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.

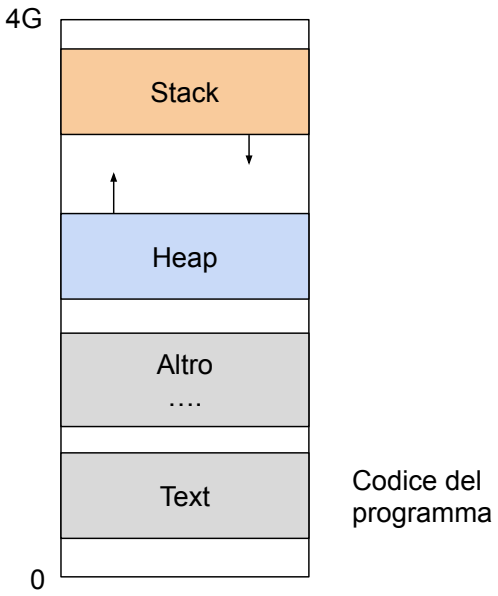

Figure 1: Layout di memoria nei processori Intel

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

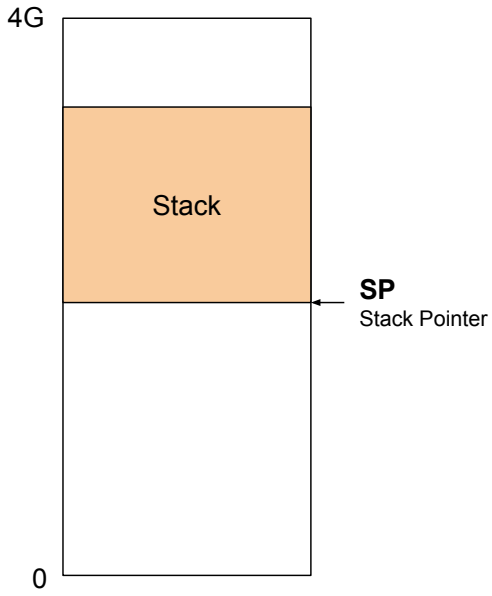

Figure 2: Stack Pointer

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

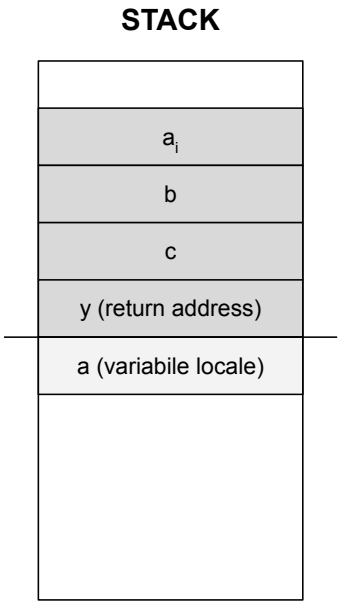

Figure 4: Stack Pointer dell'esempio precedente

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

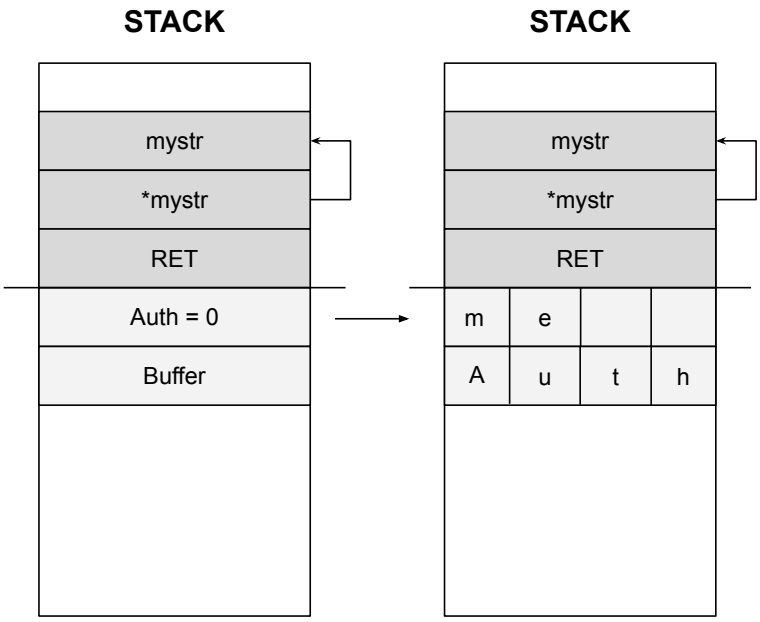

Figure 6: Stack dell'esempio precedente

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.

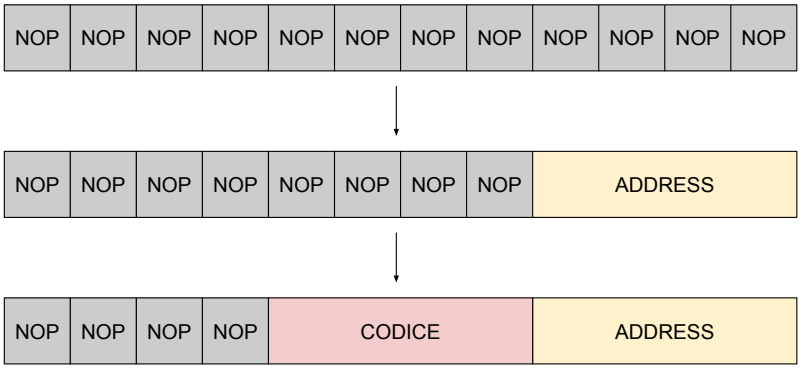

Figure 8: Costruzione di un attack vector

