Università della Calabria

DIPARTIMENTO DI MATEMATICA E INFORMATICA

Corso di Laurea in Informatica

Tesi di laurea magistrale

Dropper: uno strumento per la generazione semiautomatica di payload finalizzati all'explotation tramite Return Oriented Programming

Relatore:
Prof. Giovambattista Ianni

Candidato:
Marco Oliverio
Matricola 152469

Anno Accademico 2013–2014

A nonna Rosa

Indice

Introduzione **4**

1 Panoramica tecnica **9**
1.1 Creazione dell'immagine di un processo 9
1.1.1 Program Header Table 10
1.1.2 Global Offset Table e Procedure Linkage Table 12
1.2 Return Oriented Programming 17
1.2.1 Gadget 18
1.2.2 Address Space Layout Randomization 20
1.2.3 Return-to-plt, GOT dereferencing e GOT patching . . 22

2 Architettura di Dropper **23**
2.1 Modulo Core 26
2.1.1 Utilizzo di pianificatori 29
2.2 Modulo Gadgets 34
2.2.1 Traduzione di un gadget in formule logiche 34
2.2.2 Emulazione e catalogazione 36
2.2.3 Verifica e validazione 36
2.2.4 Verifica degli effetti secondari 37
2.3 Modulo Payload 38

3 Implementazione di Dropper **40**
3.1 Modulo Core 40
3.1.1 Interazione con l'utente 40
3.1.2 Analisi degli import 41
3.1.3 Costruzione della catena 41
3.2 Modulo Gadgets 43
3.2.1 Gadget per l'impostazione dei registri 44

3.2.2 Gadget per la scrittura in memoria 45
3.2.3 Gadget per la operazioni aritmetiche in memoria 46
3.3 Modulo Payload 48

4 Conclusioni **49**
4.1 Sviluppi futuri 51

Introduzione

A seguito dell'introduzione di misure di sicurezza come l'isolamento delle pagine di memoria, la casualizzazione dello spazio degli indirizzi (ASLR), e i controlli di consistenza dello stack, le tecniche per poter sfruttare le vulnerabilità presenti in un programma al fine di ottenerne il controllo sono cambiate radicalmente.

La tecnica forse più abusata era lo *stack smashing* [14], con la quale si compromisero centinaia di sistemi. La tecnica consiste nello sfruttare l'assenza o l'errato controllo dei limiti di un buffer memorizzato sullo stack per poter corrompere lo stack stesso, cioè veniva causato uno *stack buffer overflow*. In questo modo poteva essere "iniettato" nello stack codice arbitrario, solitamente definito *shellcode* per via del suo obiettivo più comune: ottenere una shell. Scegliendo accuratamente i dati che andranno a sovrascrivere lo stack, l'attaccante prende quindi controllo della macchina su cui gira il programma attaccato.

Infatti, nello stack sono presenti informazioni chiave per il funzionamento del codice macchina costituente un programma. Ad esempio, al momento di una chiamata a funzione, l'indirizzo dell'istruzione successiva alla chiamata stessa viene memorizzato sullo stack; in questo modo, una volta terminato il lavoro della funzione, il programma sa da dove riprendere l'esecuzione. L'attaccante, una volta preso il controllo dello stack, può sovrascrivere il valore di questo indirizzo di ritorno, controllando da dove il programma continuerà la sua esecuzione. All'attaccante quindi basta redirezionare il flusso del programma sul codice da lui stesso iniettato.

Sono state introdotte diverse misure di protezione per mitigare questo tipo di attacco. Con l'introduzione dell'isolamento delle pagine di memoria è stato possibile adottare una politica dei permessi in memoria con la granularità di una singola pagina. Il tentativo di eseguire delle istruzioni presenti in una pagina marcata come non eseguibile, così come il tentativo di scrivere su

una pagina marcata come non scrivibile causa l'arresto del programma. Il codice legittimo viene invece mappato in memoria in pagine eseguibili ma non scrivibili, mentre stack e dati, che il programma deve manipolare, vengono mappati su pagine scrivibili ma non eseguibili. In questo modo non si può più semplicemente iniettare del codice in posizioni di memoria arbitraria, in quanto le aree di memoria scrivibili (tra cui l'area di memoria riservata allo stack) non possono contenere istruzioni che possano essere eseguite, senza causare l'interruzione del programma da parte del sistema operativo.

Un'altra misura di protezione per rendere più difficile questo tipo di attacco è la casualizzazione dell'area di memoria nel quale risiede lo stack. In questo modo, non sapendo dove il codice iniettato effettivamente risiede, non si sa dove dover redirezionare il controllo del flusso per eseguire il codice iniettato.

Tuttavia l'isolamento delle pagine di memoria e la casualizzazione dello stack non evita la compromissione né del flusso del programma né dello schema dello stack. Un modo per ottenere il controllo di un programma nonostante lo stack non sia marcato come eseguibile, consiste nel riutilizzare il codice pre-esistente del programma stesso, come nella tecnica denominata *return-to-libc*[5, 8]. In un utilizzo tipico di questa tecnica, il flusso di esecuzione non viene dirottato su codice iniettato dall'attaccante ma su una funzione di una qualche libreria utilizzata dal programma. Su un'architettura a 32 bit gli argomenti vengono passati alle funzioni posizionandoli secondo un ordine preciso (definito dalla *calling convention*) sullo stack. Dato che anche lo schema dello stack può essere compromesso, l'attaccante ha, predisponendo un arrangiamento minuzioso dei valori da posizionare sullo stack, di fatto la possibilità di controllare gli argomenti da passare alla funzione. Potendo, ad esempio, richiamare la funzione `system` della libc con argomenti arbitrari è possibile eseguire un qualsiasi comando sulla macchina in cui si sta eseguendo il programma, oppure, chiamando la funzione `mprotect` (sempre nella libc) con gli opportuni argomenti, si possono modificare i permessi della pagina di memoria dove risiede lo stack, rendendola eseguibile. A quel punto redirezionare il flusso del programma sul codice iniettato sullo stack non ne provocherà l'arresto. La tecnica funziona perché il codice ad essere eseguito risiede in un area di memoria legittimamente eseguibile.

Una tecnica per mitigare questo tipo di attacco è la casualizzazione dello spazio degli indirizzi (*Address Space Layout Randomization*). Infatti nel momento in cui ASLR è attivo le librerie vengono mappate ad indirizzi di

memoria casuali, che cambiano da esecuzione ad esecuzione del programma. In questo modo non si conoscono gli indirizzi delle funzioni e un attacco di tipo return-to-libc naïve non risulta possibile. Tuttavia su 32 bit, e sotto alcune condizioni, questi indirizzi possono essere facilmente ricavati con un attacco a forza bruta[19]. Anche se su un'architettura 64 bit un attacco a forza bruta non risulta fattibile, è tuttavia possibile che si riesca a sfruttare qualche altro errore di programmazione per ottenere l'indirizzo della libreri che è fonte del codice riusabile, riuscendo così a montare comunque l'attacco.

Una generalizzazione della return-to-libc è la tecnica che prende il nome di *Return Oriented Programming*[18, 15]. In questo caso il flusso viene redirezionato non più all'inizio di una funzione, ma all'inizio di una piccola serie di istruzioni, già presenti in un area di memoria del programma marcata come eseguibile, seguita da un'istruzione *ret*. L'istruzione ret preleva dalla memoria dello stack un indirizzo e ridireziona il flusso del programma a quel dato indirizzo e viene solitamente utilizzata al termine dell'esecuzione di una funzione per tornare alla porzione di codice che l'ha richiamata. Potendo controllare lo stack l'attaccante può arrangiarlo in modo che una volta che l'esecuzione arrivi all'istruzione ret della prima serie, sullo stack sia presente l'indirizzo di una seconda serie dalle caratteristiche simili alla prima. Iterando questa tecnica più volte è possibile concatenare un numero arbitrario di queste piccole serie di istruzioni (che prendono il nome di *gadget*). L'insieme degli indirizzi dei gadget e dei parametri che se iniettati sullo stack deviano il flusso del programma in modo da eseguire una catena di gadget vengono denominati, sebbene con un piccolo abuso del termine, *payload*. La ROP è stata presentata per la prima volta in [18] e, quando sono presenti opportune tipologie di gadget, utilizzando il ROP si possono effettuare computazioni turing-complete. Tramite la ROP è possibile applicare tecniche per eludere varie misure di protezione [16] tra cui ASLR.

