SICUREZZA INFORMATICA

Buffer Overflow

Il buffer overflow è un bug che affligge il codice di basso livello, tipicamente nel C e C++ con significative implicazioni per la sicurezza. Normalmente un programma con questo bug crasha semplicemente, ma un attaccante può alterare le situazioni in modo che il programma faccia molto peggio:

- Rubare informazioni private
- Corrompere informazioni preziose
- Eseguire il codice scelto dall'attaccante

Perché sono così importanti? → Perché i buffer overflow sono ancora rilevanti oggi. C e C++ sono ancora popolari e i buffer overflow si verificano con regolarità. Molti approcci differenti sono stati sviluppati per difendere contro di essi e contro bug simili.

Sistemi critici in C e C++:

- La maggior parte dei kernel dei Sistemi Operativi e utilities
- Molti server ad alte prestazioni: nginx, MySQL, Microsoft SQL server etc
- Molti embedded systems: automobili, Mars rover, Sistemi di controllo industriale

Worm → virus che viene propagato su una macchina. Una volta preso possesso di questa macchina, se sono presenti nella stessa rete altre macchine "bucabili" questo si espande ad essa autonomamente.
Zero day → bug che non è ancora stato trovato.

Storia Buffer Overflow

1988 Morris Worm - Morris l'ha propagato attraverso delle macchine. Si è rivelato troppo aggressivo grazie allo sfruttamento di un bug. Un modo in cui si è propagato è stato un attacco di buffer overflow contro una versione vulnerabile di fingerd on VAXes. Ovvero, ha inviato una stringa speciale al finger daemon, il quale ha causato l'esecuzione del codice che ha creato una nuova copia del worm.
Risultato finale: gli è costato 10-100M di dollari in danni, libertà vigilata e ha dovuto effettuare servizio alla comunità.
Nel 2001 Code Red ha sfruttato un overflow nel Microsoft IIS server, 300.000 macchine infettate in 14 ore.
Nel 2003 SQL slammer, ha sfruttato un overflow nel Microsoft SQL server, 75.000 macchine infettate in 10 minuti.
X11/X.org → rapporto in cui veniva segnalato che era stato trovato un bug risalente al 1991. Il problema riguardava un possibile buffer overflow generato da una scanf per il caricamento dei font. Questo problema è stato presente nel codice per 23 anni.

Memory Layout

Tutti i programmi sono memorizzati nella memoria. Gli indirizzi di memoria vanno dal basso verso l'alto. Le istruzioni stesse sono in memoria.
Cmdline & env → settati quando il processo inizia.
Runtime → Heap che viene allocato dinamicamente tramite, ad esempio, delle malloc. Stack: contiene i record di attivazione delle funzioni. Lo stack possiede lo stack pointer che tiene traccia dell'ultimo elemento allocato nello stack. Solitamente in cima allo stack troviamo argc (ovvero il numero di parametri) e argv (ovvero le stringhe vere e proprie).

A tempo di compilazione conosciamo: dati non inizializzati, dati inizializzati e il text, ovvero il codice da eseguire.
Lo stack e lo heap crescono in direzioni opposte. Lo stack cresce verso il basso, mentre lo heap cresce verso l'alto.
Chiamate di funzione:

- Cosa succede in presenza di una chiamata di funzione?
- Quali dati devono essere memorizzati?

```
void func(char *arg1, int arg2, int arg3){
        char loc1[4];
        int loc2;
        …
}
```

Una chiamata a funzione è costituita dalle seguenti istruzioni macchina:

- call F → push dei parametri + Call F(address) → push ret address + JMP F

caller's data
arg3 (per primo)
arg2
arg1 (per ultimo)
???
???
loc1
loc2
….

Notiamo che gli argomenti della funzione vengono pushati in ordine inverso rispetto all'ordine presente sul codice, mentre invece le variabili locali sono messe nello stesso ordine.
Per accedere ad esempio ad una variabile locale facciamo utilizzo del puntatore **EBP** (frame pointer) che contiene l'indirizzo della base del frame corrente. **ESP** invece contiene l'indirizzo dell'elemento in cima allo stack.
Al momento del ritorno da una chiamata di funzione, EBP permette di ripristinare lo stato del record di attivazione precedente. Di conseguenza verrà modificato anche l'ESP.
EIP invece rappresenta il return address della funzione. Ci serve per ritornare al punto successivo ad un'istruzione di chiamata di funzione.

Calling function

1. Push argomenti sullo stack (ordine inverso)
2. Push del return address, ovvero l'indirizzo dell'istruzione a cui si vuole ritornare
3. Saltare all'indirizzo della funzione chiamata

Called function:

4. Push del vecchio frame pointer sullo stack
5. Set del frame pointer a dove è ora la fine dello stack
6. Push delle variabili locali sullo stack

Returning function:

7. Reset dello stack frame precedente
8. Si salta al return address

Quando torniamo da una funzione tramite la ret vengono eseguite le seguenti istruzioni macchina:

- pop per deallocare le variabili locali
- modifica del PC (program counter) che conterrà la prossima istruzione da eseguire
- JMP PC

Il chiamante della funzione deve occuparsi di deallocare le variabili locali dallo stack tramite le POP.

Buffer Overflow

Buffer → il buffer è una memoria contigua associata con una variabile o un campo. Tutte le stringhe in C sono array di caratteri.
Overflow → Si verifica quando all'interno del buffer viene messa una quantità maggiore rispetto a quanto esso può contenere. L'overflow solitamente causa un crash del programma oppure può portare ad una sovrascrittura di dati in memoria che non dovrebbero essere modificati.
In sicurezza il problema del buffer overflow viene chiamato channel problem, perché lo stack è chiamato channel.
È possibile sfruttare l'overflow, ad esempio, anche per autenticarsi senza dover inserire la password e accedere magari ad informazioni sensibili o comunque a qualsiasi tipo di informazione che deve essere tenuta non pubblica. Ovviamente deve essere fornito tramite overflow un valore valido. Una delle funzioni tramite la quale è possibile eseguire un overflow è la strcpy che ci permette di scrivere sul buffer quanto vogliamo (a meno che il programmatore non abbia effettuato controlli sulla dimensione del buffer).

Per sfruttare il buffer-overflow dobbiamo PRIMA dichiarare il buffer perché in memoria si scrive verso il basso (partendo dall'indirizzo più alto) quindi dobbiamo sforare il buffer per scrivere nella memoria successiva:

buffer[int]
var da sovrascrivere

Code Injection

Idea Principale: Caricare il mio codice all'interno della memoria in modo da far eseguire al programma ciò che voglio io (attaccante). Per realizzare ciò, occorre far puntare dal return address il codice che voglio iniettare. Il codice da caricare in memoria deve essere costituito da istruzioni in codice macchina, quindi già compilato e pronto ad essere eseguito.
In questa situazione andiamo ad eseguire codice nell'area dati, cosa insolita dato che solitamente il codice si trova da un'altra parte e l'area dati è solo leggibile o scrivibile.
Dobbiamo prestare attenzione a come lo costruiamo:

- Non può contenere byte a zero, altrimenti operazioni come scanf/gets/strcpy smetteranno di copiare.
- Non si può usare il loader, perché stiamo iniettando codice.

Lo scopo generale nell'iniettare il codice è quello di lanciare una shell con scopo generale che ci dà un accesso generale al sistema.
Il codice utilizzato per lanciare una shell è chiamato **shellcode** (lo shellcode generale di solito è di 24 bytes).
Difficoltà:

- Non possiamo inserire un'istruzione "salta al mio codice" perché non sappiamo esattamente dove il nostro codice da eseguire sia.

Sfruttiamo la possibilità di modificare il return address di una funzione in modo che, al posto di ritornare all'istruzione successiva, la chiamata di funzione punti al nostro codice malevole iniettato.
Ma come conosciamo l'indirizzo del return address? E se sbagliassimo? La CPU potrebbe trovarsi nella condizione in cui esegue dei byte che non corrispondono ad alcuna istruzione valida.
Come trovare il return address → Se non abbiamo accesso al codice, non sappiamo quanto lontano sia il buffer dal EIP salvato. Un possibile approccio è quello di provare tanti differenti valori. Scenario peggiore: se lo spazio di memoria è a 32 bit allora sono 2^32 possibili risposte, altrimenti se 64 bit abbiamo 2^64.

Tuttavia, senza la randomizzazione (dati del programma caricati ad indirizzi sempre diversi) lo stack partirà sempre dallo stesso indirizzo fissato. Lo stack cresce, ma solitamente non lo fa molto profondamente, a meno che il codice non utilizzi pesantemente la ricorsione. Tuttavia, la posizione del mio buffer può essere influenzata dalla dimensione dell'argv.
Per migliorare le nostre chance di trovare l'indirizzo del buffer contenente il nostro codice da far puntare al ret address, possiamo utilizzare quelle che si chiamano "nop sleds" ovvero "piste di atterraggio". Si tratta di istruzioni composte da un singolo byte (0x90). Queste istruzioni vengono utilizzate per aumentare il margine di errore nel tentativo di trovare l'indirizzo corretto, dato che, essendo costituite da un singolo byte, se il processore si trovasse ad eseguire una di esse, si comporterebbe semplicemente procedendo alla prossima istruzione (ciclo di clock) senza eseguire nulla. Questo succede per evitare di saltare tra dei byte che compongono delle istruzioni precise, le quali verrebbero interpretate dalla CPU come istruzioni non valide.
Per completare l'overflow, ci serviamo anche di un po' di padding che banalmente potrebbero essere dei caratteri "A" della lunghezza richiesta.

Illegal Instruction → istruzione che il processore non è in grado di capire e quindi di eseguire.
Stack protection → protezione che non permette di sovrascrivere il ret address.

Heap Overflow sui MetaDati

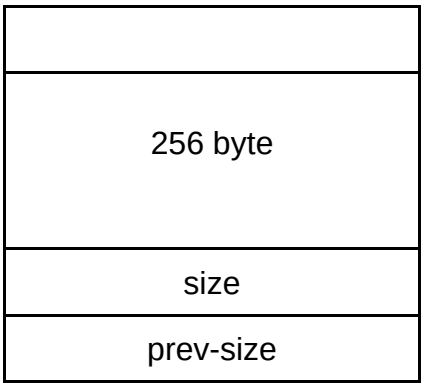
Differenze tra Stack e Heap:

- Stack:
  - Allocazioni di memoria fissate conosciute al tempo di compilazione
  - Variabili locali, return address, argomenti funzioni
  - Veloce e automatica, viene fatta dal compilatore
- Heap:
  - Allocazioni di memoria dinamica a tempo runtime
  - Oggetti, grossi buffer, strutture etc
  - Lenta e manuale, fatta dal programmatore (tramite malloc e free)

Lo heap è un pool di memoria utilizzato per allocazioni dinamiche a runtime. Vengono utilizzate prevalentemente due funzioni: malloc(), prende/alloca memoria sullo heap e la free(), rilascia memoria nello heap.

Allocatore → gestore della memoria heap.
Cosa succede dietro le quinte delle chiamate di funzione (malloc/free)?
Quando parliamo di heap abbiamo a che fare con un allocatore che si occupa di cercare zone di memoria libere e restituirle: si occupa sia della malloc che della free.
L'allocatore lavora sui chunk:
Chunk → blocco fondamentale di memoria in cui possiamo trovare sia i dati che i metadati.
Metadati → sono dati usati dall'allocatore definiti nelle librerie per allocare spazio all'interno del nostro programma: servono all'allocatore per agganciare i chunk tra loro. I metadati sono prev_size e size. La prev_size è la size del chunk precedente se libero, mentre la size è la dimensione in byte del chunk che sto considerando.

Chunk occupato → è stato allocato.
Chunk libero → rilasciata la porzione di memoria.
Quando un chunk viene liberato, viene agganciato ad una lista di chunk esistenti (lista bi-linkata). Puntatore fd punta al prossimo chunk libero, mentre backword a quello precedente. I puntatori utilizzati solo nel caso di chunk liberi, mentre quando sono occupati questi puntatori sono sovrascritti con dati. Quando faccio una malloc, e quindi richiedo allocazione di memoria, quello che faccio è prendere un chunk libero dalla mia lista.
Se facessi buf1=malloc(256), vado ad allocare lo heap nel seguente modo:



Abbiamo il puntatore AV_TOP che punterà appena sopra i 256 byte allocati. Quando viene effettuata un'allocazione, viene eseguita un'operazione di unlink, ovvero viene tolto un nodo dalla lista e messo a disposizione del programma per essere, appunto, allocato.
`p=malloc(128)` → p è il memory pointer che punta appena sopra i metadati.
Il top chunk invece, rappresenta la memoria libera rimanente dello heap. Quando una malloc viene effettuata, il top chunk viene diviso in due: la prima parte diventa chunk richiesto, mentre la seconda parte è il nuovo top chunk.
Deallocazione:

- Quando un chunk viene liberato, il bit meno significativo del campo size nei metadati del blocco successivo deve essere cancellato.
- Inoltre, il campo prev_size di questo blocco successivo sarà impostato sulla dimensione del blocco che stiamo liberando
- Ci sono più liste di chunk liberi: ogni lista contiene i chunk liberi di una dimensione specifica
- Quando un chunk viene liberato, controlla se il chunk prima di esso è già stato liberato. Nel caso in cui il chunk precedente non sia in uso, viene unito al blocco che viene liberato.
- Quando viene effettuata una richiesta di allocazione di memoria, si cerca prima un chunk libero che abbia la stessa dimensione (o un po' più grande) e si riutilizza quella memoria. Solo se non è stato trovato alcun chunk libero appropriato verrà utilizzato il top chunk.

In questo caso l'attacco può essere effettuato durante l'operazione di unlink di un nodo dalla lista di chunk liberi. L'attacco ci consente di scrivere un valore arbitrario in una locazione arbitraria. Condizione: devo avere un overflow che mi permette di sovrascrivere i metadati necessari per eseguire l'attacco.
L'attacco sfrutta la possibilità di modificare il puntatore bk del nodo FD facendolo puntare al ret address. Successivamente si modificherà il puntatore bk del nodo P in modo che punti all'indirizzo del buffer contenente lo shellcode. Questo valore verrà copiato nel nodo BK. A questo punto si effettua la seguente modifica: FD→bk=BK ovvero si va a mettere l'indirizzo del buffer all'interno del return address.
Purtroppo, questo attacco al giorno d'oggi non è più sfruttabile contro le nuove versioni della glibc perché l'unlink è stato reso più difficile. Quello che viene fatto è un altro tipo di attacco che si chiama **House of Force.**

House of Force

Condizioni: richiede che si possa sovrascrivere il top chunk, che ci sia una malloc con dimensione controllabile dall'utente (dimensione di allocazione) e infine richiede un'altra malloc.
La variabile AV_TOP punta sempre al top chunk. Lo scopo è quello di sovrascrivere AV_TOP con un valore controllabile dall'utente. Durante una chiamata a malloc questa variabile viene utilizzata per ottenere un riferimento al top chunk (nel caso nessun altro chunk possa soddisfare la richiesta); ciò significa che, se controlliamo il valore di AV_TOP e possiamo forzare una chiamata malloc che utilizza il top chunk, controlliamo dove verrà allocato il prossimo chunk. Di conseguenza, possiamo scrivere byte arbitrari a qualsiasi indirizzo. Vogliamo assicurarci che qualsiasi richiesta (di grandi dimensioni arbitrarie) utilizzerà il top chunk. Per ottenere ciò, abusiamo dell'overflow nel programma per sovrascrivere i metadati del blocco superiore. Per prima cosa scriviamo 256 byte per riempire lo spazio allocato e infine sovrascriviamo la size con l'intero (senza segno) più grande possibile.

