Tecniche di Protezione del Software

Massimo Perego

Contents

1 Spatial Memory Errors

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.

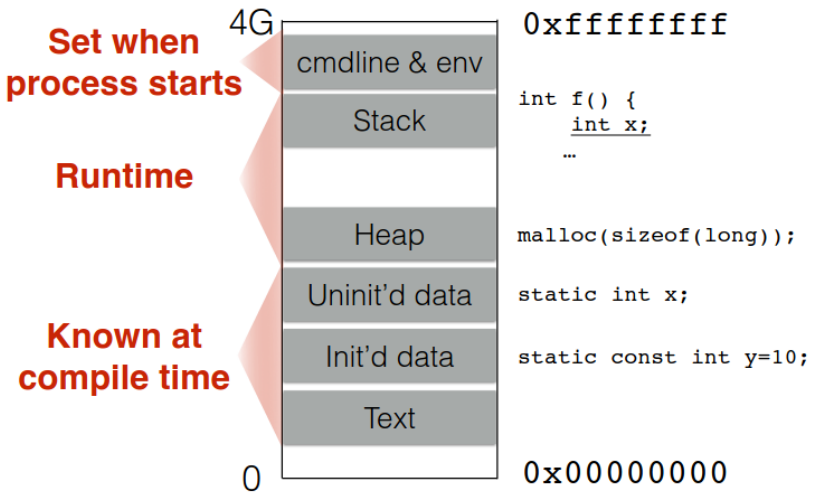
Il loader divide la memoria (tra quella adibita al programma, i 3GB di prima) in sezioni:

- **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.

1.3 Heap

1.3.1 Heap vs Stack

Lo **stack** è principalmente gestito dal compilatore per allocazioni **statiche** della memoria, conosciute a compile time. Inoltre tutti i metadati utili al programma sono memorizzati sullo stack. Un esempio di metadato è il return address per il ritorno di una funzione.

Questo è solitamente nascosto al programmatore, se la dimensione dei dati non è nota a priori viene usato l'**heap**, una memoria comandata (allocazione e liberazione) dal programmatore tramite funzioni di libreria. Generalmente più lenta ed a **gestione manuale**. Solitamente usato per oggetti, structs ed in generale elementi più grandi.

Lo stack cresce dell'alto verso il basso (indirizzi), mentre l'heap cresce dal basso verso l'alto. Una `push` sullo stack sposta il `rsp` verso il basso (e, di conseguenza, la `pop` verso l'alto). In caso di un utilizzo troppo elevato di memoria si possono "incontrare" le due zone (hai finito la memoria).

La vulnerabilità già vista sullo stack nasce dal "mischiare" dati inseriti dall'utente con metadati che permettono di alterare il flusso di controllo del programma.

le due funzioni principali per gestire la memoria nell'heap sono:

- `malloc(size)`: restituisce un puntatore ad una zona di memoria con dimensione `size`
- `free(ptr)`: dato un puntatore, libera la zona di memoria associata

Anche nell'heap sono presenti metadati, e di conseguenza la possibilità di sovrascriverli, portando a possibili vulnerabilità.

Allocatori: Definiti nelle librerie di sistema, per gestire le zone di memoria allocate servono comunque dei metadati. L'allocatore gestisce i propri dati all'interno dell'heap stesso (problema di canale).

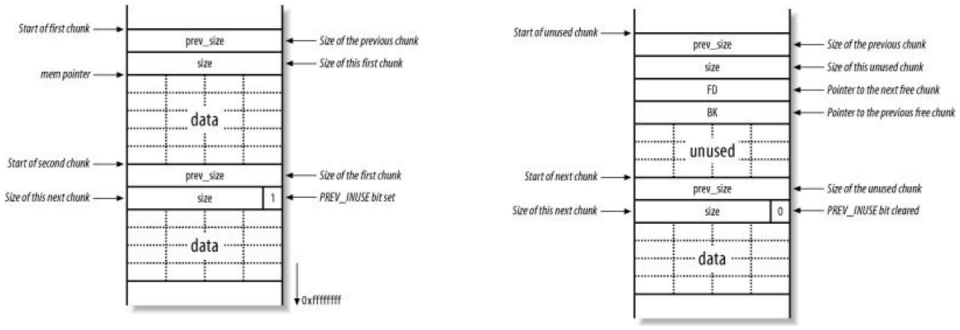
1.3.2 Heap Chunk

Heap Chunk: Struttura dati per la memoria nell'heap. Struttura:

```
struct malloc_chunk {
        INTERNAL_SIZE_T         prev_size;
        INTERNAL_SIZE_T         size;

        struct malloc_chunk*          fd;
        struct malloc_chunk*          bk;

        struct malloc_chunk*          fd_nextsize;
        struct malloc_chunk*          bk_nextsize;
        };
```