ASLR è un termine abbastanza generico e fin ora abbiamo presupposto che solo gli indirizzi delle librerie e dell'area dello stack siano casualizzati, in realtà quando un programma viene compilato affinché il suo codice sia indipendente dalla posizione in cui viene mappato in memoria (*Position Indipendent Executable*), anche il codice del programma stesso può essere caricato in un indirizzo casuale di memoria e cambiare tra un'esecuzione e l'altra. In questo caso non è più facilmente possibile montare un attacco del genere, in quanto non vi è più nessuna parte di codice che essendo mappata ad una posizione nota, possa essere sfruttata come destinazione del flusso del

programma. In altre parole anche se abbiamo il modo di influenzare il flusso del programma, non sappiamo dove redirezionarlo. Altre tecniche sono state sviluppate per mitigare attacchi di tipo ROP, anche se non risultano efficaci o l'overhead è troppo grande [7].

Lo scopo di questo lavoro di tesi è lo sviluppo di uno strumento che possa aiutare nel compito di generare un payload che, se iniettati sullo stack di un eseguibile, possa sostituire il normale flusso del programma con l'esecuzione di una serie di gadget che hanno come scopo quello di lanciare un programma arbitrario sulla macchina (ad esempio una shell). In particolare lo strumento riesce a generare catene di gadget per eludere misure di sicurezza come stack non eseguibile, casualizzazione dello spazio degli indirizzi e casualizzazione dello stack applicando, in maniera automatica, tecniche come *got patching*, *got dereferencing* e *return to plt* [16]. Lo strumento utilizza per l'analisi del binario il recente framework di sviluppo BARF [11] e supporta l'architettura x86 e x86-64. Per la costruzione di questo payload, in particolare per l'estrapolazione e l'analisi dei gadget, è decisivo l'apporto dato da tecniche e strumenti propri dell'Intelligenza Artificiale.

La *Satisfiability Modulo Theories*[3] è un'estensione della logica del primo ordine in cui l'interpretazione di alcune funzioni e predicati viene esplicitata da alcune teorie specifiche. Questo consente una modellazione naturale e efficiente di strutture dati come quelle dei bit vector e degli array, rendendolo un ottimo strumento per la modellazione dell'architettura di un elaboratore (specialmente dei registri e della memoria). Negli ultimi tempi è stata dedicata molta attenzione all'utilizzo di risolutori di formule SMT con applicazioni nell'ambito della sicurezza informatica[21]. Riuscendo infatti a modellare le istruzioni e il contesto di esecuzione di un programma in formule SMT, il risolutore può essere utilizzato, ad esempio, per verificare che un programma aderisca ad una data semantica o per estrapolarne la semantica stessa.

Anche l'esecuzione simbolica viene utilizzata sempre di più nell'ambito della sicurezza. Usando l'esecuzione simbolica, un interprete esegue le istruzioni da analizzare come in una simulazione ma, invece di utilizzare valori concreti per i valori d'ingresso, utilizza valori simbolici. Durante l'esecuzione, nelle varie operazioni vengono utilizzati i valori simbolici. In questo modo sia le espressioni che le variabili del programma, sia le condizioni che descrivono quando il flusso segue un ramo di esecuzione rispetto ad un altro, possono essere espresse in relazione ai valori simbolici. Queste relazioni possono essere espresse sotto forma di formule SMT in modo da poter interrogare

un risolutore, al fine di chiedere, ad esempio, per quali valori in ingresso il programma segua o meno un certo ramo d'esecuzione.

Nell'ambito del ROP un risolutore SMT può essere utilizzato sia per estrapolare in maniera automatica la semantica di un gadget, sia per ottenere le condizioni di partenza che consentano ad un gadget di eseguire una determinata funzione, ad esempio i valori di registro per i quali un dato gadget scrive in una specifica locazione di memoria un dato valore.

Questo elaborato è organizzato come segue: la prima parte del primo capitolo è una breve panoramica tecnica sui meccanismi principali coinvolti nell'esecuzione di un programma che ci consente di avere le basi, per analizzare, nella seconda parte dello stesso capitolo, alcune delle principali tecniche che vengono utilizzate oggi per eludere le più comuni misure di protezione attraverso la ROP. Il secondo capitolo invece è una descrizione dettagliata dell'architettura designata per Dropper, il sistema di payload generation presentato in questa tesi. Il terzo capitolo parla invece dell'attuale implementazione di Dropper. Infine, nelle conclusioni, sono presenti considerazioni generali emerse durante il lavoro di tesi nonché sugli sviluppi futuri di Dropper.

Capitolo 1

Panoramica tecnica

L'utilizzo di varie tecniche di protezione hanno reso sempre più difficile lo sfruttamento di errori nell'esecuzione di un programma al fine di prenderne il controllo. Questo capitolo introduce brevemente alcune di queste tecniche e analizza con quali metodi è possibile eluderle. Anche se alcuni concetti sono indipendenti dall'architettura e dal sistema operativo si tratterà esclusivamente di come queste tecniche sono implementate in ambiente GNU/Linux, su architettura x86 e x86-64. Nella prima parte di si parlerà brevemente dei principali meccanismi con i quali il sistema operativo, partendo dalle informazioni presenti all'interno di un file oggetto eseguibile, crea una rappresentazione dinamica del programma, detta *immagine del processo*. Nella seconda parte invece verranno analizzate alcune tecniche di exploit.

1.1 Creazione dell'immagine di un processo

Su sistemi operativi GNU/Linux per la rappresentazione statica dei programmi viene utilizzato il formato ELF. ELF (Executable and Linkable Format) può rappresentare tre tipi di file oggetto:

- *file oggetto rilocabile*: contiene dati e codice che vengono collegati ad altri file oggetto al fine di ottenere un *file oggetto eseguibile* o un *file oggetto condiviso*
- *file oggetto eseguibile*: è la rappresentazione statica di un programma che può essere eseguito, questo file ha le informazioni necessarie affinché il sistema operativo crei la sua immagine di processo

- *file oggetto condiviso* che contiene codice e dati adatti per essere collegati (nel processo di linking) in due contesti differenti. Può essere infatti utilizzato sia in fase di building insieme ad altri file oggetto rilocabili o file oggetto condivisi (*linking statico*) per creare altri file oggetto, oppure può essere collegato in fase di creazione dell'immagine del processo di un file oggetto eseguibile (*linking dinamico*).

Il formato ELF può contenere sia le informazioni necessarie per processi relativi alla fase di creazione di diversi file oggetto a partire da file sorgente (in particolare informazioni necessarie per la fase di linking), sia relativamente allo scopo della creazione dell'immagine di un processo a partire da un file oggetto eseguibile. Anche le strutture dati all'interno dell'ELF afferenti di questi due "punti di vista" sono diverse, e, a seconda della tipologia del file oggetto alcune delle strutture dati possono non essere presenti. Essendo più rilevante ai fini del nostro lavoro ci concentreremo principalmente sulle strutture dati rilevanti nelle operazioni di creazione dell'immagine del processo e nelle procedure di dynamic linking, ovvero nel processo con il quale si rendono fruibili all'eseguibile i metodi esposti da una libreria condivisa a tempo di esecuzione.