Dal lato pratico l'attacco viene svolto con la presenza di 3 malloc, una controllabile dall'utente e due invece fisse: tramite la prima allochiamo un buffer, ad esempio, di 256 byte.
Successivamente, sfruttiamo la prima strcpy che rappresenta la nostra vulnerabilità. La seconda malloc invece è quella controllabile dall'utente che ci permette di calcolare un delta che porterà l'AV_TOP a puntare appena sotto il ret address, lasciando spazio per i metadati. Infine, ho la terza ed ultima malloc che tramite la successiva strcpy mi permetterà di modificare il ret address facendolo puntare al mio codice. Non è necessario che ci sia un overflow anche su questa seconda strcpy.

25 OTTOBRE 2022

Use After Free Vulnerability

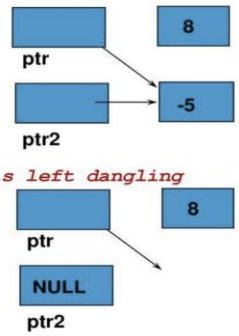
Le vulnerabilità use-after-free sono difficili da individuare durante la revisione manuale del codice perché richiedono la conoscenza dei pattern di allocazione e deallocazione che si verificano durante l'esecuzione del programma. La vulnerabilità è una di quelle temporanee, esiste solo ad un particolare punto nel tempo ovvero, quando un puntatore ad un oggetto che è stato liberato (`free()`) è deferenziato. Ciò può comportare una perdita di informazione, ma anche il fatto che l'attaccante potrebbe modificare posizioni di memoria indesiderate che possono portare potenzialmente all'esecuzione di codice.

Problema di aliasing: problema di sapere quanti puntatori puntano ad una struttura di memoria in un determinato stato temporale. Questa tipologia di attacco può essere utilizzata sia sullo heap che sullo stack, anche se su quest'ultimo è più raro.

Leaving a Dangling Pointer

```
int* ptr = new int;
*ptr = 8;
int* ptr2 = new int;
*ptr2 = -5;
ptr = ptr2;

delete ptr2;     // ptr is left dangling
ptr2 = NULL;
```



Il problema potrebbe essere peggiore in programmi grossi in cui ho tanti puntatori ad una struttura. Buona norma sarebbe quella di mettere un puntatore a NULL.

Example of Use After Free Vulnerability

```
char *retptr() {

char p ,*q ;
q = &p ;

return q ; /* deallocation on the stack */

}
```

```
int main( ){

    char *a , *b;
    int i ;

    a = malloc(16) ;
    b = a + 5;

    free(a) ;
    b[2] = 'c' ; /* use after free */

    b = retptr( ) ;
    *b = 'c' ;    /* use after free */

}
```

In questo caso si hanno due use-after-free. La prima è dovuta dal fatto che al puntatore b viene assegnato l'indirizzo di a+5, ma successivamente a viene deallocato tramite free, quindi b[2]='c' rappresenta una vulnerabilità use-after-free poiché b punta ad una zona non esistente (deallocata) e quindi è un dangling pointer: eccezione SEGFAULT. La seconda use-after-free, invece, avviene quando alla variabile b viene assegnato il valore ritornato dalla funzione retptr(), il problema che dopo la return lo stack viene deallocato e quindi b punterà ad una zona dello stack che è stata deallocata. In questo caso non ho un'eccezione perché la zona è allocata, funziona, quindi verrà scritto c, ma quando richiamerò un'altra funzione questo valore verrà sovrascritto.
Tracciabilità difficile perché su un programma complesso dovrei tracciare tutti i riferimenti a quel puntatore. Durante l'analisi statica io simulo l'esecuzione del programma e quindi non so tutti i valori dei puntatori, anche perché molti sono calcolati a runtime. Posso generare molti falsi positivi. I tool di use-after-free lavorano sui puntatori alle strutture.

Codice LAB – Buffer Overflow e uaf:

#include <stdlib.h> #include <unistd.h> #include <string.h> #include <sys/types.h> #include <stdio.h> struct auth {	

```
  char name[32];
  int auth;
};

struct auth *auth;
char *service;


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

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

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

     if(strncmp(line, "auth ", 5) == 0) {
        auth = malloc(sizeof(auth));
        memset(auth, 0, sizeof(auth));
        if(strlen(line + 5) < 31) {
           strcpy(auth->name, line + 5);
        }
     }
     if(strncmp(line, "reset", 5) == 0) {
        free(auth);
        BISOGNAVA AGGIUNGERE:
        auth = NULL;
     }

     if(strncmp(line, "service", 6) == 0) {
        service = strdup(line + 7);
     }


      if(strncmp(line, "login", 5) == 0) {
        if(auth->auth) {
           printf("you have logged in already!\n");
        } else {
           printf("please enter your password\n");
        }
     }
  }
}
```

Riceve una stringa di 128

Controlla che non ci sia un buffer overflow (controllo di <31) e poi elimino i 5 caratteri di “auth “

Comando che si chiama service: è un puntatore che si occupa del servizio richiesto che viene allocato automaticamente

Comando login che copia inline il valore del login: guardo il flag di auth->auth (se 1 vuol dire che sono già loggato) e se è 0 faccio inserire la password

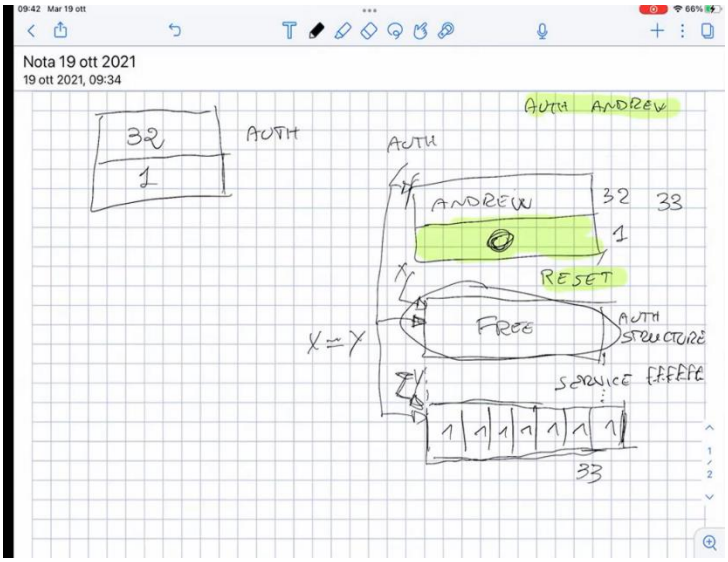
MA quand’è che viene messo a 1 in questo programma? MAI = è ciò che vogliamo simulare => si vuole modificare della memoria deallocata in modo che il sistema pensi che ci siamo loggati: inserire una funzione di write

Abbiamo a disposizione AUTH, RESET, SERVICE e LOGIN: bisogna mapparle per ottenere una sequenza che ci permetta di essere loggati, ma quale sequenza?

1. A (allocazione) = “auth”
2. D (deallocazione) = “reset”

3. W (write) = “service” MA è deallocata quindi devo riuscire ad ottenere l’indirizzo della deallocata proprio nel chunk di Service o perlomeno arrivare ad ottenere gran parte di service (perché auth è più piccola), se non tutta, spostandola nello stesso luogo di auth => per avere una parte che overlappi
4. U (utilizzo/uso) = “login” che a questo punto dato l’overlap mi dirà che sono già dentro

Allora alloco nello heap la mia struttura service [ffffff] che fa “resuscitare” il dangling pointer e l’uso successivo di auth permette di entrare nel sistema.



8 NOVEMBRE 2022

Slide-defense

In pratica, il problema del buffer overflow e dell’heap overflow è quello dei caratteri di controllo: questi, infatti, risiedono nello stesso canale di comunicazione dei dati inseriti dall’utente; come questi vengano sfruttati dipende dal tipo di attacco che si va ad implementare. Nel nostro caso (nei due attacchi che sono stati fatti precedentemente vedere) i caratteri di controllo erano il RetAddres (stack) e i metadati (heap). Questi teoricamente non dovrebbero essere conosciuti. L’attaccante è quindi in grado di controllare questi dati utilizzati dal programma stesso ed in questo modo riesce, ad esempio, a cambiare il flusso del programma. Perché si possono controllare i dati? Perché i dati di input sono mischiati con questi metadati → channel problem. Per questo motivo, quando c'è un errore programmativo che permette di controllare questi metadati c'è un problema.
Le difese costruite vanno in parte a mitigare questi problemi, due esempi: **Memory Safety** e **Type Safety**, che sono propri di alcuni linguaggi come python e java. Queste sono due proprietà che quando vengono controllate su tutte le istruzioni permettono di evitare questi tipi di problemi (memory errors). Questo è il punto in cui arrivare per eliminare i problemi discussi precedentemente. Qual è il problema di questi due tipi di difese? Le performance, per questo ad esempio C non le usa.

Tecniche di difesa implementate all’interno del sistema operativo o nel compilatore, ad esempio: **Stack canaries** (difesa a livello compilatore, servono per proteggere il ret address) e **Address Space Layout Randomization** (livello del sistema operativo). Brevemente: ASLR toglie le assunzioni sulle posizioni dei buffer per il buffer overflow quindi non si può più fare affidamento sul debug per cercare di capire dov'è il buffer da attaccare poiché ogni volta è in una posizione diversa. Tuttavia, queste difese mitigano, non escludono al 100% il problema. Vanno ad analizzare come l’attacco viene portato a termine (fasi dell’attacco) e cercano di intervenire in una

di queste fasi per disarmare l'attacco. **DEP** (Data execution Prevention) tecnica a livello del sistema operativo, le area data non sono più eseguibili, contrariamente alle assunzioni fatte fino ad ora. Non posso eseguire perché si è detto "qui ci devono essere solo dei dati, non sei abilitato ad eseguire" → annulla la possibilità di code injecting.
Queste tecniche limitano la superficie di attacco.

Attacco **ROP** (return-oriented programming): non posso fare injecting di nuovo codice allora vado a sfruttare del codice già esistente in giro per la memoria: studi hanno dimostrato che posso creare qualsiasi tipo di programma prendendo codice da un po' in giro → per contrastarlo: **CFI** (control flow integrity). Questa tecnica riguarda l'integrità di esecuzione del flusso di un determinato programma. In questo caso il codice controlla se stesso (tramite instrumentazione) ovvero, se ci si sta spostando in una zona corretta oppure no: lavora a livello compilatore. Da windows 10 applicata già anche sul kernel -- tecnica comportamentale: cerco di capire cosa farà il codice → posso rappresentare i pattern del mio programma attraverso dei grafi quindi posso definire un comportamento del programma e seguendo ciò che vedo posso capire se ci sono state delle tipologie di attacco. Quando si introducono le difese è importante andare a definire il threat model in cui si può sviluppare quell'attacco.
Secure coding → Costruire librerie particolari, che tengano conto dei possibili attacchi. Concetto di secure by design, il software andrebbe progettato fin da subito sicuro, quindi progettare magari delle librerie che contengono dei meccanismi di sicurezza che mi permettono di bloccare dei possibili attacchi (anche che non conosco). Altrimenti sarà sempre una rincorsa tra attacchi sviluppati e difese introdotte per limitarli. Le difese sostanzialmente sono delle patch introdotte per limitare gli attacchi.

Tutte queste tecniche NON sono risolutive: infatti man mano che vengono proposte ci sono sempre nuove modalità per eluderle da parte degli attaccanti; inoltre, molte di queste modalità hanno delle difficoltà di implementazione: un minimo difetto di implementazione permette una più facile elusione della tecnica. Qui si nota l'asimmetria tra attacco e difesa, la difesa ha una maggiore complessità di implementazione per vincoli di performance e necessità di copertura totale.

Memory Safety

I low-level attack sfruttano la mancanza di memory safety. Già presente in linguaggi tipo java. È divisa in due sotto proprietà: **temporal safety** (use after free) e **spatial safety** (stack overflow, heap overflow). Un linguaggio possiede la memory safety se incarna entrambe queste due proprietà su tutti i programmi scritti nel linguaggio. Un programma memory safety è meno "forte", perché solo quel singolo programma incarna entrambe le sotto proprietà. La memory safety permette di controllare che i puntatori puntino ad una zona di memoria ben definita. In particolare:

- Spatial safety: legata allo stack overflow e all'heap overflow.
- Temporal safety: legata alla use-after-free, problema temporale nell'ordine delle istruzioni che io vado ad eseguire.

a. Spatial Safety

Per esempio, quando si usano i puntatori questi vengono creati tramite modalità standard:

`p = malloc(…)`, oppure `p = &x`, oppure `p = &buf[5]`

successivamente, un puntatore deve appartenere ad una zona di memoria: vanno definiti i punti di ingresso e le memorie a cui questi puntatori appartengono.

La spatial safety è la proprietà che ci permette di definire lo spazio concreto di allocazione: dobbiamo definire lo spazio del puntatore; in particolare, può essere visto come una tripla (**p**,***b***,***e***) dove:
• **p** è il puntatore vero e proprio: si può muovere nella regione allocata (diversamente da *b*)
• ***b*** è la base della regione di memoria che può accedere
• ***e*** è l'estensione (bound) di quella regione

Inoltre, si definisce un permesso di accesso per questa regione:

iff ***b*** ≤ **p** ≤ ***e***-`sizeof`(`typeof`(**p**))

ogni volta che svolgo delle operazioni (controllo che si effettua al momento della dereferenziazione) bisogna controllare che il puntatore sia compreso all'interno della memoria allocata a cui si fa riferimento. Le operazioni con cui **p** si modifica sono, ad esempio, l'aritmetica dei puntatori. La spacial safety permette di definire una zona di memoria allocata attraverso le varie operazioni svolte.

Problema di spazio: creo un puntatore, ci faccio delle operazioni, ma poi questo può andare oltre la dimensione definita dal puntatore, a meno che il programmatore non metta dei controlli.
Quindi si va a definire una regione di memoria che permette di controllare in automatico che il puntatore sia sempre all'interno della zona di memoria in cui questo puntatore è stato definito.
Questa tecnica è implementata ad esempio in Java.
Il puntatore diventa un metadato visto come una tripletta **(p, *b*, *e*)**: ***b*** sostanzialmente è quello che mi restituisce la malloc, **p** rappresenta la posizione attuale del puntatore, mentre la ***e*** rappresenta la quantità allocata (4 byte nel caso di int). Il puntatore punta sull'ultimo byte della zona allocata.

```
int x;          // assume sizeof(int)=4
int *y = &x;    // p = &x, b = &x, e = &x+4
int *z = y+1;   // p = &x+4, b = &x, e = &x+4
*y = 3;         // OK: &x ≤ &x ≤ (&x+4)-4
*z = 3;         // Bad: &x ≤ &x+4 ≤ (&x+4)-4
```

```
struct foo {
 char buf[4];
 int x;
};
```

`int *z=y+1` → vuol dire che mi sono spostato di 4byte => sto definendo una nuova zona di memoria che è allocata rispetto al valore che ho assegnato al puntatore.
Cosa si fa successivamente? Si va ad accedere alla zona di memoria: `*z=3` – la x rappresenta lo spostamento della p nella zona di memoria: la disuguaglianza, in questo caso, **non** va bene perché z punta fuori dallo spazio di dimensione.