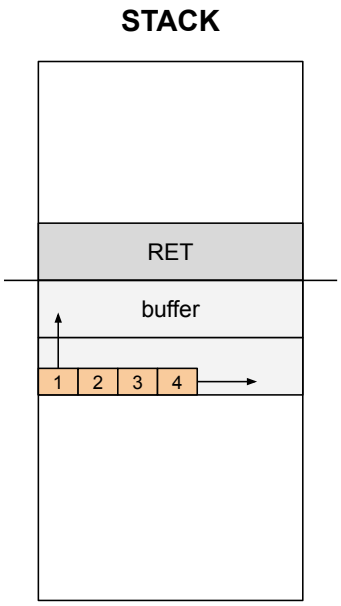

Figure 9: Processo di scrittura della memoria

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.

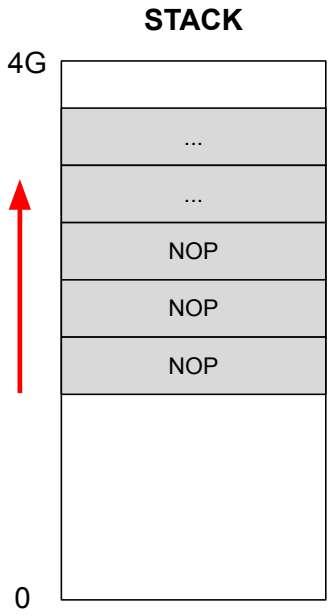

Figure 10: La lettura dello stack avviene verso l'alto

2.2 Heap Overflow

2.3 Concetti generali sull'allocazione nello Heap

L'allocazione nello Heap è gestita manualmente dallo sviluppatore e consente l'allocazione di grosse strutture dati anche dinamicamente a run time. Generalmente è più lenta rispetto allo stack.
Le due operazioni permesse sullo Heap sono:

- `malloc(bytes)`: richiede l'allocazione di tot byte nello Heap.
- `free(puntatore)`: libera una porzione di memoria precedentemente occupata.