I ordine, i parametri sono:

- `prev_size`: dimensione della zona di memoria precedente, usato solo se il chunk è libero
- `size`: dimensione della zona di memoria occupata da questo chunk, overhead compreso
- `fd`, `bk`: puntatori alle zone di memoria precedenti e successive nella lista di chunk liberi (e di conseguenza presenti solo se il blocco è libero)
- `fd_nextsize`, `bk_nextsize`: puntatori alle dimensioni degli elementi adiacenti nella lista di chunk liberi (usati solo se il chunk è libero)



I chunk occupati contengono solo `size` e i dati. Il puntatore di ritorno ottenuto tramite `malloc` punta all'inizio della zona contenente i dati.

1.3.3 Allocare e deallocare memoria

Allocazione: All'inizio dell'esecuzione si ha un top chunk, che rappresenta tutta la grandezza dell'heap. Si ha un puntatore al top chunk chiamato `av_top`; inizialmente questo puntatore coincide con la base della memoria. In seguito ad una `malloc`, viene allocata la zona di memoria richiesta, restituendo i relativi puntatori, e di conseguenza `av_top` viene spostato "sopra" al blocco allocato, deve puntare sempre alla zona di memoria "rimanente"; il top chunk decresce in seguito all'allocazione.

Deallocazione: Oltre allo spazio libero del top chunk sono presenti delle liste di free chunk, una per ogni dimensione di chunk liberi (se posso occupare la memoria esatta lo faccio, serve a ridurre la frammentazione). Quando viene richiesta un'allocazione, prima cerca se c'è una zona di memoria "grande giusta" (o poco più, se mi servono 256 byte cerco nella lista di blocchi liberi da 256 byte), se non c'è una zona adatta nelle liste di chunk liberi allora prende la memoria dal top chunk.

Nel caso in cui due chunk adiacenti diventino liberi, vengono collassati in uno solo e lo inserisce nella lista rilevante.

Allocazione:

- Richiesta di memoria (`malloc`)
- Ricerca di un blocco libero (prima dalle liste, eventualmente si usa il top chunk)
- Aggiornamento della struttura di gestione (metadati)
- Restituzione del puntatore

Deallocazione:

- Richiesta di rilascio (`free`)
- Il blocco viene marcato come libero
- Coalescenza di blocchi adiacenti (se presenti, si fondono blocchi liberi adiacenti in uno solo)
- Aggiornamento della struttura di gestione

1.4 Heap Overflow

1.4.1 Unlink

Quando viene effettuata una allocazione bisogna aggiornare i puntatori, il nodo occupato va "sganciato" dalla lista (doppiamente concatenata) di blocchi liberi, e di conseguenza i puntatori del blocco successivo e precedente vanno aggiornati (sai come si rimuove un elemento da una lista dai). La procedura è

```
1 void unlink(malloc_chunk *P, malloc_chunk *BK,
2         malloc_chunk *FD) {
3                 FD = P->fd;
4                 BK = P->bk;
5                 FD->bk = BK;
6                 BK->fd = FD;
7 }
```

Dove:

- `P` nodo da eliminare
- `BK` nodo precedente
- `FD` nodo successivo

"Stacco" il nodo da eliminare facendo puntare il puntatore `bk` del nodo successivo al nodo precedente e viceversa.

1.4.2 Exploit "Naive"

Se non è presente nessun controllo durante le scritture, una write troppo grande può andare a sovrascrivere i chunk (liberi o occupati) superiori, metadati compresi.

Nello specifico, posso sovrascrivere il `FD->bk` e `P->bk`, rispettivamente con l'indirizzo di un return address e un indirizzo di un buffer (injection vector).

```
1        FD->bk = return address address;
2        FD = P->fd;
3        BK = P->bk = address of the buffer;
4        FD->bk = BK;
```

Partendo da un heap con dei chunk liberi e la relativa lista, facendo overflow in un chunk occupato sottostante si può sovrascrivere il puntatore `bk` di due chunk vuoti con indirizzi determinati dall'attaccante (rispettivamente, un blocco con inizio di un buffer con codice malevolo, il blocco dopo con l'indirizzo del return address di una funzione).

Quando il programma andrà ad allocare il primo dei blocchi con i valori sovrascritti dovrà rimuoverlo dalla lista di blocchi liberi, quindi eseguire la procedura di unlink: il `bk` del blocco "sganciato" punta al buffer contenente qualcosa (e.g., shellcode, inizio di un buffer controllato dall'attaccante, in qualsiasi modo), mentre il `bk` blocco successivo punta al return address di una funzione, il quale verrà sovrascritto con l'indirizzo del buffer (procedura di unlink), portando il programma ad eseguire il codice all'interno del buffer una volta che il programma dovrà seguire il return address sovrascritto.