(p, *b*, *e*): fat pointer => usati anche in C in alcune zone dei programmi (non in tutto il linguaggio né in tutto il programma, ma magari sono nelle parti che potrebbero essere più soggette a certi tipi di attacchi → selezionare zone del programma su cui "applicare" la memory safety).

ARTICOLO: stack bounds protection with low fat pointer

Low Fat Pointer: si vuole ottimizzare la memoria perché i (p,b,e) vengono inseriti in una shadow memory e ciò è molto pesante → si vuole aggiungere una tecnica per inserire le codifiche della tripletta direttamente dentro il puntatore. Ciò ha conseguenze non banali: il problema principale è che il puntatore non può più puntare a **tutta** la memoria quindi la memoria viene divisa in chunk "abilitati" a cui un puntatore può puntare; ciò porta ad una nuova organizzazione della memoria virtuale.
Implementazione: https://github.com/GJDuck/LowFat

b. Temporal Safety

Assicura che la regione di memoria sia ancora attiva. Definiamo la regione come definita (allocata e attiva) o non definita (zone deallocata). Una violazione alla temporal safety si verifica quando cerchiamo di accedere ad una memoria non definita.

```
int *p = malloc(sizeof(int));
*p = 5;
free(p);
printf("%d\n",*p); // violation
```

The memory dereferenced no longer belongs to p.

→ È una proprietà molto difficile da far rispettare (enforce).

Il problema in questo caso è quello dell'aliasing ovvero ho dei puntatori a delle zone di memoria, poi vado a deallocare le zone a cui puntavano e quindi mi rimangono questi "dangling pointer". Quindi se noi vogliamo assicurare la temporal safety, dobbiamo creare un qualcosa di simile al garbage collector (tracciamento puntatori su memorie allocate), tuttavia questa cosa ha un costo elevato a livello di performance.

Infatti, in java (come in tanti linguaggi) questa difesa è implementata dal garbage collector che tiene traccia di tutti i puntatori allocati dinamicamente attraverso una tabella che viene creata, quindi durante l'esecuzione del programma. *Questa gestione, però, è molto costosa in termine di spazio, ma anche di computazione perché ad ogni momento in cui ho una dereferenziazione allora deve essere svolto un controllo.*

A livello statico alcuni puntatori non li posso classificare perché il loro valore sarà visibile solo a livello runtime. Se sapessi a livello statico tutti i valori dei puntatori, a quel punto il garbage collector potrebbe essere eliminato. Posso provare a fare delle approssimazioni per quei valori che non conosco, ma potrebbero portare a dei falsi positivi.
Quindi l'idea è quella di approssimare quanto più possibile il lavoro svolto dal garbage collector. Traccio per alcune regioni potenzialmente pericolose i puntatori a runtime.
Posso definire delle zone sensibili di memoria tramite delle euristiche. La definizione delle zone pericolose di solito viene fatta ad opera dell'analista di sicurezza.

Molti linguaggi moderni sono memory safe (hanno entrambe le proprietà) e, alcuni di questi, sono anche type safe perché ci permette di definire una proprietà che non ci dice solo che stiamo scrivendo oltre un buffer, ma anche quale dato sintattico stiamo cercando di scrivere in quel buffer.

Memory Safety For C

Non possiamo portare il C ad essere completamente memory safe, ma possiamo cercare di rendere il programma singolo memory safe (o anche solo delle parti, come già detto). È il compilatore che va ad aggiungere l'instrumentazione, sulla base dei modelli implementati. Il compilatore potrebbe aggiungere il codice per verificare le violazioni. Un accesso out-of-bounds risulterebbe in un fallimento immediato.

Le performance sono il fattore più limitante, viene introdotto un elevato overhead. Ovvero, c'è una soluzione, ma applicata a linguaggi come C e C++ crea delle poor performances. → trade-off

Progess – degli improvement
Negli anni sono stati introdotti sistemi in grado di migliorare le performance mantenendo la stessa efficienza.

- Ccured → no supporto per le librerie: i compilatori non potevano definirne la safeness
- Softbound/CETS → aumenta estremamente lo slowdown ma c'è un supporto completo (anche le librerie)
- Progetto Intel MPX → memory safety tramite supporto hardware: come migliorare delle performance software? Ci si aiuta con l'hardware.

In conclusione, i linguaggi di programmazione dovrebbero essere adattati per garantire la memory safety.

Type Safety

Proprietà molto più forte della memory safety che ci permette di identificare alcuni bug ulteriori. In alcuni casi la memory safety non è sufficiente, si possono ugualmente avere problemi legati alla sicurezza. È una proprietà che enforza (enforces: fa rispettare/applica) il tipo di un oggetto; ogni oggetto, quindi, ha un tipo. Controlla che un determinato tipo non interagisca con tipi differenti, a meno che non sia specificato diversamente.

Prototipo di un puntatore a funzione:
In questo esempio si può vedere perché la type safety è più forte della memory safety.

```
int (*cmp)(char*,char*);
int *p = (int*)malloc(sizeof(int));
*p = 1;
cmp = (int (*)(char*,char*))p;
cmp("hello","bye"); // crash!
```

Memory safe, but not type safe

Il programma crasha perché fa riferimento a p che contiene "1" che non è fuori dallo spazio dell'indirizzamento; quindi, sarebbe corretto dal punto di vista della memoria come vista prima, ma crea problemi di tipo.

Memory safe → nessun tipo di overflow
Type **un**safe → perché ho assegnato un puntatore a intero a un puntatore a funzione: errore per un problema di tipo, non si possono svolgere operazioni tra questi tipi differenti (per come sono definiti)

Dal punto di vista della memory safety l'esempio rispecchia ciò che abbiamo definito prima. La cosa che non va bene è che sono andato ad assegnare ad un puntatore di tipo funzione, un puntatore di tipo intero.

Il type safety permette di avere una visione più sofisticata del programma a livello semantico. Non solo permette di trovare il mismatch tra i tipi, ma di lavorare ad un livello più astratto sul flusso dei dati che, ad esempio, passano da un tipo all'altro. Posso definire policy all'interno del programma di tipo type safety che permettono di definire e costruire un flusso di esecuzione all'interno del mio programma stesso. *Effettuare controlli su tutto il codice diventerebbe pesante, ma si possono definire delle regole che si attivino solo su certi tipi e/o operazioni → si possono modellare molte cose.*

Per esempio, per bloccare il buffer overflow si può aggiungere un return di tipo stack → ma questo ha un costo per via dei numerosi controlli (non solo vedere se posso scrivere lì, ma anche se va bene quel tipo specifico)

Inoltre, si possono controllare delle cose dinamicamente (dynamically typed) che è molto forte ma molto costoso, oppure staticamente (statically typed). Come ottenere queste proprietà di safety? Serve progettare dei linguaggi con certe caratteristiche. Ad esempio, vengono costruite le invarianti[1] che devono essere rispettate attraverso il programma. Tramite esse si possono controllare i tipi astratti e si può creare l'isolation all'interno dei dati (si possono definire le regole che il linguaggio deve rispettare nelle sue operazioni). Il programmatore diventa protagonista di questa sorta di flusso di dati del type safety.

Esiste un linguaggio **JIF (Java with Information Flow)** → permette di definire all'interno del programma dei flussi di dati. In questo linguaggio è possibile bloccare molti degli attacchi già mostrati, grazie alle definizioni dei flussi dei valori accettati (e non: alcuni tipi possono fare certe cose, altri no e vanno bloccati).
Esempio: il flusso "carta di credito >> network component" è permesso? Se dico di **no** posso dire che i miei dati della carta non possono uscire.

Perché C non è type safe? Per questioni di performance. Definire i flussi vuol dire avere ben chiari i tipi. In java c'è una struttura che lo consente, mentre C non ha nulla e quindi si costruirebbe un qualcosa di molto pesante.
Futuro? Go, Rust, Swift forniscono feature simili a C/C++ pur essendo type safe.

Con il type checking possiamo andare a bloccare il buffer overflow. Nel nostro caso faccio controlli solo sugli user data e il control data, definisco quindi una policy in cui evidenzio il fatto che non posso passare da uno all'altro.

La type safety se costruita correttamente può andare anche a inglobare la memory safety.

15 NOVEMBRE 2022

Avoiding Exploitation

Strategie difensive di due tipi: la prima, cerchiamo di rendere difficile l'attacco. Si cerca di disinnescare l'attacco nel momento in cui si cerca di introdurlo: queste tecniche sono quindi mirate alla natura dell'attacco. Poi ci sono tecniche più generali, il cui scopo principale è proprio evitare i bug che possono causare gli attacchi.
No zeroes → questo perché lo shellcode non può contenere un byte a zero perché altrimenti viene interpretato dalla strcpy come un fine stringa. Quindi devo essere significativi a livello semantico.

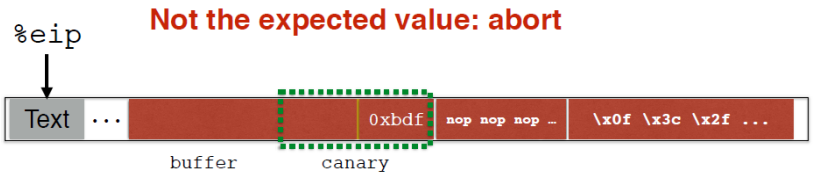
In questo contesto parliamo di attacco di **injection** dove si va a sovrascrivere il return address (%eip). Infine (nel caso di sistemi virtuali), andiamo a cercare di trovare l'indirizzo del buffer, tramite ad esempio l'utilizzo di nopsled se non siamo sicuri dell'indirizzo su cui saltiamo.
Come possiamo rendere questi attacchi più difficili? → Caso migliore: complicare l'exploitation cambiando le librerie, il compilatore e/o il sistema operativo. Non dobbiamo cambiare il codice dell'applicazione. Si sistema il design dell'architettura.

Difesa Canarino

[1] Sono delle policy che permettono di definire il flusso dei dati da un oggetto a un altro: definire dei tipi astratti che rappresentano il concetto dei flussi di dati

Chiamata così perché i lavoratori delle miniere utilizzavano i canarini per trovare la possibile presenza di gas; quindi, si tratta di “spie” che rintracciano un qualcosa di maligno. Nascono come difese con lo scopo di accorgersi della presenza di un overflow. Tecniche utilizzate da tutti i sistemi operativi moderni.

Vengono inserite tra due parti: per dividere le due parti in cui non vogliamo che ci siano interferenze.



Dopo la funzione di prologo: funzione di canary aggiunta in automatico dal sistema (push canary) e dopo aver impostato il valore faccio check canary dove controllo il valore segreto (perché se no l’attaccante potrebbe benissimo sostituirlo nel suo injection e nessuno se ne accorgerebbe) e random (posizionato in una memoria non accessibile del programma) del canarino.

Il canarino è un valore utilizzato per proteggere il return address. Viene inserito tra le variabili locali e il safe frame pointer ed è inizializzato con un valore random.
Quindi quando faccio l’overflow, succede che vado a sovrascrivere anche il canarino e quindi la “spia” mi dice che il canarino è stato sovrascritto. Ovviamente l'attaccante non deve conoscere l’esatto valore del canarino, altrimenti può sovrascriverlo tranquillamente.
Il canarino viene pushato sullo stack all’entrata nella funzione prima delle variabili locali. All’uscita della funzione il compilatore verifica che l’ESP+4 ovvero dove c’è il canarino, controlla che il valore di quest'ultimo sia quello corretto.
Difesa semplice, ma efficace. Se l’attaccante non riesce ad indovinare il canarino, non riuscirà mai a sovrascrivere il return address. Non costa molto dal punto di vista dell’overhead. Ad esempio, riesco a saltare il canarino se ho un puntatore che mi permetta di farlo ovvero se ho la possibilità di scrivere su questo puntatore. Il canarino può essere spostato anche sulle variabili locali per proteggerle, in questo caso dovrò salvarmi i valori delle variabili locali.

STACK: viene inserito il canary

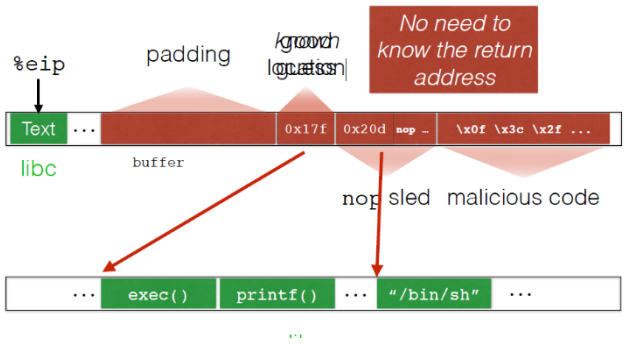
Argv1
Argv2
Ret
Sfp
Canary
Local variable

Tre tipi di canarini:

- **Random canaries** → quelli appena descritti
- **Terminator canaries** → ad esempio potrei mettere /0 sul canarino in modo che la strcpy quando sta copiando trova lo zero e termina, così che non venga raggiunto il return address.
- **Random XOR canaries** → uso un numero random per il canarino. Il return address viene xorato con il valore del canarino all’entrata della funzione. Per la proprietà dello Xor se lo rifaccio con lo stesso valore Random, ottengo nuovamente il return address originale. Quindi se qualcuno cambia il return address, quando applico lo XOR ottengo un risultato

diverso e punterà ad un'area di memoria che non esiste e quindi probabilmente il programma crasha. Permette di lavorare senza appoggiarsi a una memoria.
Si sceglie sempre un valore del canarino conosciuto come base e faccio lo XOR con il return address attuale del record di attivazione quindi dentro il canary metto il risultato dell'operazione. Quando ritorno dalla mia funzione (non push ma XOR) non faccio check ma di nuovo XOR → se il return address non è cambiato allora tutto ok.

Defense: Rendere lo stack e lo heap non eseguibile
→ in questo caso non verrà più eseguito il mio shellcode iniettato nel buffer, quindi posso comunque iniettarlo, ma poi non verrà eseguito. Quindi, anche se il canarino viene sorpassato, il mio attacco non funzionerà ugualmente.
Che cosa facciamo quindi? Abbandoniamo lo shellcode. Dobbiamo prima mandare in esecuzione del codice che superi questa protezione. Cerco di utilizzare del codice che è già presente nella macchina, nella memoria del mio processo c'è del codice che posso utilizzare.
Return-To-Libc → non torno più sul mio shellcode, ma vado a cercare in memoria e utilizzo il codice che è sempre presente in memoria ovvero quello delle librerie.
L'obiettivo è sempre quello di andare a sovrascrivere il return address che è quello che ci permette di modificare il flusso. Dobbiamo andare in memoria a cercare gli indirizzi delle funzioni ad esempio "system" o "exec". La system() è una funzione composta da due parametri, "CMD" e il parametro del comando. Devo andare a ripristinare il layout della funzione system, ovvero mettere i parametri che voglio siano eseguiti dalla funzione system.
Nuovo injection vector:
[Padding(fino al ret address)][Indirizzo system del processo][/Bin/Sh(1° parametro system)][Null]
In questo modo riesco ad eseguire ancora il mio codice scavalcando la difesa della DEP (Data Execution Prevention). → permette di ridurre la superficie d'attacco in maniera sostanziale



libc:
system("cd", "…")
execv( ),
fork( )

il return address viene modificato per andare direttamente alla system perché non ho il return address (normalmente se svolgessimo la funzione tramite una call questo ci sarebbe ma è un attacco → nessuna call, dobbiamo simulare e immaginare cosa faccia la system)

Come reazione a questa cosa è stata introdotta una nuova tecnica di difesa che è l'Address Space Layout Randomization (ASLR). È una tecnica che permette, concettualmente, di caricare un programma sempre ad indirizzi diversi, quindi, questo è sempre caricato in luoghi diversi e non si possono fare previsioni sulla memoria virtuale utilizzata. I problemi si spostano sul metodo di implementazione di questa tecnica: basta rilevare un singolo elemento e poi gli altri si potrebbero scoprire (leakage information). Quindi permette di caricare un processo e le librerie ad indirizzi sempre differenti → Con questa tecnica io non so più dove sarà la system()[2] che volevo utilizzare

[2]esecutore di comandi

nel caso precedente. Questa tecnica viene anche utilizzata come difesa per l'indirizzo del buffer che andavamo a sfruttare per modificare il flusso del programma.
Su architettura a 32 bit è possibile sfruttare la brute force, mentre sulla 64 bit no, lo spazio di indirizzamento è troppo grande.

