Tecniche di Protezione del Software

Massimo Perego

Indice

1 Spatial Memory Errors **3**
1.1 Memory Layout . . . 3
1.1.1 Stack . . . 5
1.2 Stack-based Overflow . . . 7
1.2.1 Code Injection . . . 7
1.3 Heap . . . 9
1.3.1 Heap vs Stack . . . 9
1.3.2 Heap Chunk . . . 10
1.3.3 Allocare e deallocare memoria . . . 11
1.4 Heap Overflow . . . 12
1.4.1 Unlink . . . 12
1.4.2 Exploit "Naive" . . . 12
1.4.3 House of Force . . . 14

2 Temporal Memory Errors **17**
2.1 Use After Free UAF . . . 17

3 Memory Safety and Type Safety **19**
3.1 Memory Safety . . . 19
3.1.1 Spatial safety . . . 20
3.1.2 Temporal Safety . . . 22
3.2 Type Safety . . . 23
3.3 Avoiding Exploitation: Other strategies . . . 24

4 Return Oriented Programming ROP **28**
4.1 Blind ROP . . . 31

5 Control Flow Integrity CFI **34**
5.1 Costruzione del Control Flow . . . 36

5.2 In-Line Monitor 37

6 Vulnerabilities Detection **40**
6.1 Symbolic Execution 42
6.1.1 Concolic Execution 47
6.1.2 Search 48
6.1.3 SMT Solvers 50
6.1.4 Symbolic Execution Systems 51

7 Fuzz Testing **53**
7.1 Fuzzer Architecture 53
7.1.1 Instrumentation 56
7.1.2 Sanitizers 56

8 Meltdown and Spectre **59**
8.1 Attacchi alle cache 60
8.1.1 Flush & Reload 60
8.1.2 Prime & Probe 62
8.2 Speculative Execution 62
8.3 Meltdown 63
8.4 Spectre 64

1 Spatial Memory Errors

Ci si concentrerà su linguaggi low level (e.g., C), i quali tendono a crashare in caso di errori (come buffer overflow), allo stesso tempo permettendo (potenzialmente) a un attaccante di **sfruttare le vulnerabilità** per ottenere informazioni (e.g., Heartbleed, bug SSL che permetteva di leggere tutta la memoria del programma), corrompere memoria, fino ad arbitrary code execution (la macchina esegue altro, diventa una "weird machine"), …

Il crash (ovvero `segfault`), se analizzato, può portare a un attacco, anche se non è detto che tutti i casi siano exploitabili (ma molti sì).

Questo tipo di bug hanno una lunga storia e sono tutt'ora presenti, e lo saranno finché linguaggi come C e C++ rimarranno in uso.

Inoltre è utile studiare l'evoluzione del bug stesso, assieme alle difese create per contrastarlo. Alcune caratteristiche di un attacco/difesa possono essere ritrovate anche in altri attacchi.

Solitamente l'attacco è *molto* più semplice della difesa. Per l'attacco basta un punto debole, per la difesa bisogna assicurarsi 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 a una catena di exploit che permettono di compromettere macchine (a diversi livelli), sempre con un impatto significativo.

1.1 Memory Layout

Bisogna 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.

Verrà considerato il 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. Stiamo considerando 32 bit, ma i concetti possono essere estesi a 64 semplicemente con indirizzi più lunghi.

Linux divide:

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

Di conseguenza il primo indirizzo valido per il programma è `0xbfffffff`, al di sopra si trova il kernel.

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

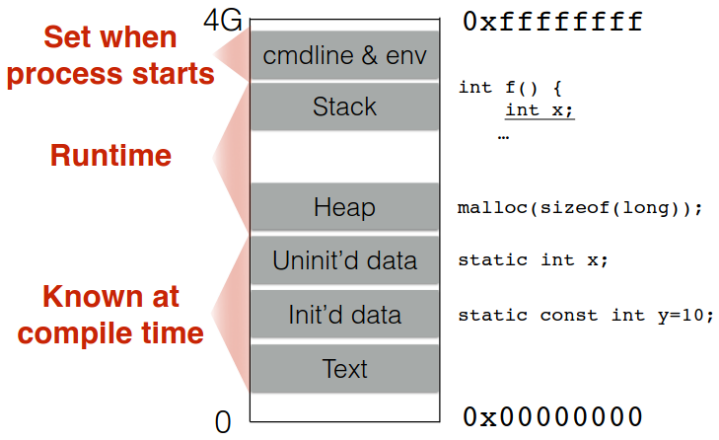
Il loader divide la memoria (tra quella adibita al programma, i 3GB di prima) in sezioni:

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

Queste hanno dimensione **nota a compile time**, mentre

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

Sono zone **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 si trovano le allocazioni dinamiche effettuate dal programmatore stesso (`malloc` e simili), mentre lo **stack** viene **gestito dal compilatore** per memorizzare elementi come le chiamate a funzione, variabili statiche, …

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`. Questa caratteristica rende possibile l'attacco di buffer overflow stack based, nel modello 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 (più recente, ultimo valore allocato, ovvero da dove si può 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 stessa.

All'**interno dello stack** vengono **memorizzate le variabili locali**: viene fatta 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, ovvero effettua una `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); fa un'ultima `pop` per ottenere dallo stack l'indirizzo a cui tornare

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 la `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 tra puntori, o almeno credo).

Il valore al di sopra (anche non immediatamente) del buffer nello stack è il **return address**: al termine della funzione, se sovrascritto, cambierà da dove il programma continua a eseguire, solitamente causando un `segfault` (se si tratta di un valore "casuale").

Senza un controllo che limiti la scrittura alla dimensione del buffer si possono sovrascrivere altre parti dello stack.

1.2.1 Code Injection

Come possiamo sfruttare questa situazione? Tramite buffer overflow si può ottenere il controllo sul return address, **control flow hijacking**.

Per arrivare a **eseguire codice arbitrario** è necessario

- 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. Bisogna quindi costruire un bytestream a partire da del codice sorgente da iniettare.

Ad esempio, un bytestream che chiama `/bin/bash` diventa uno `shellcode`.

Bisogna forgiare un bytestream, prendendolo manualmente da codice eseguito o tramite tool appositi (solitamente più facile).

Injection Vector: Per inserire il codice in memoria, il posto ideale sarebbe il **buffer già a disposizione**.

L'**input** viene **copiato nel buffer**, il quale è sullo stack, quindi inserendo il bytestream all'interno dell'input lo si può inserire nel buffer.

Vogliamo **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 precedente ci saranno 10 byte per lo shellcode (dimensione allocata per il buffer) e 4 byte per il return address (8 byte se `x64`).

In questo modo, quando il programma torna dalla funzione, porrà nel PC l'indirizzo del buffer, contenente il bytestream, il quale sarà interpretato come codice. Abbiamo dirottato il control flow, portando all'esecuzione di codice arbitrario.

Spatial memory error: Stiamo "mischiando" dati utente e control channel (comandi di controllo), problematica comune a più vulnerabilità e ambiti.

Sovrascriviamo nello spazio dei caratteri di controllo. L'esecuzione di codice arbitrario avviene al ritorno della funzione vulnerabile.

Adesso esistono protezioni che bloccano l'esecuzione di codice posto all'interno dello stack, non è una zona di memoria che dovrebbe mai contenere codice eseguibile (senza considerare casi particolari), quindi 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; anche tutti i metadati utili al programma sono memorizzati sullo stack, come ad esempio il return address.

Se la dimensione dei dati non è nota a priori viene usato l'**heap**, una memoria controllata (allocazione e deallocazione) dal programmatore tramite funzioni di libreria (i.e., `malloc()` e `free()`).

Generalmente più lenta e a **gestione manuale**. Solitamente usato per oggetti, structs e 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 (insomma, 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: possibile vulnerabilità.

Allocatori: Definiti nelle librerie di sistema, si occupano di gestire la memoria, allocando e liberando le zone di memoria, quando necessario.

Per gestire le zone di memoria allocate servono comunque dei metadati. Problema di canale: l'allocatore gestisce i propri dati all'interno dell'heap stesso.

1.3.2 Heap Chunk

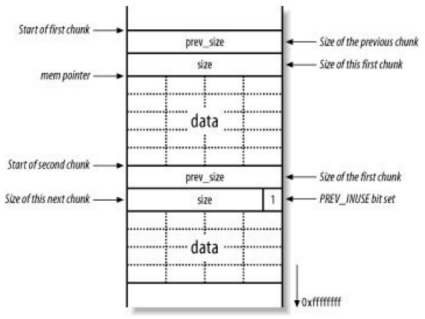
Un **Heap Chunk** è una struttura dati per la memoria all'interno dell'heap.

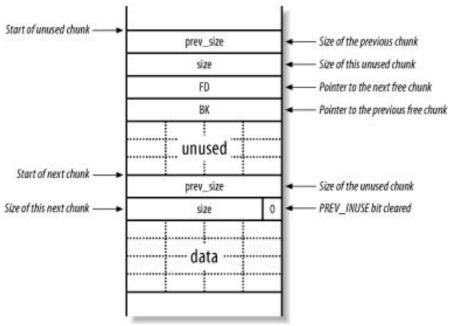
Struttura:

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

    struct malloc_chunk*    fd;
    struct malloc_chunk*    bk;

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

I ordine, i parametri sono:

- `size`: dimensione della zona di memoria occupata da questo chunk, overhead compreso
- `prev_size`: dimensione della zona di memoria precedente, usato solo se il chunk è libero
- `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)





Quindi 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, con un puntatore chiamato `av_top`. Inizialmente questo puntatore coincide con la base della memoria.

In seguito a una `malloc()`, viene allocata la zona di memoria richiesta, restituendo i relativi puntatori, e di conseguenza spostando `av_top` "sopra" al blocco allocato, punta 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, l'allocatore prima cerca se c'è una zona di memoria "grande giusta" (o poco più, se servono 256 byte cerca nella lista di blocchi liberi da 256 byte), se non c'è una zona adatta nelle liste di chunk liberi allora prende dalla memoria dal top chunk.