Questo exploit è stato patchato controllando che `FD` e `BK` puntino effettivamente l'uno all'altro, non si può più sparare allo stack.

1.4.3 House of Force

Dopo la patch per la correzione dell'unlink sono nate nuove tecniche per sfruttare l'heap overflow, le principali si chiamano:

- The House of Prime
- The House of Mind
- **The House of Force**
- The House of Lore
- The House of Spirit
- The House of Chaos

Verrà mostrata solo la House of Force, spiegazioni ed esempi per le altre possono essere trovate `a questo indirizzo`. Ognuna di queste sfrutta diverse funzionalità dell'allocatore per attuare un attacco.

Tutte le tecniche citate hanno delle **condizioni per poter essere utilizzate**, la sola presenza della vulnerabilità non vuol dire che possa essere sfruttata.

Il **fuzzing** è il processo di trovare vulnerabilità nel programma, cercando crash del programma. Dopo aver trovato la vulnerabilità (i.e., il crash) bisogna analizzare se nel punto del programma che causa il crash ci sono le condizioni per un attacco vero e proprio, c'è da capire su che possibile vulnerabilità bisogna concentrarsi.

Un possibile ambito di ricerca sono gli AEG Automatic Exploit Generation: software per costruire in automatico attacchi a partire da possibili vulnerabilità (crash).

Esempio di programma vulnerabile ad House of Force:

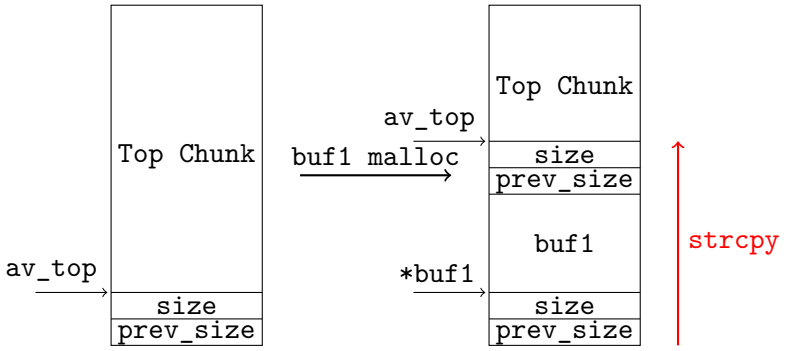
```
char *buf1, *buf2, *buf3;

buf1 = malloc(256);
strcpy(buf1, argv[1]);

buf2 = malloc(strtoul(argv[2], NULL, 16));
buf3 = malloc(256);
strcpy(buf3, argv[3]);

free(buf3);
free(buf2);
free(buf1);
```

Condizioni necessarie per questo exploit:

- avere una prima malloc su un numero arbitrario di byte (fisso, non importante, `buf1` nell'esempio)
- avere una `strcpy()` sul buffer precedente (`buf1`)
- avere un'altra `malloc` comandata dall'attaccante, i.e., la cui dimensione è definita in modo dinamico tramite input, in qualche modo (`buf2`)
- avere un'altra `malloc` non comandata dall'attaccante, in cui l'attaccante può scrivere (anche in modo controllato, basta poter scrivere)

Al termine del programma ci saranno le free.

Cosa succede sull'heap quando queste tre condizioni sono soddisfatte?



Quindi possiamo arrivare a sovrascrivere la `size` del top chunk, permettendo di dire al programma quanto spazio libero rimane all'interno. Questa è il primo problema: metadato size del top chunk modificabile dall'attaccante. A questo punto posso "incrementare" la memoria dell'heap (dimensione di `size`) fino a **tutta la memoria del processo indirizzabile**, la `size` è determinata unicamente in modo software. Posso andare ad indirizzare nell'intera memoria del processo.

Una volta "allargato tutto", l'attaccante può andare in un punto arbitrario della memoria da sovrascrivere. L'heap adesso include stack, .text ed in generale tutta la memoria. Posso sovrascrivere una zona di memoria arbitraria con un qualsiasi valore. Ad esempio, sovrascrivendo un `return address` sullo stack.

Abbiamo ingannato l'allocatore per **includere lo stack nello spazio indirizzabile dall'heap**, permettendoci di sovrascrivere una zona di memoria arbitraria.

La seconda `malloc` ha un'ampiezza comandata dall'attaccante, il che ci permette di **raggiungere la base dello stack**, dove c'è il `return address` che si vuole sovrascrivere (questa è una dimensione variabile, Δ tra posizione `dell'av_top` e posizione del `return address`, da calcolare).