1.1.1 Program Header Table

L'ELF header (che è l'unica struttura dati che ha una posizione fissa all'interno del file) funge da una sorta di mappa, e ci consente di localizzare tutte le altre strutture dati presenti nel file.

Tra queste vi è la *Program Header Table*. La Program Header Table è un array di strutture, ognuna delle quali contiene o informazioni su come costruire un segmento della memoria dell'immagine di un processo o informazioni per la preparazione dell'immagine stessa. Un elemento all'interno della Program Header Table (su un'architettura a 32bit) è rappresentata dalla seguente struttura C:

```
1 typedef struct {
2   uint32_t   p_type;
3   Elf32_Off  p_offset;
4   Elf32_Addr p_vaddr;
5   Elf32_Addr p_paddr;
6   uint32_t   p_filesz;
7   uint32_t   p_memsz;
8   uint32_t   p_flags;
```

```
 9    uint32_t   p_align;
10  } Elf32_Phdr;
```

Listing 1.1: Program Header Table entry

La struttura relativa ad un'architettura a 64bit è pressoché identica, l'unica differenza sta nella posizione degli attributi. L'attributo `ptype` indica il tipo di entry. Per il nostro scopo ci limiteremo a descrivere le entry di tipo `PT_LOAD`, `PT_INTERPETER` e `PT_DYNAMIC`. Le entry di tipo `PT_LOAD` rappresentano informazioni su un segmento da caricare in memoria. Il campo `p_offset` e `p_filesz` indicano rispettivamente l'offset e la sua grandezza all'interno del file. `p_vaddr` e `p_memsz` invece indicano rispettivamente il base address nel quale caricare in memoria il segmento e la dimensione che avrà in memoria (che potrebbe essere più grande rispetto a quella sul file). `p_flags` è molto rilevante per il nostro lavoro e indica con quali flags il segmento andrà ad essere caricato in memoria ovvero se il segmento sarà o meno scrivibile, leggibile e/o eseguibile. Tuttavia nel caso il segmento contenga codice compilato affinché risulti indipendente dall'indirizzo base nel quale risiede in memoria, il campo `p_vaddr` potrebbe essere nullo e la posizione del segmento in memoria potrebbe essere casualizzata.

La voce all'interno della Program Header Table con il tipo `PT_INTERP` invece contiene una stringa che rappresenta il percorso di un file oggetto eseguibile o condiviso all'interno del filesystem, detto interprete. Il sistema operativo crea l'immagine del processo dell'interprete, dandogli il controllo. Sarà compito poi dell'interprete creare l'immagine del processo necessaria all'esecuzione dell'eseguibile. Affinché questo possa accadere l'interprete avrà accesso alle informazioni presenti nel file principale. Normalmente l'interprete è costituito da codice indipendente dalla posizione nella quale viene caricato, che viene casualizzata evitando conflitti tra gli spazi di memoria utilizzati dall'eseguibile principale e quelli dell'interprete stesso.

Quando viene creato del codice oggetto che utilizza delle librerie dinamiche, il linker aggiunge alla Program Header Table un elemento di tipo `PT_INTERPRETER` con impostato come interprete il *dynamic linker*, che si occupa di trovare le librerie necessarie all'eseguibile, caricarle in memoria, caricare in memoria i segmenti dell'eseguibile, risolvere le relocation verso i simboli delle librerie, e ridare poi controllo all'eseguibile stesso. Come vedremo la risoluzione di un simbolo potrebbe essere rimandata fin quando non sia realmente necessario. A supporto di questi processi troviamo alcune strutture dati, anch'esse aggiunte al file oggetto eseguibile durante la fase di linking. Queste

strutture risiedono all'interno di segmenti che vengono caricati in memoria, e sono quindi disponibili durante l'esecuzione del programma. Informazioni su dove trovare queste strutture dati possono essere ricavate attraverso un'altra struttura dati del file ELF, la *Section Header Table*, che è un array di strutture che descrivono, per l'appunto, le parti che compongono il file, dette sezioni. Tuttavia, anche se nel nostro lavoro di tesi le sezioni vengono utilizzate per ottenere informazioni sul binario esaminato, è da notare che esse non sono necessarie per un file oggetto di tipo eseguibile. Un file oggetto infatti può essere caricato esclusivamente con le informazioni presenti nella Program Header Table (lo strumento sstrip[12] elimina appunto le sezioni da un file eseguibile). Tuttavia è conveniente per scopi illustrativi riferirsi alle strutture dati relative al processo di dynamic linking riferendosi alle rispettive sezioni, che sono:

- Una sezione `.dynamic` che contiene gli indirizzi di altre strutture necessarie al processo di dynamic linking e la lista delle librerie necessarie all'esecuzione del file oggetto eseguibile
- Una sezione `.hash` che contiene una tabella dei simboli
- Le sezioni `.got` e `.plt` contengono rispettivamente due tabelle: *la Global Offset Table* e *la Procedure Linkage Table*. Nelle sezioni successive vedremo come queste due strutture dati vengono utilizzate dal dynamic linker per risolvere i simboli e le chiamate a funzioni presenti nelle librerie dinamiche

1.1.2 Global Offset Table e Procedure Linkage Table

Un programma che sia indipendente dalla posizione in cui viene caricato in memoria non può contenere al suo interno indirizzi assoluti. Le Global Offset Table (`GOT`) contengono indirizzi assoluti in un'area di memoria privata destinata ai dati, non compromettendo quindi l'indipendenza del codice dalla posizione in memoria (e quindi che sia condivisibile da più immagini di processo, come nel caso di una libreria dinamica). Il linker, una volta creata l'immagine di un processo, processa tutte le rilocazioni (strutture dati che contengono informazioni per la risoluzione dei simboli) di tipo `R_386_GLOB_DAT` e per ognuna di esse calcola l'indirizzo assoluto. Il linker conoscendo l'indirizzo di tutti i file oggetto caricati in memoria ha tutte le informazioni necessarie

per calcolare il valore di questi indirizzi. Una volta calcolati il linker inserisce i valori assoluti nei rispettivi elementi della GOT, permettendo all'eseguibile di accedervi attraverso posizioni relative.

A noi interessa principalmente come la GOT è coinvolta nel momento in cui un file oggetto eseguibile esegue una chiamata ad una funzione di una libreria dinamica, processo in cui è coinvolta anche una seconda tabella, la Procedure Linkage Table (PLT).

La PLT redireziona chiamate a funzioni che utilizzano indirizzi relativi a indirizzi assoluti calcolati a tempo d'esecuzione. In fase di building di un eseguibile il linker fa in modo che chiamate a funzioni presenti in librerie dinamiche vengano direzionate a elementi della PLT. Anche se la PLT risiede nel segmento di memoria destinato a contenere il codice di un eseguibile, utilizza valori nella Global Offset Table, non compromettendo così né l'indipendenza dalla posizione ne la condivisibilità del codice. È compito del dynamic linker calcolare i valori degli indirizzi della funzione chiamata e impostarli nei relativi elementi della GOT. Nella restante parte di questa sezione andremo a descrivere come il dynamic linker utilizza queste due tabelle per risolvere gli indirizzi, facendo riferimento all'implementazione su un'architettura a 32bit. Tuttavia il concetto principale resta pressoché invariato anche per un'architettura a 64bit. Nel listato 1.2 è illustrata la struttura della PLT.

```
1  .PLT0:
2      pushl   got_plus_4
3      jmp     *got_plus_8
4      nop;    nop
5      nop;    nop
6  .PLT1:
7      jmp     *name1_in_GOT
8      pushl   $offset
9      jmp     .PLT0@PC
10 .PLT2:
11     jmp     *name2_in_GOT
12     pushl   $offset
13     jmp     .PLT0@PC
14     ...
15     ...
```