Nel caso in cui due chunk adiacenti diventino liberi, vengono collassati in uno solo per poi inserirlo 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 presenti all'interno dei blocchi liberi. 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

```
void unlink(malloc_chunk *C, malloc_chunk *prev_C,
    malloc_chunk *next_C) {
        next_C = C->fd;
        prev_C = C->bk;
        next_C->bk = prev_C;
        prev_C->fd = next_C;
}
```

Dove:

- `C` nodo da eliminare
- `prev_C` nodo precedente
- `next_C` 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 potrebbe andare a sovrascrivere i chunk (liberi od occupati) superiori, metadati compresi.

Nello specifico, si possono sovrascrivere `next_C->bk` e `C->bk`, con, rispettivamente, l'indirizzo di un return address e l'indirizzo di un buffer (injection vector).

```
next_C->bk  = return address;
next_C      = C->fd;
prev_C      = C->bk = buffer address;
next_C->bk  = prev_C;
```

Partendo da un heap con dei chunk liberi e la relativa lista, facendo overflow fino ad arrivare a un chunk occupato sottostante si può sovrascrivere il puntatore `bk` di due chunk vuoti con indirizzi determinati dall'attaccante, rispettivamente, un blocco conterrà l'inizio di un buffer con codice malevolo, il blocco successivo 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)
- Il `bk` del blocco successivo punta al return address di una funzione, il quale verrà sovrascritto con l'indirizzo del buffer (procedura di unlink)

Questo porta il programma a eseguire il codice all'interno del buffer una volta che il programma dovrà seguire il return address sovrascritto.

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

TL;DR: Dato che durante l'unlink il puntatore `bk` del blocco allocato (quello da rimuovere dalla lista di chunk liberi) va a sovrascrivere il puntatore `bk` del chunk successivo, inserendo l'indirizzo di un indirizzo contenente un buffer con codice malevolo e l'indirizzo del return address di una funzione (rispettivamente), la procedura di unlink sovrascriverà il return address con l'indirizzo del buffer, portando il programma a eseguire il codice malevolo quando dovrà uscire dalla funzione intaccata.

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.

Dopo aver trovato una vulnerabilità (i.e., il crash) bisogna analizzare se in quel punto del programma si possono presentare le condizioni per un attacco vero e proprio, per capire su che possibile vulnerabilità concentrarsi.

Esempio di programma vulnerabile ad House of Force:

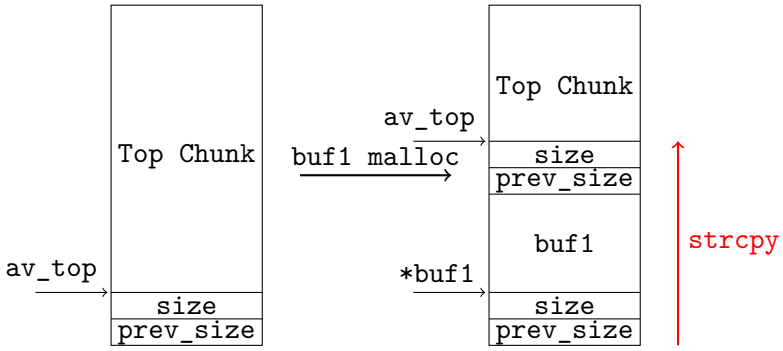
```
1  char *buf1, *buf2, *buf3;
2  // malloc arbitraria
3  buf1 = malloc(256);
4  // Scrittura controllata sulla malloc
5  strcpy(buf1, argv[1]);
6  // malloc di dimensione controllata
7  buf2 = malloc(strtoul(argv[2], NULL, 16));
8  // malloc in cui l'attaccante può scrivere
9  buf3 = malloc(256);
10 strcpy(buf3, argv[3]);
11 // rilascio delle risorse
12 free(buf3);
13 free(buf2);
14 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()` controllata dall'attaccante 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 `strcpy()` in seguito a una nuova `malloc()` non comandata dall'attaccante, all'interno della quale l'attaccante può scrivere (anche in modo controllato)

Al termine del programma ci saranno le `free()`.

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



Si può arrivare a sovrascrivere il parametro `size` del top chunk, permettendo di dire al programma la quantità di spazio libero rimanente.

Questo è il primo problema: metadato `size` del top chunk modificabile dall'attaccante.

A questo punto si può incrementare la memoria dell'heap (dimensione di `size`) fino a **tutta la memoria del processo indirizzabile**, la `size` è determinata unicamente in modo software. Si può andare a 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 e in generale tutta la memoria.

Si può 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 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).

Quando comincia a scrivere nell'ultimo buffer (ultima `strcpy()`, comandata dall'attaccante) starà 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.

Step:

- `malloc()` e `strcpy()` sull'allocazione precedente per sovrascrivere parametro `size` del top chunk (overflow), allargando l'heap a tutta la memoria indirizzabile
- La seconda `malloc()`, di lunghezza comandata, viene usata per raggiungere i dati da sovrascrivere alla base dello stack
- La terza `malloc()` tornerà l'indirizzo dell'`av_top`, quindi l'indirizzo da sovrascrivere nello stack
- L'ultima `strcpy()` permette di sovrascrivere i dati nello stack

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 sulla rottura della 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 a 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 a 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 bisogna 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.

Esempio di UAF:

```c
// 1) Alloca e inizializza
char *e = malloc(SIZE);
// ... Uso del puntatore

// 2) Libera la memoria
free(e);

// 3) Nel frattempo, alloca un buffer di uguale dimensione
// per "occupare" la stessa area
char *hijack = malloc(SIZE);
strcpy(hijack, "allowed");

// 4) Uso ancora il puntatore liberato
if (strcmp(e, "allowed") == 0){
    // privileged stuff
}
```

TL;DR: Se una zona viene allocata, deallocata e poi dereferenziata si potrebbe avere un undefined behavior.

L'allocatore potrebbe ri-usare il chunk deallocato quando viene fatta un'allocazione di dimensione pari a quella precedente (magari dimensioni controllate dall'attaccante), portando ad avere puntatori che puntano a dati possibilmente controllati dall'attaccante.

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 l'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 (vedi 4), e per contrastare questi esiste la Control Flow Integrity CFI (vedi 5, controllo sulle transizioni del programma, se va "fuori dagli schemi", ad esempio modificando il return address, la transizione è bloccata).

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 a 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

Permette di far valere la proprietà di **sicurezza spaziale della memoria**, ovvero controllare 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 per 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 a 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 e necessita memoria aggiuntiva per i metadati legati a ogni puntatore.

Il compilatore di un linguaggio memory safe deve istanziare memoria aggiuntiva, *per ogni puntatore*, e aggiungere il codice per i controlli, *a ogni accesso*.

Implementazione naive:

```c
typedef struct {
    int *ptr;
    size_t e;
} fint_ptr;

fint_ptr make_fint(size_t n) {
    fint_ptr f;
    f.ptr = malloc(n * sizeof(int));
    f.e = n;
    return f;
}

bool get_fint(fint_ptr f, size_t i, int *out) {
    if (i < 0 || i > f.e)
        return false;
    *out = *f.ptr + i;
    return true;
}

void free_fint(fint_ptr *f) {
    free(f->ptr);
    f->ptr = NULL;
    f->e = 0;
}
```

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, permettendo alla rappresentazione nativa del puntatore di inglobare i metadati stessi.

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 a 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).

Per implementarli, in generale:

- Viene riservata una grande zona di memoria virtuale all'inizio dell'esecuzione del programma, per poi dividerla in sotto-regioni, una per taglia di allocazione possibile
- Ogni dimensione richiesta è arrotondata a quella superiore più vicina e l'allocatore restituisce un puntatore appartenente alla zona relativa
- Ogni sezione è grande quanto una potenza di 2 (uguale per tutte le regione, definita), quindi guardando solo i primi n bit del puntatore (dove $2^n =$ `region_size`) si può ottenere la base della regione, quindi la dimensione
- Dalla dimensione si può controllare se l'accesso al puntatore è valido

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 a 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)
```

In questo caso `s` rimane dangling.

Serve una tabella di memoria per ogni puntatore che punta a tale zona. 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 a 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.*

Esempi memory safe ma non type safe:

```
int x = 69;
float y = 6.9;
printf("Valore: %d \n", *(int *)&x);
printf("Valore: %d \n", *(int *)&y); // Not type safe
```

Non causa "problemi" ma stampa valori senza senso.

Oppure

```
int (*cmp) (char*,char*);
int *p = (int*) malloc(sizeof(int));
*p = 1;
cmp = (int (*)(char*,char*)) p; // Memory safe, not Type safe
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 a un'eccezione. Tutto è permesso, ma se a 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, i quali rimangono costantemente veri 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

```
int{Alice -> Bob} x;
int{Alice -> Bob, Chuck} y;
x = y;  //OK: policy on x is stronger
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, si tenta di evitare bug, ma vengono aggiunte 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 sicura 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 a 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 a un valore casuale al posto che all'indirizzo segnato. Permette di risparmiare sullo stack lo spazio della canary.

Per aggirarle:

- Brute force: su sistemi a 32 bit, è possibile indovinare il valore della canary in tempo "ragionevole"

- Leak: vulnerabilità che permettono di stampare informazioni, ad esempio format string, possono portare a un leak del canary, da usare poi nel payload di overflow

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 si può 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. Viene modificato il control flow del senza iniettare codice.

Address Space Layout Randomization ASLR: Randomizzare il layout di memoria, ogni volta che il processo va in memoria vengono usate zone di memoria differenti; su `x64` soprattutto, 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 implementazioni, ma in generale bisogna notare che:

- sposta solo l'offset delle zone di memoria, non le posizioni relative all'interno di esse
- potrebbe essere applicato solo alle librerie (sempre position indipendent), ma 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)

Per aggirare l'ASLR:

- Sfruttare un information leak per leggere un puntatore e calcolare la posizione di stack/librerie
- L'offset tra istruzioni è fisso, cambia solo la posizione generale in memoria, sovrascrivere solo gli ultimi byte con un offset noto permette quindi di arrivare alle istruzioni richieste

4 Return Oriented Programming ROP

Prevenzione e 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 ROP 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 già 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 (tipicamente sempre buffer overflow, necessita comunque di sovrascrivere un return address).

Le "sfide" di questo attacco sono

- trovare i gadget
- collegarli

Ma *cosa sono i gadget*? Semplicemente 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

I gadget trovati in memoria (pezzi di codice terminati da `ret`) vengono concatenati 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` del valore puntato da `%esp`) e prosegue da lì (codice puntato) l'esecuzione del programma.