2.3.1 Heap Chunck

Lo Heap è organizzato in **chunck**, struttura dati rappresentata in seguito.

```
struct malloc_chunk {
  /* Size of previous chunk (if free).  */
  INTERNAL_SIZE_T      prev_size;
  /* Size in bytes, including overhead. */
  INTERNAL_SIZE_T      size;

  /* double links -- used only if free. */
  struct malloc_chunk* fd;
  struct malloc_chunk* bk;

/*Only used for large blocks: pointer to next larger size.*/
/*double links -- used only if free. */
  struct malloc_chunk* fd_nextsize;
  struct malloc_chunk* bk_nextsize;
}
```

fd e **bk** stanno per **forward** e **backwards**. Nello Heap, i blocchi liberi sono organizzati in liste doppiamente linkate con chunk di dimensioni simili. I blocchi occupati, non hanno tali puntatori.
Per ogni chunk, se il chunk precedente è libero, il bit meno significativo di INTERNAL_SIZE_T è posto a 0, altrimenti va posto a 1.

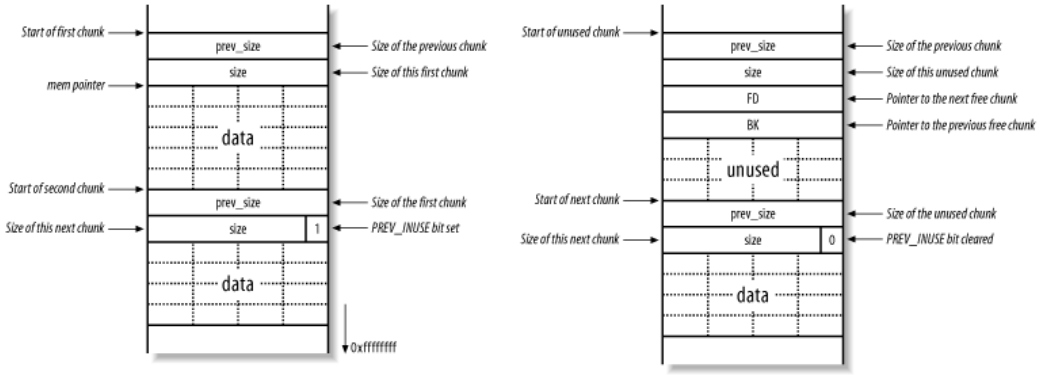

Figure 11: Sezione di Heap prima e dopo la free di un chunk

2.3.2 Allocazione Memoria

Il chunck che rappresenta la memoria libera nello Heap è detto **Top Chunk**. Ogni qualvolta voglio allocare nuova memoria, il top chunk viene diviso in due: una parte diventa memoria allocata, la parte rimasta libera diventa il nuovo top chunk.
Ovviamente si ricorre allo splitting del top chunck solamente se non ci sono chunk liberi di dimensioni idonee nelle liste di chunck liberi.

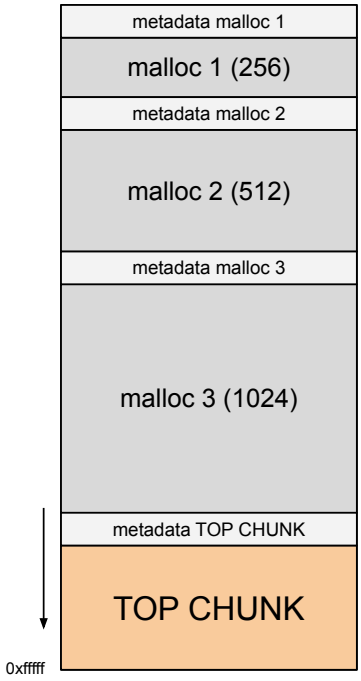

Figure 12: Heap dopo un malloc di 256, uno di 512 e uno di 1024 in quest'ordine

2.4 Deallocazione memoria

Quando dealloco memoria svolgo le seguenti operazioni:

- Pongo a 0 il bit meno significativo del campo size del blocco successivo a quello che sto deallocando.
- Pongo nel campo `prev_size` del chunk successivo la dimensione del chunk che sto allocando.
- In caso anche il chunk precedente sia libero, unisco i due chunk.