Listing 1.2: Procedure linkage table

La risoluzione dell'indirizzo può essere sommarizzata nei seguenti punti:

- Quando viene creata l'immagine del processo i primi due valori nella GOT assumono valori particolari (spiegati di seguito)

- Quando l'eseguibile esegue una chiamata di una funzione in una libreria dinamica il flusso del programma viene indirizzato nella `PLT`. Nel nostro esempio viene chiamata la funzione `name1`, che avrà come indirizzo di destinazione l'istruzione marcata dall'etichetta `.PLT1`

- Il programma a questo punto esegue un `jmp` all'indirizzo contenuto nell'elemento nella GOT relativo a `name1`. Al momento di caricamento del programma, a eccezione di casi particolari come descritto di seguito, questo elemento è impostato con il valore dell'istruzione successiva al `jmp` stesso, cioè all'istruzione alla linea 7 del listato 1.2

- Il codice a questo punto salva sullo stack l'offset all'interno della tabella delle rilocazioni che permette di individuare la rilocazione relativa al simbolo `name1`. La rilocazione permette di ricavare sia l'elemento della `GOT` relativo a quel simbolo che il nome del simbolo stesso, fornendo le informazioni necessarie al dynamic linker per capire quale simbolo è stato chiamato e qual è l'elemento da modificare nella `GOT`.

- A questo punto il programma salta al primo elemento della `PLT` e, dopo aver salvato sullo stack il secondo elemento della `GOT`, salta all'indirizzo contenuto nel terzo elemento della got, che dà il controllo al dynamic linker

- Il dynamic linker esamina lo stack, controlla quale simbolo è stato chiamato, calcola l'indirizzo e imposta il valore nell'elemento relativo nella `GOT`. In questo modo una seconda chiamata a `name1` non causerà una seconda chiamata al linker ma salterà direttamente all'indirizzo corretto

Le figure 1.1 e 1.2 riassumono i flussi di esecuzioni del programma illustrativo presente nel listato 1.3 evidenziando le differenze tra la prima chiamata alla funzione `printf` e la seconda.

L'approccio appena descritto, nel quale gli indirizzi delle funzioni vengono risolti solo al momento in cui vengono chiamate, viene definito *lazy binding*. Se invece il programma viene eseguito con la variabile d'ambiente `LD_BIND_NOW` impostata gli indirizzi vengono tutti calcolati e scritti nella `GOT` al momento del caricamento dell'eseguibile. Questo comportamento avviene

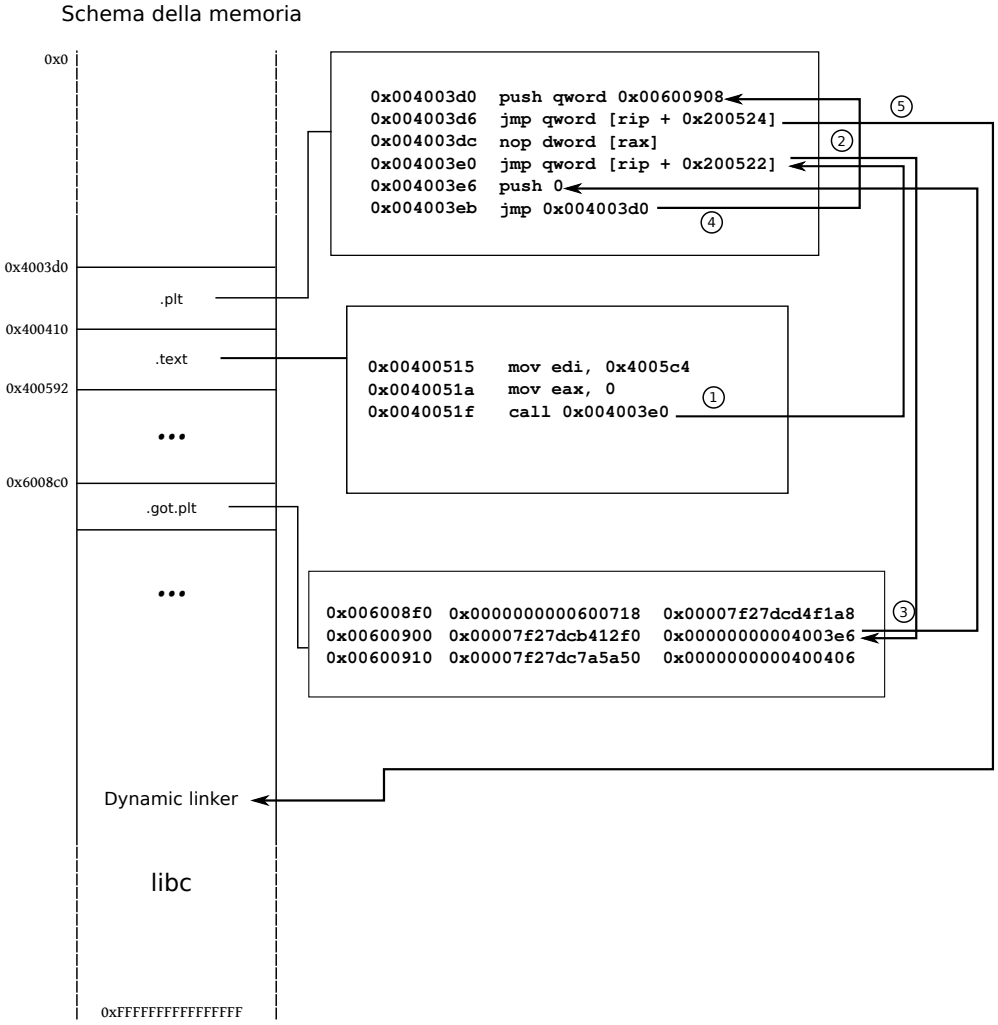

Figura 1.1: Fasi della chiamata ad una funzione di una libreria condivisa prima che nella GOT sia presente l'indirizzo della funzione

