Appunti di Sicurezza Informatica

Caccaro Sebastiano
A.A.2019/2020

Contents

1 Introduzione

Sito del corso: `http://security.di.unimi.it/sicurezza1920/sec2.shtmls`
L'esame sarà a febbraio
Modalità esame: parte a quiz fatta computer + parte pratica sempre fatta a computer.

2 Memory Errors

Specialmente in certi linguaggi di programmazione, come C e C++, che non hanno meccanismi di controllo su quello che fa il programmatore con la memoria. Posso quindi sovrascrivere zone di memoria che non sono di mia competenza. Posso quindi creare degli **unexpected behaviour**.
Posso usare queste falle per comportamenti malevoli, come eseguire codice scritto da un attaccante ecc, per rubare dati ecc.
Sono scritti in C o C++ componenti che devono essere velocissimi, come sistemi operativi e sistemi critici, server web, embedded sistems.
Altri linguaggi non hanno questi problemi perchè inseriscono dei controlli, ma a scapito delle perfomance.

2.1 Buffer Overflow

2.1.1 Storia

Comincia del 1988 con il **Morris Worm**. Nel giro di 3-4 ore butta giù tutta la DarpaNet, prendendo una multa di 10-100 milioni di dollari, galera ecc.

> **Worm**
> Programma che si autoreplica e si diffonde in varie macchine

Nel 2001 **Code Red** infetta 300.000 macchine in 14 ore, nel 2003 **SQL Slammer** infetta 75.000 macchine in 10 minuti.
Molte vulnerabilità non vengono mai corrette, perchè comunque l'applicazione funziona lo stesso.

2.1.2 Layout di memoria

> **Memoria Virtuale**
> Modalità di visualizzazione della memoria nella quale un processo vede tutta la memoria come se fosse assegnata solo a lui.

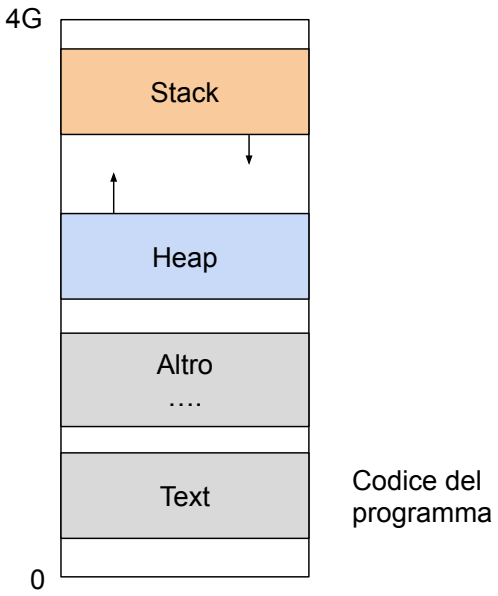

Figure 1: Layout di memoria nei processori Intel

Mentre lo Stack cresce verso il basso, lo Heap cresce verso l'alto. Quindi, se nello stack per allocare devo sottrarre (andare giu), nello heap devo andare verso l'alto.
Lo Heap contiene perlopiù variabili allocate dal programmatore. Nello Heap, per allocare memoria, devo usare `malloc(sizeof(tipo))`. Una volta che non mi serve più quella zona, la libero con un `free`.

2.1.3 Funzionamento dello Stack

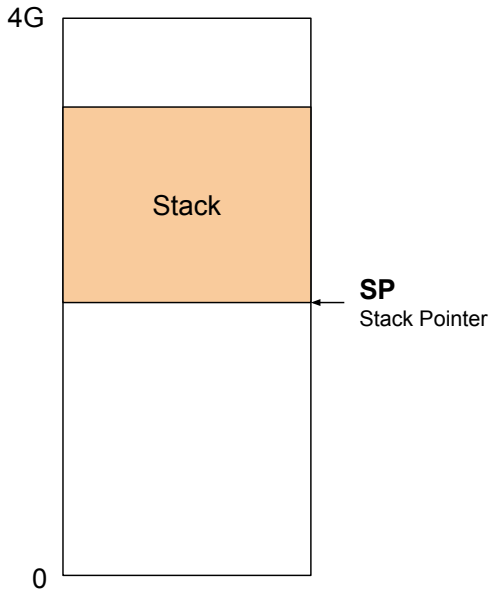

Figure 2: Stack Pointer

Solitamente lo stack è utilizzato dal programma, in genere per le chiamate a funzioni. Lo stack è delimitato dallo **stack pointer**.

```
function(a1,b,c)
  int z
  strcpy(a,a1)
  return

main(...)
  function(a,b,c)
  Y:RET
```