Quindi, se sullo stack è presente l'indirizzo del gadget successivo, 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 dei gadget e i relativi parametri; ogni volta che ne viene eseguito uno (a partire dal primo), lo stack pointer scende e trova il successivo.

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 bisogna 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: saltare in mezzo a un'istruzione porta a una codifica diversa del programma.

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'istanza di istruzione `b`. In questo modo si possono 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? Generalmente 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 malevolo da 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). Prende il programma, definisce i gadget necessari, 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 `nginx`
- L'attaccante non ha nè binario (programma eseguibile) nè source code
- ASLR, canary, DEP attivi
- Conoscenza di una vulnerabilità, da qualche parte (in questo caso vulnerabilità nota per la versione di nginx)

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

- 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 causa ASLR
- la lunghezza sarà la dimensione dell'applicazione, approssimativamente nota

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

1. Leakare la canary
2. Defeat the ASLR
3. Trovare in modo blind i gadget necessari per la `write`
4. Ottenere il binario

Leakare la canary: Trovo la dimensione del buffer (tirando a indovinare), per poi provare a sovrascrivere un byte della canary alla volta, provando tutti i valori finché smette di crashare, per poi passare al byte successivo.

Defeat the ASLR: Provo a indovinare il return address, in maniera simile a come fatto per la canary, bisogna scoprire *più o meno* dove si trova il programma in memoria.

Una volta scoperta la canary, si può tirare a indovinare possibili return address finché non si trova uno degli indirizzi dov'è posizionato il programma (non crasha quando prova a uscire).

Blind search: Bisogna 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 questi gadget vanno trovati senza avere il binario.

Per indovinare il gadget si può sfruttare la differenza di comportamento tra `pop` e altre istruzioni: `pop` sposta il registro `esp`.

Dopo aver trovato la canary si tira a indovinare, sovrascrivendo il return address, per trovare due gadget:

- **idle gadget**: un indirizzo che mantiene aperta la socket (non crasha)
- **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 a 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 si possono mappare i gadget in memoria a run-time, si trovano le posizioni delle `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 a indovinare nella GOT.

5 Control Flow Integrity CFI

Si tratta di una tecnica di protezione basata su un'idea diversa dalle altre difese descritte precedentemente, le quali "complicavano" l'attacco prevenendone le fasi.

Qua il paradigma di difesa è diverso, si vuole una cosa più generale, l'idea è quella di **definire un modello di comportamento del programma** e se l'esecuzione effettiva esce da questo modello viene segnalata un'anomalia.

Le "sfide" di questa idea sono:

- definire il modello di comportamento, qual'è il comportamento aspettato
- rilevare efficientemente e velocemente deviazioni dalle aspettative
- evitare possibili compromissioni del detector/monitor che controlla il programma

Definire expected behavior: Come si può **caratterizzare il comportamento di un programma** durante l'esecuzione? Una caratteristica fondamentale di tutti i programmi è il control flow, rappresentabile tramite **Control Flow Graph CFG**, in cui ogni nodo è una chiamata a funzione (insiemi di istruzioni) e gli archi sono i passaggi di stato del programma.

Il CFG rappresenta il flusso del programma in termini di insiemi di istruzioni e transizioni tra di essi. Se durante l'esecuzione effettiva il comportamento non rispetta le transizione definite: anomalia.

Rilevare deviazioni efficientemente: Il controllo deve essere rapido, quindi il monitor deve essere efficiente. Non può essere esterno perché diventerebbe troppo invasivo. Il controllo viene quindi inserito all'interno del programma: **In-line reference monitor IRM**: istruzioni di controllo inserite all'interno del programma stesso.

I controlli vanno fatti rispetto a un **threat model** ben definito: si suppone quali siano le possibili azioni che l'attaccante può o non può compiere e i controlli vengono inseriti di conseguenza.

Ad esempio: si presuppone che l'attaccante

- può fare stack e heap overflow, ROP

- non può disabilitare l'IRM, non può modificare il codice in memoria (DEP)

Va sempre definito un threat model con ciò che è e non è lecito fare da parte dell'attaccante. Se l'attaccante riesce a rompere questi presupposti il sistema di difesa viene effettivamente superato (ad esempio, disabilita l'IRM saltando i check).

In generale, si presuppone che l'attaccante può scrivere cosa e dove vuole, ma non può modificare o inserire codice. A partire da questo si vuole evitare che ci siano deviazioni illegittime del flusso di controllo, permettendo transizioni solo verso destinazioni valide.

Evitare compromissione del detector: Per evitare che l'IRM venga disabilitato sicuramente serve che il codice sia immutabile e che sia presente una sufficiente randomness all'intero dei controlli stessi.

L'in-line monitor deve essere protetto da **immutabilità** e **randomness**.

In termini di **efficienza**: il **Classic CFI** (2005 di Microsoft Research) portava un overhead medio del 16%, con 45% nel caso peggiore; funzionava solo su eseguibile, non sulle librerie. Nel 2014 si è raggiunto il `Modular CFI` (`pagina autore`), un modello di CFI modulare con un overhead molto più basso (5% in media, 12% worst case), il quale è in grado di lavorare anche su librerie.

Sicurezza: MCFI permette di ridurre del 95.75% i gadget ROP; definendo punti di entrata e uscita dell'esecuzione non si può portare avanti un attacco ROP, il quale, per definizione, modifica il flusso del programma; MCFI blocca i salti illeciti.

La percentuale di riduzione viene dal fatto che i gadget non possono essere utilizzati in quanto non parte del flusso di controllo originale, significativamente riducendo la superficie di attacco. Si possono usare solo i gadget presenti all'interno del flusso di esecuzione (pochi).

Un'altra metrica usata è l'**Average Indirect-target Reduction (AIR)**, ovvero la percentuale di riduzione di possibili bersagli di salti indiretti che un attacco potrebbe sfruttare (punti di controllo del control flow). Le istruzioni di salto possono essere categorizzate in

- **direct calls**: viene definito esplicitamente l'indirizzo a cui saltare (direct, salta a quell'indirizzo)

- **indirect calls**: tutte le operazioni in cui l'indirizzo finale del salto va calcolato a runtime

In sostanza, la percentuale di bersagli per salti indiretti che vengono eliminati dalla CFI, ovvero quasi tutti.

5.1 Costruzione del Control Flow

La costruzione del control flow di un programma è divisa in due fasi:

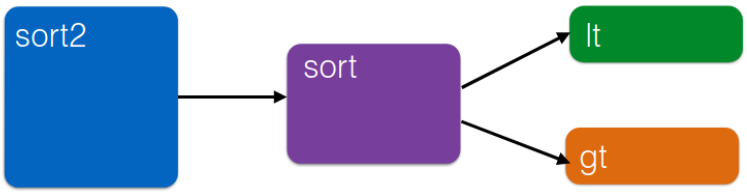
- analisi statica
- analisi dinamica

Analisi statica: Il compiler costruisce il grafo del control flow (già presente, viene usato per ottimizzazione, "strumenta" il programma).

Il primo passo dell'analisi statica è il **call graph**, ovvero il grafo che rappresenta le chiamate a funzione; quale funzione chiama cosa. Esempio:

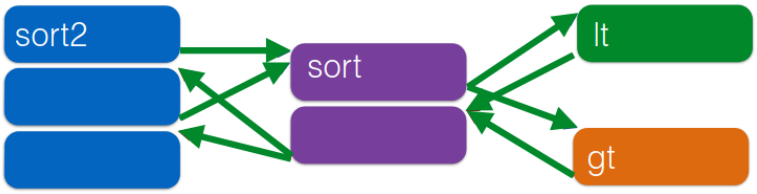
```
sort2(int a[], int b[], int len)
{
    sort(a, len, lt);
    sort(b, len, gt);
}
```

```
bool lt(int x, int y) {
    return x<y;
}
bool gt(int x, int y) {
    return x>y;
}
```



Permette di rappresentare il **flusso di chiamate** all'interno del control flow, ma si tratta di un modello piuttosto permissivo: rappresenta le chiamate ma non il flusso reale delle funzioni.

Idealmente si vorrebbe un modello *più restrittivo*, ovvero **il CFG**; si introducono i **basic blocks**: contenitori di istruzioni sequenziali, terminate da un'istruzione di salto, di qualsiasi tipo (`jmp`, `ret`, …). Permette di distinguere `call` da `return`, il grafo sul codice di prima diventa:



Ogni funzione viene "spezzata" in moduli, ognuna delle quali ha una `call`/`return`, definendo meglio il modello di esecuzione del programma. Ogni basic block fa una `call` o una `return` (praticamente un gadget).

In sostanza:

- viene calcolato il CFG di `call`/`return` a compile time (o comunque partendo dal binario)
- il control flow del programma viene monitorato dall'interno, assicurando che segua percorsi definiti dal CFG
- le direct calls non vanno monitorate (assumendo che il codice sia immutabile, l'indirizzo target non può essere cambiato), quindi **solo le indirect calls vanno controllate** (`jmp`, `call`, `ret` con target non costanti, tutti i salti parametrici)

5.2 In-Line Monitor

Bisogna implementare all'interno del programma un metodo efficace per controllare "da dove arriva" un salto, inoltre deve essere una "trasformazione" del programma.

Per fare ciò vengono usate delle **label**:

- viene inserita come commento una label all'inizio del basic block destinazione del salto, riga non di codice
- prima di saltare, viene fatta una `cmp` tra il valore presente nella destinazione (dove punta il registro su cui viene fatto il salto) e il valore che dovrebbe avere la label
- se coincidono tutto a posto, manda avanti l'esecuzione
- se non coincidono blocca il programma

Il codice

Source			Destination	
Opcode bytes	**Instructions**		**Opcode bytes**	**Instructions**
FF E1	jmp ecx	; computed jump	8B 44 24 04	mov eax, [esp+4] ; dst
		…		

Diventa:

Source			Destination	
Opcode bytes	**Instructions**		**Opcode bytes**	**Instructions**
81 39 78 56 34 12	cmp [ecx], 12345678h	; comp ID & dst	78 56 34 12	data 12345678h ; ID
75 13	jne _error_label	; if $\neq$ fail	8B 44 24 04	mov eax, [esp+4] ; dst
8D 49 04	lea ecx,[ecx+4]	; skip ID at dst		…
FF E1	jmp ecx	; jump to dst		

Viene inserita **una label per ogni basic block**, quando c'è un indirect call il programma controlla che la destinazione abbia la label corretta (o una delle label corrette) prima di saltare. Viene controllato che il salto sia alla destinazione corretta.