2.4.1 Unsafe Unlink

Quando vado a deallocare un chunck preceduto o succeduto da un un chunk libero, posso andare a mergiare questi due chunk.
Analizziamo il caso in cui il chunk precedente (p) a quello che devo deallocare sia libero. Svolgerò le seguenti operazioni:

- Incremento la size di p, inglobando il chunk successivo.

- Eseguo l'**unlink** di p (che quindi ora ingloba anche il chunk che voglio deallocare)
- Posiziono p nella lista di chunk più appropriata

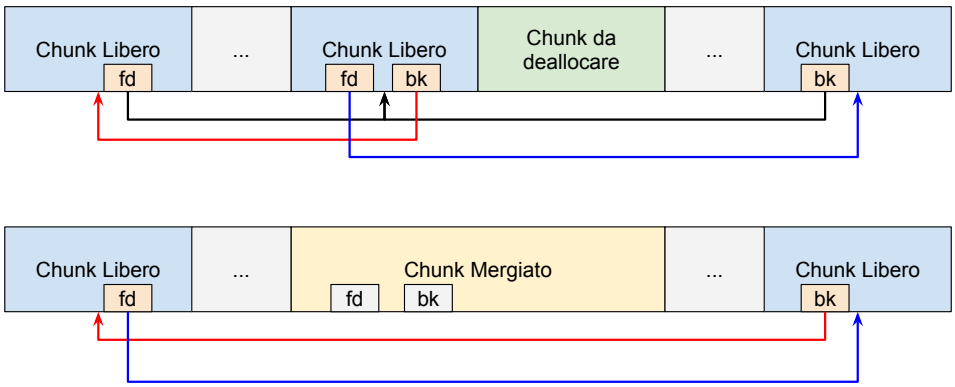

Figure 13: Situazione prima e dopo unlink

Questa procedura avviene tramite il seguente codice

```
#define unlink(P, BK, FD) {
  FD = P->fd;
  BK = P->bk;
  FD->bk = BK;
  BK->fd = FD;
}
```

http://etutorials.org/Networking/network+security+assessment/Chapter+13.+Application-Level+Risks/13.5+Heap+Overflows/

2.5 User After Free

È più legato a proprietà temporali della memoria che a proprietà spaziali. Rispetto ad altre vulnerabilità, è più difficile da individuare e da correggere. Ad oggi, è una delle vulnerabilità più usate e sfruttate. Ne sono state trovate in browser come Chrome, ma anche in altri browser.
In linguaggi come Java queste cose non sono possibili perchè sono presenti vari controlli sulla memoria. Rispetto a questa specifica vulnerabilità, Java si protegge tramite il garbage collector.
La user after free avviene quando una zona di memoria viene liberata, ma provo comunque a dereferenziare il puntatore che la puntava. Si basa quindi sui **dangling pointers**. Una volta deallocata una zona di memoria, è buona pratica settare il suo puntatore a 0.
Sostanzialmente, quello che si può fare è usare un dangling pointer per accedere a zone di memoria che dovrei lasciare stare.

2.5.1 Esempio di Use After Free

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

struct auth {
  char name[32];
  int auth;
};

struct auth *auth;
char *service;

int main(int argc, char **argv)
{
  char line[128];

  while(1) {
      printf("[ auth = %p, service = %p ]\n", auth, service);
```

```
        if(fgets(line, sizeof(line), stdin) == NULL) break;

