

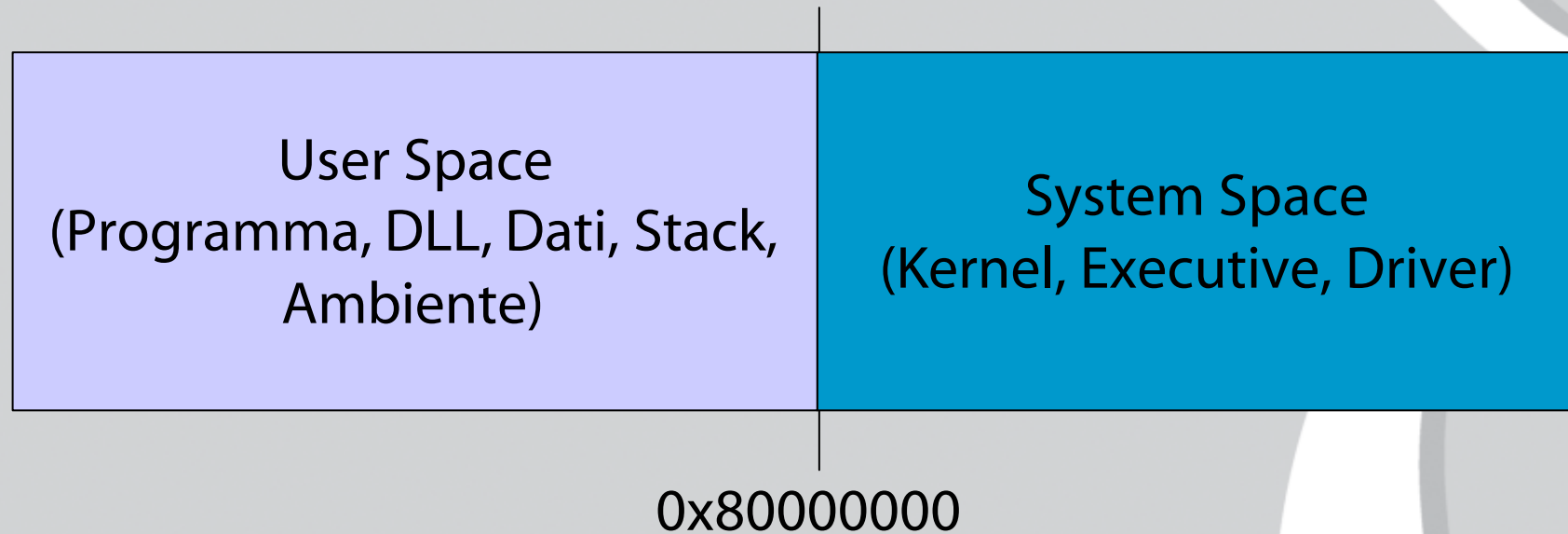
Introduzione alle tecniche di Exploit

Luigi Mori
Network Security Manager
Intrinsic
lm@intrinsic.it

IFOA
24 Luglio 2003

- Memory layout di un processo Win2000
- Stack Frame
- Exploit
- Shellcode
- Note finali

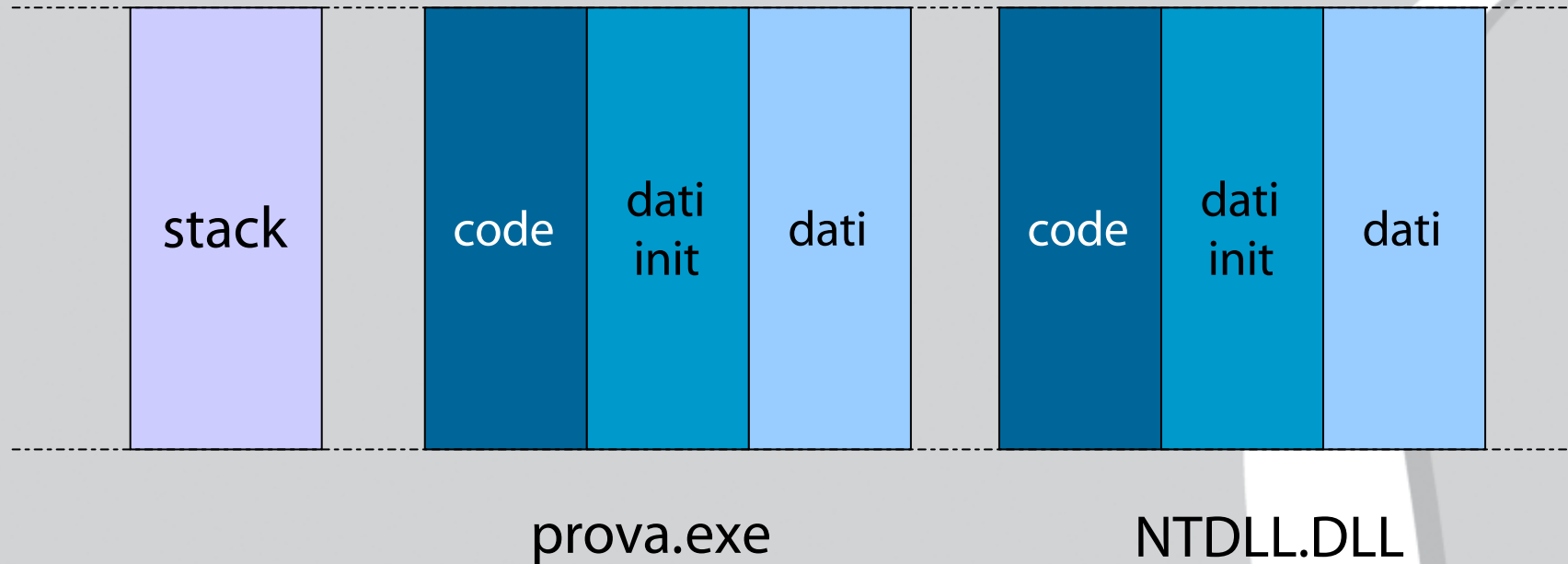
- Lo spazio di indirizzamento è flat
- Suddiviso in due regioni principali
 - System space (Kernel)
 - User space (Programma e DLL)
- All'interno dello user space la memoria può essere suddivisa in ulteriori sezioni
 - Codice
 - Stack
 - Dati inizializzati
 - Dati non inizializzati