Tipi di labeling: Il **labeling** può essere:

- **semplice**: una stessa label per tutti i blocchi; evita chiamate fuori dal grafo ma non impedisce salti errati; permette "quello che vuoi" all'interno del grafo
- **dettagliato**: una label diversa a seconda dei punti di ritorno, permettendo di controllare anche che i salti all'interno del grafo siano corretti

Can we defeat CFI?

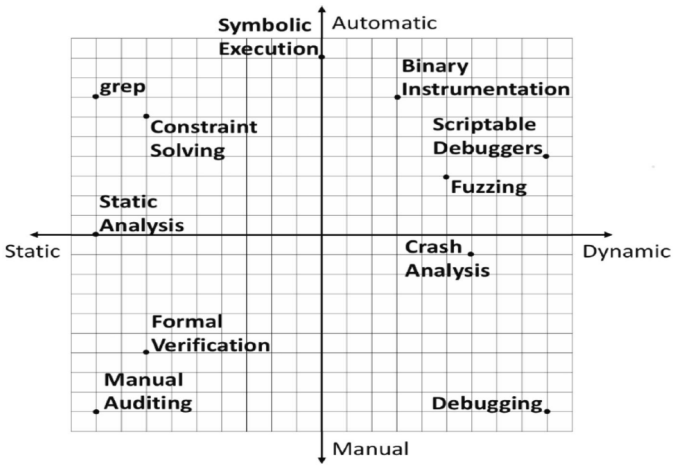
- Code injection con label corretti: non si può fare perché i dati non sono eseguibili (DEP)
- Modificare label nel codice per permettere il control flow desiderato: il codice è immutabile
- Modificare lo stack durante il check per controllarne l'esito: l'attaccante non può modificare i registri in cui sono caricati i dati rilevanti al controllo (No time-of-check, time-of-use bug TOCTOU)

Garanzie: Il CFI garantisce di impedire attacchi che modificano il control flow (ROP, ret2libc, …). Ma non

- attacchi che manipolano il control flow seguendo le label (mimicry attacks, soprattutto con labeling semplice)
- data leaks: `Heartbleed` non sarebbe stato bloccato
- corruptions: modificare variabili di controllo per portare al flusso di esecuzione desiderato (e.g, overflow di una variabile "`authenticated`" su cui viene fatto un controllo)

6 Vulnerabilities Detection

Dato un programma, si vuole cercare in modo proattivo se sono presenti vulnerabilità. Esistono molti metodi per cercare vulnerabilità, possono essere più o meno automatici e possono operare in maniera statica o dinamica.



Di conseguenza, una tecnica può essere

- Dinamica e automatica, esempio: binary instrumentation (inserire codice a runtime) o fuzzing (inviare input con l'obiettivo di trovare comportamenti anomali)
- Dinamica e manuale, esempio: debugging (eseguire il programma passo dopo passo per osservarne lo stato interno)
- Statica e manuale, esempio: manual auditing del codice (revisione manuale), verifica formale (dimostrare matematicamente la correttezza)
- Statica e automatica, esempio: analisi statica, esecuzione simbolica

Prima di effettuare le analisi è necessario definire delle proprietà attribuibili ai tool usati per effettuare le analisi. Le proprietà principalmente sono:

- **Soundness**: una tecnica è *sound* se è in grado di dire che un programma non ha nessuna vulnerabilità, ovvero se c'è una vulnerabilità la trova. Possono esserci **falsi positivi** (si tratta di un'approssimazione astratta), ma se è presente una vulnerabilità vera la trova. Di conseguenza, un programma è unsound se ci sono dei falsi negativi

- **Completeness**: una tecnica è *complete* se, quando una vulnerabilità viene trovata, questa è vera. Può offrire **falsi negativi**, ovvero "mancare" vulnerabilità, ma se la trova è sicuramente una vulnerabilità reale. Di conseguenza, una tecnica incomplete può avere falsi positivi

TL;DR: Le tecniche possono essere **sound**: trova tutti i bug possibili, potrebbe essere un falso positivo, oppure **complete**: può non trovare tutti i bug, ma se lo trova è una vulnerabilità reale. Ovviamente non esiste un tool che le ha entrambe.

Da queste proprietà si possono derivare altre caratteristiche, ad esempio:

- se un tool è sound allora vuol dire che considera tutti i path di esecuzione possibili del programma, in quanto requisito per trovare tutti i bug possibili; in maniera dinamica questo non è fattibile, viene fatto tramite tecniche di static analysis, costruendo un'astrazione dell'esecuzione del programma
- se un tool è complete allora deve mostrare esecuzioni del programma che permettono di verificare le vulnerabilità; quindi l'analisi è più dinamica, il programma viene lanciato con un set di input che possono causare problemi; la tecnica fornisce input concreti che triggerano vulnerabilità

La soundness è più legata alla static analysis (analisi su modello astratti del programma), mentre la completeness alla dynamic analysis (esecuzioni reali del programma).

La **static analysis** consiste generalmente di modelli matematici astratti per l'esecuzione del programma per poi fare inferenza su quali possibili path potrebbero portare a vulnerabilità.

La **dynamic analysis** invece utilizza esecuzioni più concrete per mostrare vulnerabilità effettive.

Solitamente, i tool di detection hanno un approccio *ibrido*, usando tecniche statiche e dinamiche per ottenere un buon compromesso tra soundness e completeness. Il trade off si traduce in quanti falsi positivi e falsi negativi saranno presenti.

Una differenza importante tra static e dynamic execution sono le performance: la static analysis è una analisi del codice, di conseguenza è molto

più veloce dell'esecuzione vera e propria del programma (la dynamic deve eseguire davvero il programma, quindi può essere molto lenta; quanto ci può mettere il programma?).

6.1 Symbolic Execution

La static analysis è una analisi off-line a partire da codice sorgente o binario del programma, quindi produce risultati riguardo la qualità del codice. Vengono definite delle proprietà a priori sul codice e i tool di static analysis controllano che queste vengano rispettate.

Un classico esempio di analisi statica è il compilatore di un programma: per la compilazione e ottimizzazione bisogna effettuare un'analisi del codice; molti tool di analisi statica si basano su ciò che produce il compilatore. L'analisi statica può usare o meno l'esecuzione simbolica.

La symbolic execution è una tecnica di static analysis che permette di simulare in maniera off-line un'esecuzione del programma; "finge" un'esecuzione del programma.

Si chiama "simbolica" perché vengono usati valori simbolici, viene calcolata un'approssimazione del programma per ottenere formule che rappresentano lo stato del programma stesso in diversi punti di esecuzione.

Il testing funziona: i bug riportati sono reali, ma ogni test è sostanzialmente un'asserzione, ogni condizione di test può controllare solo *una* possibile esecuzione. In breve, complete but not sound. Si spera che il caso di test generalizzi abbastanza, ma non ci sono garanzie.

La symbolic execution generalizza quello che è il testing di casi singoli, vuole essere "più sound", rappresentando più in generale l'esecuzione del programma. Da asserzioni su input concreti passa ad asserzioni su simboli generici, esempio:

`assert(f(3) == 5)` $\longrightarrow$ `y = a; assert(f(y) == 2*y-1);`

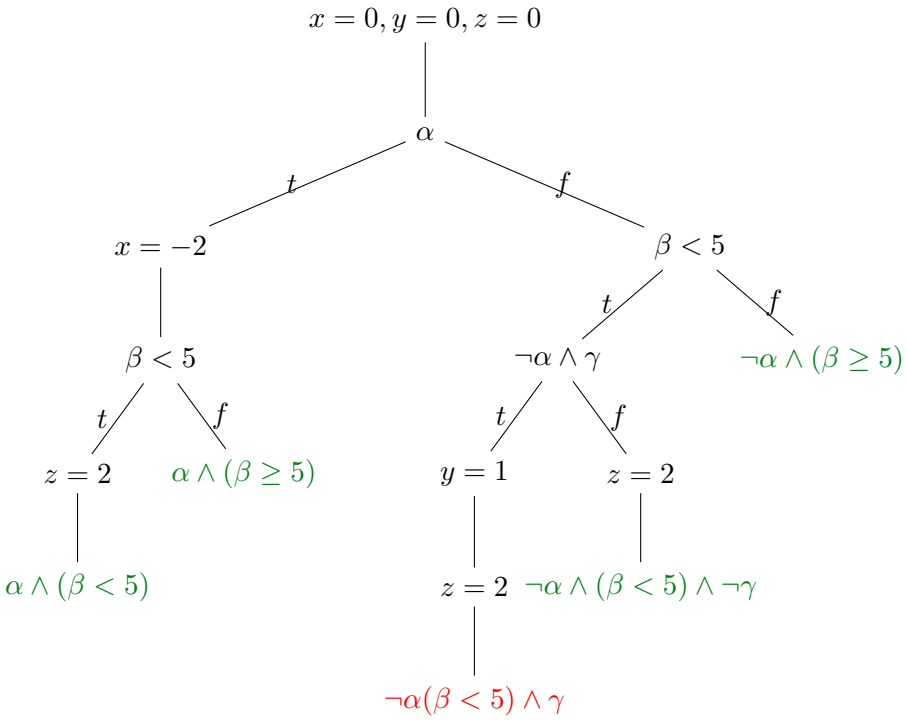
Se il path di esecuzione dipende da variabili sconosciute, l'esecuzione simbolica viene divisa concettualmente

```
int f(int x)  if (x > 0) then return 2*x - 1; else return 10;
```

Una "esecuzione simbolica parallela".

Esempio:

```
int a = alpha, b = beta, c = gamma; // symbolic!
int x = 0, y = 0, z = 0;
if (a) {
    x = -2;
}
if (b < 5) {
    if (!a && c) { y = 1; }
    z = 2;
}
assert(x+y+z != 3)
```



Non si vuole eseguire realmente il programma, piuttosto verificare che non ci possa mai essere un'esecuzione del programma che viola determinate proprietà. Si vuole cercare una classe di input che permette di violare le condizioni di esecuzione del programma, se questa esiste.