        if(strncmp(line, "auth ", 5) == 0) {
            auth = malloc(sizeof(auth));
            memset(auth, 0, sizeof(auth));
            if(strlen(line + 5) < 31) {
                strcpy(auth->name, line + 5);
            }
        }
        if(strncmp(line, "reset", 5) == 0) {
            free(auth);
        }
        if(strncmp(line, "service", 6) == 0) {
            service = strdup(line + 7);
        }
        if(strncmp(line, "login", 5) == 0) {
            if(auth->auth) {
                printf("you have logged in already!\n");
            } else {
                printf("please enter your password\n");
            }
        }
    }
}
```

Il programma è un loop continuo che chiede all'utente che servizi vuole. Posso scegliere fra 4 comandi, a seconda di quello che voglio fare. Il comando auth vuole una sintassi del tipo `auth admin`. Reset libera la memoria puntata da auth. Service copia la linea inserita e la mette dentro dentro `service`. Login controlla se io mi sono già loggato o no. Voglio autenticarmi anche se non so la password, mettendo auth− >auth a 1. I passaggi per loggarsi sono i seguenti:

- Chiamo il comando `auth`, scrivendo tipo `auth seba`
- Chiamo il reset, andando a deallocare auth
- Chiamo service e una stringa lunghissima, tipo `service AAAA[...]AAA`. Non posso metterla troppo lunga però, altrimenti andrei nel top chunk. Meglio andare ad allocare piccole stringhe "a tentoni" o guardando bene con gdb

- Ora, se chiamo login, sono dentro. Questo perchè sto provando a leggere la zona di memoria di auth, che è stata deallocata (è però rimasto un dangling pointer) ma sostituita dalla nuova stringa service, quindi `auth = 1`

3 Gdb

Gdb serve per debbugare e fare gli hackerini. Si lancia con `gdb ./nome-programma`. Ecco una lista di comandi utili:

- `disasasemble` scompone il codice in formato assembler. Se vedo tutti gli zeri, devo caricare gli indirizzi. Scrivo:
  - `*b 0x0` Inserisco breakpoint
  - `r` per lanciare il programma, mi da errore ma ok
  - `delete breakpoints`
  - `disassemble main`
- Per caricare lo heap devo: far partire il programma , fare un'allocazione e poi fare ctrl + c.
- `set pagination off` toglie la paginazione.
- Con il comando `info reg` ottengo l'indirizzo di tutti i registri, fra cui lo **stack pointer** e l'instruction pointer.
- Per vedere cosa succede nello stack, scrivo x/40x e poi l'indirizzo dello stack. Sto dicendo a gdb di stampare 40 words in esadecimale. È più comodo scrivere `x/40x $sp`
- Per analizzare lo heap, con `info proc mappings` trovo il mappaggio in memoria del programma e quindi anche nello heap. Quindi, lanciando `x/40x indirizzo-heap`, riesco a visualizzare lo heap. La sequenza di comandi è:
  - `r`
  - eseguo almeno un'operazione sullo heap
  - `ctrl + c`
  - Proseguo con quello che devo fare
  - metto

Posso creare comandi ad hoc tramite `command`. Posso fare comandi come:

- `echo stringa` per stampare una stringa.ì
- `print nomevariabile` per stampare il contenuto di una variabile
- Gli stessi comandi di prima, come `x/40x ecc`.