Figure 3: Esempio di chiamata a funzione

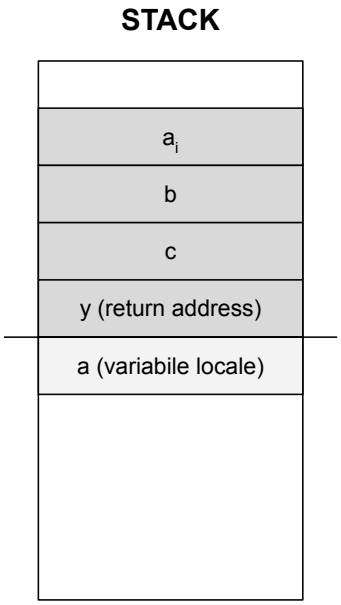

Figure 4: Stack Pointer dell'esempio precedente

Quando vado ad eseguire la funzione, alloco anche un return address, che mi serve per capire dove andare alla fine della funzione. I parametri della funzione vengono poi espressi in funzione della posizione dello stack pointer (SP+16 bit ecc). Alla fine della funzione, sposto l'instruction a pointer viene impostato con il valore presente in Y, ovvero il return address. È compito del chiamante poi togliere dallo stack gli altri parametri (`a`, `b` e `c`). Cosa succede quindi ad una chiamata?
Chiamando la funzione:

1. Push degli argomenti nello stack
2. Push del return address
3. Jump all'indirizzo della funzione

Tornando alla funzione principale:

1. Ritorno allo stack frame precedente
2. Torno al return address

2.1.4 Problematiche

> **Buffer Overflow**
> Operazione di scrittura che va a eccedere la locazione di memoria allocata a un dato dato.

```
void func(char *arg1)
{
 int authenticated = 0;
 char buffer[4];
 strcpy(buffer, arg1);
 if (authenticated) {...
 ...
}

int main()
{
 char *mystr = "AuthMe!";
 func(mystr);
 ...
}
```

Figure 5: Programma che esegue un buffer overflow

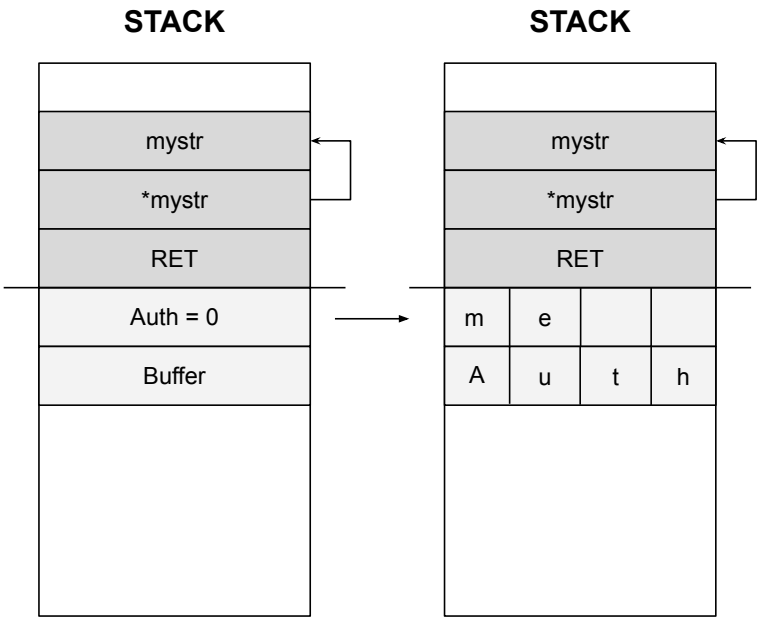

Figure 6: Stack dell'esempio precedente

Sto provando a copiare una stringa "`AuthMe`" nel buffer a 4 byte. Siccome un carattere corrisponde a un byte, riesco a copiare solamente "`Auth`". Quindi, per scrivere il "`Me`" devo andare sopra nello stack. Quindi ho scritto "`Me`" nella variabile `Authenticated`, che adesso è diversa da 0. Ora quindi sono autenticato!
Con questo metodod potrei tranquillamente sovrascrivere tutto lo stack. Potrei, ad esempio, **sovrascrivere il return address**, saltando quindi in qualsiasi punto all'interno del programma, addirittura a parti del codice inserite in modo malevolo.

2.1.5 Code Injection

Lo scopo di una **Code Injection** è quello di fornire il codice da eseguire all'interno del programma. Purtroppo non posso metterla all'interno della sezione `Text`, perchè è read-only. Mettiamo quindi il codice all'interno dello stack.

```
void func(char *arg1)
{
 char buffer[4];
 sprintf(buffer, arg1)
}
```

Figure 7: Snippet

L'idea è quella di sovrascrivere buffer fino al return address. Il mio scopo è quello di inserire il mio codice all'interno di buffer, fino ad arrivare alla zona di memoria del return address. Nel return address andrò quindi a puntare l'indirizzo di memoria nel buffer, andando quindi ad eseguire il codice contento all'interno di essa.
Ma come faccio a sapere l'indirizzo di memoria attuale del buffer? Siccome la memoria è virtuale, riesco a sapere più o meno un range dove si trova l'indirizzo del buffer. Quindi inserisco prima del codice una sequenza di `NOP`, una cosidetta **NOP-sled** (pista d'atterraggio) per il mio codice.