Dopo una `malloc` di dimensione Δ, l'`av_top` sarà nello stack, la terza allocazione di dimensione n (fissa), partirà dall'`av_top` (in questo momento stiamo presupponendo non ci siano liste libere, altrimenti bisogna stare un po' più attenti) e quando comincio a scrivere in quest'ultimo buffer (ultima `strcpy`, comandata dall'attaccante) starò sovrascrivendo il `return address`, magari con un indirizzo di una zona di memoria il cui contenuto è controllato dall'attaccante (anche nell'heap, generalmente la zona dati non è eseguibile, ma ci sono dei casi in cui potrebbe esserlo).

TL;DR: Inganniamo l'allocatore nel pensare che il top chunk sia più grande di quanto non sia realmente, tramite overflow, per poi andare a sovrascrivere una zona di memoria arbitraria in una scrittura successiva.

2 Temporal Memory Errors

Le vulnerabilità viste fin'ora si basavano su un "problema" nello spazio relativo alla memoria (overflow di qualcosa), invece, le vulnerabilità basate sul rompere la temporal memory si focalizzano su una sequenza di esecuzione in ordine temporale, la **vulnerabilità si presenta in un certo istante di esecuzione** del programma, ovvero in uno stato specifico in cui il programma si trova.

Sono difficili da individuare con una revisione manuale del codice dato che serve conoscere la sequenza esatta di allocazione e deallocazione durante l'esecuzione del programma, difficilmente individuabile su codice complesso.

2.1 Use After Free UAF

Una Use-After-Free accade quando si **usa un puntatore che è stato precedentemente liberato** (dereferenziazione di un puntatore che punta ad una zona di memoria liberata, dangling pointer).

Esempio:

```
1  char *a , *b; int i;
2
3  a = malloc(16) ;
4  b = a + 5;
5  free(a) ;
6
7  b[2] = 'c' ; /* use after free */
8  b = retptr( ) ;
9  *b = 'c' ; /* use after free */
```

Conoscere l'insieme di puntatori che puntano ad un oggetto, senza una struttura dati come il garbage collector, non è un problema semplice (**aliasing** problem). Un analizzatore statico solitamente non riesce a trovare tutti gli aliasing, inoltre possono esserci puntatori definiti dinamicamente sulla base dell'oggetto.

Per avere una UAF devo individuare:

- una allocazione
- una deallocazione
- una dereferenziazione su un qualcosa di deallocato

Dopo averla individuata bisogna sfruttarla, tramite la funzione dell'allocatore che effettua la ricerca nelle liste di blocchi liberi un elemento della esatta grandezza richiesta, i.e., se la grandezza è giusta, riallocherà la zona allocata in precedenza.

Se c'è la possibilità, dopo la deallocazione, di eseguire una `malloc` di dimensione e valori controllati, il dangling pointer precedente punterà ai dati scritti dall'attaccante (quelli che sono rimasti nella zona deallocata e successivamente re-allocata).

La UAF è la vulnerabilità più frequente *in natura*, più difficile da scovare e spesso permette di arrivare a esecuzione di codice arbitrario.

3 Memory Safety and Type Safety

Tutti gli errori descritti nelle sezioni precedenti nascono da un **utilizzo errato della memoria**, nei linguaggi che lo permettono. Utilizzare **linguaggi memory safe** permetterebbe di evitare le problematiche descritte precedentemente, ma il degrado delle performance potrebbe non essere accettabile per alcuni casi d'uso.
Memory safety e type safety sono due proprietà intrinseche ad alcuni linguaggi di programmazione che permettono di bloccare la possibilità di effettuare gli attacchi descritti in precedenza.

Le **difese** possono essere a **livello** di:

- **compilatore**: come le canary, usate per rilevare una eventuale sovrascrittura del return address
- **sistema operativo**: come il Address Space Layout Randomization ASLR, che impedisce di sapere le posizioni esatte dei valori in memoria
- **architetturale**: componenti hardware che permettono di bloccare gli attacchi

Per contrastare ASLR sono nati gli attacchi Return Oriented Programming ROP, e per contrastare questi esiste la Control Flow Integrity CFI (controllo sulle transizioni del programma, se va "fuori dagli schemi" la transizione è bloccata, come ad esempio cambiando il return address).

3.1 Memory Safety

Sicurezza della memoria, si tratta di una proprietà fondamentale di alcuni linguaggi (detti memory safe). Un programma scritto in un linguaggio memory safe:

- Permette di creare puntatori solo attraverso alcuni mezzi standard; si vogliono intercettare i punti di creazione dei puntatori
- Permette di usare puntatori solo per accedere a memoria che "appartiene" a quel puntatore; un puntatore che appartiene ad una zona di memoria deve accedere effettivamente a quella zona di memoria

Combina le idee di temporal safety (accedere solo a memoria attualmente allocata/valida) e spatial safety (accedere solo a memoria valida).