- Spraying attack: metto lo shellcode (delle sue repliche) in modo da riempire tutta la memoria con gli shellcode e poi nel ret metto un indirizzo a caso → potrei riuscire a beccare il mio shellcode e riesco a sincronizzarne l'esecuzione (usato di solito nei sistemi a 32 bit)

Un modo per ridurre in questo caso ulteriormente la superficie d'attacco è quello, ad esempio, di caricare in memoria solamente le librerie che vengono effettivamente utilizzate (guardo quello che fa il mio programma e carico in memoria solo le funzioni che effettivamente il programma chiama, quindi il num di funzioni che l'attaccante potrebbe chiamare diminuisce – e poi ci sono funzioni più e meno potenti: printf, al contrario di system ed execv, è poco potente e poco utile), quindi se il programma non esegue una system, non posso utilizzarla. Però se si attaccano programmi potenti il problema rimane, risultato: ASLR.
Come tecnica di attacco in risposta a questa limitazione, è stato introdotto il ROP.

Quindi:

Injection code ====> DEP (Data execution Prevention) Return into libc ====> ASLR (e function reduction) Function reduction (attack surface reduction) ====> ROP

ROP (Return Oriented Programming)

Idea: piuttosto di usare una singola funzione libc per eseguire il mio shellcode, metto insieme pezzi di codice esistente chiamati gadgets per farlo, costruendo così un programma vero e proprio. **Programma Turin Completo** è un programma con cui posso costruire tutto, anche un compilatore.
Si entra nel codice della funzione e si usano pezzi del suo codice per comporre codice arbitrario. Quindi non chiamo più la funzione, ma vado al suo interno e utilizzo suoi pezzi di codice.
Mettiamo insieme pezzi di codice (frammenti) che possiamo chiamare gadget. Questo perché se dis-assembliamo il codice delle funzioni, sono costituiti da una marea di istruzioni assembly.
Challenge:

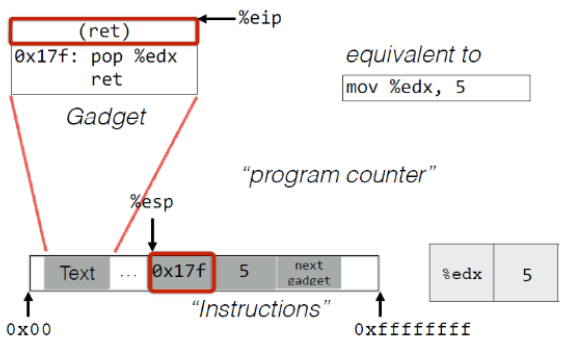
- Devo trovare i gadget che mi servono in memoria
- Devo avere un meccanismo per metterli insieme

Il ROP non è una tecnica di exploitation, ma si tratta di una tecnica applicata al buffer overflow oppure all'heap overflow.

I **gadget**[3] sono gruppi di istruzioni che terminano con una `ret`**.** Per ogni gadget devo definire dei parametri: come si prendono questi valori? I parametri vengono presi con l'istruzione POP dallo stack. → Lo stack è visto come la memoria RAM dove caricherà il programma.
L'ESP[4] è ciò che manda avanti l'esecuzione di questa macchina astratta (vettore di attacco), definisce quindi il program counter. I gadget vengono agganciati tra loro tramite le funzioni ret, ed ognuno può essere visto come un blocco funzionale, ovvero come un insieme di istruzioni che riceve dei parametri (i valori dei registri e della memoria a cui sto facendo riferimento). Come già detto, il gadget riceve i parametri attraverso le funzioni pop e li inserisce sullo stack.
I gadget sono identificati dall'indirizzo (del nostro programma vittima) e da un valore che viene caricato tramite pop (parametro).

[3] *unità di istruzioni che vanno eseguite (?)*
[4] indirizzo dell'elemento in cima allo stack

Esempio => programma a cui vogliamo ambire: `mov'%edx,'5` ovvero mettere 5 nel registro indirizzi. Bisogna trovare dei gadget che simulano questo.



Indirizzo del gadget: 0x17f dove in memoria c'è il *mio* gadget (ho trovato l'indirizzo) quindi quando con l'esp ritorno lì allora faccio la pop %edx (gadget) dove ho messo 0x5 (valore che posso sfruttare come attaccante) e poi fa ret (dove io posso far riflettere l'esecuzione di un altro gadget). Come sono concatenati i gadget? Tutti devono terminare con ret che è ciò che mi devo trovare sullo stack perché l'esp prende il nuovo ret e passa al nuovo gadget. Quindi componendo questi frammenti di istruzione si può ricostruire il programma che si vuole simulare.

Side Effect dei gadget → dato che sono già presenti nel codice e non li stabilisco io, i gadget potrebbero contenere istruzioni che non mi interessano e che potrebbero anche modificare il risultato che volevo ottenere inizialmente.
Esistono tool che automatizzano la ricerca dei gadget all'interno del mio programma. ROPgadget Tool ti dice dove trovare i gadget che fanno cose specifiche e permette i salti disallineati → così rop diventa turing compatibile

Oppure ROP compiler → scrivo il mio programma in alto livello e lui me li traduce in maniera automatica:
Bubble sort => ROP compiler => stack
^^ libc/pc

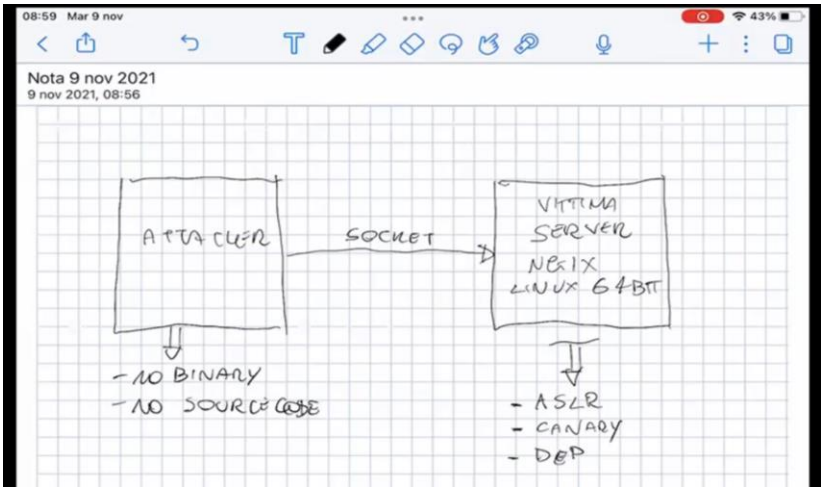
Intel x86 (architetture cisc) è un instruction set denso perché ogni possibilità ha una codifica particolare. Istruzioni che vanno da 1 byte a 13 byte. Questa variabilità mi permette tante cose, esempio: se prendo un'istruzione da 13 e ci salto in mezzo, avrò una codifica differente, ma sempre valida di questa istruzione. È come se avessi infiniti gadget nascosti.
A seconda di dove parto avrò sempre una codifica diversa. Su ARM invece non è possibile questa cosa, perché ha le dimensioni fisse per una codifica, se sono 4 byte e io salto in mezzo a questi 4, quella codifica non ha alcun significato.

Blind ROP: basta una piccola breccia nel sistema (legato alle performance) che permette di fare un blind ROP (non so nulla), ma con questa tecnica riesco a dumpare via rete il mio programma su una macchina remota. http://www.scs.stanford.edu/brop/

29 NOVEMBRE 2022 (22 nov saltata)

Hacking Blind - ARTICOLO

Difesa: Randomizzazione della locazione del codice su macchine a 64-bit che rende gli attacchi molto difficili. Gli attacchi recenti e pubblicati sono spesso per versioni a 32 bit di eseguibili.
Blind ROP è stata costruita sfruttando il punto debole di un server reale (nginx). Non si ha a disposizione il processo, viene fatto tutto alla "cieca" ovvero senza conoscere nulla della macchina vittima. L'attaccante non sa quali sono i gadget all'interno di questa macchina, non ha a disposizione né il binary né il source code. La macchina vittima ha ASLR, canarino e DEP. Sulla macchina vittima è presente un buffer overflow.



Lo scopo dell'attaccante è quello di andare a trovare in modo blind un gadget che dovrebbe esistere nel binario, che potrebbe essere una write, per poter copiare il binario della macchina vittima sulla propria – in particolare l'attaccante vuole leakare il binario (perché dopo da lì si possono trovare tutti i gadget); quindi si usa un gadget che va in memoria di questa macchina e permette di leggere indietro il binario. L'attaccante deve indovinare il socket descriptor.

```
write(sd, buffer, length)
```

`sd` è il socket descriptor, `buffer` è l'indirizzo di memoria dove inizia il binario da dumpare, `length` la lunghezza che voglio scrivere; fatto ciò, l'attaccante può applicare il classico ROP. Tuttavia, si devono anche scavalcare tutte le protezioni viste le volte precedenti (cioè superare la difesa del canarino e quella dell'ASLR).
L'attacco è permesso da due nuove tecniche:

- Abbiamo bisogno di una primitiva di stack reading per poter andare a leggere il canarino, il return address ecc. Il return address in questo caso ci dà informazioni per sapere dove è caricato il programma. Quindi lo stack reading ci permette di trovare informazioni molto sensibili.
- Blind ROP → questa tecnica localizza in remoto i gadget ROP.

THREAD STACK:

ARGV1			
RET			
SFP			
25	36	77	88
BUFFER			

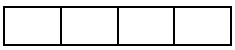
=> valori del canary (esempio)

si fa uno stack reading → ho da 0 a 256 tentativi per beccare ognuno dei valori, quando sbaglio il valore la connessione viene staccata (il thread viene chiuso), ma se si becca il valore di quel byte allora la funzione va avanti normalmente – anche in caso di overflow.

L'implementazione dello stack reading sfrutta una debolezza intrinseca nell'implementazione dell'ASLR. Questa debolezza riguarda il fatto che quando io implemento il canarino o l'ASLR scelgo un valore che è valido per quel processo nel momento in cui viene caricato. Quando parliamo di server di rete (nginx), abbiamo a che fare con una programmazione multithread. Ogni thread ha il suo stack, gestisce il suo flusso di esecuzione. La vulnerabilità sfruttata è il fatto che se un thread crasha, non porta al crash dell'intero processo. Quindi, quando crasha, il processo non viene ricaricato e i valori di canarino e ASLR rimangono uguali. Comportamento visibile

dall'esterno perché quando il thread crasha la connessione viene chiusa. Se vado ad eseguire qualcosa di sbagliato, ad esempio dando un valore sbagliato al canarino, verrà ucciso il processo e quindi chiusa la connessione. In questo modo io posso dedurre se ho sbagliato il valore (crash del thread) oppure l'ho azzeccato e quindi il thread continuerà l'esecuzione.

L'attaccante prova ad indovinare i singoli byte del canarino, che sarà una word di 4 byte. Ciascun byte ha un valore da 0 a 255. Se lo trova il programma va avanti e capisco che ho indovinato il valore, se non lo trova, il programma crasha cioè la connessione viene chiusa. Una volta indovinato il byte, procede a tentare di indovinare gli altri.
Il canarino non viene cambiato ad ogni crash, è univoco (non viene cambiato per questioni di efficenza). Se venisse cambiato ad ogni crash ci sarebbe un problema di overhead e di complessità (problemi di spazio di memoria, computazione ecc).
Una volta indovinato tutto il canarino provo ad andare ad indovinare byte a byte il return address. Il guessing del return address potrebbe anche non essere precisissimo.

→ dopo devo cercare di ottenere i valori dentro il RET 
sempre usando una tecnica di stack reading - "Return address guessing" per l'ASLR.

Gadget necessari per la chiamata alla funzione write:
prima challenge
1) pop rdi; ret (socket) → scrivo sulla socket descriptor
2) pop rsi; ret (buffer) → dov'è caricato il programma in memoria
3) pop rdx; ret (length) → quanto è lungo
=> queste tre sono anche l'input della write(sd, buffer, len)
seconda challenge
4) pop rax; ret (write syscall number)
5) syscall (invocazione)

Problema: bisogna costruire una tecnica blind che permette di trovare in memoria i gadget per eseguire questa write e poi eseguirli.

Per prima cosa devo andare a trovare le prime 3 pop che solitamente sono abbastanza presenti (quindi facili da trovare) e poi fare uno sforzo ulteriore per andare a trovare le ultime due pop. Solitamente le ultime due istruzioni non sono presenti nel codice binario. Questo perché quando un programma chiama una system call non lo fa mai direttamente, passa attraverso le librerie quindi quello che esiste nel binario è la chiamata alla libreria write. Quando faccio la write non trovo la chiamata alla syscall all'interno del binario, ma quest'ultimo chiamerà attraverso una tabella che si chiama PLT un certo indirizzo, la PLT lo risolverà con il linker che punterà all'interno delle librerie.
Quello che viene fatto (per le prime 3 pop) è definire dei gadget, in particolare due tipologie:

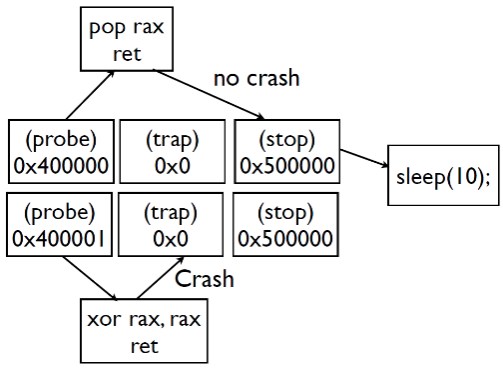
- **Dead** (crash) **Gadget** → nel momento in cui salto a questi gadget mi viene chiusa la connessione.
- **Stop Gadget** → gadget che servono a mettermi in loop la connessione lasciandola aperta ad esempio con delle sleep() e non mi fanno crashare la connessione.

Questi sono gadget di controllo utilizzati per poter costruire l'osservazione. Non so dove sono, ma posso osservarli. Sovrascrivo il canarino che conosco (quindi posso "andare su") e modifico i byte del return address, specifico un determinato indirizzo X e osservo cosa succede. Cosa è successo?

(a) produce qualcosa che si è fermato? Allora ho uno stop gadget.
(b) ha prodotto una chiusura? Allora lo definisco come dead gadget.