Per ogni terminazione di path di esecuzione sarà presente una **path condition**: formula che permette di caratterizzare tutti gli input che permettono di eseguire quel determinato path (nell'esempio: in verde se soddisfacibili, rosse altrimenti); se tutte le condizioni sono verificate (e quindi l'equazione

logica è risolvibile) vuol dire che gli input permettono di entrare in quel determinato path di esecuzione.

Ogni foglia dell'albero di esecuzione ha legata la sua equazione logica che permette di stabilire i potenziali input che portano l'esecuzione in quel determinato path.

Se viene trovata una violazione delle asserzioni in un determinato path, risolvendo la path condition si ottiene la classe di input che causa la violazione dell'asserzione. Dalla formula si può derivare l'input concreto per la violazione. Trovare input per la violazione diventa problema per un `constraint solver` (come ad esempio `Z3`).

Ogni path dell'esecuzione simbolica rappresenta più esecuzioni del programma: esattamente il set di esecuzioni i cui valori concreti soddisfano la path condition. In questo modo si possono coprire molte parti di esecuzione e testing.

Vista come analisi statica, la symbolic execution è

- **complete**, ma **non sound** (generalmente non termina, si "incastra" spesso, la terminazione non è garantita)
- dipendente da path, flow e contesto

L'idea è piuttosto vecchia, ma aveva limiti che la rendevano poco pratica, in particolare: può essere molto compute-intensive: i path possibili possono essere *tanti*, con path condition lunghe e difficili da verificare (anche solo la soddisfacibilità); quindi soggetta all'hardware del tempo: a oggi hardware e algoritmi per i SMT/SAT solver sono molto migliorati.

Nel 2005-2006 è stato ripresa l'idea di symbolic execution per l'ambito del bug finding, con l'aggiunta di euristiche per ridurre lo spazio di esecuzione.

Symbolic variables: All'interno delle espressioni (quindi all'interno del linguaggio) vengono **aggiunte variabili simboliche**, le quali rappresentano valori sconosciuti. Vengono introdotte quando sono presenti input forniti al programma (`mmap`, `read`, `write`, …). Se un bug viene trovato queste permettono di riprodurlo. Sono dei "placeholder" per valori sconosciuti a compile time (noti solo durante l'esecuzione).

Vogliamo fare in modo che il linguaggio possa includere espressioni simboliche. Normalmente, le variabili in un programma contengono valori, ora

possono contenere anche espressioni simboliche.

Esempio:

```
1 x = read();
2 y = 5 + x;
3 z = 7 + y;
4 a[z] = 1;
```

La memoria simbolica conterrà:

$$\begin{aligned} \texttt{x} &\mapsto \alpha \\ \texttt{y} &\mapsto \texttt{5+}\alpha \\ \texttt{z} &\mapsto \texttt{12+}\alpha \end{aligned}$$

E se `a` non è abbastanza grande, il valore di α potrebbe causare problemi.

Il **controllo del programma** può essere **condizionato dai valori simbolici** (il programma dipende anche dai valori esterni, generalmente). Esempio:

```
1 x = read();
2 if (x>5) {
3    y = 6;
4    if (x<10)
5       y = 5;
6 } else
7    y = 0;
```

E possiamo rappresentare l'influenza dei valori simbolici attraverso le path conditions. Esempio: l'esecuzione arriva alla riga `3` solo se la path condition $\pi = \alpha > 5$, mentre alla riga `5` si arriva solo se $\pi = \alpha > 5 \wedge \alpha < 10$.

Una path condition può essere insoddisfacibile: **unfeasible path** (non c'è soluzione alla formula logica). Le soluzioni a path constraints possono essere usate come input concreti.

Si possono introdurre asserzioni che permettono di determinare se ci sono vulnerabilità sfruttando la feasibility dei path. Esempio: prima di accedere a un array si introducono dei bound check:

```
1 x = read();
2 y = 5 + x;
3 z = 7 + y;
4 if(z < 0)
5    abort();
6 if(z >= len(a));
7    abort();
8 a[z] = 1;
```

Le due condizioni permettono di controllare che non si presentino buffer

overflow all'interno del programma. Trovare delle soluzioni alle path condition rilevanti (nell'esempio, per la riga `5` $\pi = 12 + \alpha < 0$ e `7` $\pi = \neg(12 + \alpha < 0) \wedge 12 + \alpha \geq 4$) vuol dire trovare classi di input che permettono di avere la vulnerabilità per cui si sta testando (buffer overflow in questo caso).

Ogni volta che si presenta un branch si ha una fork dell'esecuzione simbolica: si ha un symbolic executor per ogni path di esecuzione. Accade quando si possono avere soluzioni sia alla path condition che alla sua negazione.

Libraries e codice nativo: L'esecuzione simbolica prima o poi raggiungerà i "limiti" dell'applicazione: librerie, sistema o chiamate a codice assembly.

Le soluzioni a questo problema possono essere:

- far entrare il symbolic executor nella libreria, ma potrebbero essere *molto* complicate (probabilmente si incastrerà)
- fornire un modello delle librerie per l'esecuzione

TL;DR: Si vuole fingere un'esecuzione del programma con variabili simboliche, percorrendo ogni path e tenendo traccia delle condizioni necessarie per arrivare a ogni determinato stato del programma (path condition).

Una volta trovata una vulnerabilità (violazione di asserzioni stabilite per il programma) si usa un SMT solver (e.g., Z3) per risolvere la path condition del percorso, ottenendo la classe di input concreti che portano alla vulnerabilità (se la path condition è risolvibile).

In generale:

- Analisi statica: analizza il codice senza eseguirlo, deduce proprietà tramite modelli astratti, migliore scalabilità grazie all'astrazione di singoli percorsi (ad esempio, path sempre feasible)
- Symbolic execution: esegue il programma con valori simbolici, usati per considerare tutti i possibili input, i percorsi di esecuzione diversi vengono considerati tramite espressioni logiche (path conditions), scalabilità più limitata dovuta alla path explosion (2^n path, n numero di condizioni) e alla complessità delle path condition (rilevanti le performance dell'SMT solver)

6.1.1 Concolic Execution

Anche chiamata **dynamic symbolic execution**, il nome viene da concrete+symbolic. Lavora con gli stessi concetti della symbolic execution ma è un ibrido tra analisi statica e dinamica.

Si instrumenta il programma per fare symbolic execution durante l'esecuzione: a partire da un input concreto si esegue il programma, ma si traccia l'esecuzione simbolica durante l'esecuzione concreta; si tiene traccia delle path condition (shadow memory). Ogni volta che viene trovato un ramo condizionale nel flusso di controllo si tiene traccia del ramo percorso concretamente e la condizione per percorrerlo.

Terminata l'esecuzione di un percorso, si ottiene una path condition "completa", per generare un percorso alternativo da eseguire si nega una delle condizioni e si usa un constraint solver per trovare un nuovo input che soddisfi tali condizioni.

Si esplora un percorso alla volta tramite valori concreti. Permette di avere sempre dei valori concreti per ogni esecuzione e si possono anche seguire chiamate esterne al programma (perdendo symbolic-ness). Si possono gestire casi senza necessariamente far esplodere la complessità per il SMT solver.

In breve, la concolic execution parte da input simbolici concreti, associando alle variabili simboliche un valore concreto in modo da ottenere un'esplorazione del programma secondo il valore scelto.

Per ogni diramazione del programma viene salvato lo stato che ha portato ad una determinata scelta all'interno di una shadow memory; vengono salvate tutte le condizioni che vengono attraversate.

Una volta arrivato alla fine del path (di una esecuzione) si può tornare indietro per determinare una condizione che permette di esplorare il path simmetrico (un altro path).

TL;DR: Si esegue il programma con un input concreto, ma si tiene traccia simbolicamente del percorso preso. Una volta terminata l'esecuzione si nega una delle condizioni del percorso e si usa un SMT solver per trovare un input che soddisfa la path condition modificata.

Differenze: Rispetto alla Symbolic execution:

- L'input è anche concreto, non solo simbolico, si ha un'esecuzione concreta
- Usando valori concreti è più facile gestire interazioni con il sistema operativo/librerie/puntatori e simili

In generale, la concolic execution permette migliore coverage nei casi in cui vengono richiesti input esterni (in quanto forniti concretamente), ma può essere meno efficiente in casi di branching logico pesante, in cui la symbolic execution riesce ad analizzare tutti i path risolvendo le path condition, mentre la concolic deve trovare input reali.

6.1.2 Search

Le principali problematiche legate all'utilizzo della symbolic execution sono:

- come effettuare l'esplorazione all'interno del flusso di esecuzione di un programma?
- risolvere molte formule logiche, anche di grandi dimensioni, quindi serve un SMT solver e sono rilevanti le performance

Path Explosion: Solitamente la symbolic execution *non può essere svolta in modo esaustivo*, il numero di esecuzioni è (solitamente) esponenziale nel numero di branch (ad esempio, per ogni `if` di controllo, se ci sono 3 variabili simboliche booleane si possono avere 2^3 path possibili; ma anche per ogni loop, in quanto possono essere visti come una serie di condizioni).

Comparandola all'**analisi statica**: quest'ultima permette di approssimare *sempre* l'esecuzione di un programma, loop e condizioni possono essere approssimati come "always feasible"; questo semplifica l'analisi ma può portare a **falsi positivi**.

Questo non è vero nella symbolic execution: va **sempre simulata l'esecuzione di un programma**, con la relativa risoluzione di formule logiche.

Bisogna **definire** dei **metodi di esplorazione** del path di esecuzione in modo da **guidare l'esploratore simbolico**, in quanto non è possibile eseguire il programma esaustivamente, bisogna aggiungere delle euristiche per capire *dove andare*.