3.1.1 Spatial safety

Ci permette di far valere la proprietà di sicurezza spaziale della memoria, ovvero andare a vedere che un puntatore non vada a puntare oltre una zona di memoria stabilita.

Fat Pointers: Un puntatore non è più "solo un puntatore", ma diventa una tripla (p, b, e) dove:

- p è il puntatore effettivo
- b è la base della zona di memoria a cui può accedere
- e è l'estensione/limite della zona a cui può accedere

L'accesso è permesso se e solo se

$$b \leq p \leq e - \texttt{sizeof(typeof(}\ p\ \texttt{))}$$

Ogni puntatore ha un tipo, quindi "quanto si può spostare" è determinato anche dalla dimensione del tipo puntato (se è su int di 4 byte non posso puntare al penultimo byte, andrei fuori per gli ultimi 3). L'aritmetica sui puntatori modifica solo p, senza toccare b ed e; p si sposta, gli altri due rimangono lì a stabilire i limiti. Deve sempre valere la disuguaglianza.

I Fat Pointers sono puntatori che hanno dati aggiuntivi, come la tripla descritta in precedenza; metadati aggiunti ai puntatori. Effettuare il controllo **ad ogni accesso** degrada le performance (soprattutto quando non è l'unico controllo da effettuare, questo è solo per la spatial safety). Il controllo è oneroso ed è necessaria una memoria aggiuntiva per i metadati legati ad ogni puntatore.
Il compilatore di un linguaggio memory safe deve instanziare memoria aggiuntiva per ogni puntatore e aggiungere il codice per i controlli, ad ogni accesso.

Low Fat Pointers: Si tratta di una variante che vuole ridurre l'overhead in termini di spazio e prestazioni. L'idea è quella di "codificare" all'interno dell'indirizzo stesso i metadati, la rappresentazione nativa del puntatore può inglobare i metadati. La disposizione della memoria viene usata per ricavare le informazioni prima salvate nei metadati in tempo costante.

La memoria virtuale viene divisa in "regioni", ciascuna riservata ad oggetti di dimensioni simili. Questo consente di dedurre la dimensione di un oggetto basandosi solo sull'indirizzo del puntatore (se appartiene a quella regione deve avere una certa dimensione).

3.1.2 Temporal Safety

Le regioni di memoria possono essere:

- **definite**: allocate e attive
- **non definite**: non inizializzate, non allocate o deallocate

Quando un puntatore punta ad una zona di memoria non definita è un problema. Per evitare errori serve tener traccia di dove un puntatore punta all'interno delle regioni di memoria. Dobbiamo evitare dangling pointers. Esempio:

```
1 p = malloc(4)
2 s = p
3 free(p)
```

`s` rimane dangling.

Serve una tabella di memoria per ogni puntatore che punta a tale zone. Quando la zona viene deallocata/diventa non definita, tutti i puntatori che fanno riferimento a quella zona devono essere resi non più validi. Nell'esempio precedente, `s` dovrebbe essere messo a `NULL` dopo la deallocazione. Questo richiede un controllo sulla validità del puntatore ad ogni dereferenziazione.

La combinazione di spatial e temporal safety si chiama **memory safety**. Il modo più semplice per ottenerla è utilizzare un linguaggio memory safe. C/C++ non sono memory safe, ma permettono di scrivere codice memory

safe, il problema è che *non ci sono garanzie*. Il compilatore potrebbe aggiungere codice per controllare le violazioni, ma rimane sempre il problema dell'inevitabile degrado delle performance (bisogna solo stabilire quanto e se questo è accettabile).

3.2 Type Safety

La type safety ha lo scopo di definire su che tipo di dato sono ammissibili quali operazioni, riducendo così le possibili problematiche durante l'esecuzione del programma.

Ogni oggetto ha un **tipo associato** (`int`, `int pointer`, `float`, …). Una volta determinati i tipi, posso decidere **quali operazioni** sono ammissibili per **quali tipi**. Le operazioni fatte sugli oggetti devono *sempre* essere sempre compatibili con il tipo dell'oggetto, evitando errori, anche run-time.

In generale la type safety è *più forte della memory safety*. Esempio memory safe ma non type safe:

```
1 int (*cmp) (char*,char*);
2 int *p = (int*) malloc(sizeof(int));
3 *p = 1;
4 cmp = (int (*)(char*,char*)) p; // Memory safe, not Type safe
5 cmp("hello","bye"); // crash!
```

In questo caso è memory safe in quanto abbiamo messo un valore "entro i limiti" all'interno del puntatore a funzione `cmp`, ma sta tentando di mettere un intero all'interno di un `type address` (al posto dell'indirizzo della funzione ho `1`), quindi non è type safe (quindi crasha, `SEGFAULT`). Se il tipo dei due valori è disallineato la type safety **nega l'assegnamento**.