L'interfaccia fra i due spazi è la NTDLL.DLL

- Diviso in una serie di regioni
- Ogni regione ha dei permessi di accesso
 - codice: XR
 - dati non inizializzati: RWX
 - dati inizializzati: RX
 - stack: RWX

Esempio User Space

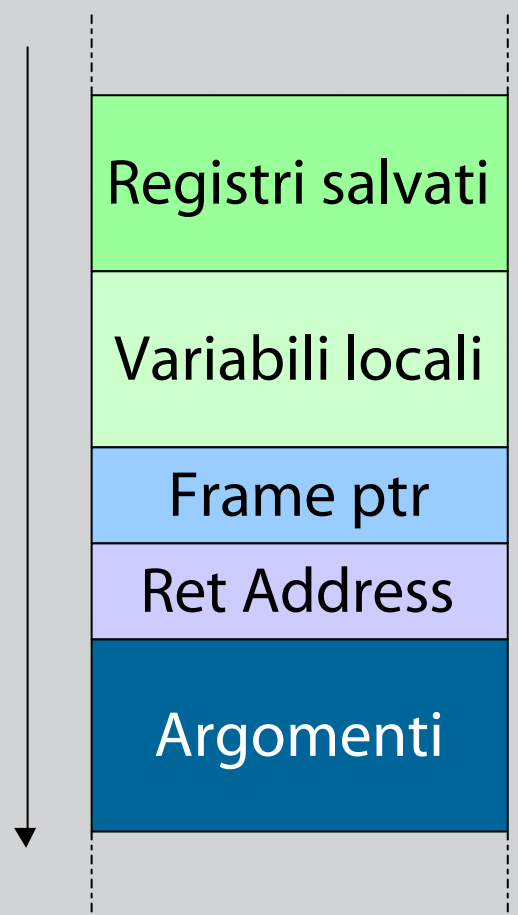


Stack Frame di una funzione C

```
int  
f(char *s)  
{  
    char c[16];  
    int i;  
    i = 1;  
    ...  
    return i;  
}
```

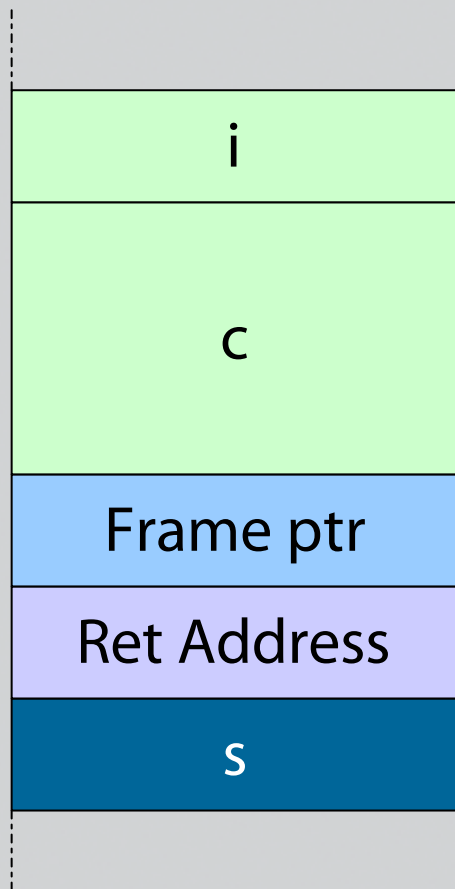
```
push ebp  
mov ebp, esp  
sub esp, 20h  
mov [ebp-20], 1  
...
```

Stack Frame di una funzione C



- Registri Salvati. Sono i registri salvati dalla calling conv usata
- Frame pointer. E' il valore salvato del registro usato per indirizzare le variabili locali (EBP)
- Ret Address. Indirizzo di ritorno
- Argomeni. Argomenti passati

Stack Frame di *f*



- Nessun registro salvato
- Le variabili locali sono contigue

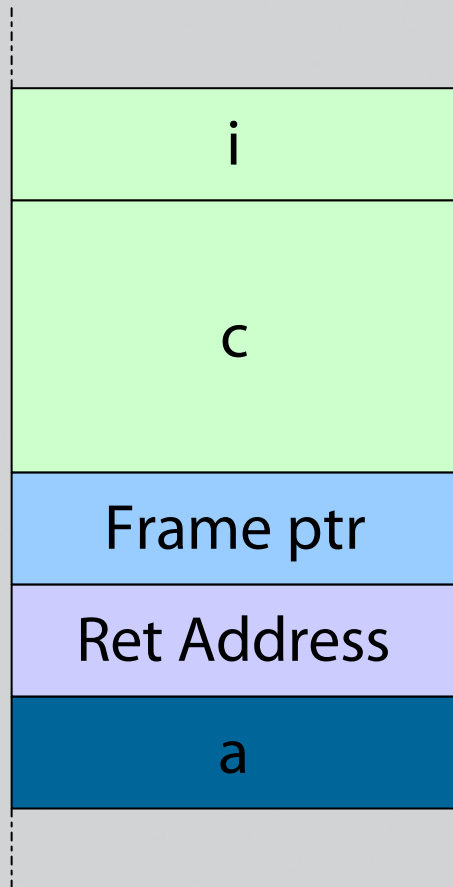
- Buffer overflow
 - classico
 - frame pointer
 - heap
- Format String Vulnerabilities
- Integer overflow