> **NOP**
> Short for NO Operation. Istruzione assembly che dice al processore di non fare niente per un ciclo di clock

Devo quindi avere le seguenti cose:

- Distanza da sovrascrivere
- Codice da inserire (inteso come codice macchina, binario, ancora più basso dell'assembly)
- Indirzzo del buffer

Creo quindi il cosidetto **attack vector**.

> **Attack Vector**
> Input che devo fornire al programma per inserire il mio exploit

Per capire un po'meglio dove è il return address, provo a sovrascrivere con dati a caso sempre più grandi. Se il programma crasha, vuol dire che sta

probabilmente cercando di saltare a un indirizzo a caso. Quindi è probabile che abbia trovato il return address.

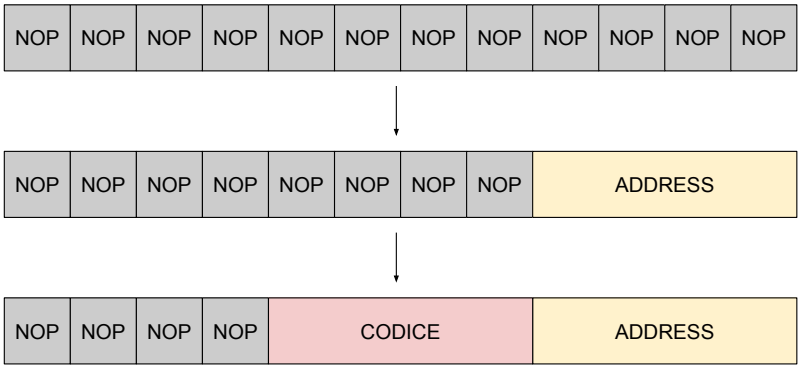

Figure 8: Costruzione di un attack vector

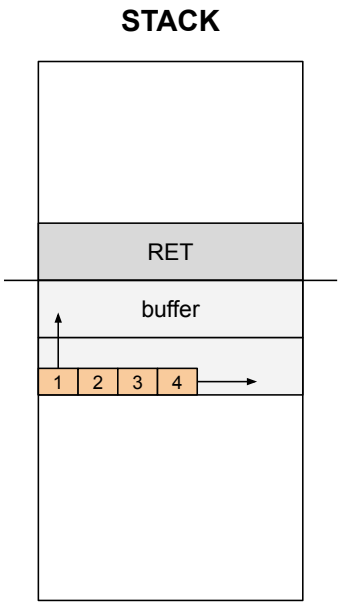

Figure 9: Processo di scrittura della memoria

Il mio scopo solitamente è quello di acquisire maggiori privilegi. Lo faccio tramite qui programmi denominati **set whith rooot**, ovvero quei programmi che possono essere eseguiti da un utente normale ma che girano con permessi di root.

2.1.6 Esempio di Code Injection

Esercizio 1 disponibile qui `https://github.com/andrealan/Software-Security-Lab/tree/master/bof-exercise`

> **Shell Code**
> Codice che esegue una nuova finestra della shellS

Il programma attaccante dichiara lo **shell code**. Nel main viene creato l'attack vector.

1. Creo un buffer
2. Lo riempio di NOP
3. Dichiaro l'offset rispetto al stack pointer, potrei doverlo cambiare
4. Inserisco il valore dell'indirizzo all'inizio
5. Scrivo lo shellcode dopo l'indirizzo

IMPORTANTE

La lettura nello stack avviene verso l'alto. Quindi la NOP sled più che come una discesa, può essere pensata come un uno skilift.

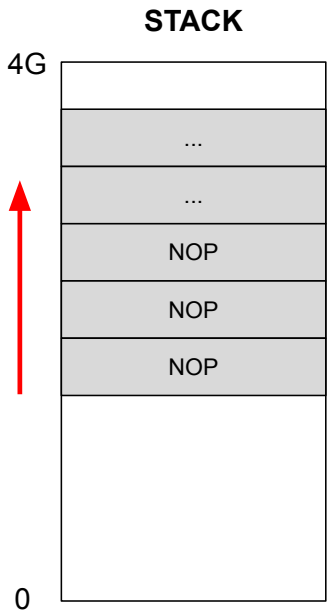

Figure 10: La lettura dello stack avviene verso l'alto