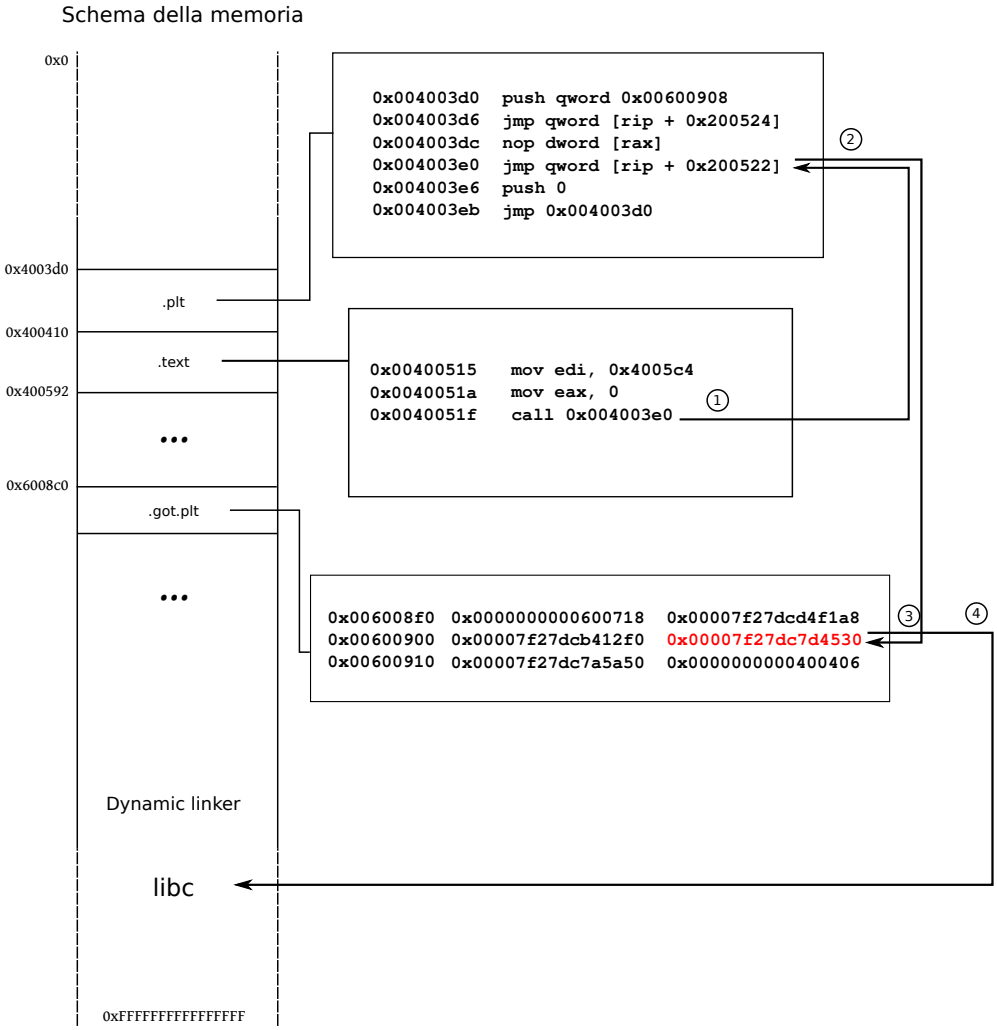

Figura 1.2: Fasi della chiamata ad una funzione di una libreria condivisa dopo che la GOT sia stata modificata con l'indirizzo della funzione dal dynamic linker