Buffer overflow classico

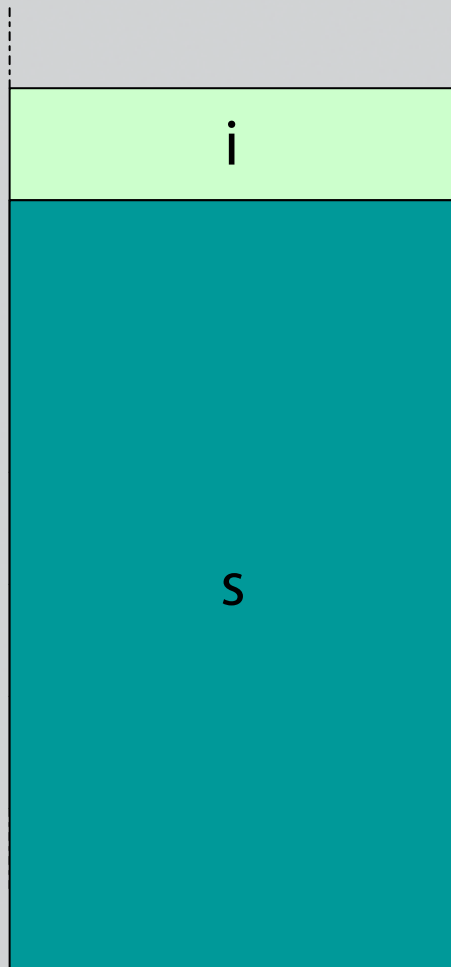
- Il padre di tutti i buffer overflow
- Ideato da Aleph1 nell'articolo "Smashing the stack for fun and profit" nella e-zine Phrack #49

```
int  
f(char *s)  
{  
    char c[16];  
    int i;  
    i = 1;  
    strcpy(c,s);  
    return i;  
}
```

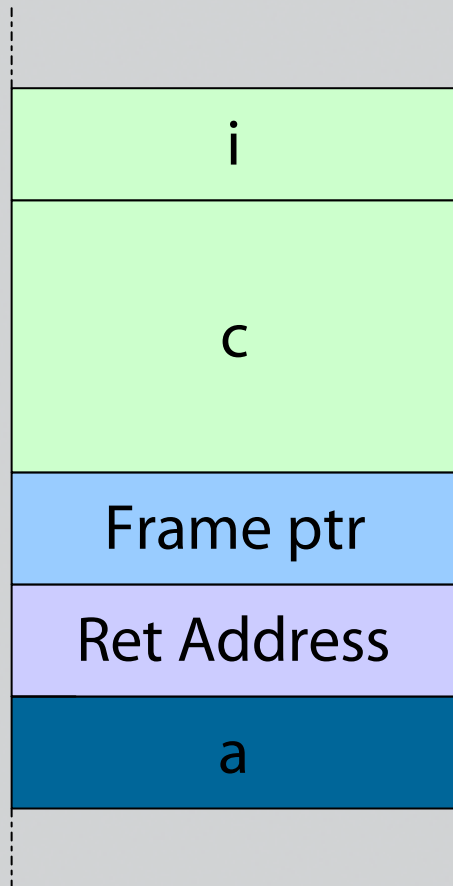
- La funzione **strcpy** copia i dati da **s** in **c** senza controllare la dimensione
- Se la stringa puntata da **s** è più lunga di 16 byte ci sono dei problemi



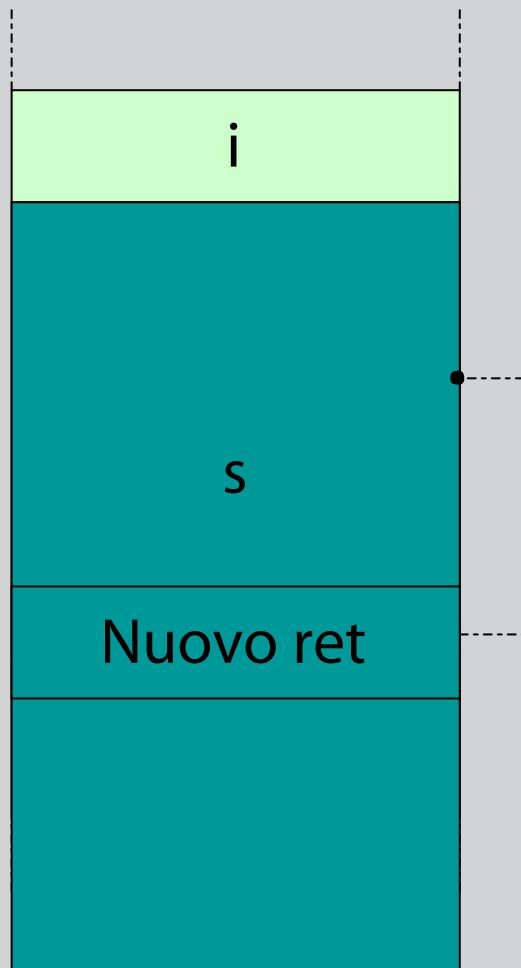
- Se s è abbastanza lunga può andare a sovrascrivere frame pointer, return address e altro ancora
- Questo può essere sfruttato da un attaccante