C/C++ implementano tipi primitivi ma non c'è nessun controllo su *cosa viene assegnato a cosa*. Effettuare questi controlli può essere oneroso, vanno fatti su **ogni operazione** tra dati. In breve, la type safety costa performance, quindi, anche se ci sono soluzioni, non sempre hanno overhead accettabile.

Dynamically Typed Languages: All'interno dei linguaggi dynamically typed, tutti gli oggetti hanno **un solo tipo: dinamico**. Ogni operazione su un oggetto di tipo dinamico è permessa, ma tale operazione potrebbe non essere implementata, portando ad un'eccezione. Tutto è permesso, ma se run-time il tipo effettivo non implementa l'operazione viene sollevata un'eccezione.

Enforce Invariants: Gli invarianti sono delle **formule logiche** per garantire determinate proprietà sull'esecuzione di dei pezzi di codice, rimane costantemente vero in certi punti del programma. Possono essere fatti valere tramite type safety. Una proprietà che deve rimanere sempre vera durante l'esecuzione del programma.

Types for security: Gli invarianti possono essere usati anche per la sicurezza, generalmente riguardano il controllo del flusso di dati, prevenendo errori logici. Tale controllo del flusso di dati, anche all'interno dello stesso programma, permette di non "*far uscire*" un dato da determinate zone.

Esempio: Java with Information Flow (JIF), estensione di Java

```
1 int{Alice -> Bob} x;
2 int{Alice -> Bob, Chuck} y;
3 x = y;          //OK: policy on x is stronger
4 y = x;          //BAD: policy on y is not as strong as x
```

3.3 Avoiding Exploitation: Other strategies

Sapendo che ogni attacco ha determinati prerequisiti, un modo per prevenirli è fare in modo che le condizioni non si possano presentare. Questo aumenta la complessità dell'attacco, rendendo l'exploit più difficile da sfruttare.

Quindi, tento di evitare bug, ma aggiungo protezioni nel caso qualcosa sfugga. Per evitare i bug esistono secure coding practices e tecniche di code review avanzate, come program analysis, fuzzing, ….

Per evitare l'exploitation, quali sono le fasi di un attacco di stack smashing?

- scrivere il codice dell'attaccante in una zona di memoria
- fare in modo che `%eip` esegua il codice dell'attaccante
- trovare il return address

Vogliamo inibire una di queste fasi.

Come si possono rendere più difficili questi attacchi? Il caso migliore è **modificare librerie, compilatore e/o sistema operativo**, in modo tale da non dover cambiare il codice dell'applicazione ma avere una soluzione a **livello architetturale**.

Canary: Per inibire la fase di overflow, ci si è ispirati ai canarini usati dalle miniere: se il canarino muore c'è gas. Possiamo fare una cosa simile per lo stack, scriviamo un valore prima del return address e se al termine dell'esecuzione della funzione (prima di fare il `ret`) il valore è non è quello definito all'inizio c'è stato un tentativo di stack smashing. Il valore originale va salvato in una zona di memoria safe, e read-only.

Come viene scelto il valore della canary:

- terminator canaries (`CR`, `LF`, `NUL`, `-1`): valori non ammessi dallo `scanf`, l'attaccante non li può inviare come input;
- numero random, scelto ad ogni inizio del processo;

- Random `XOR` canaries: si sceglie un valore random, ma il return address diventa `ret` := `ret` $\oplus$ `canary`, tornando al valore originale al termine della funzione, "sabotando" un eventuale indirizzo di ritorno sovrascritto in quanto tornerà ad un valore casuale al posto che all'indirizzo segnato. Permette di risparmiare sullo stack lo spazio della canary.

Data Execution Prevention DEP: La seconda fase dell'attacco è scrivere in memoria il codice ed eseguirlo. Per evitare che codice dell'attaccante possa essere eseguito si possono **rendere alcune zone di memoria**, come stack e heap, **non eseguibili**. In questo modo, se anche viene bypassata la canary, il programma va in panico prima di eseguire codice posizionato nello stack/heap. Nelle zone solo dati non si può eseguire codice (generalmente, esistono casi particolari). Si chiama Data Execution Prevention, non posso iniettare codice eseguibile.

`Return-to-libc:` Metodo trovato per evadere la DEP, l'idea è inserire nel return address funzioni presenti all'interno della libreria di sistema, le quali sono ovviamente eseguibili. L'injection vector sovrascrive il return address con l'indirizzo di una funzione di sistema, preparando lo stack con i parametri corretti per l'esecuzione di tale funzione. Modifico l'esecuzione del codice senza iniettarlo.