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main(int argc, char *argv[]) {
5   printf("prima");
6   printf("seconda");
7   return 0;
8 }
```

Listing 1.3: codice del programma a cui fanno riferimento le figure 1.1 e 1.2

anche quando si compila un eseguibile con l'opzione `RELRO`. Inoltre, in questo caso, una volta modificata la `GOT` viene resa non più scrivibile, in modo da mitigare attacchi di tipo got patching.

1.2 Return Oriented Programming

Per mitigare classici attacchi di tipo stack-smashing sono state elaborate diverse tecniche per precludere la possibilità di eseguire il codice iniettato da un attaccante. La prima di questa tecniche è stata implementata in una patch di Solar Designer [9], che modifica lo schema della memoria dell'immagine di un processo al fine di rendere le istruzioni presenti nello stack non eseguibili. Dato che nella maggior parte dei casi l'attaccante utilizzava lo stack come locazione del codice iniettato questa patch rendeva innocui molti di questi attacchi. Una tecnica più completa, chiamata "$W \oplus X$", assicura invece che non ci sia una pagina di memoria all'interno del processo che sia scrivibile ed eseguibile nello stesso momento. Per eludere questa tecnica un attaccante è costretto a non utilizzare più codice da lui stesso iniettato ma codice già presente nell'immagine del processo bersaglio (il primo a suggerire quest'approccio è stato Solar Designer [8]). Dato che la libreria C standard è praticamente utilizzata in quasi tutti i programmi Unix e contiene funzioni utili ad un attaccante (come `system` o `execve`) il codice utilizzato solitamente è proprio quello della libc, da cui il nome con cui vengono catalogati questo tipo di attacchi: *return-to-libc*. Tra le misure per mitigare attacchi di questo tipo fu ipotizzato la rimozione di alcune funzioni dalle libc al fine di rendere questo tipo di attacchi meno efficace. Shacham in [18] ha evidenziato come questo tipo di protezione fosse in realtà inefficace introducendo per la

prima volta la tecnica da lui battezzata come *Return Oriented Programming* (ROP), in cui venivano usate piccole serie di istruzioni di codice già presente nell'immagine del processo e non intere funzioni come in attacchi di tipo return-to-libc. Anche se, in questa prima applicazione, la tecnica estrapolava queste serie di istruzioni dalla libc, in [17] è stato evidenziato come questa tecnica restasse valida nonostante le istruzioni venissero direttamente prelevate dal testo dell'eseguibile e non dalle sue librerie. In questo modo si può utilizzare la ROP per eludere tutta un'altra serie di protezioni che si basano sulla casualizzazione della posizione di aree di memoria chiave all'interno dell'immagine del processo di un eseguibile (in particolar modo delle aree che contengono le librerie condivise e di quella riservata allo stack) detta appunto *Addres Space Layout Randomization*(ASLR). Nell'applicazione della ROP vengono individuate piccole serie di istruzioni con delle particolari caratteristiche, chiamate *gadget*. Arrangiando minuziosamente lo stack si possono eseguire gadget uno dietro l'altro, cicli o computazioni arbitrarie. Un punto importante è che, almeno per quanto riguarda l'architettura intel, possono essere individuate all'interno di un file oggetto molte sequenze di istruzioni che non sono "intenzionali", ovvero inserite dal compilatore come le istruzioni relative alla traduzione di codice sorgente. Infatti, essendo la codifica delle istruzioni in linguaggio macchina molto densa e non allineata, se si iniziano ad interpretare le istruzioni partendo dal centro di un'altra istruzione, vi è molta probabilità di decodificare una sequenza di istruzioni alternativa e valida. Questo è dovuto alla natura stessa della codifica, quello che Shacham chiama geometria.

1.2.1 Gadget

In questa sezione analizzeremo più in dettaglio i gadget e le loro caratteristiche. Bisogna innanzitutto considerare che essendo i gadget composti da una piccola serie di istruzioni questo tipo di attacchi agisce ad un livello più basso rispetto a quello degli attacchi di tipo return to libc. I gadget possono essere visti come delle istruzioni di basso livello di uno strano calcolatore. Queste istruzioni non vengono concatenate in maniera standard ma arrangiando minuziosamente i valori presenti nello stack. I gadget sono terminati dall'istruzione `ret` e nelle istruzioni che lo compongono non sono contenuti salti e altre istruzioni che deviano in qualche modo il flusso del programma (tecniche che utilizzano anche questo tipo di gadget sono state sviluppate,

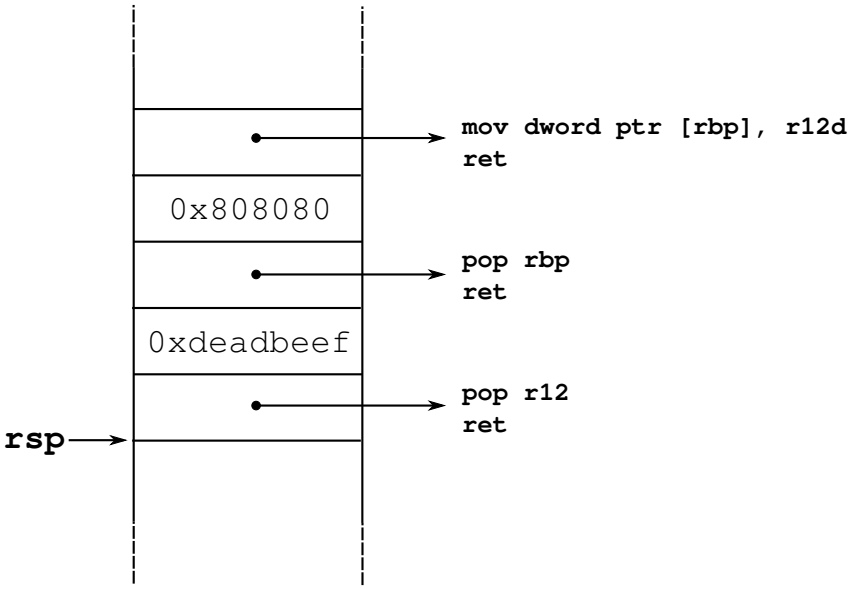

Figura 1.3: Payload per la scrittura in memoria di una costante

come in [6]). Aggiustando lo stack in modo che nel momento in cui il flusso del programma arrivi all'istruzione `ret` di un gadget il registro dello stack punti alla zona di memoria che contiene l'indirizzo del prossimo gadget si riesce a concatenare un gadget al successivo. I gadget, soli o in combinazione, possono svolgere diverse funzioni. Per quanto riguarda, ad esempio, le operazioni di lettura o scrittura possiamo notare che:

- gadget del tipo `pop REG;ret` ci permettono di caricare una costante in un registro
- gadget del tipo `mov reg1 ,[reg2 + imm];ret` ci permettono di leggere un valore dalla memoria (il valore di `reg2` può essere impostato con un gadget del punto precedente) e scriverlo su un registro
- gadget del tipo `mov [reg1 + imm], reg2` ci permettono di scrivere in una posizione di memoria (i valori di `reg1` e `reg2` possono essere impostati con gadget che ci consentono di caricare una costante di un registro)

Un esempio di come una serie di indirizzi e valori possano essere posizionati accuratamente sullo stack per effettuare una scrittura in memoria (la costante `0xdeadbeef` nella locazione `0x808080` è rappresentato in figura 1.3).

In base all'eseguibile che stiamo considerando però potrebbero non essere disponibili gadget esattamente come questi. In particolare potrebbero presentarsi altre istruzioni tra quella che a noi effettivamente interessa (perché svolge la funzione a noi "utile") e l'istruzione `ret`. A volte modificando opportunamente lo stack si può fare in modo che gli effetti di queste istruzioni non incidano sulla nostra computazione. Ovviamente altri gadget, o sequenze di gadget, possono svolgere funzioni equivalenti e, il fatto che una categoria di gadget non è disponibile non implica che la stessa funzione non possa essere svolta utilizzando gadget e/o strategie differenti. Ad esempio se non abbiamo a disposizione un gadget che ci permetta di impostare il registro `eax` tramite `pop eax;ret` un'opportuna combinazione di `mov eax, 0xFFFFFFFF;ret` e `inc eax;ret` ci permette (non considerando restrizioni sulla lunghezza del payload) di impostare valori arbitrari in `eax`. Estrapolare, in maniera automatica, sequenze e combinazioni di gadget che ci permettano di raggiungere condizioni desiderate gestendo gli effetti secondari delle istruzioni "superflue" rappresenta la parte più impegnativa dell'implementazione di Dropper.

Le operazioni che l'architettura ci permette di eseguire non si limitano ad operazioni di lettura e scrittura. Infatti possiamo eseguire operazioni aritmetiche, operazioni logiche e utilizzare sequenze di gadget che ci permettono di controllare il flusso del programma. Come considerato nel paragrafo precedente spesso i metodi per eseguire queste operazioni non sono diretti, ma è necessario considerare gli effetti secondari e operare solo con i gadget a disposizione. Ad esempio possiamo avere a disposizione solo operazioni che operano su dimensioni di un byte, e per eseguire operazioni su dimensioni maggiori è necessario concatenare più volte la stessa operazione eseguendola un byte per volta, oppure un'operazione potrebbe essere eseguita utilizzando una serie di più operazioni ma che nel complesso sia equivalenti all'operazione voluta.

1.2.2 Address Space Layout Randomization

L'*'Address Space Layout Randomization* (ASLR) è una tecnica di protezione che consiste nel caricare in posizione casuale regioni della memoria del programma. Questo è possibile perché un'istruzione può riferirsi ad una locazione di memoria tramite la distanza tra il proprio indirizzo (contenuta nel registro instruction pointer) e la locazione in esame. Non utilizzando quindi più indirizzi assoluti il codice può risultare indipendente dalla posizione di

memoria in cui è caricato a patto che che gli offset relativi restino invariati (cioè che il codice non venga "mischiato"). Il dynamic linker e alcune strutture dati vengono utilizzate per far funzionare porzioni di codice indipendenti anche dagli offset relativi (come nel caso del testo dell'eseguibile e le librerie dinamiche), utilizzando un livello di indirezione (vedi sez. 1.1.1 e 1.1.2).

L'utilizzo dell'ASLR porta due complicazioni principali nell'utilizzo di tecniche basate sulla ROP. Prima di tutto non conoscendo l'indirizzo di una porzione di codice non si possono conoscere neanche gli indirizzi dei gadget contenuti all'interno. Questo riduce notevolmente la quantità di gadget a disposizione, e, a seconda di quanto codice venga casualizzato può rendere impossibile un attacco che utilizzi la ROP. Infatti anche se solitamente, per questioni di efficienza, solo le librerie vengono casualizzate, è possibile compilare un'eseguibile in modo che una volta eseguito tutte le aree che contengono codice vengano casualizzate (*Position Indipendent Executable* (PIE)). In quest'ultimo scenario, a meno di non ottenere informazioni sulla posizione di memoria in altri modi (ad esempio tramite altre vulnerabilità che espongano indirizzi di strutture dell'eseguibile), l'applicazione della ROP non risulta possibile.

Inoltre a venire casualizzate non sono solo porzioni che contengono codice, ma anche la porzione di memoria riservata allo stack. Non conoscendo la posizione dello stack risulta molto più difficile creare sequenze di gadget che possano eseguire operazioni di controllo del flusso. Una possibile soluzione potrebbe essere quella di iniettare gli indirizzi dei gadget in un'altra posizione (nota) e modificare il valore del registro dello stack perché punti a quella zona di memoria, ovvero creare un *Fake Stack Frame*. In più non avendo più accesso a librerie dinamiche come la libc, difficilmente si riesce a trovare nella porzione di codice non casualizzato istruzioni che ci consentano di lanciare una syscall (solitamente il programma fa affidamento alla libreria C per eseguirle e quindi dovrebbe comparire tra le sequenze di istruzioni non "intenzionali"). Tuttavia in [17] è stato dimostrato come anche utilizzando gadget che si incontrano in porzioni di codice relativamente piccole è possibile eseguire computazioni arbitrarie, addirittura concatenando questi gadget in maniera automatica. In più come vedremo nelle prossime sottosezioni è possibile, sotto alcune condizioni, ricavare gli indirizzi di funzioni presenti nella libc partendo dalle strutture dati coinvolte nelle operazioni di dynamic linking. Se le giuste tipologie di gadget sono disponibili è possibile quindi eseguire comandi arbitrari (come lanciare una shell) anche in contesti in cui

stack e librerie sono casualuzzati.

1.2.3 Return-to-plt, GOT dereferencing e GOT patching

In questa sottosezione presupponiamo che il codice del testo dell'eseguibile non sia stato casualizzato. Anche se le librerie sono casualizzate se un'eseguibile utilizza una funzione presente in una libreria dinamica il linker, in fase di building, inserisce un elemento della PLT relativo a quella funzione (vedi sez. 1.1.1). Se la posizione in cui viene caricato il codice è nota lo è anche quella della PLT e quindi l'indirizzo dell'elemento. Questo ci permette di poter "utilizzare" nella nostra catena le funzioni che utilizza l'eseguibile bersaglio, dirottando il flusso all'indirizzo del relativo elemento nella `PLT`, da qui il nome *return-to-plt*. Una volta utilizzata una funzione il valore dell'indirizzo assoluto di quella funzione si verrà a trovare nell'elemento relativo nella GOT (vedi sez. 1.1.2). Come descritto in [16] questo ci permette di utilizzare due tecniche di attacco particolari: *GOT dereferencing* e *GOT patching*. Tutte e due le tecniche utilizzano l'indirizzo assoluto di una funzione contenuto nell'elemento relativo nella GOT per calcolare l'indirizzo assoluto di una funzione della libc desiderata, di fatto eludendo ASRL. Nella prima tecnica il valore viene letto, modificato tramite un'operazione aritmetica e viene poi eseguito un salto a questo indirizzo. Un esempio di gadget che consentono questo tipo di attacco sono: `add eax,[ebx+off];ret` e `jmp [eax]`. La seconda tecnica invece modifica l'elemento della GOT in loco, utilizzando poi un return-to-plt per richiamare la funzione voluta. Un esempio di gadget che può essere utilizzato per modificare la funzione in loco è:

```
adc byte ptr [esi + 0x5f], bl ; pop ebp ; ret
```

Infatti concatenando diverse istruzioni del genere insieme ad opportuni gadget che impostarino i registri `esi o ebx` è possibile aggiungere un offset arbitrario in una locazione di memoria arbitraria. È da notare che questa tecnica non funziona nel caso la GOT sia in un'area di memoria non scrivibile (ad esempio se l'eseguibile è stato compilato con l'opzione RELRO) ma i gadget utilizzati in questa seconda tecnica sono di un tipo più comune rispetto a quelli utilizzati nella GOT dereferencing.

Capitolo 2

Architettura di Dropper

In questo capitolo si fa una descrizione dettagliata dei requisiti e delle funzionalità presenti nell'architettura designata per lo sviluppo di Dropper.

L'obiettivo di Dropper è quello di generare, analizzando un file oggetto eseguibile, il payload corrispondente ad una catena di gadget che se iniettato nello stack dell'immagine del processo associato a quell'eseguibile, consente di prenderne il controllo, eseguendo un comando arbitrario sulla macchina. Lo strumento applica tecniche note per eludere misure di protezioni esistenti come la casualizzazione degli indirizzi (`ASRL`), stack e aree dati non eseguibili ("W $\oplus$ X") o la read-only relocation, automatizzando il più possibile la scelta della strategia appropriata.

Dropper analizza il binario utilizzando alcune delle funzionalità del recente framework per l'analisi dei binari `BARF` [11]. Il linguaggio utilizzato per la scrittura di Dropper è python e, nella sua prima versione, supporta le architetture `x86` e `x86-64` su piattaforme GNU/Linux (il formato degli eseguibili supportato è ELF). Tuttavia come tratteremo meglio nel corso di questo capitolo, queste restrizioni possono essere facilmente superate nelle future versioni grazie all'astrazione data dall'utilizzo di un linguaggio intermedio per rappresentare le istruzioni.

Una tipica sessione dell'utilizzo di Dropper può essere grossolanamente divisa in quattro fasi:

1. Analisi del binario
2. Estrapolazione e analisi dei gadget
3. Scelta della strategia da utilizzare

4. Creazione del payload

Nella prima fase, quella in cui si analizzano le informazioni presenti nel file oggetto eseguibile esaminato, vengono ricavate le informazioni sulle misure di protezione attualmente in uso sul file oggetto eseguibile, le funzioni utilizzate e le librerie dinamiche importate. Queste informazioni verranno utilizzate specialmente nella terza fase.

Dropper, nella seconda fase, ricerca all'interno del file binario i gadget disponibili. I gadget trovati vengono tradotti, dal framework BARF, in un linguaggio intermedio: REIL [20]. Effettuare le analisi su un linguaggio intermedio anziché sulle istruzioni dell'architettura vera e propria porta una serie di vantaggi:

- le operazioni di analisi vengono astratte e rese indipendenti dalla specifica architettura. In questo modo è possibile aggiungere il supporto di altre architetture allo strumento in modo molto più semplice, operazione che si riduce a sviluppare un traduttore da questa architettura al linguaggio intermedio
- REIL ha un insieme di istruzioni molto ridotto (solo quattordici istruzioni), questo facilità notevolmente lo sviluppo di strumenti di analisi
- le istruzioni di REIL non presentano nessun effetto secondario implicito, come invece avviene per la maggior parte delle istruzioni delle architetture reali

Una volta trovati e tradotti in un linguaggio intermedio è necessario estrapolare la semantica dei gadget. Questo permetterà nella fase successiva l'assemblaggio di piccole catene che consentano di eseguire operazioni particolari (come modificare il valore dello stack, chiamare una funzione, settare un registro, scrivere in memoria, operazioni aritmetiche, leggere dalla memoria, ecc). Dropper utilizzerà nella fase di creazione del payload queste piccole catene per assemblare catene via via più lunghe che eseguono operazioni più complesse e articolate. L'architettura prevede che la generazione di queste sequenze avvenga sia in maniera programmatica, in cui la procedura per la sua costruzione è codificata in un algoritmo, sia in maniera dichiarativa, ad esempio utilizzando linguaggi di planning come `PDDL`[2]. Un analisi sui vantaggi e sui problemi che si incontrano in una codifica del genere è riportata nella sezione 2.1.1.

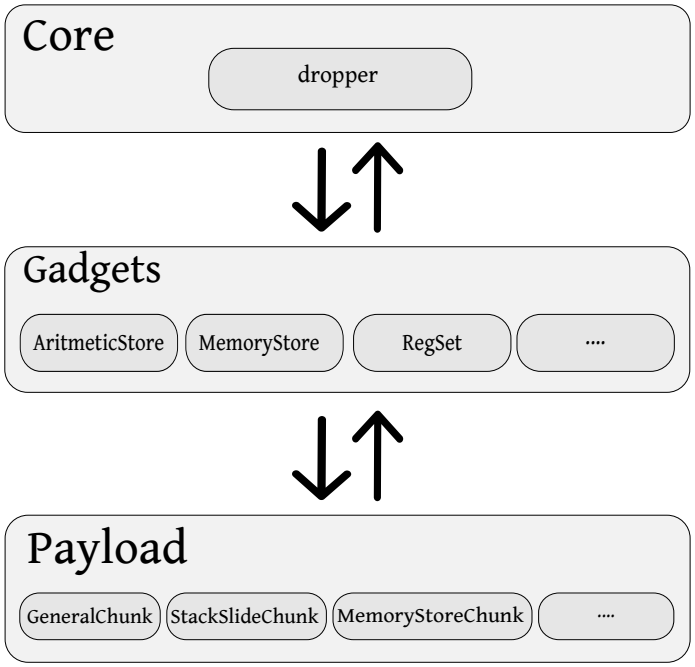

Figura 2.1: Architettura di Dropper

Nella terza fase, per scegliere la tecnica di exploit appropriata, Dropper si basa sia su alcune informazioni che fornisce l'utente in merito all'errore che si sta sfruttando (ad esempio se può controllare un file descriptor, se ci sono limiti nella dimensione del payload o nei caratteri ammessi) sia su quelle ricavate dalla prima fase di analisi dell'eseguibile. Dev'essere semplice estendere Dropper con nuove strategie.

L'ultima fase si occupa di assemblare gli indirizzi dei gadget e valori concreti per la generazione di un payload che, una volta iniettato sullo stack, comporti l'esecuzione delle varie catene assemblate nelle fasi precedenti.

L'architettura di Dropper si compone di tre moduli logici, rappresentati in figura 2.1:

- Core: È il modulo che si occupa di a) leggere il binario, b) coordinare le operazioni dei restanti moduli, c) interagire con l'utente per ottenere informazioni aggiuntive sul binario e/o sullo scenario nel quale dev'essere prodotta la catena e d) scegliere la strategia di exploit più appropriata

- Gadgets: È il modulo che si occupa dell'estrapolazione, dell'analisi, della scelta e dell'assemblaggio dei gadget da utilizzare nella composizione della catena e dei vari sui frammenti
- Payload: È il modulo che si occupa di trasformare una sequenza ordinata di gadget in un payload vero e proprio da iniettare sullo stack, gestendo il posizionamento corretto degli indirizzi e dei valori utilizzati dai vari gadget

2.1 Modulo Core

È il modulo che si occupa di leggere il file binario, interagire con l'utente, scegliere le varie strategie da adottare e di coordinare le azioni degli altri moduli.

Dropper parsa l'ELF e ricava la lista delle funzioni che il file oggetto eseguibile utilizza, i relativi elementi nella *Procedure Linkage Table* (vedi 1.1.1), le aree di memoria all'interno dell'immagine del processo dove è possibile scrivere e da dove è possibile leggere e le misure di protezione utilizzate dal file binario. Per il parsing dell'ELF Dropper utilizza una libreria apposita, pyelftools ([4]).

L'utente poi, interagendo con il modulo, fornisce informazioni aggiuntive necessarie alla generazione della catena e linee guida sulla strategia da utilizzare. Tra le informazioni che può fornire l'utente troviamo:

- La possibilità da parte dell'utente di controllare un file descriptor
