Tecniche di Protezione del Software

Massimo Perego

Contents

1 Buffer Overflow

Ci concentriamo su linguaggi low level (e.g., C), i quali tendono a crashare in caso di errori (come buffer overflow), ma un attaccante può **sfruttare le vulnerabilità** per ottenere informazioni (e.g., Heartbleed, bug SSL che permetteva di leggere tutta la memoria del programma, che su SSL insomma, peso), corrompere memoria, fino ad arbitrary code execution (la macchina comincia ad eseguire altro, diventa una "weird machine"), ecc.
Il crash (ovvero `segfault`), se analizzato può portare ad un attacco, anche se non tutti i casi sono exploitabili (molti sì).

Questo tipo di bug hanno una lunga storia e sono tuttora presenti, e lo saranno finché C e C++ saranno usati. Inoltre è utile studiare l'evoluzione del bug stesso, assieme alle difese create per contrastarlo. Alcune caratteristiche di un attacco/difesa possono risultare presenti anche in altri attacchi.

Inoltre, solitamente, l'attacco è molto più semplice della difesa. Per l'attacco mi basta un punto, per la difesa devo essere sicuro di aver coperto tutti i possibili punti di attacco, senza degradare troppo le performance.

I sistemi C e C++ sono ancora molto presenti e spesso sono parte di sistemi critici come:

- OS, Kernel e relative utilities
- Server che richiedono alte prestazioni (Apache httpd, nginx, MySQL, redis)
- Sistemi embedded (risorse limitate, le performance sono importanti)

La prima versione di buffer overflow funzionante è del 1988: Morris Worm, per poi dare inizio ad una catena di exploit che permettono la compromissione della macchina stessa (a diversi livelli), in qualsiasi caso con un impatto significativo.

1.1 Memory Layout

Dobbiamo sapere come un programma viene caricato in memoria, in quali zone e di conseguenza cos'è lo stack e quali sono gli effetti delle chiamate a funzione. Parleremo del modello Linux `x64`, anche se il concetto dell'attacco è universale l'implementazione può cambiare in base a dettagli tecnici (architectural dependent).

Ogni processo ha un proprio **layout di memoria**, con indirizzi che vanno da `0x00000000` a `0xffffffff` (per 32 bit, con 64 sarebbero troppo lunghi da scrivere), quindi 4GB di indirizzamento totali.

Linux divide:

- 1GB per il sistema operativo, dall'alto
- 3GB per le applicazioni

Di conseguenza il primo indirizzo valido per il programma è `0xbfffffff`, sopra c'è il kernel.

Il **loader carica in memoria** un **processo** quando questo viene chiamato, occupando la page table ed allestendo la memoria per l'uso del programma.

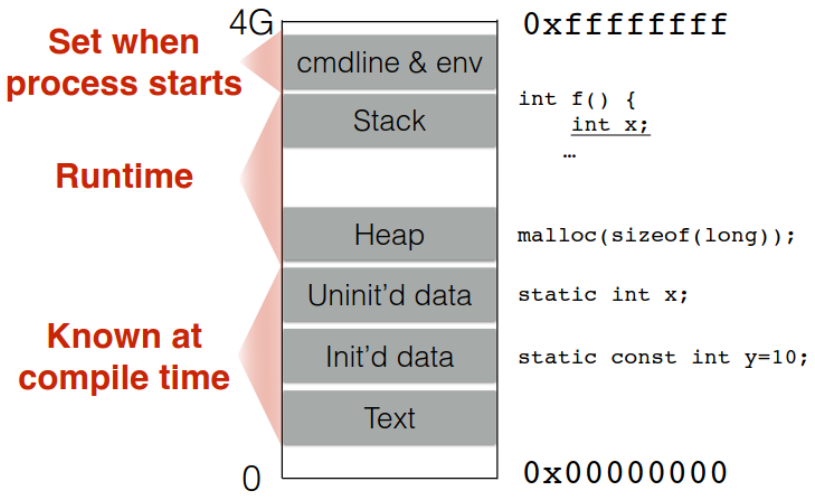
Il loader divide la memoria (tra quella adibita al programma, i 3GB di prima) in sezioni:

- **Text** per memorizzare il codice del programma
- Zone per dati inizializzati (**Data**) e non (Block Started by Symbol **BSS**)

Queste sono **conosciute a compile time**, mentre

- **Stack**
- **Heap**

Sono **dinamiche** e permettono la gestione del programma.



Stack e Heap quindi sono zone dinamiche che **crescono in direzioni opposte** (stack verso il basso).
Nell'heap ci sono le allocazioni dinamiche effettuate dal programmatore stesso (`malloc` e simili), mentre lo **stack** viene **gestito dal compilatore** per memorizzare cose come le chiamate a funzione.

1.1.1 Stack

All'interno dello stack viene gestita l'esecuzione del programma. Gli **indirizzi dello stack crescono verso il basso**, partendo da `0xbfffffff` e scendono. Questo rende possibile l'attacco di buffer overflow stack based, nel modo attualmente esistente.

Man mano che viene allocata memoria, lo stack alloca spazio dall'alto verso il basso. `Push` decrementa il valore dell'indirizzo, `Pop` lo aumenta.

Stack Pointer: Su architetture Intel, si tratta del **registro** che tiene conto dell'**indirizzo a cui è arrivato lo stack**, il valore dello spazio allocato più in basso (ultimo valore allocato, da dove posso ricominciare ad allocare).