L'idea più semplice possibile è una BFS/DFS, con la relativa queue/stack per tenere traccia di cosa visitare; visito un grafo in maniera "banale". Questo però ha dei problemi:

- il principale è che non sono guidati da nessun tipo di informazione aggiuntiva, non viene cercato nulla di "particolare" all'interno del programma
- la DFS potrebbe "bloccarsi in una zona del programma"
- la BFS è marginalmente meglio, ma non è facilmente implementabile come concolic

Search strategies: Dobbiamo fornire delle priorità per la ricerca, vogliamo andare verso path che *probabilmente* contengono errori. Vogliamo modellare l'esecuzione del programma come un DAG (Directed Acyclic Graph) in cui i nodi sono gli stati del programma e gli archi indicano la possibile transizione da uno stato all'altro. Di conseguenza dobbiamo decidere *che tipo di algoritmo usare per l'esplorazione del grafo.*

Non è noto *a priori* che path vanno presi, quindi una certa quantità di randomness può essere utile. Alcune idee:

- scegliere il path da esplorare in modo uniformemente casuale
- random restart se la ricerca non ha trovato nulla di interessante *da un po'*
- scegliere casualmente da path con la stessa priorità

Uno dei problemi con la randomness è la riproducibilità: vanno **mantenuti i valori dell'esecuzione**, altrimenti potrebbe non essere chiaro come un bug si è presentato; la comparsa dell'errore può essere legata a una certa sequenza di stati precedenti, i.e., non è un singolo input che fa crashare il programma ma una sequenza; nel concreto vuol dire **tenere traccia dei path percorsi**, generalmente tramite il seed di un generatore pseudo-random.

Coverage-guided heuristic: Tecnica di ricerca basata sul **numero di volte in cui viene visitata una determinata istruzione**. Il programma è composto da un determinato numero di istruzioni, viene aggiunto uno "score" per indicare quante volte è stata eseguita ogni istruzione.

L'idea è che gli errori sono spesso in aree del programma difficili da raggiungere, quindi si punta a coprire la maggiore percentuale del programma possibile. Quando il punteggio è troppo alto si cercano input per zone non ancora esplorate.

Per arrivare in una certa zona di memoria potrebbero servire determinate precondizioni, non sempre soddisfacibili, non è detto che l'esecuzione riesca sempre a visitare tutte le zone del programma.

Generational search: Ibrido tra BFS e coverage-guided

- la prima generazione comincia con una esecuzione random completa
- tutte quelle successive negano un branch dell'esecuzione ottenuta nella generazione precedente in modo da ottenere un nuovo percorso, con path prefix diverso

Vengono usate coverage heuristics per determinare delle priorità.

Combined search: Effettuare più ricerche allo stesso tempo, non esiste una soluzione one-size-fits-all, quindi potrebbe essere un'idea usare algoritmi di ricerca diversi (magari in maniera alternata) per raggiungere parti diverse del programma, sperando di avere una ricerca il più completa possibile.

TL;DR: Si vuole modellare l'esecuzione del programma come un DAG, bisogna esplorarlo in maniera *sensata* per ottenere performance soddisfacenti. Alcuni metodi di ricerca:

- BFS/DFS: tecnicamente funzionano, ma estremamente inefficienti, non usano informazioni aggiuntive, si bloccano facilmente
- Coverage-guidede heuristic: si cerca di coprire il maggior numero possibile di istruzioni, tenendo traccia del numero di volte in cui viene visitata ogni istruzione
- Generational search: si comincia con una generazione di esecuzioni complete random, le generazioni successive negano un branch di una esecuzione precedente per ottenere un nuovo percorso. Si usano coverage heuristic per determinare delle priorità nella ricerca
- Combined search: più algoritmi assieme, sperando in una ricerca il più completa possibile

6.1.3 SMT Solvers

Per esplorare nuovi path serve risolvere formule logiche, anche molto complesse, quindi le **performance degli SMT solver** sono fondamentali per

effettuare symbolic execution.

Da una parte si cerca di **semplificare le formule** (ad esempio con la concolic execution, inserire input concreti riduce le variabili della formula logica), ma dall'altro si cercano **ottimizzazioni per la risoluzione delle formule stesse**.

I constraint solver usano tecniche per migliorare le performance, come ad esempio tradurre strutture di memoria in formule logiche efficientemente, teoria degli array (mappare array a formule logiche), caching di richieste precedenti, rimuovere variabili ridondanti, ….

Alcuni SMT solver:

- `Z3, da Microsoft research`
- `Yices, da SRI`
- `STP, da Vijay Ganesh, now @ Waterloo`
- `CVC3, primariamente da NYU`

6.1.4 Symbolic Execution Systems

Dopo l'ideazione, il primo revival della symbolic execution arriva con due sistemi:

- `DART: Godefroid and Sen, PLDI, 2005`
- `EXE: Cadar, Ganesh, Pawlowski, Dill, and Engler, CCS 2006`

SAGE: Concolic executor sviluppato da Microsoft Research, nato dal lavoro su DART, usa generational search e viene usato principalmente per trovare bug all'interno di file parser (probabilmente terminano, il comportamento è solo input/output, la concolic execution funziona bene).

Viene usato in produzione da MS dal 2007, rappresenta il laboratorio di fuzzing più grande al mondo (più di 500 machine years), con più di 3.4 miliardi di constraint controllati (più grande uso di SMT solver di sempre) per trovare centinaia di bug in centinaia di applicazioni.

KLEE: Esegue simbolicamente bytecode LLVM (compila file a `.bc`, KLEE lavora su questi). Sviluppato a partire da EXE, lavora usando diverse strategie di ricerca, principalmente random path + converage guided. Simula

l'ambiente per gestire system calls, accessi ai file, etc. Disponibile assieme a LLVM.

Mayhem: Sviluppato da CMU (Brumley et al), lavora su binari. Usa una ricerca BFS-style assieme a native execution; combina symbolic e concolic. Genera automaticamente report quando trova bug.

Mergepoint: Estende Mayhem con una tecnica chiamata *veritesting*, combina symbolic execution con analisi statiac del codice:

- l'analisi statica viene usata per blocchi di codice completi
- la symbolic execution per zone più difficili da analizzare (e.g., con loop indefiniti, aritmetica dei puntatori, syscall)

Permette un miglior bilanciamento tra solver ed executor, risparmiando tempo e trovando bug più velocemente (copre più del programma nello stesso tempo). Usato per 11,687 bug in 4,379 applicazioni su distribuzione Linux (inclusi nuovi bug in codice altamente testato).

Altri: Alcuni altri esecutori simbolici:

- **Cloud9**: Parallel, multi-threaded symbolic execution; Extends KLEE (available)
- **jCUTE, Java PathFinder**: symbolic execution for Java (available)
- **Bitblaze**: Binary analysis framework (available)
- **Otter**: directed symbolic execution for C (available); give the tool a line number, and it try to generate a test case to get there
- **Pex**: symbolic execution for .NET

Alla fine: La symbolic execution generalizza il semplice testing ed è usato in production code per molte applicazioni (SAGE per Microsoft, Mergepoint per Linux), ed esistono molti tool disponibili liberamente.

7 Fuzz Testing

Il **fuzzing** rientra tra le tecniche di ricerca di vulnerabilità. Nella sua forma più semplice consiste nell'inviare un'elevata quantità di input automaticamente generati a un target (programma o meno, qualsiasi cosa si debba testare).

L'obiettivo della ricerca è **trovare stati invalidi** del programma. Per esempio, su un binario, il fuzzing genera input fino a far crashare il programma.

Ci sono diverse **tipologie di fuzzer**, si possono categorizzare per due elementi fondamentali:

- il metodo di generazione dell'input
- introspection: livello di conoscenza del programma

Introspection: Si possono avere 3 livelli principali:

- **black box**: non sanno nulla di come è formato il programma all'interno
- **white box**: conoscenza completa del programma, visione totale dello stato interno del programma
- **grey box**: non hanno una visione completa dello stato, ma possono estrarre informazioni a runtime tramite instrumentazione

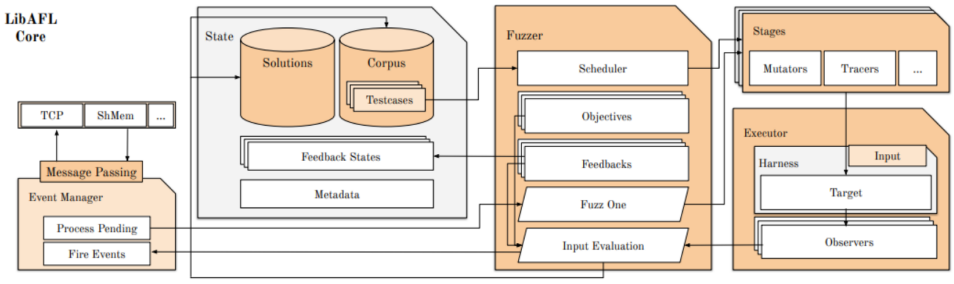
Metodo di generazione input: Esistono fuzzer che hanno una generazione dell'input di tipo:

- **generazionale**: partono da zero, generano secondo una grammatica, espressioni regolari o regole più complesse (online learning, ML)
- **mutazionali**: viene fornito un set di input di partenza, a ogni iterazione il fuzzer muta il set di input alla ricerca di altri input "*interessanti*" (va definito cosa significa)
- **ibrido**: combina tecniche di generazione e mutazione

7.1 Fuzzer Architecture

Come è fatto un fuzzer? Nella forma più semplice si hanno da un lato il fuzzer, il programma dall'altro: vengono mandati input da una parte all'altra.

LibAFL: Libreria Rust che permette la creazione di fuzzer arbitrariamente complessi. Per andare un po più nel dettaglio



Executor: Si tratta del componente che **riceve l'input** da parte del fuzzer vero e proprio **per poi eseguirlo**. Si possono avere diversi livelli di complessità per un executor, il più semplice (per un binario) è un fork server (fork del processo per poi applicare l'input, la fork evita di dover ricaricare il programma in memoria, il che sarebbe troppo lento).

Altre tipologie di executor, ad esempio per un kernel, dato che non si può caricare un kernel ex novo ad ogni test, usano VM o Hypervisor (wrapper attorno all'executor vero e proprio).

Input: Rappresentazione astratta della **tipologia di input accettata** dal programma. Potrebbe essere una bytestring (binario), una sequenza di syscall (kernel), una lista di transazioni (smart contract), …; a seconda di ciò che viene testato è importante conoscere cosa prende in input il programma.

Corpus: Gli input vanno salvati da qualche parte. Il Corpus contiene due elementi:

- **Solutions**: input che assolvono al loro compito; ovvero causano crash o comunque raggiungono l'obiettivo
- **Interesting**: input che raggiungono zone *interessanti* del programma (e quindi vengono messi in coda per eventuale mutazione)

Quelle non interessanti vengono scartate (a cosa servono?).