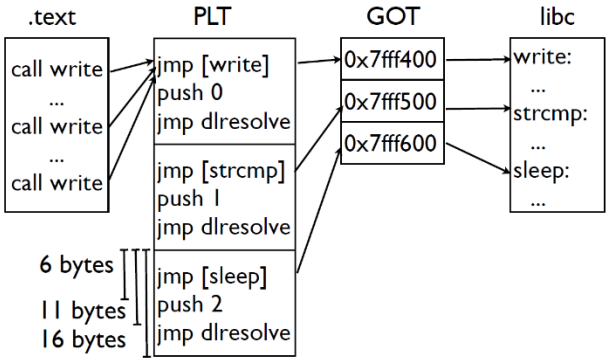
Questi sono definiti come probing gadget, perché poi mi danno un'idea sul resto (*probing: indagatore, che funziona da sonda*), mi permettono di capire dove sono le pop.

Una volta fatto ciò vado a cercare i gadget che mi servono (i primi 3 per la write).



→ voglio capire cosa c'è nei "probe" (dove probe l'indica l'indirizzo del gadget di cui viene fatto lo scanner): c'è una pop? O altro?

L'ultima parte dell'attacco è la seguente:
Nel binario solitamente quando chiamiamo una funzione di libreria sfruttiamo due tabelle: PLT e GOT. Esempio:

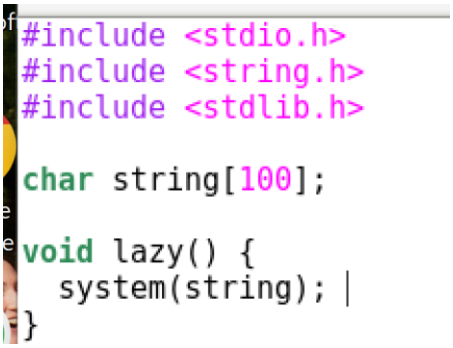


.text è nel binario

Call write è un aggancio alla PLT che è statica (read only), l'indirizzo della chiamata "call write" non è l'indirizzo della funzione libreria, ma della PLT. La prima volta che chiamo la funzione la PLT viene aggiornata tramite un'altra tabella, la GOT, che contiene l'indirizzo del linker dinamico. PLT e GOT sono due tabelle che lavorano in tandem. PLT chiama la GOT che a sua volta essendo un linker dinamico, fornisce l'indirizzo della funzione della libreria da chiamare (funzioni di libc). Quando si scrive nella PLT non si ha un crash del sistema, se mi sposto di un byte, non crasha. Quindi si individua la tabella modificando il ret address e poi si cerca di capire dove è la write. Si provano i vari indirizzi e si osserva quale corrisponde alla write.

L'ultima cosa che devo fare è provare la combinazione dei numeri di socket descriptor. Il parametro può variare da 0 a 256. Provo finché non vedo che stanno arrivando i dati ovvero il binario con una certa lunghezza. In questo modo prelevo il binario dalla macchina remota e lo porto sulla mia macchina locale.

Codice LAB - ROP



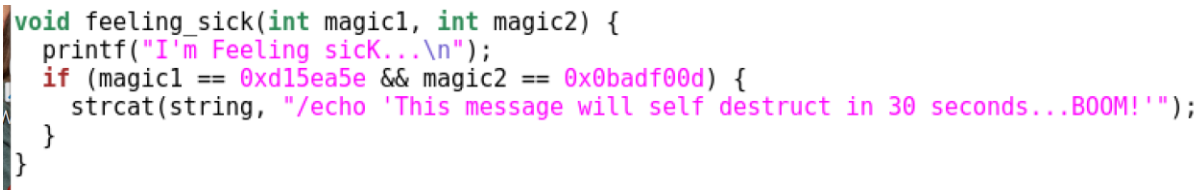
```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

char string[100];

void lazy() {
  system(string);
}
```

→ system(string) esegue quello che c'è in stringa e devo fare in modo che sia "bin/echo 'This…' "

I due parametri di feeling_sick devono corrispondere:



```c
void feeling_sick(int magic1, int magic2) {
  printf("I'm Feeling sicK...\n");
  if (magic1 == 0xd15ea5e && magic2 == 0x0badf00d) {
    strcat(string, "/echo 'This message will self destruct in 30 seconds...BOOM!'");
  }
}
```

Inoltre, nel file exploit.py dobbiamo fare in modo che l'elenco dello stack sia

1. food (che va nel RET per primo) → primo parametro: quello che serve per matchare ed entrare nella funzione cioè un parametro, magic, che è "deadbeef"
   [il record di attivazione di food ha il suo return address e parametro sopra (dal basso verso l'altro)] → dopo food in string c'è "bin" faccio una RET nel gadget pop RET, perché? Devo pulire il parametro di food perché è 'tutto in mano a me'. La pop RET pulisce, porta lo stack pointer, e poi ritorna su feeling sick.
2. feeling sick → ha due parametri: quindi nel RdA ha magic1, magic2 e la pop RET. Lui esegue la strcat e aggiungo "echo 'This…' " + eseguo le due eliminazioni dei parametri (pop pop RET)
3. lazy: farà system di quello che c'è in string e a questo punto l'esecuzione del programma sarà quello che c'è

in fondo ci sono SFP e Buffer

6 DICEMBRE 2022

Control Flow Integrity

Tecnica implementata da molti sistemi operativi, non sempre automatica, ma si può attivare.
→ utilizzata per contrastare i memory errors. Sistema di detection/prevention che cerca di bloccare il programma, ma in particolare si vuole capire il comportamento del programma (non si basa sull'attacco in sé).

Tecnica che non si concentra su un attacco in particolare: è un esempio di come un sistema di difesa può essere progettato in generale (secure by design): si basa sul comportamento del programma. Ho un programma da difendere e quello che voglio fare è definire un comportamento di questo programma. Una volta definito il comportamento, creo un monitor che osserva che questo comportamento sia effettivamente rispettato, ogni deviazione mi porta ad avere un'anomalia (ad es. un attacco).
Quindi, le tre challenge del CFI sono:

- Dobbiamo definire il comportamento aspettato (come il programma si comporta durante l'esecuzione) – *(anche "com'è definito un comportamento?")*

- Rilevare le deviazioni dal comportamento efficientemente => equipaggiato con un detector che deve essere protetto perché non deve essere possibile attaccarlo
- Evitare che il mio detector (colui che si occupa di osservare) venga compromesso. Il come proteggere il detector ovviamente è una parte fondamentale.

Come definiamo il comportamento "aspettato"? Ma cos'è un comportamento? Esempio: si potrebbero guardare le syscall che vengono svolte dal programma.

Ovvero come il programma si comporta durante l'esecuzione, in questo caso
→ **Control Flow Graph** (CFG): rappresenta i possibili path di esecuzione (percorsi leciti nel grafo) della mia applicazione. Prendo quindi come modello questo grafo per definire un comportamento aspettato. Oppure **Call Graph**: grafo delle chiamate; vengono entrambi esaminati in maniera più specifica dopo.

Rilevare deviazioni in modo efficiente (con basso overhead) dal comportamento aspettato
→ **In-line reference monitor** (IRM) sistema di difesa solitamente implementato a livello del compilatore, molto veloce perché è all'interno dell'applicazione stessa. L'obiettivo del monitor è quello di bloccare l'attacco prima che vada in esecuzione. Avrò quindi una fase offline in cui vado a costruire il modello che rappresenta il comportamento del mio programma, una fase online in cui il programma viene eseguito e il controllo del modello che ho costruito deve essere integrato nel mio programma tramite in-line (controllo all'interno del programma stesso che porta ad una velocità massima) reference monitor. In particolare, vado a vedere se vengono svolti i path che avevo immaginato nella fase di learning, ma non è detto che siamo riusciti a immaginare TUTTI i path leciti per cui è possibile avere dei "false positive": non tutti i path illeciti sono degli attacchi! Si vanno ad inserire dei codici di controllo dentro il programma tramite il compilatore: protezione self contained, tutta dentro la nostra applicazione.

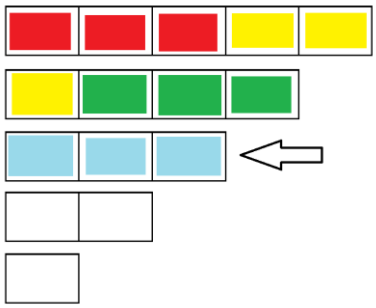
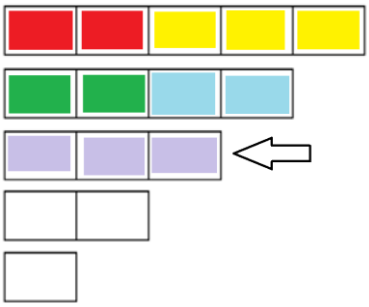
Evitare che il detector venga compromesso
→ **Sufficient randomness:** (creazione di segreti non predicibili) – label non predicibili
→ **Immutability:** tutti gli attacchi che cercano di eseguire codice arbitrario non devono poter modificare la mia instrumentazione.

<u>Efficienza</u>:
Quanto è l'overhead di questo sistema?

- Classic CFI (2005 – articolo 8-cficcs) → impone 16% di overhead in media, 45% nel caso peggiore. Ogni programma ha caratteristiche diverse, e quindi ho un caso medio e uno peggiore, in base a quanti oggetti devo proteggere. Non comprendeva le librerie (per una questione di complessità + non poteva essere fatta offline, richiedeva un lavoro più ingegneristico non possibile a quel tempo) => lavorava su eseguibili arbitrari
- Modular CFI (2014) → impone 5% di overhead in media, 12% nel caso peggiore. Applicabile solo sul programma C attraverso un compilatore, LLVM, e lavoriamo a livello di codice sorgente.

<u>Efficacia</u>: definita da quanto è sicuro questo sistema
Quanti attacchi vengono beccati?
MCFI (il modular, quello del 2014) può eliminare il 95,75% dei ROP gadgets su versione x86-64bit. Benchmark è una sorta di dataset su cui si calcola l'overhead. I ROP gadgets <u>non vengono eliminati fisicamente</u> dal MCFI, ma solamente non permette di utilizzarli.

NOTA: Istruzioni intel cisc = moduli di lunghezza variabile → però c'è una proprietà che porta a ri-sincronizzarsi con il flow del programma quindi se usati nel modo giusto potrebbe essere che non venga riconosciuto l'attacco (uno ha usato un gadget ma non è stato identificato)

		anche scegliendo in modo diverso le istruzioni, per cui magari sempre da 3 blocchi, oppure alternando da 2 a 3 si arriva a fare quello da 3 insieme ALLORA ci si allinea all'esecuzione normale del codice e non si viene scoperti

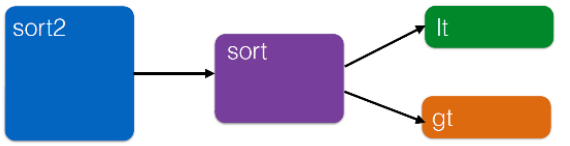
Riduzione degli indirect-target (percentuale di possibili bersagli di salti indiretti che CFI esclude), ovvero i punti di attacco più utilizzati che devono essere protetti dal CFI. Più riusciamo a ridurre l'uso di questi indirect-target e più siamo riusciti a proteggere il nostro sistema. Protezione anche in questo caso alta: 99% (staticamente si riesce a dare un risultato quasi certo di dove salterà). Già dalla progettazione del sistema, non vado a colpire un attacco, ma definendo il comportamento di un programma, vado a beccare le deviazioni.

Esistono diversi grafi con cui posso esprimere il comportamento di un programma:

- **Call Graph** (grafo delle chiamate a funzione)

```
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;
}
```



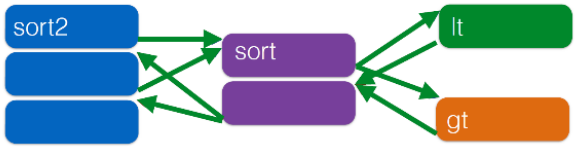
→ grafo che rappresenta le chiamate di funzione effettuate all'interno del mio programma a livello statico. Nell'esempio non è presente il numero di volte che la funzione sort viene chiamata. Quello che faccio è quindi andare all'interno del programma e determinare tutte le funzioni che ho nel programma e poi definire il grafo. Posso lavorare a due livelli: intra-procedural ovvero all'interno della funzione stessa o inter-procedural, lavoro esternamente collegando tutte le altre funzioni con i loro grafi. Questo rappresenta già un modo di definire un comportamento.

Questo non ci basta, perché noi vorremmo definire il flusso in modo più granulare e non grossolano come in questo caso. Allora parliamo di Control Flow Graph. Questo viene fatto per avere un maggior controllo sugli attacchi. Occorre ragionare sempre in modo generale.

- **Control Flow Graph**

```
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;
}
```



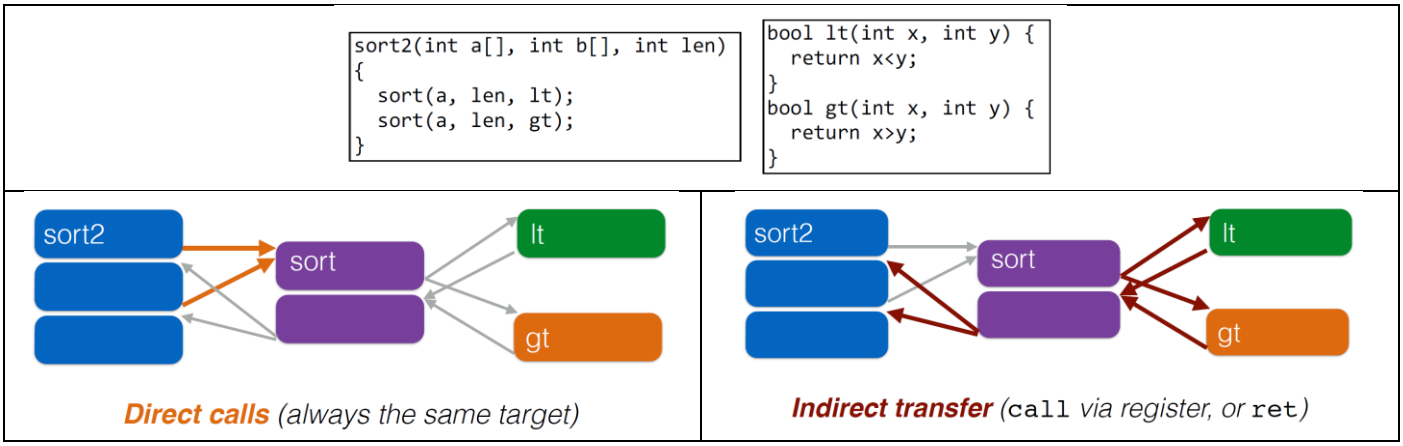
→ vengono definite delle unità base, che non sono più le funzioni, ma andiamo all'interno del codice delle funzioni stesse. Vado a trovare dei macro-blocchi operativi all'interno delle funzioni che sono chiamati **basic blocks.** Il blocco solitamente termina sempre con istruzioni che cambiano il flusso del programma (ad esempio istruzioni di jump). Quindi ho una serie di istruzioni sequenziali che non mi modificano il flusso del programma terminate poi da una che lo modifica. Ogni basic

block può essere distinto da una call (qualcuno che lo chiama) e da una return. Una definizione del flusso più reale del flusso del programma con il Control Flow Graph. → si può vedere una maggiore definizione: sort2 non è più un blocco unico, ma è suddivisa in tre e ci vengono dati anche possibili punti di return delle funzioni (e in alcuni casi questi sono anche diversi in occasione di una chiamata a una stessa funzione)

Quindi il CFG definisce il flusso del programma: basato su mattoncini (basic block formati da istruzioni che possono terminare con dei salti, ovvero ci sono serie di istruzioni aritmetiche sequenziali che terminano con un salto (sulla falsa riga dei gadget che però terminano con una RET)) che sono collegati tra loro.