Address Space Layout Randomization ASLR: Randomizzare il layout di memoria, ogni volta che il processo va in memoria vengono usate zone di memoria differenti, ho un sacco di spazio, metto dove voglio il programma. In questo modo i valori esatti degli indirizzi di memoria sono sconosciuti, permette di inibire l'ultima fase di sovrascrittura del return address: se non so dove sia il mio codice non so dove far puntare il return address. Permette anche di evitare gli attacchi come `return-to-libc` randomizzando la posizione delle librerie di sistema.

Esistono diverse implementazione, ma in generale bisogna notare che:

- sposta solo l'offset delle zone di memoria, non le posizioni relative all'interno di essere
- potrebbe essere applicato solo alle librerie (sempre position indipendent), non al codice del programma (potrebbero esserci riferimenti

"statici" a zone del programma)

- servono abbastanza bit random, altrimenti si può fare brute-force (su architettura 32 bit non è sicuro, su 64 è già molto meglio)

4 Return Oriented Programming ROP

Prevenzione ed attacchi si "inseguono" sempre:

- **Difesa**: stack/heap non eseguibile per prevenire iniezione di codice; **attacco**: jump/return to libc
- **Difesa**: nascondere gli indirizzi di memoria e return address con ASLR; **attacco**: ricerca brute force (32 bit) o information leak (format string)
- **Difesa**: non usare codice in libc ma usare solo codice all'interno del text del programma, si riducono le funzioni di libreria legate all'utilizzo del processo stesso, vengono caricate solo le parti necessarie; **attacco**: costruire le funzionalità che servono tramite Return Oriented Programming ROP

Il Return Oriented Programming nasce dall'esigenza di limitare il numero di funzioni che un processo usa per la propria esecuzione. Introdotta per la prima volta nel 2007 da Hovav Shacham (*`The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls`*).
L'idea di base è: prendo pezzi di codice dalle funzioni presenti in memoria e le unisco per "costruire" un attacco, ovvero il programma voluto dall'attaccante. Si può dimostrare che, con una codebase sufficiente, gli elementi costruibili sono Turing-compatibili. I pezzi di codice usati si chiamano **gadget**. Si tratta di una nuova tecnica per eseguire codice sfruttando una vulnerabilità (sempre buffer overflow).

Le "sfide" di questo attacco sono

- trovare i gadget
- collegarli

Ma *cosa sono i gadget*? Si tratta di sequenze di istruzioni (assembly) che terminano con un `ret`. Si "trasforma" il processo di esecuzione (il programma diventa una `weird machine`), lo stack diventa il "codice" per l'attaccante; non si può iniettare codice all'interno dello stack ma nella weird machine descritta:

- `%esp` diventa (una sorta di) program counter
- i gadget sono invocati tramite una `ret` (si parte dalla prima, quella sovrascritta da un buffer overflow per esempio)

- i gadget hanno parametri, passati tramite lo stack, quindi tramite pop, …

Si ha una trasformazione della memoria del programma. L'idea è

- prendere il codice che vogliamo eseguire
- emulare l'assembly tramite i gadget

Concateno gadget trovati in memoria (pezzi di codice terminati da ret) per simulare il comportamento voluto. L'overflow del buffer termina sovrascrivendo l'indirizzo del primo gadget in memoria.

Esempio: volendo simulare il codice

```
mov %edx, 5
```

Avendo come gadget

```
pop %edx
ret
```

E il valore 5 sullo stack, possiamo fare in modo che il valore puntato da `%esp` sarà caricato in `%edx` ed `%esp` spostato sopra. Al termine di ogni gadget si ha il `ret`, fondamentale per concatenare i gadget. Cosa fa una `ret`? Prende il primo valore sullo stack (`pop` di cosa punta `%esp`) e continua da lì (codice puntato) l'esecuzione del programma. Quindi se sullo stack è presente il codice del prossimo gadget, l'esecuzione proseguirà da lì.
In altre parole: cambio l'indirizzo di ritorno con l'indirizzo del primo gadget, sullo stack ci sarà la chain di return address di gadget e parametri usati dagli stessi, ogni volta che ne viene eseguito uno (a partire dal primo), lo stack pointer scende e trova quello dopo.
Esistono tool automatici per automatizzare la ricerca e unione dei gadget, chiamati ROP Compiler.

Sequenza di codice	Equivalente ROP

`0x17f: mov %eax, [%esp]`
`mov %ebx, [%esp+8]`
`mov [%ebx], %eax`
	`0x17f: pop %eax`
	`ret`
	`…`
	`0x20f: pop %ebx`
	`ret`
	`…`
	`0x21a: mov [%ebx], %eax`

Sostanzialmente, nello stack saranno presenti:

- indirizzo del gadget
- parametro/i del gadget
- indirizzo del gadget
- …

Quindi devo sovrascrivere lo stack con un layout di questo tipo.

Trovare i gadget: I gadget per costruire un exploit possono essere trovati con una ricerca automatica del binario (cercando `ret` e andando a ritroso, `esempio di ROP gadget finder`).