Il compilatore utilizza lo stack quando vengono chiamate le funzioni, **nel momento in cui avviene una chiamata a funzione**:

- viene effettuata la `push` (istruzione macchina) dei parametri della funzione sullo stack
- l'istruzione macchina `call` viene chiamata, portando l'esecuzione all'indirizzo di memoria del codice della funzione
- la `call` effettua anche la `push` sullo stack dell'indirizzo di ritorno di una funzione, ovvero da dove proseguire l'esecuzione al termine della funzione

Dopo queste istruzioni comincia l'esecuzione della funzione. All'**interno dello stack** vengono **memorizzate le variabili locali**, quindi viene effettuata una `push` di queste variabili all'interno dello stack.

Al termine dell'esecuzione ci sarà un'istruzione `ret` che fa tornare l'**esecuzione all'indirizzo puntato dal return address** memorizzato in precedenza sullo stack (anche senza `return` esplicito, serve a continuare l'esecuzione del programma dopo la funzione).

La funzione di `ret`:

- libera la zona dedicata alle variabili locali, `pop` di tutte le variabili memorizzate sullo stack
- carica nell'instruction pointer (o program counter, registro che tiene traccia dell'istruzione da eseguire) il valore del return address (indirizzo della prossima istruzione che deve eseguire il processore)

Bisogna deallocare anche i parametri allocati sullo stack ma chi lo effettua dipende dalla calling convention del compilatore, quindi può farlo il chiamato o il chiamante (i.e., il pop di quei valori verrà effettuato prima o dopo il `ret`).

In ordine, dall'alto verso il basso, all'interno dello stack saranno presenti:

- Parametri
- Return address
- Variabili locali

1.2 Stack-based Overflow

Esempio di bug:

```
1 void f (par){
2         char buf[10];
3         strcpy(buf, par);
4 }
```

La funzione **non controlla dimensioni di sorgente e destinazione**, quindi cosa succede se l'**elemento da copiare è più grande** della **memoria** che gli è stata **allocata** (ovvero la dimensione del buffer destinazione)?

Lo **stack** sarà **composto da**:

- parametri della funzione, `par` in questo caso
- return address
- variabili locali, qui solo il buffer destinazione

Se la dimensione del buffer da copiare è maggiore del buffer allocato il programma **andrà a sovrascrivere i valori precedenti nello stack** (lo stack alloca dall'alto verso il basso, ma gli indirizzi del buffer vanno dal basso verso l'alto, l'indice 1 è più in basso dell'indice 8, per mantenere coerente l'aritmetica con i puntori, I guess).

Il valore sopra il buffer nello stack è il return address, che porterà a tornare ad un indirizzo casuale se sovrascritto, portando ad un `segfault`.

Non c'è un controllo che limiti la scrittura alla dimensione del buffer, portando a sovrascrivere altre parti dello stack.

1.2.1 Code Injection

Come possiamo sfruttare questa situazione? Tramite buffer overflow posso avere il controllo sul return address; control flow hijacking.

Per arrivare ad **eseguire codice arbitrario** devo

- definire il codice
- iniettarlo in memoria
- cambiare il valore del return address in modo che punti a quella zona di memoria

Definire il codice: Il processore legge solamente stringhe di byte che corrispondono alle istruzioni da eseguire. Noi dobbiamo costruire un bytestream a partire da del codice sorgente che vogliamo eseguire. Un bytestream che chiama `/bin/bash` diventa uno shellcode.
Dobbiamo forgiare un bytestream adatto alle nostre esigenze, manualmente prendendolo da del codice eseguito o tramite tool appositi (più facile solitamente).

Injection Vector: Per metterlo in memoria, il posto ideale sarebbe il **buffer** che abbiamo **già a disposizione**. L'**input** viene **copiato nel buffer**, il quale è sullo stack. Quindi inserendo il bytestream all'interno dell'input posso inserirlo in memoria all'interno del buffer.

Dobbiamo **costruire un input** che fa partire l'esecuzione del codice voluto (chiamato injection vector). Sarà quindi composto dal **bytestream del codice** (e.g., shellcode) e dal **valore che sovrascriverà il return address**, ovvero l'indirizzo del buffer (come trovarlo?).
Nell'esempio sopra ci saranno 10 byte per lo shellcode (dimensione allocata per il buffer) e 4 byte per il return address (considerando architetture a 32 bit).
In questo modo, quando il programma torna dalla funzione, porrà nel program counter l'indirizzo del buffer, contenente il bytestream forgiato da noi. La sequenza di istruzioni posta nel buffer sarà quindi interpretata come codice. Abbiamo dirottato il control flow, portando all'esecuzione di codice arbitrario.

Spatial memory error: Stiamo "mischiando" dati dell'utente e control channel (comandi di controllo), problematica comune a più vulnerabilità e possibili ambiti. Sovrascriviamo nello spazio dei caratteri di controllo. L'esecuzione di codice arbitrario avviene al ritorno della funzione vulnerabile.

Adesso ci sono protezioni che bloccano esecuzione di codice all'interno dello stack, non è una zona di memoria che dovrebbe contenere codice (anche se ci sono anche casi particolari) e l'esecuzione di codice presente in zone di memoria simili è bloccata.