CFI compliance con CFG:

- Calcolo le chiamate e i ritorni CFG in anticipo, durante la compilazione o dal binario (offline)
- Monitorare il flusso di controllo del programma e assicurare che segua solo percorsi consentiti dal CFG. Il flusso è monitorato in base alle informazioni ottenute dal mio basic block
- Osservazione: le chiamate dirette non devono essere monitorate. Assumendo che il codice sia immutabile, l'indirizzo target non può essere cambiato. Quindi il codice del programma non può essere modificato dall'attaccante. Ho una call con un indirizzo fissato, calcolato dal mio programma (dal compilatore, fase di generazione del codice)
- Perciò: monitorare solo le chiamate indirette. JMP, call, ret con target non costante (ad esempio: JMP %EAX RET, CALL %EAX). In questo caso EAX è un registro non un valore. Call indirette sono calcolate a runtime, cioè non so a quale indirizzo potrò saltare, dipende dalla scelta effettuata dall'utente. L'attaccante può cambiare il valore di un registro tramite ad esempio un gadget

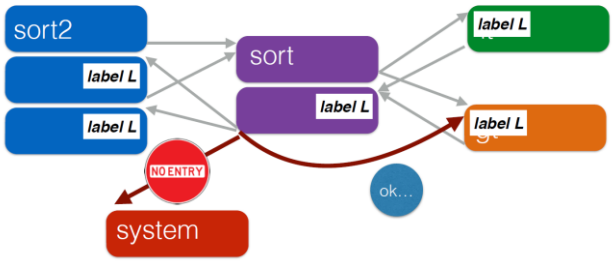


Direct call è quella alla funzione sort perché è una funzione ben definita che verrà sempre chiamata.
Le chiamate da sort, ovvero quelle a lt e gt (due puntatori a funzione), saranno due chiamate condizionate dal parametro che gli passo. Queste si trattano di call a valori presenti nei registri. Questo perché posso andare o in lt o in gt e quindi l'attaccante potrebbe tranquillamente cambiarle. Anche le ret da una funzione sono call indirette, perché è determinato dal valore che ho sullo stack nel ret address.

In-Line Monitor

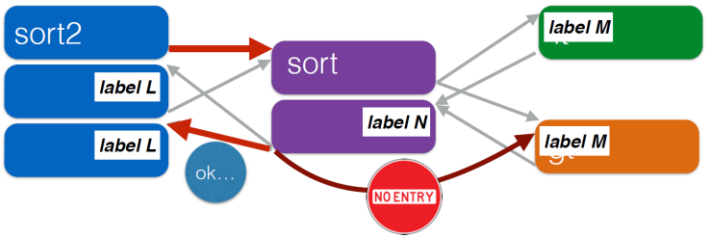
Implementa il monitor in-line, come una trasformazione di programma (instrumentazione del codice). Costruiamo dei plugin del compilatore che cercano zone (call indirette) e mettono dei

codici di controllo sulle call indirette. Inserisce una label proprio prima dell'indirizzo target di una chiamata indiretta. Inserisce del codice per controllare la label del target a ogni trasferimento. Abortisce se la label non combacia. Le labels sono determinate dal CFG (visione che mi sono costruito in precedenza).
Posso saltare solo tra i target etichettati con la stessa label. Siccome sto usando la stessa label (ci sono più blocchi con label L) posso fare salti anche tra due basic block che non sarebbero collegati direttamente → **simplest labeling**



È possibile utilizzare stesse etichette per ogni nodo (modello più semplice). Mentre, se salto ad un blocco, come ad esempio la system, che non è etichettato con la stessa label, non viene permesso (NO ENTRY).

Possiamo definire anche delle label diverse a cui puntare per le call e le ret (**detailed labeling**):



Nell'esempio posso passare dalla label M ad una label N con la ret, ma non posso saltare da una Label N ad un M – perché c'è un maggior tipo di dettaglio rispetto all'esempio di prima per cui non è più un salto indistinto, si sono messe delle label definite per definire, su di loro, i possibili salti. Dentro uno stesso blocco si definiscono due label uguali perché quando si fa il return da un blocco, staticamente non si sa dove si andrà quindi devono essere uguali (permesse entrambe allo stesso modo quindi – opzioni statiche delle chiamate che si hanno). Questo rappresenta il modo in cui evito l'utilizzo di gadget.

ARTICOLO CFI:

source | **destination**
jmp ecx -> per saltare nella destinazione | **mov eax, [esp+4]**
In questo caso ho un unico salto.

Instrumentazione: *serve per vedere che sto saltando alla label corretta*
cmp [ecx], 12345678h -> confronta il valore puntato da ecx con il valore dell'etichetta (label) (12345678h). Se l'etichetta aspettata è uguale a quella effettivamente contenuta nel nodo target (destination), allora salto l'errore (**jne**). Se non è uguale allora ho un errore e salto ad un handler.
jne error_label
lea ecx, [ecx+4] -> prende il valore di ecx+4 e carica il valore ottenuto in ecx.
jmp ecx
Nella destinazione viene messo un commento ";" (in assembly è un commento)
cmp [ecx], 12345678h | **data 12345678h**
jne error_label | **mov eax, [esp+4]**

```
lea ecx, [ecx+4]
jmp ecx
```

Can we defeat CFI?

- Code Injection con label legali:

→ non funziona perché assumiamo che la zona di memoria dati non sia eseguibile.

- Modificare la label per consentire il flusso di controllo desiderato:

→ non funziona perché il codice è immutabile – read only (le label sono all'interno del codice).

- Modificare lo stack durante un controllo, per far sembrare che abbia successo:

→ non funziona perché l'avversario non può cambiare i registri in cui carichiamo i dati rilevanti.

Controllo l'etichetta del ret, vado avanti ad eseguire altre istruzioni e successivamente potrebbe esserci una finestra temporale che mi permette di modificare il ret address, ma solo in caso di presenza di thread, non nel caso di esecuzione sequenziale. Si sfruttano delle particolari race condition (time of check, time of use). – applicazioni multi thread che vanno a cambiare i valori dello stack: spill-over dei registri che cambia le carte in tavola

Assicurazioni di CFI:

- Efficace contro tipologie di attacchi che modificano il controllo del flusso
- No manipolazione del flusso di controllo che è consentito dalle labels/grafo → Chiamati **mimicry attacks,** mimano la possibilità di chiamare in loop delle funzioni che sono nel flusso corretto del programma e che mi possono portare ad effettuare degli attacchi (sfrutto il fatto che la mia visione statica non riesce a determinare ciò che succederà a runtime). La semplice single-label CFG è suscettibile a questi
- No protezione da perdita di dati o corruzioni → heartbleed (bug che era stato trovato in SSL: il protocollo chiedeva un "certificato lungo 512", ma se tu vai a modificare la richiesta della grandezza del certificato, con 1024, 2048, etc allora veniva data anche parte della memoria) non sarebbe impedito, nemmeno l'overflow di autenticazione in cui la modificazione del controllo è consentita dal grafo.
- No **data control attack**, sono una classe di attacchi (difficili da beccare e potenzialmente hanno un impatto molto alto) il cui overflow non vanno a modificare il ret address (non modifico il controllo di flusso del programma), ma vado a modificare delle variabili, il valore dei dati. Ho un cambio di flusso quindi lecito (cioè non mi accorgo che è invece illecito). Sono effettuati su una serie di applicazioni reali, molto presenti su embedded system.

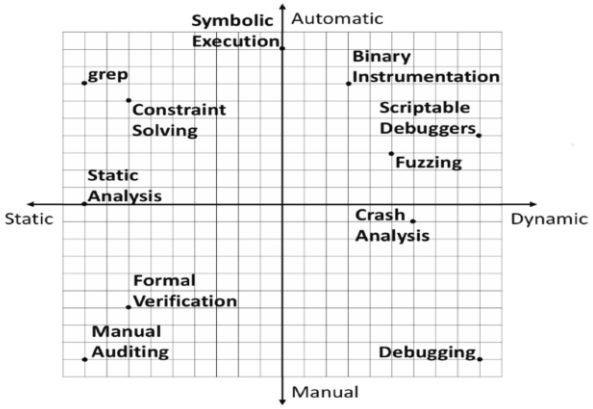
13 DICEMBRE 2022

Static Analysis VS/and Dynamic Analysis

Control flow integrity → molto efficace ma non riesce a bloccare i tipi di data attack dove invece di modificare il ret address vado a modificare le variabili. Questa parte è dedicata ai tool che analizzano in maniera automatica i tool per bloccare degli attacchi.

La code analysis ha due applicazioni: **source** code analysis (analisi del codice sorgente) o **binary** analysis (analisi del codice binario). L'obiettivo è quello di costruire tool automatici in grado di scovare bug, perché solitamente con un'analisi manuale (in particolare di grandi programmi) possono sfuggire interazioni, errori ecc. Problematica dei tool: vengono classificati in base al rate di falsi positivi e negativi.

Falsi positivi → bug che sono stati trovati, ma che in realtà non lo sono effettivamente.
L'analisi manuale ci serve per validare l'analisi effettuata dal tool automatico.



STATIC ANALYSIS: ho un codice, non lo eseguo, ma lo rappresento e vado a vedere sulla rappresentazione se rispetta certe proprietà di sicurezza o se presenta dei bug → si controlla dal punto di vista statico → analisi completa dei path di esecuzione del programma
Nell'analisi statica io costruisco un modello di esecuzione matematico del mio programma in modo che possa simulare (simbolica) l'esecuzione del mio programma (tutti i possibili percorsi), ma non la eseguo realmente. Mi permette di valutare la maggior parte degli input che il mio programma potrebbe ricevere, valutando gli stati che potrebbe avere il programma. Il vantaggio principale dell'analisi statica: è molto veloce perché una volta definito il modello, verificare alcune proprietà è molto veloce.
Svantaggio → è solo un'approssimazione dell'esecuzione del mio programma, può portare a dei falsi positivi o falsi negativi. La precisione dell'analisi dipende un po' da cosa sto cercando di fare.
Constraint solving → metodi per capire quali sono gli/le input/condizioni per arrivare ad un certo punto del programma.
Symbolic execution esiste sia nella forma statica che dinamica.

DYNAMIC ANALYSIS: si esegue il codice e si analizza ciò che accade durante l'esecuzione
Analisi dinamica → ho un'effettiva esecuzione del programma. Traccio ad esempio il contenuto delle variabili, dove potrebbero venire copiati i dati. Vantaggio: è più precisa perché tutti i dati sono reali dato che sto effettivamente eseguendo il programma. Difetto più grosso: posso analizzare un input alla volta, difatti è molto lenta. Infatti, do un input e vado a verificare cosa succede a fronte di questo input.
Binary instrumentation → instrumentazione del nostro programma per tracciare le system call, gli accessi in memoria e il flusso dei dati.
Fuzzing → creazione di input potenzialmente malevoli per la mia applicazione e verifica di quello che succede al programma.
Solitamente tra analisi statica e dinamica si vanno a prendere degli ibridi perché assicurano soundness e completeness.

Detection of the vulnerability

Rileva la presenza di vulnerabilità nel codice durante lo sviluppo, il testing e la manutenzione.
Quando parliamo di tool per rilevare le vulnerabilità parliamo di due proprietà principali: soundness e completeness.

Se noi troviamo un tool che è **sound** rispetto ad un buffer overflow, vuol dire che è in grado di dirci se il programma ha o meno vulnerabilità di tipo buffer overflow. Al contrario, una tecnica è unsound se non è in grado di rilevare tutte le vulnerabilità effettive, ha dei falsi negativi.

Il tool per essere sound ovviamente deve essere in grado di analizzare tutto il programma.
Una tecnica è **complete** se ogni volta che trova una vulnerabilità, questa è effettivamente reale, non crea cioè falsi positivi. Quindi, se rileva un buffer overflow siamo sicuri che lo è realmente. Al contrario una tecnica è incomplete se ci da falsi positivi. Sound e complete ci servono quindi per giudicare ciò che un tool fa.

La **soundness** lavora sulla parte statica del programma. Costruisce quindi un'astrazione e cerca di considerare tutte le possibili esecuzioni del nostro programma, è più legata all'analisi statica. Inoltre, è difficilissima da ottenere perché presuppone che noi sappiamo tutti gli input che il programma può ottenere. La **completeness** può essere raggiunta tramite l'esecuzione del programma che mostra le vulnerabilità. Il tool deve essere in grado di fornire un input concreto che, se eseguito sul programma, manderà lo stesso in crash.

Input astratti → sono input simbolici, che rappresentano una categoria di input.
Input concreti → quelli, ad esempio, che forniamo noi per fare l'exploit.
Posso far crashare il programma, ma non è detto che sia exploitabile tramite un attacco. Più in generale: non è detto che se ho una vulnerabilità questa sia anche exploitabile.

Static Analysis VS Symbolic Execution
Analisi statica → è un qualsiasi calcolo offline (costruzione modello teorico) che ispeziona il codice e produce risultati sulla qualità del codice. Applicabile al codice sorgente o binario e può comprendere più tecniche, tra cui ad esempio le tecniche del compilatore. Può usare la symbolic execution e ispezionare la formula risultante, ma non è associata ad essa. Oppure può usare altre tecniche come espressioni regolari.

Symbolic execution→ è un tipo specifico di calcolo offline che calcola alcune approssimazioni di ciò che effettivamente fa il programma costruendo formule che rappresentano lo stato del programma in vari punti. Si chiama "simbolico" perché l'approssimazione è solitamente una sorta di formula che coinvolge variabili di programma e vincoli sui loro valori

➔ Non esaurisce la static analysis

Il fatto che un tool sia sound deve valere per tutte le tipologie di programmi, non solo per uno.

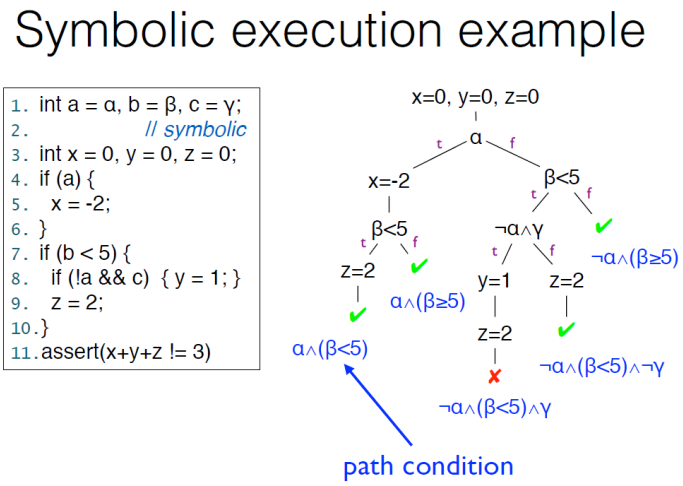
Symbolic Execution

Dall'altra parte c'è l'analisi dinamica che si effettua eseguendo il codice → più lenta, ma ha meno limiti (perché nell'analisi statica ci sono cose che non posso scoprire perché si scoprono solo eseguendo il codice: tipo puntatori, o gestione degli input dell'utente) però non comunque completo perché non posso avere TUTTI gli input.