Ma quanti gadget sono presenti? Ogni programma, in architettura Intel, può essere visto come n rappresentazioni diverse: dato che è un'architettura CISC le operazioni possono avere diverse lunghezze, quindi saltare in mezzo ad un'istruzione porta ad una codifica diversa. Esempio: se l'istruzione `a` ha opcode `0x0a0b` e l'istruzione `b` ha opcode più breve `0x0b`, "saltando" il primo byte dell'istruzione `a` ho effettivamente trovato un'istruzione `b`. In questo modo posso trovare `ret` o qualunque cosa in maniera più semplice. Diventa più difficile su architetture RISC (tutti i byte di istruzione sono allineati).

I gadget sono sempre sufficienti per portare avanti un attacco? Sì, Shacham ha provato che per code base non triviali (e.g., libc), i gadget sono Turing completi.

Un ROP Compiler prende in input:

- codice che voglio eseguire ad alto livello
- programma vittima

E restituisce in output l'injection vector, ovvero il layout dello stack per effettuare l'attacco (eseguire il codice input). Quindi prende il programma, definisce che gadget servono, li trova nel programma e crea il layout dello stack.

4.1 Blind ROP

Si tratta di un attacco pubblicato da stanford (`qui la pagina`) che mostra come il ROP si possa applicare in condizioni reali. Il contesto è:

- remoto, si tratta di un server ngix
- l'attaccante non ha il binario (programma eseguibile), nè il source code
- ASLR, canary, DEP attivi
- conoscenza di una vulnerabilità, da qualche parte

L'idea è: su un eseguibile PIE a 64 bit in esecuzione su un server, se in seguito ad un crash il server riparte ma non ri-randomizza i valori posso:

- leakare canary e return address dallo stack
- trovare gadget (run-time) per leakare il binario
- trovare i gadget per la shellcode

Lo scopo del programma è andare a leakare il binario, ovvero usare un gadget che va in memoria, prende il programma e lo restituisce all'attaccante. Per fare questo bisogna far fare al server una write su una socket `sd`, con buffer e relativa lunghezza, dove

- il buffer sarà l'indirizzo del programma, sconosciuto per ASLR
- la lunghezza sarà la dimensione dell'applicazione, approssimativamente nota

Dato che ad ogni connessione il server mantiene gli stessi valori (nuovo thread, stessi dati, fork), anche se poi il thread crasha, le fasi sono:

- leakare la canary
- defeat the ASLR
- trovare blind i gadget per fare una write
- ottenere il binario

Leakare la canary: Trovo la dimensione del buffer (tirando ad indovinare) e poi provo a sovrascrivere un byte della canary, provando tutti i valori finché non crasha per poi passare al byte successivo.

Defeat the ASLR: Provo ad indovinare il return address, similmente a come fatto per la canary, bisogna scoprire *più o meno* dov'è il programma; una volta conosciuta la canary posso tirare ad indovinare finché non scopro uno degli indirizzi dov'è posizionato il programma.

Blind search: Dobbiamo trovare in modo blind i gadget che permettano di fare la write. Per una `write(sd, buffer, length)` le istruzioni che servono sono

```
pop rdi  //sd
pop rsi  //buffer
pop rdx  //length
pop rax  //write syscall num
syscall
```

Quindi servono i gadget per fare

```
pop rdi;  ret
pop rsi;  ret
pop rdx;  ret
pop rax;  ret
 syscall
```

Il `sd` si può indovinare (8 bit, facile), ma devo trovare questi gadget senza avere il binario. Per indovinare il gadget posso sfruttare la differenza di comportamento tra `pop` ed altre istruzioni: `pop` sposta il `esp`. Dopo aver trovato la canary si tira ad indovinare, sovrascrivendo il return address, per trovare due gadget:

- idle gadget: un indirizzo che mantiene aperta la socket
- stop gadget: un indirizzo che fa crashare la socket

I gadget reali si trovano sfruttando questi due: sullo stack metto, in ordine,

1. indirizzo del gadget cercato
2. stop gadget
3. idle gadget.

Provando ad eseguire, il gadget cercato può essere:

- un'istruzione normale (non `pop`): l'`esp` non viene spostato, il gadget eseguito dopo è lo stop e la socket crasha
- una `pop`: viene spostato l'`esp`, lo stop gadget non viene eseguito, la socket rimane aperta

In questo modo posso mappare i gadget in memoria a run-time, so dove ci sono le `pop`. Per capire su che registro viene fatta la `pop`: non lo capisco, si provano a caso le `pop` trovate e quando torna indietro qualcosa dalla socket ho beccato la combinazione giusta (ha fatto la write).

L'indirizzo per la write lo si trova tirando ad indovinare nella GOT.