Generator: Il componente che si occupa di generare input, secondo grammatiche o regole di generazione. In generale, gli input generati devono

rispettare una struttura.

Scheduler: Si può avere un numero indefinito di test case (tra generazione e mutazione gli input possono aumentare molto in fretta), quindi è importante orchestrare in maniera corretta la **selezione dei test case**. Va definito uno scheduler, nel caso più semplice può essere una coda/stack, ma può avere un impatto significativo sulle performance.

Stage: Definisce una serie di operazioni da effettuare sull'input (ricevuto dallo scheduler), come mutazioni, taint tracking, ….

Mutator: Si occupa della **mutazione degli input**. A seconda del test case e di ciò che si vuole ottenere ne esistono di diverso tipo, il metodo più semplice è un bit flip, ma anche splicing, block swapping, truncating, …; possono essere usate tecniche arbitrariamente complesse.

Observer: Si vuole avere conoscenza di ciò che accade all'interno del programma, al fine di definire *cosa è interessante* (tranne nei casi di fuzzer black box, dove non si ha la nozione di interessante, vanno più o meno a caso, le uniche cose interessanti sono le soluzioni).

L'observer si occupa di **estrarre** (in qualche modo) **informazioni dal programma** in fase di analisi per poi rimandarle al fuzzer.

Un esempio potrebbe essere una coverage map: controlla gli edge, vengono tracciati i possibili flussi di esecuzione e si tiene traccia in un bytearray di quante volte l'esecuzione passa attraverso ogni determinata zona del programma.

Feedback: Modulo che si occupa di **analizzare le informazioni estratte**, controparte dell'observer. Riceve l'output dell'observer, lo analizza per determinare caratteristiche riguardo al test case eseguito, ad esempio per stabilire se è interessante e aggiungerlo o meno al corpus.

Cosa manca? Altri componenti che potrebbero esserci, meno nel dettaglio:

- se il fuzzer è su una macchina diversa dall'eseguibile, bisogna collegarlo alla rete
- **Harness**: sotto-componente dell'executor che effettivamente passa l'input al programma

7.1.1 Instrumentation

Come si fa a implementare un observer? Vogliamo estrarre informazioni da un programma a runtime, solitamente si fa con tecniche di **instrumentazione**: tipicamente a partire da una intermediate representation (per poterlo analizzare più facilmente in maniera automatica, senza rischiare di modificare il comportamento originario del programma), si analizza il programma per identificare zone "interessanti" e si aggiungono istruzioni che permettono di capire quando queste vengono raggiunte.

Esempio: mi interessa (per qualche motivo) contare le chiamate di sistema del programma, cerco all'interno della intermediate representation tutte le istanze di syscall e assegno un ID ad ognuna di queste, il codice iniettato tramite instrumentazione incrementerà in un array la casella corrispondente alla syscall effettuata, per ogni chiamata all'interno del programma.

Altro esempio di instrumentazione: i fat pointer, obiettivo diverso, ma sostanzialmente è la stessa cosa (visti qui: 3.1.1).

TL;DR: Instrumentazione vuol dire modificare il binario (solitamente non da sorgente, ma da rappresentazione intermedia), per rilasciare informazioni a runtime.

7.1.2 Sanitizers

Vogliamo identificare crash del programma, ma potrebbe non bastare per trovare vulnerabilità. Per questo motivo sono nati i sanitizer, pezzi di codice, **instrumentazioni aggiuntive** che permettono di **rilevare comportamenti anomali** quando accadono.

Esempi:

- `AddressSanitizer (ASan)`, rileva errori di memoria
- `UndefinedBehaviorSanitizer (UBSan)`, rileva undefined behavior
- `ThreadSanitizer (TSan)`, rileva race conditions

Quando rilevano un'anomalia fanno crashare il programma, con un log relativo all'errore, il quale può essere più o meno dettagliato.

Esempio: ASan (tra le altre cose) aggiunge delle "red zone" attorno alle zone di memoria, ovvero zone senza permessi di lettura o scrittura: quando il programma prova ad accedervi si genera un `segfault`. Instrumenta il

programma per cercare tutte le allocazioni, tutti gli utilizzi della memoria in generale (con le relative ottimizzazioni, non così semplice).

Ovviamente i sanitizer non vengono usati in produzione, non si tratta di un comportamento desiderato, oltre al fatto che rallentano l'esecuzione.

Confronto con symbolic/concolic execution: Rispetto alla concolic execution:

- Non richiede necessariamente una conoscenza profonda del codice interno (black/grey box)
- L'input è generato/mutato casualmente (secondo euristiche), mentre per la concolic deriva da vincoli logici (più computazionalmente costosi); la symbolic execution richiede di trovare input concreti con SMT solver
- Ha un overhead basso rispetto alla necessità di avere analisi simbolica e SMT solver
- Inefficace nel caso di vincoli complessi, mentre la concolic genera input "sicuri"
- La symbolic/concolic esplora sistematicamente tutti i percorsi, il fuzzing potrebbe avere peggiore coverage (l'esplorazione si basa comunque su un metodo casuale)
- Migliore efficienza su grandi progetti, ma copertura meno approfondita di percorsi complessi

In generale, il fuzzing funziona meglio in casi contenenti input esterni, syscall e condizioni facili da triggerare casualmente. Esempio:

```
1 void func(char* input) {
2     char buffer[64];
3     strcpy(buffer, input);  // Crash con input > 64 bytes
4     // Chiamata esterna complessa
5     FILE* file = fopen("/dev/random", "r");
6 }
```

La symbolic execution potrebbe bloccarsi sulla syscall in quanto non può emulare dispositivi esterni; in generale fa più fatica con chiamate esterne e funzioni che deve trattare come black box.

Invece la symbolic execution è più efficiente quando le condizioni richieste sono molto specifiche e difficilmente raggiungibili in maniera casuale. Esempio:

```c
void checksum(char* input) {
    int sum = 0;
    // Vincolo complesso: 1000 come somma dei valori ASCII
    for (int i = 0; i < 8; i++) {
        sum += input[i];
    }
    // Branch "nascosto" al fuzzing
    if (sum == 1000) {
        //...
    }
}
```

Il fuzzing difficilmente troverà una combinazione di valori corretta, la quale può essere trovata facilmente tramite symbolic execution.

Notes

State of the Art: AFL++: Stato dell'arte per fuzzing su binario, `presentato nel 2020` come gray box fuzzer con edge coverage guidance, derivato dall'originale AFL. Abbastanza customizzabile, scritto in C++.

NB: Si può fuzzare un po' tutto, non solo binari. Ad esempio:

- `Syzkaller:` unsupervised coverage-guided kernel fuzzer
- `Smartian:` Smart Contract Fuzzing with static and dynamic data-flow analyses
- `Nyx-Net:` coverage-guided network testing

OSS-Fuzz: Progetto lanciato da Google nel 2016 per fare fuzzing di repo open source di grossi progetti, individuando un buon numero di vulnerabilità in più di 1000 progetti.

8 Meltdown and Spectre

Pubblicati nel 2017, si tratta di attacchi side channel, vanno a distruggere alcune delle assunzioni solitamente fatte sull'hardware. Per quanto riguarda la sicurezza, solitamente vengono fatte assunzioni su alcune proprietà (come ad esempio si presuppone sia presente isolation tra processi, grazie all'MMU), con questi attacchi alcuni componenti hardware non sono più considerabili trusted, **rompendo il tipico threat model considerato**.

Con questi attacchi è stato mostrato come l'hardware possa essere compromesso, *senza particolari requisiti a livello software* (ovvero funzionano anche con un software bug free). Questi attacchi mostrano come, tramite un processo senza bug non privilegiato, si possa andare a leggere la memoria del kernel. Vengono **sfruttate problematiche di progettazione hardware** per andare a minare le solite assunzioni sulle componenti fisiche.

Questi attacchi mettono in crisi *tutta la progettazione hardware fino alla loro pubblicazione*, i requisiti di sicurezza non sono mai stati tenuti in considerazione durante la progettazione, sono sempre state guardate prima le caratteristiche funzionali (performance). Non esistono soluzioni ottimali, a meno di ricostruire tutta l'architettura.

Ci sono due principali componenti coinvolti in questi attacchi:

- CPU
- Cache

Le cache possono essere viste come una matrice in cui

- le linee si chiamano cache set
- le colonne si chiamano cache line

Tramite una manipolazione dell'indirizzo virtuale si può andare a capire se un determinato dato è presente all'interno della cache (ogni indirizzo virtuale può far riferimento ad una sola linea all'interno della cache). L'indirizzo virtuale (dopo essere stato tradotto in fisico) viene usato per determinare la linea di cache in cui i dati verranno posti.

Quando un processore deve accedere ad un certo indirizzo, richiede certi dati, prima interroga la cache, solo dopo, se il dato non è stato trovato, viene interrogata la RAM (in seguito ai livelli di cache inferiori).

In generale, in una CPU esistono più tipologie e livelli di cache

- ogni core ha una cache L1 e L2
- divisa tra i core si ha una cache L3

Più le cache sono vicine al core, più sono veloci. Queste tipologie di attacchi lavorano principalmente su cache L3.

Micro-architecture attack: Attacchi alla micro-architettura, sono attacchi che si posizionano nel layer tra software e hardware.

Side channel attacks: Attacchi (studiati soprattutto in crittografia) che studiano l'ambiente di esecuzione di un processo/dispositivo; anche senza guardare fisicamente all'interno del dispositivo, si possono fare inferenze sul contenuto del dispositivo? Si tratta di inferenze a partire dall'ambiente circostante per dedurre il contenuto.

Esempio: considerando una smart card, l'unica interazione possibile con una smart card è inviare del testo in chiaro e ricevere del testo cifrato (si presuppone che esistano difese hardware contro l'aprire fisicamente la smart card e leggere). Si può guardare l'ambiente esterno alla smart card per capire la chiave, ad esempio, guardando quanta energia elettrica usa la smart card per cifrare il testo: guardando l'onda di esecuzione, ogni testo avrà un'onda differente e si sfrutta conoscenza sull'algoritmo di cifratura per ottenere informazioni sulla chiave usata; ad esempio, alcune ottimizzazioni in base al contenuto di testo e chiave usate dall'algoritmo potrebbero essere indizi. Magari non si può capire la chiave completa, ma capirne il 90% per poter fare brute force sul resto è *good enough*.