È una tecnica proveniente dal software engineering. Nel testing work, quello che si fa è di introdurre delle assert che testano delle condizioni particolari rispetto ad un punto del programma. Es: `assert(f(3)==5)`. Questa asserzione è basata su un input particolare dell'applicazione. Non è sound perchè non esplora tutti i casi.
Symbolic execution è un'esecuzione statica simbolica del mio programma, il suo scopo non è quello di trovare bug. Permette variabili simboliche nelle valutazioni. $y=\alpha$ rappresenta un input simbolico. In questo modo posso generalizzare la mia asserzione. Concettualmente ho dei path (if) che si diramano. Può includere delle condizioni (path condition) che in qualche modo

identificano un certo percorso nel mio programma. Non è sound perché effettivamente non va a verificare tutti i casi: generalizza ciò che è il testing ed aggiunge più sound.
Le assert valutano condizioni particolari rispetto a particolari zone del programma.

È nata come un'esecuzione simbolica e questo si è espresso tramite delle asserzioni per generalizzare il testing → dove si introducono le var simboliche del programma



L'esecutore va a costruire un grafo di esecuzione in cui ogni volta che incontra delle diramazioni (definite dagli if statement) crea una diramazione in cui mette come principale nodo della diramazione la condizione che ci fa spostare da un ramo a un altro.

Qui viene fatto manualmente ma normalmente non è così → alla fine si vanno a valutare le asserzioni e si capisce quando queste sono in contraddizione con l'asserzione che abbiamo messo: a questo punto vado a capire qual è l'input che porterebbe al crash (ad esempio l'assert centrale fallisce).

Le condizioni rappresentano percorsi che il nostro programma può prendere, ma dato che io non so che valori l'utente darà in input, vado ad attribuire dei simboli alle variabili.

path condition → è la formula che rappresenta simbolicamente l'input che devo dare per arrivare in una determinata posizione del programma. Alla fine di ogni percorso (path) verifico che l'asserzione sia verificata. Si tratta di espressioni logiche.
Dopo aver valutato l'asserzione, mi chiedo quali sono i valori reali dell'input che devo dare per arrivare in un determinato punto del programma. In particolare, voglio arrivare in fondo al path in cui l'asserzione non è verificata. Quello che faccio è agganciare un constraint solver.

Il constraint solver prende la formula logica e mi dice quali sono gli input (una soluzione della formula, se esiste) che se dati al programma fanno arrivare alla violazione dell'asserzione.
Ogni percorso di esecuzione simbolico rappresenta molte esecuzioni effettive del programma: si può coprire molto più spazio di esecuzione del programma rispetto al testing.
La copertura totale del programma definisce la correttezza totale di esso e a noi a livello di sicurezza non ci interessa molto.

Su programmi molto grossi la symbolic execution è molto difficile che termini, quindi non è sound.

La symbolic execution ha avuto un utilizzo molto pratico. I ricercatori sono riusciti a creare un modello di calcolo molto efficiente. Presentata la prima volta da King nella sua tesi di dottorato.
Perché non è subito decollata? La symbolic execution richiede molte risorse, è CPU intensive. Avevamo un problema tecnologico e algoritmico (legato all'esplosione path del programma, soddisfacibilità formula logica). Oggi i computer sono più veloci e più grossi, ma ci sono anche algoritmi migliori (SMT/SAT). Inoltre, si è passati anche a delle tecniche ibride (Concolic execution).

Nel 2005-2006 ritorna interesse per la symbolic execution. Area di successo (security): bug finding, nasce l'esigenza di applicare la symbolic in particolari zone del programma. Ricerca euristica nello spazio delle possibili esecuzioni. Trova bug davvero interessanti.
L'esecuzione simbolica può essere fatta su particolari zone del programma dove, per esempio, troviamo operazioni sulla memoria.

Basic symbolic execution - Symbolic variables

Come è costruito l'esecutore simbolico?
L'esecutore simbolico è un programma che lavora sul linguaggio su cui applica la symbolic execution.

$e ::= \alpha \mid n \mid X \mid e_0 + e_1 \mid e_0 \leq e_1 \mid e_0 \;\&\&\; e_1 \mid \ldots$

- $n \in N$ = integers, $X \in Var$ = variables, $\alpha \in SymVar$

Partiamo quindi dal linguaggio che definisce delle espressioni e andiamo ad estendere queste espressioni già presenti con un altro insieme di simboli rappresentate da alfa che comprende tutte le variabili simboliche. I simboli vanno a sostituire i valori di input. Ogni volta che troviamo funzioni associate all'input (read, fgets, ecc.), ho un mapping delle variabili simboliche su queste funzioni di input del linguaggio stesso.
Dobbiamo costruire un calcolatore simbolico che valuti e componga le espressioni calcolate prima dal punto di vista simbolico. Quello che l'esecutore simbolico deve saper fare è concatenare le espressioni con valori simbolici, ad esempio α+5, "hello"+α.

→ Symbolic expressions: il nostro esecutore simbolico deve avere la capacità di valutare le espressioni composte dai valori normali degli input uniti ai nostri valori simbolici.

Esempio sottostante: anche nel caso simbolico riesco a dire che 12 supera lo spazio dell'array a 4 posizioni

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

Concrete Memory	Symbolic Memory
x ↦ 5	x ↦ α
y ↦ 10	y ↦ 5+α
z ↦ 17	z ↦ 12+α
a ↦ {0,0,0,0}	a ↦ {0,0,0,0}
Overrun!	*Possible overrun!*
	We'll explain arrays shortly

Come vengono calcolate le path condition?
Per ogni diramazione in quel punto particolare posso definire una path condition: per ogni istruzione una condition.

Sono espressioni logiche le cui soluzioni delle formule logiche rappresentano l'input che mi permette di arrivare ad un particolare punto del programma. Sono associate alle espressioni del mio linguaggio e vengono assegnate alle istruzioni di condizioni rappresentate dall'if statement. Esistono nei punti di istruzioni del programma che per essere raggiunte devo rispettare delle

condizioni (solitamente un if). La soluzione dell'espressione logica mi dice se esiste un input per arrivare in quel particolare punto del programma.

path infeasible → non ho alcuna condizione che mi possa portare a quel path. La fattibilità del path ci permette di verificare le asserzioni. Le asserzioni devono essere tradotte in qualche modo dal compilatore.
Soluzione ai path constraints possono essere usati come input per casi concreti di test che eseguiranno quel path.

Paths and assertions

I due if dell'esempio sottostante vengono aggiunti solo in caso si va a fare testing o analysis perché altrimenti introdurrebbe overhead:

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

π = true
π = true
π = true
π = true
$\pi = I2+\alpha<0$
$\pi = \neg(I2+\alpha<0)$
$\pi = \neg(I2+\alpha<0) \wedge I2+\alpha \geq 4$
$\pi = \neg(I2+\alpha<0) \wedge \neg(I2+\alpha \geq 4)$

Sono introdotti dal risolutore simbolico o dall'analista per verificare se ci sono casi di input che mi possono portare ad un overflow del buffer.
Se entrassi in una delle due abort, significherebbe che ci possono essere dei valori dati in input che mi causerebbero l'overflow. Se invece i due abort non sono mai raggiungibili, sono path infeasible allora non ci sarà buffer overflow.

True vuole dire che ci arrivo sempre. Poi arrivo a calcolare le prime condizioni – arrivato alla fine devo vedere se entrare in quei due if è possibile, perché se non lo sono allora si può stare tranquilli; in caso contrario, ci sono dei casi concreti che ci fanno andare negli abort().

Forking execution

Esecutore simbolico → nella sua analisi ogni volta che trova una if statement si forca. Analizza un path alla volta, singolarmente. I forking prendono strade diverse una dall'altra che verranno analizzate sequenzialmente.
Viene creato un task iniziale in cui tengo il program counter, le varie path conditions di quel particolare branch vengono aggiunte in una word list e tutte le variabili simboliche.

Più ho valori simbolici e più la complessità della computazione aumenta.
Fare l'esecuzione simbolica sulle libc è veramente complicato, sono scritte male e perciò non seguono le deadlines generali. Quindi, non si fa l'esecuzione simbolica della singola libreria, ma viene applicato un modello semantico alle librerie di sistema. In questo modo l'esecutore simbolico sa come comportarsi.

NOTA: Ad un certo punto bisogna analizzare le librerie: in alcuni casi vengono eseguite simbolicamente (molto difficile in alcuni casi, come nei driver (??)) oppure vengono inseriti modelli simulati del comportamento della libreria.

Concolic execution → fornisce un input concreto e semplifica/minimizza le path condition (più facili da eseguire dal sat solver, che era un punto difficile dell'uso di questi algoritmi) → è un'esecuzione concreta del programma su determinati input con l'aggiunta dell'esecuzione simbolica sopra descritta. Il programma viene instrumentato, vengono messi dei **watchpoint** negli if del programma. Inizialmente si considerano degli input concreti, eseguo quel path particolare. Ogni volta che incontra una condizione if (espressione associata alla condizione) la memorizza nella shadow memory che ha creato; quindi, tengo il valore di quell'espressione basato sul valore concreto. Una volta terminato il ramo attraversato tramite l'input concreto che ho fornito, torno indietro (nella shadow memory ho le condizioni salvate) e parto da queste condizioni applicando la symbolic execution per esplorare altre parti del programma. L'input concreto quindi mi serve per determinare un'area concreta che mi interessa ad esempio un'area con tante operazioni di memoria e poi da lì mi sposto in altri punti. Quando ho scoperto ciò che dovevo scoprire poi posso usare la symbolic execution.

La symbolic execution non è sound perché non termina e quindi alcune parti del programma non vengono eseguite simbolicamente, ma è complete perché ci assicura che se viene rilevato un bug, non si tratta di un falso positivo. In questo modo non do il carico al constraint solver di andarsi a calcolare dei valori particolari, perché io già so che con quel valore arriverò in quel particolare punto. Ci permette di concretizzare e rimpiazzare i valori simbolici riducendo la complessità della formula che sto andando a considerare. Ogni input definisce uno e un solo path.
Vantaggi:

- direziono il mio esecutore simbolico in maniera più precisa
- Semplificazione della formula logica (e più velocità)

20 DICEMBRE 2022

https://github.com/SVF-tools/SVF

10 GENNAIO 2023

Basic Symbolic Execution - **Symbolic execution as search, and the rise of solvers**
Search e SMT

Sono i due campi che portano dei limiti alla mia analisi. Search computazionalmente costosa a causa del numero di path che ho da analizzare. Altro limite è quello legato alla SMT(fattibilità della soluzione delle nostre formule logiche).
La symbolic execution è computazionalmente costosa. Occorre considerare la fattibilità della soluzione delle nostre formule logiche. Anche se non andiamo concretamente nei valori dobbiamo considerare le varie opzioni, i path che ci sono vanno esplorati tutti
Primo problema che si può considerare: **Path explosion** → esplosione dei rami (path) che ho da analizzare. (le combinazioni dei costrutti portano ad un'analisi molto costosa)
Struttura ramificata esponenziale (condizioni if-else), ad esempio con 3 variabili ho 2^3 path (perché sono opzioni binarie: 2). Ancora peggio è la situazione nel caso dei loop:

```
1. int a = α;    // symbolic
2. while (a) do ...;
3. …
```

- Potentially 2^31 paths through loop!

Se la variabile è una variabile intera avremo 2^31 paths attraverso il ciclo.

In questo caso avremo una non terminazione mentre invece nell'analisi statica avevamo una terminazione, perché abbiamo un'approssimazione dei branch, dei loop etc. Con l'analisi statica posso avere dei falsi positivi proprio perché io faccio approssimazioni; quindi, devo poi andare a verificare se si tratta di un bug vero e proprio oppure no.

Quindi si vanno a trovare delle metodologie di ricerca. Vengono create delle strategie di ricerca che si basano sui principi detti prima: prioritizzare la ricerca su porzioni di codice che potrebbero avere dei bug, basarsi su grafi, algoritmo di esplorazione del grafo.

Inizialmente, essendo un grafo, vengono proposte i classici algoritmi di esplorazione dei grafi, ovvero esplorazione in profondità e in ampiezza. Questi metodi hanno aspetti negativi, ovvero non sono guidati. A livello computazionale potenzialmente dannosa, perché potrebbe causare il blocco dell'analisi, per via dei path explosion ad esempio. BFS è una scelta migliore perché spostandosi in ampiezza non rimane bloccata nei loop.
È necessario prioritizzare la ricerca, ovvero definire strategie per andare verso zone particolari. Pensare ad un programma come ad un grafo di esecuzione (DAG). Definire algoritmi di prioritizzazione su questo grafo (DAG).

Primo algoritmo considerato è l'algoritmo randomness. Questo algoritmo ci permette di uscire da quelle condizioni particolari (ad esempio loop) che bloccano la nostra esplorazione (soprattutto se ci bloccano per un periodo di tempo lungo).

- Si sceglie un path senza sapere esattamente dove si sta andando
- Successivamente verifichiamo dove la nostra ricerca ci sta portando, se sta entrando in loop o verso zone che non ci portano a niente. A questo punto ripartiamo da zero con un path diverso.
- Scegli tra path di uguale priorità a caso.

È un algoritmo che dà buoni risultati perché riesce a svincolarsi meglio nelle esplorazioni effettuate e inoltre ci permette magari di trovare dei nuovi bug che non avevamo pensato. Il problema principale è dato dalla riproducibilità, ovvero devo in qualche modo andare a riprodurre quelli che sono gli stati del mio percorso e dato che sto scegliendo in modo random, non ho traccia di quello che è successo precedentemente. In particolare, è legato alla riproducibilità di bug che sono difficili da trovare, legati al fatto che il trigger del bug non è relativo al singolo ramo che sto esplorando al momento, ma relativo agli stati dei rami esplorati precedentemente. Se io so che il bug esiste, ma non riesco a riprodurlo è un problema perché vuol dire che non riesco a correggerlo.

Coverage-guided heuristics
Idea: provare a visitare statement che non abbiamo visto prima.

Ogni volta che visito un nodo metto un flag perché con questa tecnica voglio una copertura del codice massima → illusione di aver controllato il massimo che potevo (possibile non sia così perché magari si beccano solo i bug superficiali e non su quelli che si ottengono solo con determinati input/combinazioni di azioni)
Vogliamo testare più codice possibile. Quindi una volta che ho testato un pezzo di codice, desidero andare a visitare un'area di codice diversa. L'approccio consiste nell'attribuire un punteggio ad uno statement che indica il numero di volte che è stato visto. Il passo successivo, quindi, è quello di andare a scegliere lo statement che ha il punteggio più basso.

Può funzionare perché solitamente i bug si trovano in parti del programma difficili da raggiungere. Questo perché spesso per far crashare un programma servono condizioni particolari, che magari in 20 anni dall'esistenza di questo programma non erano mai capitate perché la zona in cui era presente il bug non è mai stata testata.
Può non funzionare perché potrebbe non essere in grado di raggiungere uno statement se non abbiamo impostato bene le precondizioni.

Generational search

Tecnica ibrida di ricerca dei bug: È una soluzione ibrida tra la BFS e la coverage-guided. In questo caso cerchiamo di avere la massima copertura del codice possibile.

Generation 0: Prendiamo un programma random e lo eseguiamo fino al completamento
Generation 1: prendiamo i paths dalla gen. zero, neghiamo una condizione del ramo su un path per produrre un nuovo path prefix, troviamo una soluzione per quel prefix e poi prendiamo il path risultante.
Generation n: similmente facciamo il branching off della gen n-1.

Si possono anche definire delle aree di priorità perché riteniamo ci sia più probabilità.