- Termino l'inserimento di comandi con `continue` e `end`.

Questo command verrà eseguito ad ogni breakpoint.
Quando eseguo con gdb, lo stack del mio programma viene shiftato in giù di qualche unità.

31 Ottobre 2019

4 Memory Safety

> **Memory Safety**
> Insieme di meccanismi di controllo che salvaguardano la sicurezza della memoria rispetto ad attacchi come overflow ecc. È la combinazione di **Temporal Safety** e **Spatial Safety**

La memory safety è utilizzata dalla maggior parte di tutti i linguaggi moderni.

4.1 Spacial Safety

Il concetto principale è che ogni puntatore abbia solo una determinata zona di memoria a cui può puntare e non può uscirci.
Bisogna vedere i puntatori come triple:

- **p** è il puntatore vero e proprio
- **b** è la base della memoria
- **e** è la lunghezza della sezione di memoria

Posso accedere un area di memoria solo se `b` $\leq$ `p` $\leq$ `e-sizeof(typeof(p))` . Le operazioni sul puntatore vanno a interessare solo **p**, senza toccare **b** e **e**. Tramite questo metodo, sto praticamente prevenendo i buffer overflow. L'unica downside è che ogni volta che dereferenzio un puntatore, devo eseguire i controlli.

4.2 Temporal Safety

Si assicura che la memoria a cui sto puntando sia ancora **definita** (= allocata). Quindi, mi impedisce di derefenziare puntatori a sezioni di memoria non allocate, evitando attacchi come **Use After Free**. Solitamente è implementata tramite una tabella che traccia le zone in uso o meno, detta **garbage collector**.

4.3 Type Safety

Ad ogni oggetto è associato un tipo, le operazioni eseguite devono rispettare il tipo dei puntatori. È più forte della Memory Safety. Il controllo del tipo può essere fatto sia a compile time che a run time.

5 Tecniche di difesa

Esistono tecniche per eliminare del tutto o rendere più difficile l'exploitation di un certo bug. Queste tecniche vanno implementate non nel codice sorgente, ma nelle **librerie**, nel **compilatore** e nel **sistema operativo.**

5.1 Canaries (Canarini)

Riferimento ai canarini che si portavano nelle miniere. Il **canarino** cerca di dividere i dati utente dai dati di controllo. È una sezione di memoria prima del `RET` che viene sovrascritto quando provo a fare un buffer overflow. Essendo il valore del canarino modificato, è possibili individuare il tentativo di exploitation. Esistono vari tipi di canarino:

- **Terminator Canaries**: Riempiti con caratteri che non posso scrivere in memoria con scanf ecc, come `CR, LF, NUL, 0` .
- **Random Canaries**: Creo una stringa random e la salvo da qualche parte in memoria, poi vado a controllare che corrisponda.
- **Random XOR Canaries**: Come quelli random, solo che non salvo una copia ma il suo XOR.

Ogni volta che entro in una funzione `F`, eseguo i seguenti passaggi:

- Inizializzo la funzione, quindi pusho il `RET` e cose ausiliarie
- Genero un valore random
- Eseguo il push di questo valore sullo stack
- Salvo il valore anche in memoria
- ..... Eseguo il corpo della funzione.....
- Prima di leggere il `RET`, controllo che il canarino corrisponda al valore in memoria

