  - Se possibile opera su cartelle non condivise
  - Funzioni tmpfile(3) e tmpnam(3) (lib. stdio.h) non sono sicure
  - Per utilizzare file temporanei:
    - Creare un nome di file random
    - Aprire il file con open(2) , i seguenti flag O_CREAT|O_EXCL e privilegi minimi
    - Cancellare il file temporaneo al termine dell'utilizzo

# Metodi di programmazione 5

- Fidarsi solo di dati provenienti da canali affidabili:
  - **Indirizzi IP e MAC sorgente** possono essere contraffatti
  - **Indirizzi email mittente** possono essere contraffatti
  - **DNS query** possono essere contraffatte
- Prevenire Cross-site Malicious Content
  - Problema diffuso in applicazioni web
  - Utilizzare filtri o codifiche
- Contrastare attacchi semantici
  - e.s.: http://www.bloomberg.com@badguy.com

# Metodi di programmazione 6

- Seguire alcuni principi di sicurezza (S&S):
  - Semplicità
  - Design aperto: incoraggia gli altri a revisionare il codice!
  - Controllare tutti gli accessi.

# Richiesta di risorse esterne

# Richiesta di risorse esterne

- Invocare solo **routines di librerie sicure**
  - Alcune librerie implementano funzioni non sicure
  - Se non esistono, scriverle di proprio pugno.
- **Controllare i parametri** con cui sono invocate le routines
- Verificare ogni ritorno di System & Library Call
- Interfacciare due programmi attraverso **APIs** e non attraverso le interfacce destinate all'interazione umana.

# Richiesta di risorse esterne

- Controllare/proibire i metacaratteri nella composizione di comandi per la shell o evitare completamente le invocazioni alla shell.
  - & ; ‘ .\.| * ? ~ < > ^ ( ) [ ] { } $ \n \r
  - Gli spazzi bianche sono separatori di parametri …
  - Altri caratteri problematici: ! # -
- Stessi controlli sui metacaratteri per chiamate ad altri programmi (SQL)
- Cifrare le informazioni sensibili:
  - Utilizzare SSL/TLS per l’invio di dati privati su Internet
  - Cifrare i dati salvati sul disco specialmente se sono dati critici

# Controllare gli output

# Un Output oculato

- **Minimizzare i feedback**
  - Log degli errori, non dare troppe informazioni sull'errore a utenti non fidati
  - Non comunicare la versione del programma in caso di errore (es. login remoto)
- Gestire casi di **comunicazioni lente o interrotte** da programmi esterni
  - Possibili attacchi di tipo DoS

# String Format Attack

Controllare la formattazione dei dati in uscita:

- *int printf ( const char * format, ... );*
- **Sbagliato**: printf(stringFromUntrustedUser);
- **Corretto**: printf("%s", stringFromUntrustedUser);
- Alcuni parametri di formattazione pericolosi:
  - **%n** : prende come argomento un puntatore e scrive in memoria i caratteri seguenti a partire dall'indirizzo specificato
  - **%x** : legge dati dallo stack
  - **%s** : legge una stringa dalla memoria del processo

# String Format Attack

Controllare la formattazione dei dati in uscita:

- *int printf ( const char * format, ... );*
- **Sbagliato**: printf(stringFromUntrustedUser);
- **Corretto**: printf("%s", stringFromUntrustedUser);
- Alcuni parametri di formattazione pericolosi:
  - **%n** : prende come argomento un puntatore e scrive in memoria i caratteri seguenti a partire dall'indirizzo specificato
  - **%x** : legge dati dallo stack
  - **%s** : legge una stringa dalla memoria del processo

# Esempio 1

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

int main (int argc, char **argv)
{
      char buf [100];
      snprintf ( buf, sizeof buf, argv [1] ) ;
      buf [ sizeof buf -1 ] = 0;
      printf ( “Buffer size is: (%d) \nData input: %s \n” ,
                                                  strlen (buf) , buf ) ;
      return 0 ;
}
```

```
$./formattest “Bob”
Buffer size is (16)
Data input : Bob

$./formattest “Bob %x %x”
Buffer size is (27)
Data input : Bob bffff 8740
```

# Esempio 2

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

int main (int argc, char **argv)
{
      printf ( “User: ”);
      printf (argv[1])  ;
      return 0 ;
}
```

```
$./formattest “Bob”
User: Bob

$./formattest “%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s%s”
```

# Un Output oculato
# Caso PHP

- PHP < 4.0.3 error logging format string
  - Chiamata della funzione *php_syslog()* con dati forniti dall'utente
  - *php_syslog()* utilizza la chiamata a printf in maniera insicura
  - L'attaccante potrebbe indurre il programma a sovrascrivere le variabili nel suo stack con dati arbitrari
  - Questo permette degli attacchi remoti che consentono di acquisire i privilegi del processo PHP (solitamente i privilegi del web server)

# Direttive per linguaggi specifici

# Direttive per linguaggi specifici

- **Perl**
  - Abilitare le modalità –w (warn) e –T (taint)
  - Utilizzare la funzione open() con tre parametri che disabilita i caratteri speciali sui path
- **Shell** (sh, csh)
  - Non usare per operazioni di tipo setuid/setgid
  - In generale evitare script shell in casi di programmazione sicura
    - Uno script shell si può considerare affidabile quando tutti gli input provengono da sorgenti fidati (startup/shotdown scripts )
- **PHP**
  - Impostare *register_globals* su “off”
  - Usare le versioni 4.1.0+ e usare $_REQUEST per i dati esterni
  - Filtrare i dati utilizzati da *fopen()*

# Problemi legati a linguaggi specifici

- **C/C++**
  - Massimo rigore sulla tipizzazione
    - Utilizzare enum, unsigned in maniera appropriata
    - Fare attenzione ai char con e senza segno
  - Compilare con il più alto livello di avvisi e cercare di correggerli
    - gcc –Wall –Wpointer-arith –Wstrict-prototype –O2
  - Fare attenzione al buffer overflow

# Consigli generali e tools

# Consigli generali

- Numeri random: usare /dev/(u?)random
- Non inviare “in chiaro” passwords su Internet
- Autenticazione di utenti:
  - In intranet, utilizzare sistemi di autenticazione come Kerberos
  - Evitare l’autenticazione “in chiaro” su web

# Consigli generali

- Proteggere i segreti nella mem. dell'utente
  - Disabilitare *core dump* attraverso *ulimit*
  - *mlock(), mlockall()* impedisce che la memoria sia trasferita su disco per motivi di swap
  - Cancellare i dati segreti immediatamente dopo l'uso
- Utilizzare algoritmi crittografici e protocolli ***aggiornati*** e non cercare di ***inventarli personalmente***

# Tools

- Source Code Scanners
  - Flawfinder, RATS,LCLint, cqual
- Gestione della memoria
  - Valgrind
- Verifica sui crash attraverso test casuali
  - BFBTester

# Conclusioni

- Cercare di evitare tutte le vulnerabilità note, questo riduce notevolmente il rischio di attacchi
- Cercare di mettersi nei panni di un attaccante che vuole sfruttare delle vulnerabilità del nostro programma
- In ambito della programmazione sicura la ***paranoia*** è una ***virtù***

# Domande