Combined Search

Eseguo più ricerche contemporaneamente. Andare a velocizzare e trovare dei path all'interno di alcune aree in cui mi concentro e combino l'utilizzo di più tecniche. Dopo aver analizzato una certa area riparto da zero e mi sposto.
Idea: nessuna soluzione valida per tutti. Dipende dalle condizioni necessarie per mostrare il bug. Potrebbe utilizzare algoritmi diversi per raggiungere diverse parti del programma. L'idea è quella di guidare l'esecutore simbolico verso zone particolari.

Nota di Lanzi:
La symbolic execution utilizzata nell'ambito della sicurezza, per quello sappiamo dove/come muoverci nella ricerca dei bug/analisi.

SMT solver performance

SMT Solver (constraint solver che ci permettono di risolvere le nostre path condition) sono programmi logici che risolvono equazioni logiche che lavorano su un problema di tipo Sat.

I SAT solvers sono il cuore degli SMT solver. Esistono vari SMT solver:

- Z3
- Yices
- STP
- CVC3

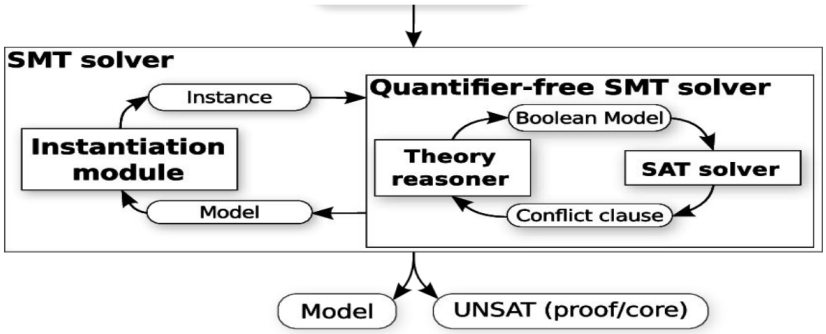
Caratteristiche diverse in base alla velocità di risoluzione di certe formule

Esempio: In questo caso ci sono 2^100 possibili path di esecuzione → questo rende difficile trovare il bug

```
int counter = 0, values = 0;
for (i = 0; i<100; i++) {
  if (input[i] == 'B') {
    counter++;
    values += 2;
  }
}
assert(counter != 75);
```

SLIDE – Z3: an efficient SAT/SMT solver

SAT Problem → problema di soddisfacibilità delle formule del primo ordine. Quello che si fa è di cercare di ridimensionare la complessità di queste formule che vengono considerate. È un problema NP completo ovvero non risolvibile in tempo polinomiale. Potrei avere istanze di soluzioni che potrebbero metterci troppo tempo per essere risolvibili.
La seconda problematica: quando analizzo con formule logiche (traduzione path condition in formule logiche), mi imbatto in quelli che sono i costrutti del linguaggio stesso (array, strutture di memoria, tipi di dati, ecc).
In prima istanza non siamo in grado di tradurre questi costrutti in formule logiche.
SMT cerca di tradurre in SAT tutti i costrutti di un linguaggio. Può essere considerato un'estensione di SAT e include tutte le teorie dei costrutti presenti nei linguaggi di programmazione.
Z3 è un SAT/SMT Solver di Microsoft.



Tutta l'architettura è chiamata SMT Solver. Inizialmente viene creata un'istanza del problema che poi viene passata al Quantifier-free SMT Solver (SMT). Theory reasoner cerca di creare un modello booleano passandolo poi al SAT solver che cerca di risolvere questa formula booleana. C'è un ciclo all'interno dell'architettura per la costruzione del modello booleano perché potrebbero esserci dei conflitti. Una volta che non riesce a risolvere la formula, il modello viene ripassato al Theory reasoner che cerca di semplificare il modello.
Ci sono zone di caching in modo da non ricalcolare parti di soluzioni a formule già calcolate in modo da essere molto più performanti. Cerca di rimuovere variabili non utilizzate, inutili.
SMT Solver è il componente più costoso della nostra analisi e quindi si cerca di ritardare sempre il suo utilizzo.

Get-model input concreto per arrivare a quel punto del programma, questo è un esempio “SAT Variable”, ci sono anche altre sintassi per Functions, Big Number, If Statement:

```
(declare-const a Int)
(declare-const b Int)
(declare-const c Int)
(declare-const d Int)
(assert
(and (> (* 2 a) (+ b c)) (> (* 2 b) (+ c d))
(> (* 2 c) (* 3 d)) (> (* 3 d) (+ a c))))
(check-sat)
(get-model)
```

Ha tutta una sua sintassi che permette di definire un’interfaccia di comunicazione con il mio componente SAT solver.

Symbolic execution systems

Esistono degli esecutori simbolici come DART e EXE. Sono i primi che sono stati utilizzati. **DART** basato sul model checking, formal systems background – un po’ datato ma ancora utilizzato. Cerca attraverso la modellizzazione dei programmi, di trovare formule logiche e in qualche modo di risolverle. **EXE** basato più su security (ricerca dei bug). SMT solver chiamato STP.

SAGE

Uno degli esecutori simbolici più utilizzati. È un concolic executor sviluppato da Microsoft Research (ha un laboratorio di più di 500 macchine che fanno analisi del codice tramite SAGE ed utilizza la generational search. Focalizzazione sui parser ovvero componenti software che definiscono il layout dell’applicazione, ad esempio come leggere la JPEG. I parser sono molto sensibili alla sicurezza, probabilmente se io modificassi il layout sotto alla JPEG o il parser se ne accorge oppure crasha. Nella maggior parte dei casi crasha, perché il parser si aspetta che si segua un determinato layout.

KLEE → symbolic executor che lavora su file sorgenti compilati con LLVM che è integrato in GCC. Utilizza i fork() per gestire stati multipli. Impiega diverse strategie di ricerca, random path e coverage-guided. Simula l'ambiente per gestire chiamate di sistema, accessi ai file.

Mayhem → lavora sui binari. Combina il meglio della strategia simbolica con la strategia concolica. Quando trova un bug, genera automaticamente un exploit funzionante.

Mergepoint → è per Linux, è un’estensione della Mayhem. Combina la symbolic execution con l’analisi statica. Usa l'analisi statica per blocchi di codice completi. Usa la symbolic execution per le parti più difficili da analizzare. Miglior bilanciamento tra il solver e l’esecutore: bug trovati più velocemente e più copertura del programma allo stesso tempo.

Ogni esecutore simbolico si specializza su una specifica parte: non ne esiste uno che copre tutto.

Fuzzing

Nato come una sorta di random testing, ovvero una delle strategie della symbolic execution. Lo scopo è quello di trovare comportamenti anomali del nostro programma, ovvero eccezioni che

vengono rilevate, quindi crash, non terminazione ecc. È complementare alle funzionalità di testing, si concentra principalmente sulla costruzione dell'input random da mandare all'applicazione, per far crashare quest'ultima o creare delle eccezioni. Si narra che il fuzzing sia nato dopo che in America (Wisconsin) a seguito di un temporale, erano crashati tutta una serie di computer provocando errori inaspettati nelle applicazioni che si stavano eseguendo.
Tipi di fuzzing:

- **Black box**: non conosco nulla dell'applicazione e non faccio inferenza su nulla. Il tool non conosce niente sul programma o sui suoi input (mando input random con caratteri e lunghezza strane). Input che non matchavano con gli input normali per quell'applicazione. Facile da utilizzare, ma esplora solo superficialmente la mia applicazione. L'input randomico o viene gestito al primo livello, quindi esce e dice mal formato oppure non lo gestisce e crasha (nel caso non ci siano controlli sull'input).
- **Grammar** (""grey"") **based**: Il tool genera input formati da una grammatica. In questo caso sono quindi consapevole di come vengono costruiti gli input, perché sono appunto basati sulla grammatica del mio linguaggio. Si riesce ad andare più in profondità nel mio programma. L'input può essere mal formato in parte, nel senso che posso fornire una parte basata sulla grammatica che mi permette di arrivare ad un certo stato e poi da qui entra in gioco la parte malformata (che va oltre la grammatica del linguaggio) per testare quel determinato stato.
- **White Box**: è quella più "consapevole", la concolic execution le è paragonabile. Symbolic execution è un fuzzer white box. Il tool genera nuovi input almeno parzialmente informati dal codice del programma da essere fuzzato. Quindi creo input ben precisi per raggiungere determinate parti del mio programma. Si tratta di una tecnica molto più precisa perché appunto so come arrivare in particolari punti del programma (ad esempio le parti in cui ho copie in memoria). È un fuzzing spesso facile da usare, ma computazionalmente costoso.

Fuzzing vuole creare input che vadano in punti specifici del mio programma, ma senza il costo computazionale di una tecnica come la symbolic execution. In questo caso parliamo quindi di input concreti. Le tecniche di fuzzing sono ad hoc, ci sono fuzzer per il kernel, fuzzing per android ecc.

Come si creano gli input? Bisogna trovare degli input che siano in grado di esplorare il codice in maniera consistente.
Mutation:

- Si prende un input legale e lo si muta. Lo si muta andando a modificare i bit attraverso, ad esempio, tecniche di machine learning ecc. Se ad un certo punto noto che finisco sempre nello stesso ramo della mia esecuzione, si modifica la parte di bit che viene mutata in modo da poter magari finire in un ramo diverso. Lo scopo è sempre quello di coprire il più possibile il mio programma. Input legali ad esempio generati dall'uomo, da una grammatica o da un SMT solver. Le mutazioni possono essere di diverso tipo, *genetiche (?)* che guardano quali sono le parti che fanno più cambiare la copertura, oppure manuali/umani e faccio dei dataset di input reali e usando tutti questi input e li muto

Generational:

- Genera input da zero, ad esempio da una grammatica (di ciò che sto testando, degli input di quello che testo – es. parser che prende in input un pdf, creo un input non valido cambiando i campi, creando situazioni limite)

Combinations:

- Genera un input iniziale, lo si muta n volte, si genera nuovi input
- Genera mutazioni secondo la grammatica.

Lo scopo è trovare degli input concreti e poi trovare un crash. Il problema principale però è trovare il punto in cui è vulnerabile (root cause) perché quest'ultima potrebbe non essere presente nel punto in cui è crashato. Questo perché vulnerabilità e crash non sono direttamente correlati (ad esempio quando vado a sovrascrivere qualcosa, ma solo tornando all'esecuzione e al controllo c'è il crash: l'origine del problema era prima quindi bisogna individuare il punto di vulnerabilità). Altro problema è che io posso utilizzare mille input diversi che mi rilevano sempre lo stesso bug. Si cerca di clusterizzare gli input in modo da sapere che una determinata classe di input mi rileva un determinato bug.
Ci si chiede inoltre se il crash del programma segnala una vulnerabilità che sia effettivamente sfruttabile per un possibile attacco. Questa può essere una fase opzionale.
La root cause è difficile da vedere ad occhio. I programmi sottoposti a fuzzing solitamente vengono instrumentati con dei tool (per trovare dei memory errors), come per esempio ASAN (Address Sanitizier) che instrumenta puntatori, array, buffer ecc che permettono di reperire più facilmente il punto in cui è la root cause.

NOTA: ASAN
Prende il programma, utilizza compilatori (strumenta ogni operazione che c'è all'interno del programma) e tiene traccia con una shadow memory - ci sono delle red zone in questa memoria ausiliaria - di dove è avvenuto lo sbufferamento, l'overflow - quando avviene il crash lui ha una mappa del programma dove sono avvenuti gli overflow.

Problema: è molto molto pesante dal punto di vista computazionale perché devo strumentare tutte le read/write del programma. → performance ed efficienza basse: posso anche non instrumentare niente, ma poi occorre tenere a mente che avrò tutta un'analisi manuale che potrà occupare moltissimo tempo.

In poche parole: prendiamo un'applicazione, instrumentiamo i buffer → quando avviene il crash, questo viene intercettato dall'ASAN che fa il dump dello stack e fornisce informazioni sui buffer instrumentati per capire in quale punto c'è stato overflow (usato anche per identificare use after free). → successivamente lo si fuzza

Se il programma crasha con un errore segnalato da ASAN, allora occorre preoccuparsi riguardo a bug sfruttabili per possibili attacchi.

Fuzzing vs Symbolic Execution
Consideriamo una condizione x>5 e x<10. Supponiamo x sia un intero, la probabilità di generare randomicamente un input x che triggera l'errore è 4/2^32, dove 4 sono i valori 6,7,8,9. In questo caso particolare la symbolic execution vince, la probabilità nel caso del fuzzing è piuttosto alta e quindi il fuzzer potrebbe rimanerci per molto tempo a lavorare. Sono evidenti in questi casi i limiti del fuzzing.
Quello che si fa è quindi utilizzare una soluzione ibrida, quindi un fuzzing ibrido. Utilizzo fuzzing e Symbolic execution per ottenere la massima efficienza possibile. Da tenere presente, comunque, che la symbolic execution è molto costosa.
angr lavora su binario.
KLEE (usa concolic execution) rallenta di 3000 volte l'esecuzione nativa dell'applicazione, mentre angr 300.000 volte. La symbolic execution ha dei limiti anche sul constraint solver. Non è realistico conoscere tutte le informazioni riguardante le APIs. Dobbiamo definire dei confini per la nostra

applicazione. Vanno definiti modelli che rappresentano le system call, modelli che rappresentano le funzioni di libreria. Il fuzzing invece con l'input concreto non ha problemi di questo tipo.

https://klee.github.io/tutorials/

Hybrid Fuzzing

L'attuale stato dell'arte è il fuzzing ibrido, che combina sia greybox che whitebox. L'idea è di utilizzare greybox fuzzer come ricerca globale per campionare rapidamente lo spazio degli stati. Quando si blocca, usa la whitebox pesante come ricerca locale. Hybrid fuzzing è stato recentemente proposto. Combina sia il fuzzing che la concolic execution con la speranza che il fuzzer esplori velocemente input banali e la concolic execution risolva rami complessi.

Idea nata dalla cyber grand challenge: emerso che la tecnica ibrida è quella che funziona di più.

Qsym

È un hybrid fuzzer (concolic+fuzzer) nato per ottimizzare fuzzer come Vuzzer e Driller riuscendo ad essere molto più veloce e a trovare molti più bug. È riuscito a trovare 13 nuovi bug. Hanno fatto delle ottimizzazioni riguardanti l'intermediate representation, agli snapshot e alla modellizzazione dell'ambiente esterno. Quando parliamo di intermediate representation, ogni volta che viene fatta una compilazione, instrumentazione deve essere fatta tramite intermediate representation, ovvero una rappresentazione self contained che mi permette di facilitare l'analisi. È un'operazione costosa e quindi hanno ridotto la traduzione della IR perché a priori loro sapevano dove applicare la symbolic execution. Riduzione degli snapshot i quali vengono utilizzati quando devo cambiare nella symbolic execution da un path all'altro. Noi l'avevamo chiamata shadow memory, ma in realtà quello che si fa è di fare degli snapshot su ogni statement. Anche questa operazione è molto costosa. Loro hanno trovato delle tecniche di concolic testing che permettevano di eliminare lo snapshot e ridurre tutta la parte relativa al salvataggio degli stati intermedi. Inoltre, hanno utilizzato non più dei modelli fittizi per l'ambiente esterno, ma sono riusciti a trovare degli input concreti per avere una visione più concreta permettendo così di trovare nuovi bug dell'ambiente esterno (API, system call ecc).