- Se s è abbastanza lunga può andare a sovrascrivere frame pointer, return address e altro ancora
- Questo può essere sfruttato da un attaccante

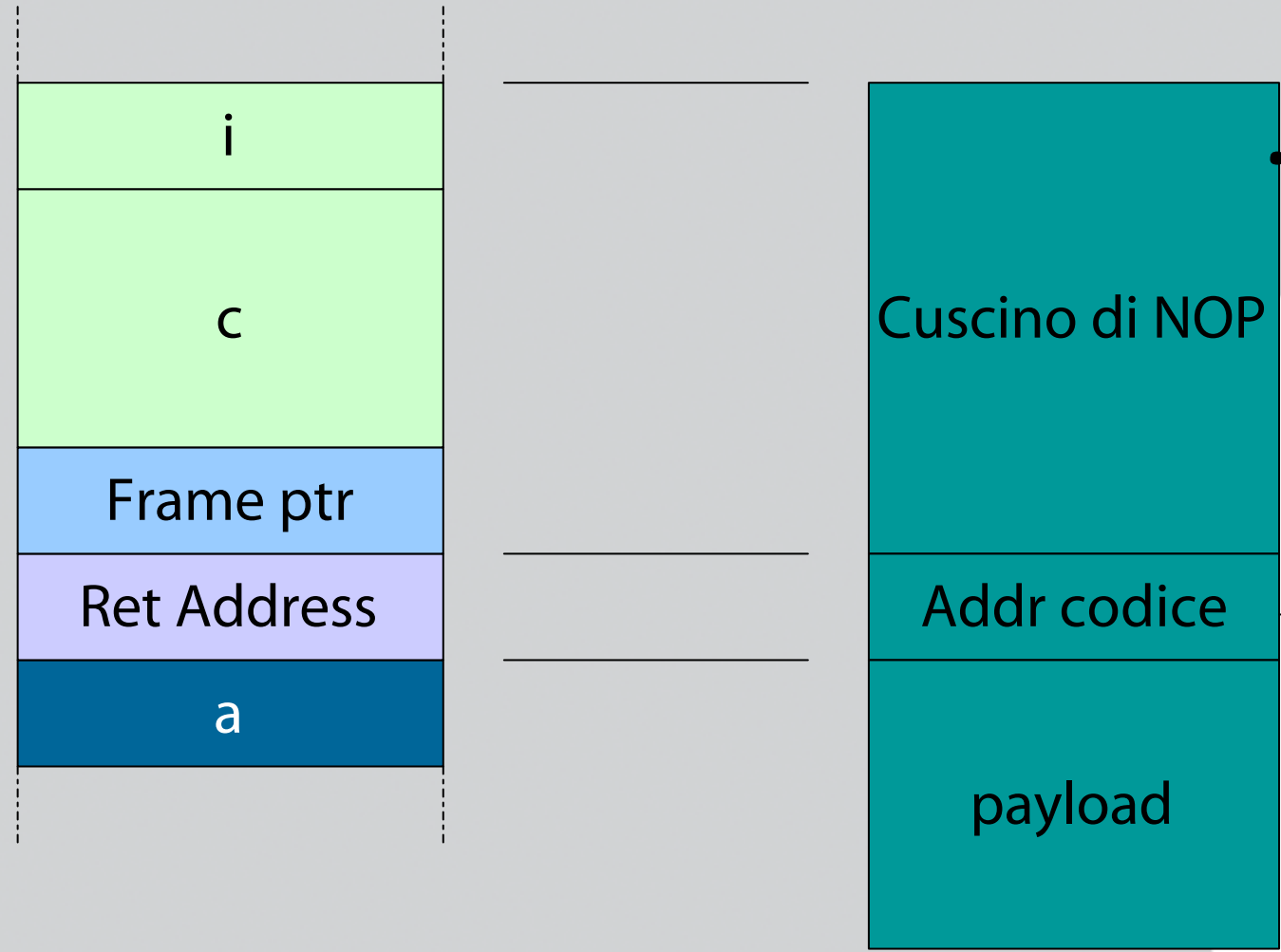


- Posso costruire **s** in modo che la parte che va a sovrascrivere l'indirizzo di ritorno punti all'interno dei dati iniettati
- Quando **f** finisce il suo lavoro invece di saltare indietro nella funzione chiamante salta nel buffer



- Posso costruire s in modo che la parte che va a sovrascrivere l'indirizzo di ritorno punti all'interno dei dati iniettati
- Quando f finisce il suo lavoro invece di saltare indietro nella funzione chiamante salta nel buffer

Anatomia di un exploit



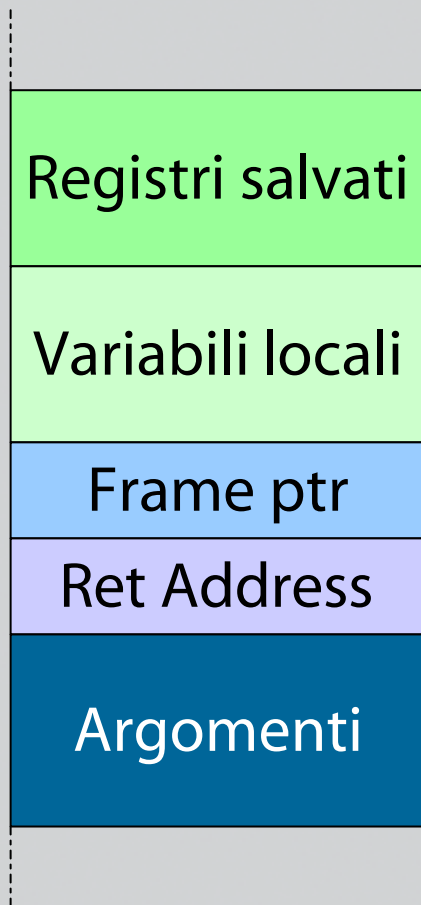
Anatomia di un exploit

- Il cuscino di NOP iniziale serve per non dover essere eccessivamente precisi nello specificare l'indirizzo di jump
- Il creatore dell'exploit deve prestare attenzione ai check fatti sull'input (alfanumerico, solo 7bit, ...)
- Il payload è la parte dell'exploit che compie l'azione vera e propria