In generale, l'analisi dell'ambiente circostante ad un dispositivo permette di dedurre informazioni riguardo l'esecuzione.

8.1 Attacchi alle cache

Le cache si prestano a questa tipologia di attacchi in quanto condivise, ogni processo condivide la stessa cache.

Meltdown e Spectre non sono completamente hardware, ma richiedono interazione tra software e hardware (microarchitectural attack).

8.1.1 Flush & Reload

Lo scenario di attacco è: una shared memory (ad esempio, una libreria in comune) e due processi software: un "attaccante" $P1$ e una "vittima" già in

esecuzione sulla macchina $P2$.

Si suppone che $P1$ conosca il codice di $P2$ (ad esempio, implementa un algoritmo noto), ma ovviamente non conosce lo stato di esecuzione del programma (ovvero valori usati all'interno del programma, presenti solo all'interno della memoria del processo $P2$). L'attaccante quindi vuole "spiare" lo stato di esecuzione di $P2$ sfruttando la memoria condivisa, supponendo siano due programmi indipendenti e bug-free.

Il processo $P1$ fa:

- **flush**: operazione assembly fornita dall'architettura, permette di svuotare totalmente la cache

In seguito $P1$ rilascia l'esecuzione e fa andare in esecuzione $P2$, il quale caricherà i suoi valori/eseguirà il suo codice. Una volta eseguito del codice, ogni indirizzo virtuale relativo alle istruzioni eseguite da $P2$ farà riferimento a una linea della cache, i.e., le istruzioni eseguite da quando la cache è stata svuotata saranno state caricate in cache.

Quando $P1$ torna in esecuzione, in seguito a $P2$, farà

- **reload**: carica (accede) a ogni indirizzo virtuale presente all'interno del codice, per poi calcolare il tempo di accesso a ogni indirizzo (access time)

Dal punto di vista della cache, gli unici indirizzi al suo interno sono quelli eseguiti da $P2$:

- se un indirizzo non è in cache andrà preso dalla memoria, più lento
- se il valore è già in cache allora l'accesso sarà molto più veloce

Quando viene trovato un tempo di accesso più veloce a una linea di cache, $P1$ può capire quali istruzioni ha eseguito il programma $P2$, ottenendo informazioni anche sullo stato e contenuto della memoria di $P2$. L'unica linea con accesso più veloce è la linea della libreria comune ai due processi già caricata in cache, ovvero le istruzioni appena usate dal secondo processo.

Riassunto:

- $P1$ fa `flush` della cache
- $P2$ va in esecuzione, carica in cache delle istruzioni
- $P1$ tenta di accedere a ogni indirizzo virtuale, quelli con tempo di accesso più breve rispetto agli altri sono le istruzioni eseguite da $P2$

Sapendo il mapping di $P2$ e il timing della cache posso inferire lo stato di esecuzione di $P2$.

8.1.2 Prime & Probe

Si hanno lo stesso scenario e assunzioni di prima (so dove come viene mappata virtualmente la memoria di $P2$), ma non si ha una memoria shared in uso da entrambi i programmi.

L'attacco consiste di:

- $P1$ fa **prime** della cache: riempie completamente la cache (semplicemente accedendo a tutti gli indirizzi del suo spazio di indirizzamento che sappiamo essere mappati all'interno di $P2$)
- va in esecuzione $P2$, eseguirà il suo blocco di codice, caricando x e y in cache
- $P1$ fa timing sulla cache: prova tutte le linee, quelle eseguite da $P2$ avranno tempo più lento; tutti i dati sono in cache, il codice di $P1$ è diverso da $P2$, quando $P1$ riprova ad accedere a tutta la cache ci saranno delle linee più lente, in quanto richiedono lo scarico e carico della cache: $P2$ avrà usato le linee virtuali corrispondenti a quello slot di cache

Ancora una volta, misurando le differenze di timing, si può capire l'indirizzo virtuale utilizzato da un altro processo, sapendo il mapping virtuale di questo secondo processo si può arrivare alle istruzioni eseguite, portando a inferenze sullo stato (utili o meno).

Questi attacchi vengono portati su un unico core, "isolando" l'esecuzione di programma target e attaccante, per evitare che ci sia "noise" a livello di cache dovuto all'esecuzione di altri processi (scheduling).

8.2 Speculative Execution

Meltdown e spectre usano vulnerabilità all'interno della CPU. Un'ottimizzazione (abbastanza spinta) della CPU è la **speculative execution** (presente all'interno di ogni CPU moderna).

Un programma è costituito di esecuzioni sequenziali: lento. L'esecuzione speculativa è "*tentare di indovinare*" cosa dovrà fare la CPU nei passaggi successivi, parallelizzando il più possibile l'esecuzione delle istruzioni (a livello di micro-operation, più in basso dell'assembly).

Il processore deve capire che istruzioni sono dipendenti tra loro, quali operazioni vanno eseguite in sequenza e quali invece possono essere parallelizzate.

All'interno di una reservation station (shadow memory) la CPU memorizza le istruzioni da eseguire sequenzialmente.

Ma la "prossima istruzione" che sta tentando di eseguire in anticipo non è detto che andrà effettivamente eseguita (detta transient instruction, sta effettivamente speculando sull'esecuzione); potrebbe succedere qualcosa tra *adesso* e quando dovrà davvero essere eseguita l'istruzione "speculata" che ne evita l'esecuzione (come, ad esempio, un `segfault`).

I risultati dell'esecuzione speculativa sono tenuti all'interno di un buffer (ROB), non vengono scritti immediatamente nei registri/memoria dell'architettura finché l'esecuzione non risulta confermata. Tutta la speculazione non è visibile all'architettura finché l'esecuzione non è confermata.

La domanda che si sono posti meltdown e spectre è: *posso usare questa cosa per fare leak di informazioni?* La risposta è *sì*.

8.3 Meltdown

`Meltdown` è una vulnerabilità che sfrutta:

- speculative execution
- cache attacks

Idea dietro il codice:

```
1 data = kernel_access[I];
2 access(probe_array[data*4096]);
3 address = probe_array + data*4096;
```

La prima istruzione è un **accesso ad un indirizzo del kernel**: siamo in un programma non privilegiato, quindi causerà un `segmentation fault`. Le istruzioni dopo sono "fattibili" (accessi a un array definito all'interno del programma), ma non verranno mai eseguite, il programma si fermerà sempre prima, questo però la speculative execution non lo sa e le vede come indipendenti tra loro.

Ognuna delle operazioni successive verrà divisa in microop ed eseguite in parallelo, il dato non viene faultato subito ma viene eseguito, anche se di nascosto (*se* vince la race condition in cui la seconda istruzione finisce prima

della prima all'interno del buffer, ma basta fare qualche tentativo, anche nel peggiore dei casi).

Le istruzioni che sono finite in shadow memory vengono **eseguite e poi bloccate**, l'architettura non le vede, ma finiscono comunque all'interno della **cache**. Sarà presente all'interno di una linea della cache il valore a cui è stato fatto l'accesso, ma che non doveva essere caricato.

Tramite prime & probe si può risalire a quale linea kernel (privilegiata) è stato fatto l'accesso. *Ma si può scoprirne il valore?* Per fare ciò entra in gioco la terza istruzione del listato: `probe_array` è un valore noto (definito dall'attaccante), `4096` è la grandezza di una linea della cache (potrebbe essere diversa, esempio di valore) e `data` è il valore che vogliamo scoprire. In questo modo viene caricata in memoria una cache line diversa in base al valore del dato.

Grazie al timing possiamo capire il valore: l'accesso è più veloce solamente alla linea x: il valore di `data`, caricato durante l'accesso out-of-order era x.

Esempio semplificato di codice:

```
// Indirizzo del valore cercato
volatile uint8_t *addr = &secret_value;
// Array per il side-channel (cache timing)
uint8_t array[256 * 4096]; // 256 pagine * 4KB (dim cache)
// Flush dell'array dalla cache
for (int i = 0; i < 256; i++)
    _mm_clflush(&array[i * 4096]);
// Tentativo di accesso, genererà segfault
value = *addr;
// Accesso speculativo, influisce sulla cache
array[value * 4096] = 0x42; // Dipende dal valore segreto
// Misura del tempo di accesso per trovare il valore
val = cache_timing();
```

8.4 Spectre

`Spectre` è una vulnerabilità che sfrutta:

- speculative execution
- cache attacks

- branch prediction

La branch prediction unit è un componente all'interno delle CPU, fa parte (circa) dell'esecuzione speculativa, tenta di prevedere l'esito dei branch per fare speculazione; viene fatto anche in base all'esecuzione passata (e.g., se 99 volte su 100 è entrato nel loop, probabilmente entrerà di nuovo).

Quindi, facendo un accesso a un array, con un indice valido, 99 volte, e alla 100esima viene messo un indice out of bound, la branch prediction eseguirà lo stesso l'istruzione, prima che il programma si accorga che l'accesso non è valido. Il valore caricato dalla branch prediction unit verrà, ancora una volta, caricato in cache.

Similmente a meltdown, un'istruzione che non doveva essere eseguita è stata anticipata, con effetto sulla cache, quindi ne si può capire il valore.

Esempio di codice spectre:

```
unsigned int array1_size = 16;
uint8_t array1[16] = {/* ... */};
uint8_t array2[256 * 512]; // Array per side channel
// Victim function
void victim_function(size_t x) {
    if (x < array1_size) {
        // Accede speculativamente anche se x è fuori limite
        uint8_t value = array1[x];
        uint8_t temp = array2[value * 512];
    }
}
void main() {
    size_t malicious_x = out_of_bound_value;
    // Pulisce array2 dalla cache
    for (int i = 0; i < 256; i++)
        _mm_clflush(&array2[i * 512]);
    // "Allenamento" con input valido
    for (int i = 0; i < 100; i++)
        victim_function(i % array1_size);
    // Chiamata speculativa con indice malevolo
    victim_function(malicious_x);
    // Timing delle righe di cache
    val = cache_timing();
}
```