Questa tecnica, con qualche modifica, può essere usata anche contro la sovrascrittura dei metadati sullo heap.

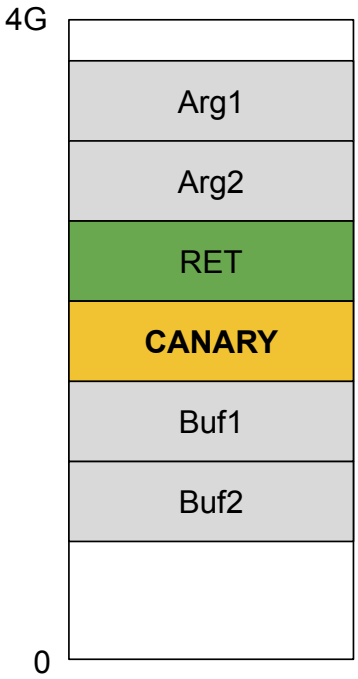

Figure 14: Stack con canarino

5.2 DEP Data Execution Prevention

Vengono resi lo stack e le heap non eseguibili (non dovrebbero contenere codice), e se provo a metterci dello shellcode il sistema operativo lancia un'eccezione.
Quindi, come faccio ora a fare l'hackerino?

5.2.1 Return-to-lib-c

In memoria ho comunque molto codice. Nelle librerie c che vengono caricate automaticamente è presente `SYSTEM()`. Quindi, al posto del `RET` metto l'indirizzo del system, e passo `"/bin/sh"` come argomento.

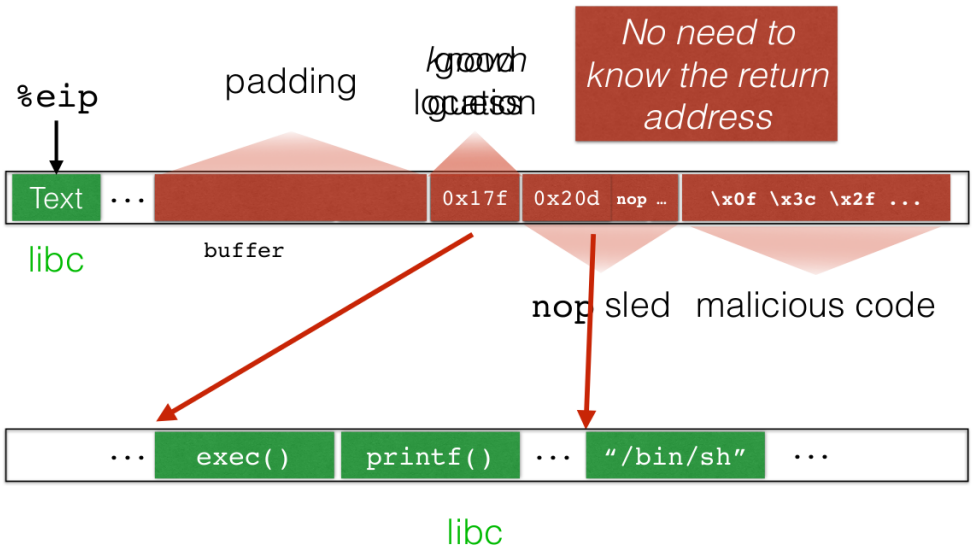

Figure 15: Return to libc

5.3 Adress space layout randomization

Ogni volta che ho un programma, carico le zone di memoria (HEAP, TEXT, STACK, BSS) in ordine diverso. Ho alcuni problemi:

- Shifto solo i blocchi, ms non le posizioni relative all'interno di essi.
- È più applicabile alle librerie che al codice del programma vero e proprio.
- Funziona male con 32 bit, perchè posso fare il brute force dell'indirizzo.

7 Novembre 2019

6 Return-oriented Programming (ROP)

Attacco che usa pezzi di codice già esistenti, detti **gadget**, che vengono composti in modo **automatico** per comporre del codice malevolo. Bisogna quindi trovare i gadget e poi concatenarli insieme.
Un gadget è un gruppo di istruzioni che termina sempre con un `ret`. Concateno questi gadget passando ad ogni gadget l'indirizzo del gadget successivo, attraverso le istruzioni `pop` dello stack. Alla fine, riesco sempre a trovare dei gadget, perchè l'architettura Inter x86 è CISC, ed è piena di istruzioni che possiamo usare, posso saltare in mezzo alle istruzioni, e il codice

è self-reparing.
Per scrivere il codice malevolo, si usa un cosidetto **ROP Compiler**. Questo tool si occupa di prendere il codice dell'attacco (scritto in C) e il programma vittima, e mi fornisce il codice assembler da inserire al momento dell'attacco.

6.1 Blind ROP

Attacco che ha come vittima un server remoto (linux). Non conosco l'ambiente, e quindi non so dove sono i gadget. Ci sono canarini, randomization ecc. L'idea è quella di farsi dare una copia dell'eseguibile, in modo tale da poter poi costruire l'exploit. Condizioni:

- Macchina a 64 bit
- ASLR abilitato
- Canarini abilitati

Ad ogni collegamento ad un server, viene creato un thread nel server per gestire la connessione con il client. Questo thread prende le caratteristiche del processo padre, ma in caso di crash viene ricreato **senza randomizzazione**. L'attacco si svolge così:

- **Stack reading:** leggo i canarini e il return address sullo stack.
- Trovo i gadget che mi permetto di usare la funzione `write`.
- Trovo i gadget per trovare tutti gadget per costruire lo shellcode.
- Ora posso eseguire un rop classico.

Quando individuo i gadget, li divido nelle seguenti categorie:

- **Crash:** causano il crash del programma
- **Infinite Loop:** gadget che possono essere chiamati indefinitamente. Sono particolarmente interessato a quelli che eseguono operazioni sullo stack.