Frame pointer overflow

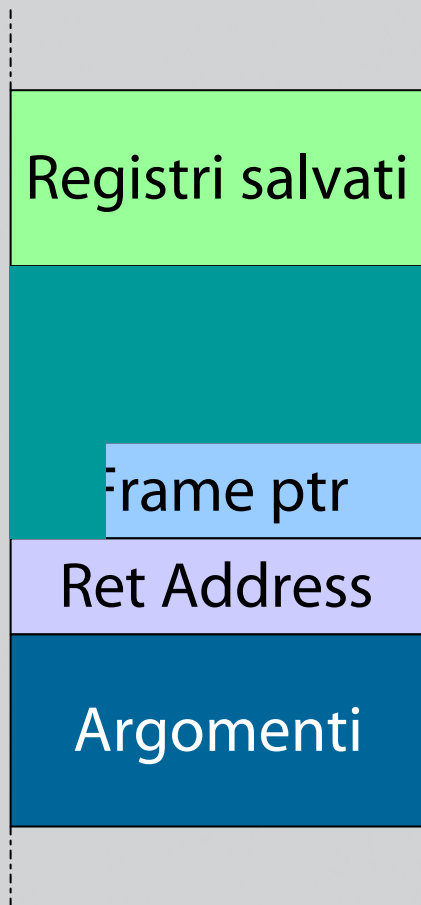
- Buffer overflow in cui è possibile sovrascrivere solo il primo byte del frame pointer
- Vengono detti off-by-one
- Spesso sono dovuti a differenti implementazioni di una funzione su più piattaforme (strncpy su Linux, Win32, ...)

Frame pointer overflow



- Solamente un byte del frame pointer viene sovrascritto
- Dovuti ad errori nel controllo della lunghezza (\leq invece di un $<$ stretto)

Frame pointer overflow



- Solamente un byte del frame pointer viene sovrascritto
- Dovuti ad errori nel controllo della lunghezza (\leq invece di un $<$ stretto)

Come è possibile ?

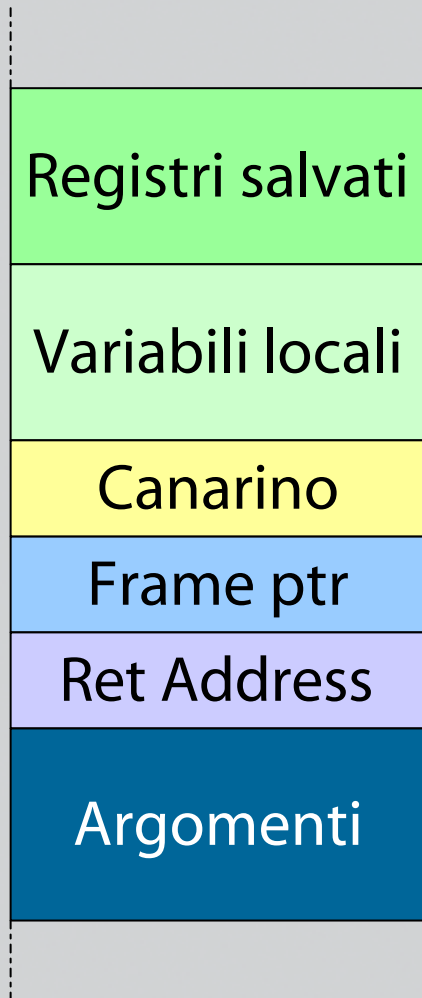
```
g()  
{  
  ...  
  f();  
  ...  
}
```

- La funzione **f** è attaccabile
- Molte funzioni hanno all'inizio un codice del tipo:
 push ebp
 mov ebp, esp
- Al termine invece:
 mov esp, ebp
 pop ebp
- Se dentro **f** il frame pointer viene cambiato al termine di **g** viene modificato lo stack pointer

- Stack non eseguibile
 - Al segmento di stack vengono rimossi i diritti di esecuzione, i dati presenti sullo stack non possono essere eseguiti
 - Alcuni compilatori usano lo stack per creare trampolini verso altro codice
 - Il codice per l'exploit potrebbe già trovarsi all'interno del programma è sufficiente passargli i dati giusti

- Canarino
 - Il compilatore viene modificato in modo da inserire un valore tra Frame Pointer e variabili in entrata ad ogni funzione
 - All'uscita da ogni funzione si controlla se il canarino è ancora vivo (intatto)
 - Il canarino può essere un valore casuale oppure semplicemente 0

Difendersi con il canarino

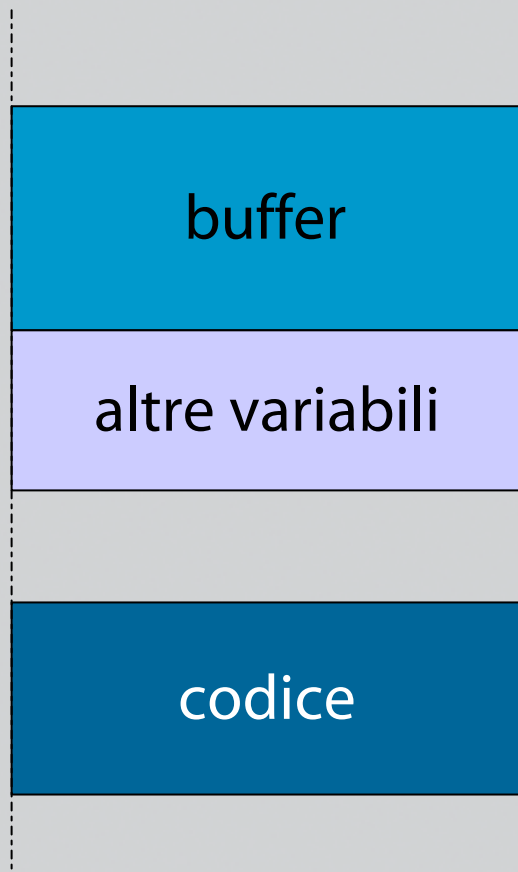


- Il canarino viene inserito in entrata alla funzione
- Viene controllato in uscita
- Se non è intatto significa che è stato sovrascritto e l'esecuzione viene interrotta
- Si evita di eseguire il codice dell'exploit

Heap-based overflow

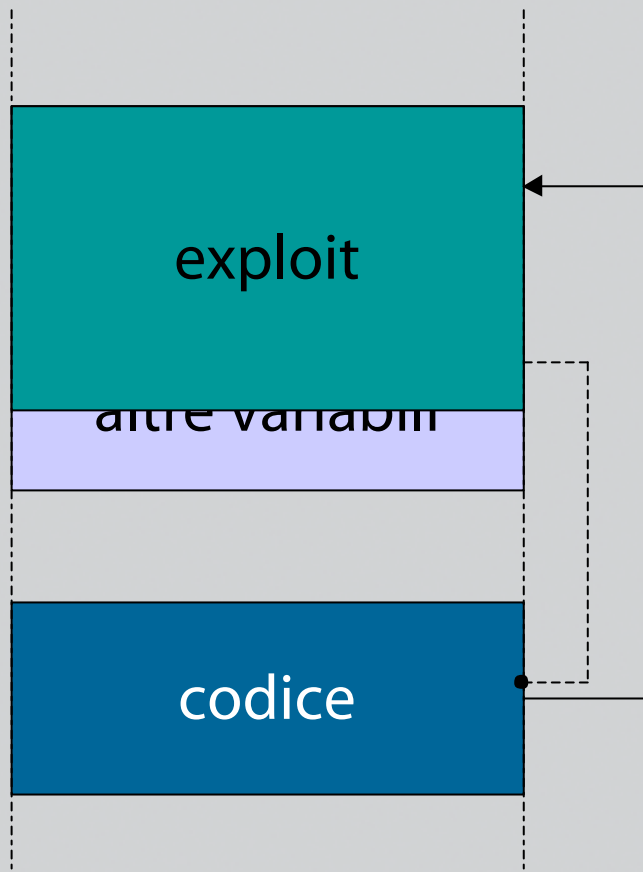
- Fino ad ora abbiamo visto solo attacchi allo stack
- E' possibile anche attaccare buffer presenti nell'area dati (heap)

Heap-based overflow



- Una volta sovrascritto il buffer è necessario trovare un modo per saltarci dentro
- L'idea è di sovrascrivere un puntatore al codice in modo da farlo puntare dentro al buffer

Heap-based overflow



- Una volta sovrascritto il buffer è necessario trovare un modo per saltarci dentro
- L'idea è di sovrascrivere un puntatore al codice in modo da farlo puntare dentro al buffer

Esempio di programma affetto

```
void
g()
{
}

void
main(int argc, char **argv)
{
    static struct {
        char buffer[10];
        void (*f)();
    } s;

    s.f = g;
    argv++;
    strcpy(&s.buffer, *argv);
    printf("%08x %08x\n",
           &s.f, &s.buffer);
    s.f();
}
```

- Dando un argomento abbastanza lungo è possibile sovrascrivere il puntatore a funzione f e direttore il flusso del programma

- Classi C++ (mix di puntatori a funzioni e dati)
- Funzioni di libreria che contengono puntatori statici a funzioni (atexit, ...)
- ...

- Sfruttano funzioni tipo ***printf***

```
printf(format_string, arg1, arg2, ...)
```

- Quando la format string può essere influenzata dall'utente è possibile che sia presente una Format String Vulnerability

Format String Functions

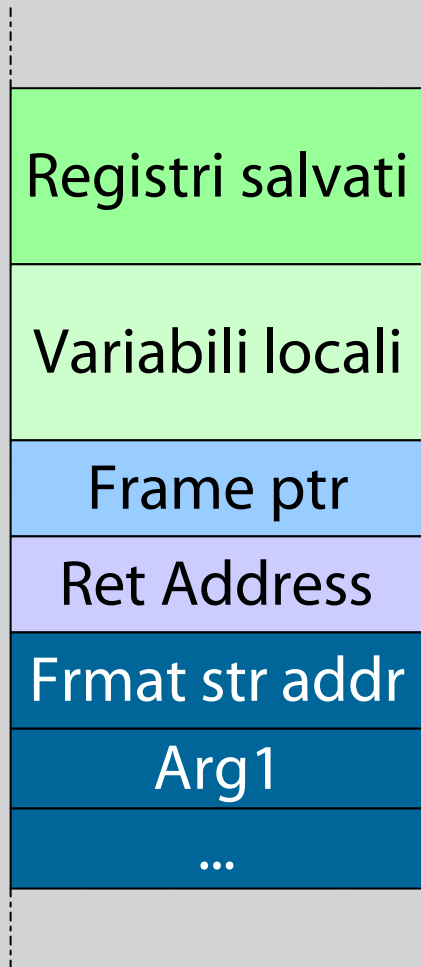
- Le funzioni tipo ***printf*** accettano una format string ed un numero variabile di argomenti
- In output viene generata una versione trasformata della format string

```
void  
f()  
{  
    int i;  
  
    i = 5;  
    printf("prova %d %08x\n",  
           i, &i);  
}
```

output

prova 5 00130000

Format functions stack frame



- Il chiamante conosce il numero di argomenti passati, il chiamato no
- Il primo argomento è la format string
- Il chiamato deduce dalla format string il numero di argomenti

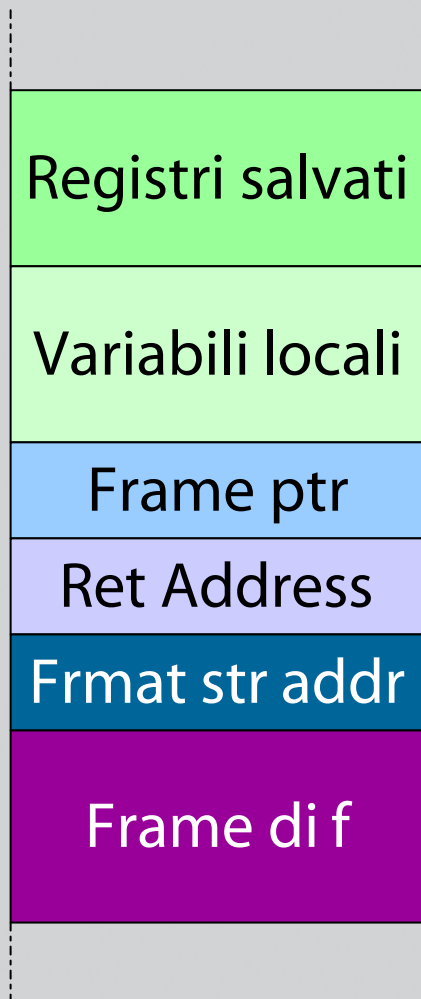
```
void  
f(char *s)  
{  
    char buff[512];  
    char outbuff[512];  
    sprintf(buff, "Err: %s",  
s);  
    sprintf(outbuff, buff);  
}
```

- Con il primo sprintf viene creata la format string per il secondo
- Passando alla funzione una stringa che funzioni da format string possiamo sovrascrivere lo stack

Esempio di format string exploit

- Passo alla funzione **f** una stringa tipo
“%S%S%S%S”
- Dopo il primo **sprintf** ho nel buffer
buff la stringa
“Err: %S%S%S%S”
- Nel secondo **sprintf** questa stringa
viene usata come format string

Esempio di format string exploit



- La funzione ***sprintf*** si aspetta di trovare 4 argomenti sullo stack (uno per ogni %s)
- Quando cerca di accedere a questi argomenti va a toccare il frame della funzione chiamante, in questo caso ***f***
- ***sprintf*** andrà a stampare valori “casuali” della memoria del processo. Crash del programma

Eseguire codice arbitrario

- Sfruttando una format string vulnerability è possibile anche far eseguire codice arbitrario al processo
- Il procedimento è abbastanza delicato ma lo scopo ultimo è sempre lo stesso: far saltare l'esecuzione all'interno del buffer riempito con il codice dell'exploit

- Sono gli errori più difficili da sfruttare
- Sfruttano una delle caratteristiche fondamentali dell'aritmetica dei calcolatori

- Nel linguaggio ci sono diversi tipi di interi
- Ogni tipo di intero è rappresentato da una variabile con un certo numero di bit
- Esempio su piattaforma Win32
 - int è un intero a 32 bit
 - short è un intero a 16 bit
- Ogni intero può avere un valore massimo pari a $2^{(\text{dimensione in bit})} - 1$

Esempio di funzione vulnerabile

```
int  
catvars(char *buf1, char *buf2,  
        int len1, int len2)  
{  
    char buff[256];  
  
    if ((len1+len2) > 256)  
        return -1;  
    memcpy(buff, buf1, len1);  
    memcpy(buff+len1, buf2, len2);  
  
    ...  
    return 0;  
}
```

Il controllo $(len1+len2)$ non tiene conto di un possibile integer overflow. Al memcpy è quindi possibile sovrascrivere lo stack e si ricade nel classico stack-based buffer overflow

Idea base degli exploit

- Tutti le tecniche di exploit che abbiamo visto si dividono in due parti:
 - iniezione del codice arbitrario in un buffer
 - dirottamento del programma verso quel buffer
- E' il dirottamento la parte più delicata e che differenzia una tecnica dall'altra

- Viene detto il **payload** dell'exploit
- E' il codice che ha il compito di compromettere il sistema
- Scrivere questo codice è complesso e delicato
- In genere il suo obiettivo è quello di aprire una porta di accesso al sistema (shellcode)

- Uno scrittore di shellcode deve affrontare diversi problemi
- Il principale è la differenza tra le varie versioni del sistema operativo
- Il secondo è ottenere tutte le funzioni che servono per svolgere il lavoro

- Sotto Win32 le funzioni di sistema sono suddivisi in librerie caricate dinamicamente dal programma (DLL)
- Gli indirizzi delle funzioni all'interno delle librerie cambiano da versione a versione

Import & Export Tables

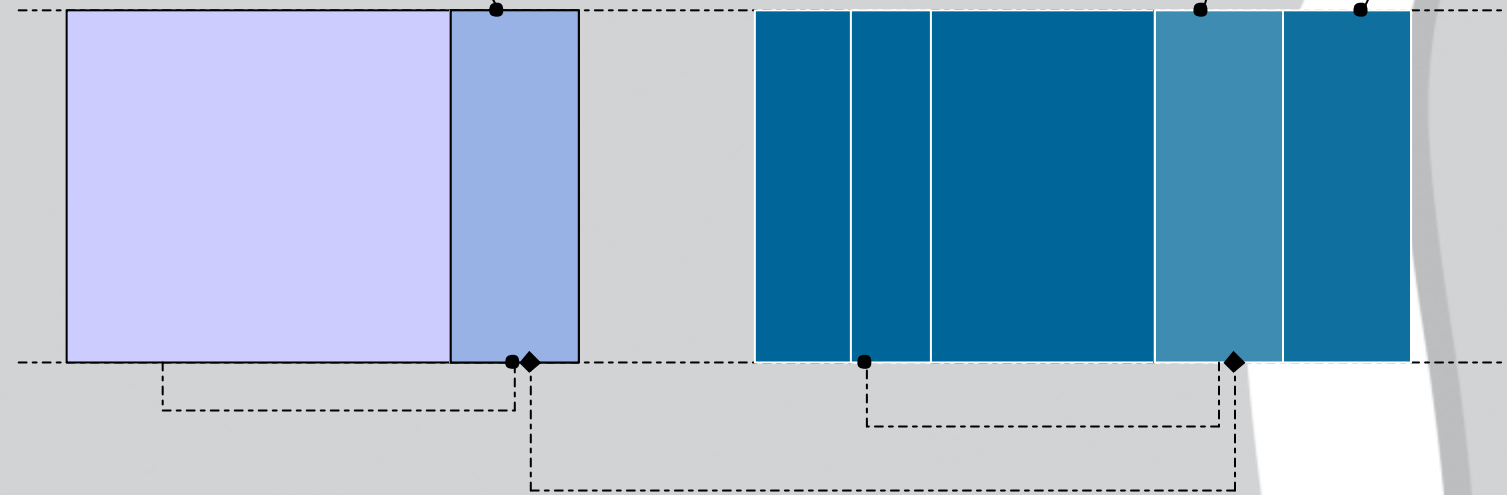
import table di foo.exe

import table di kernel32.dll

export table di kernel32.dll

foo.exe

kernel32.dll



Import & Export Tables

- In ogni eseguibile sotto Win32 (PE) ci sono una import ed una export table
- Nella Import Table ci sono le librerie e le funzioni necessarie al funzionamento del programma
- Nella Export Table ci sono gli indirizzi delle funzioni dell'eseguibile accessibili agli altri programmi

Import & Export Table

- Al momento dell'esecuzione il loader di Win32 si occupa di caricare tutte le DLL necessarie
- Completa quindi la Import Table dei vari componenti con il contenuto delle altre Export Table

Accesso alle funzioni run-time

- Il codice iniettato deve avere un modo per accedere alle funzioni run-time
- Non può usare un indirizzo fisso perché potrebbe cambiare da versione a versione
- Può utilizzare le funzioni `GetProcAddress` e `LoadLibrary`
- Con queste chiamate può avere accesso a tutte le altre funzioni
- Per usarle però deve trovare il loro indirizzo

Shellcode Win32 (IAT scanning)

- Per prima cosa il payload scandisce la IAT del suo eseguibile per trovare le chiamate LoadLibrary e GetProcAddress
- Sfrutta quindi queste due funzioni per ottenere l'indirizzo di tutte quelle che gli servono

- Ci sono altre tecniche usate dagli scrittori di exploit:
 - PEB
 - SEH
- Il problema che devono risolvere è sempre lo stesso: trovare una costante tra le varie versioni del sistema operativo

- Abbiamo visto una serie di attacchi che possono essere portati verso programmi con errori di programmazione
- Il problema è che questi errori non sono macroscopici ma spesso sono inezie
- Basta poco per rendere un programma vulnerabile

- Il **Tipo** di una variabile definisce l'insieme dei valori che può assumere durante l'esecuzione
- I linguaggi di programmazione si possono classificare in base al loro sistema di tipi e alle sue caratteristiche

- Gli errori run-time possono essere distinti in due classi
 - **trapped errors.** Gli errori che vengono rilevati
 - **untrapped errors.** Errori che non vengono rilevati
- Un buon sistema di tipi può eliminare completamente la seconda classe di errori rendendo il linguaggio **safe**

- Un exploit che sfrutta un qualche overflow genera un untrapped error
- E' possibile eliminare questo tipo di errori utilizzando linguaggi di programmazione safe
- Il linguaggio C non è safe

- **Memory layout Win2000**
 - Solomon, Russanovich “Inside Microsoft Windows 2000”, Microsoft Press
 - Articoli di Matt Pietrek su MSJ
- **Stack overflow**
 - Aleph1 “Smashing the stack for fun and profit”, Phrack Magazine 49
 - Dark Spyrit “Win32 Buffer Overflows”, Phrack Magazine 55
 - Dark Spyrit “Klog - The Frame Pointer overwrite, Phrack Magazine 55
- **Heap overflow**
 - Matt Conover “Heap overflows”
- **Sistemi di tipi**
 - Luca Cardelli “Type Systems”