- I file oggetto condivisi utilizzati dall'eseguibile sulla macchina dove si intende far eseguire la catena. Questi file vengono analizzati al fine di ricavarne gli offset da utilizzare nelle tecniche di sovrascrizione della GOT (vedi sez. 1.2.3)
- Eventuali caratteri che non possono comparire all'interno del payload
- Eventuali limiti sulla lunghezza del payload
- Il comando da eseguire

Con tutte queste informazioni il modulo può quindi procedere nello stabilire quale strategia utilizzare e richiedere al modulo Gadgets le sequenze di

gadget necessarie. Lo schema principale delle varie strategie, si può dividere in due fasi:

1) preparare e scrivere in memoria gli argomenti per eseguire una chiamata ad una funzione di libreria che ci consenta di eseguire un comando arbitrario (`execve` o `system`) e 2) eseguire quella funzione.

Per scrivere in memoria è possibile utilizzare diverse strategie, Dropper deve scegliere la più adatta o quella specificata dall'utente. Il modulo deve essere facilmente estendibile per aggiungere nuove strategie. Alcune delle strategie più efficienti per scrivere in memoria sono:

- Nel caso sia possibile inserire arbitrariamente dei valori in un file descriptor (ad esempio se l'utente può inserire uno stream di dati nell'input del programma) allora è comodo utilizzare la funzione `read` della `libc` per copiare questi dati nell'area di memoria di destinazione. Dropper fornisce i valori da "inserire" nello stream dati del file descriptor. L'attaccante deve fare in modo che quando la funzione `read` venga richiamata siano letti dal file descriptor proprio quei dati

- Utilizzo di catene di gadget che consentano di scrivere valori costanti in memoria. Per poter utilizzare questa tecnica devono essere disponibili i gadget delle categorie necessarie, in più i valori da caricare in memoria non devono presentare caratteri che non possono essere presenti nel payload. In questa tecnica i dati non vengono scritti tutti in insieme ma a chunk di alcuni bytes per volta (a seconda dell'architettura o del tipo di gadget). A seconda del tipo di gadget per ognuna di queste iterazioni è necessario inserire nel payload un certo numero di bytes di "padding". Per questi motivi l'utilizzo di questa tecnica tende a generare payload piuttosto lunghi

- Utilizzo delle funzioni `strcpy` o `memcpy` per copiare byte presenti in posizioni note all'interno dell'eseguibile nella posizione desiderata [13]. Questa tecnica, copiando verosimilmente solo un byte per volta tende anch'essa a generare payload abbastanza lunghi ma ha il grande vantaggio di poter copiare anche valori che non possono essere presenti nel payload. È necessario che i valori dei byte che voglio copiare siano già presenti nell'eseguibile, condizione che, a meno di file oggetto eseguibili molto piccoli, risulta abbastanza comune

- Nel caso si riesca ad ottenere in qualche modo (ad esempio attraverso una seconda vulnerabilità) la conoscenza dell'indirizzo dello stack i valori possono essere inseriti direttamente insieme al payload

Una volta scritti gli argomenti per la funzione da richiamare in memoria è necessario chiamare la funzione stessa. La funzione deve permetterci di eseguire un comando arbitrario. All'interno della `libc` sono presenti due funzioni con queste caratteristiche: `execve` e `system`. Un ritorno diretto alla funzione solitamente non può avvenire, in quanto il valore del suo indirizzo in memoria è casualizzato. Tuttavia, se una di queste due funzioni è utilizzata all'interno del programma, basterà ritornare al suo elemento nella Program Linkage Table, che ha una posizione nota e fissa all'interno della memoria dell'immagine del processo. Se così non fosse un'alternativa è quella di adoperare tecniche per ricavare l'indirizzo della funzione partendo dall'indirizzo di un'altra funzione della stessa libreria che invece viene utilizzata dal programma. Alcune considerazioni su queste tecniche, descritte nella sezione 1.2.3, che riguardano l'architettura di Dropper sono:

- Devono essere noti gli offset tra le funzioni all'interno del file oggetto condiviso della libreria utilizzato sulla macchina bersaglio. L'utente può fornire a Dropper un file di questo tipo dal quale verranno ricavati i relativi offset. È da notare che, variando in un range relativamente piccolo, non dovrebbe essere impossibile, se le condizioni lo consentono, montare un attacco di tipo forza bruta per ricavare questi offset

- Come visto in 1.2.3 per poter utilizzare la tecnica di GOT patching il binario deve presentare una sequenza di gadget che ci consenta di eseguire un'operazione di somma (o un'operazione che ci consenta di simulare una somma, come una sottrazione) tra un offset di tre/quattro byte e un'area di memoria arbitraria (l'area dove è memorizzato l'indirizzo della funzione usata come riferimento, solitamente nella GOT). Molto spesso però abbiamo gadget che ci consentono di addizionare solo un byte per volta. Questo è un problema perché in un'operazione di somma byte per byte dev'essere considerato un eventuale valore di riporto tra un byte e il successivo. Un modo per risolvere questo problema è utilizzare gadget che utilizzino un'istruzione di tipo `adc` (add with carry), facendo attenzione che gli effetti secondari degli altri gadget coinvolti nella catena non vadano a modificare il carry flag. Quest'istruzione viene utilizzata proprio per le addizioni multi-byte e aggiunge

anche il valore del carry flag quando esegue l'operazione di somma. In realtà anche non avendo a disposizione questo tipo di gadget, per via dell'allineamento in memoria, gli ultimi bit dell'indirizzo di una funzione non variano tra un'esecuzione e la successiva. Con questa parziale conoscenza del valore iniziale si può comunque creare una catena con un alta percentuale di successo come descritto nella sezione 3.2.3

È interessante notare come, anche se vi è una guida nelle operazioni da seguire basata sulla strategia scelta, la sequenza di operazioni che conducono allo stato del sistema desiderato sono molteplici. Ad esempio l'operazione aritmetica per il calcolo dell'indirizzo della funzione bersaglio potrebbe avvenire in memoria, oppure tra due registri e poi salvata in memoria. Qui si notano i vantaggi che potrebbe avere un approccio che modelli la situazione in un problema di planning rispetto ad un approccio di tipo programmatico. Infatti, con un approccio programmatico, tutte le combinazioni vanno codificate nell'algoritmo, una procedura la cui completezza è più difficile da dimostrare e che è molto più soggetta ad errori.

2.1.1 Utilizzo di pianificatori

Il problema di generare una catena (quindi una sequenza) di gadget (che possono essere visti come azioni) per arrivare ad uno stato del sistema desiderato si traduce naturalmente in un problema di planning. Una volta tradotto si può utilizzare un planner per ottenere una soluzione valida e/o (dipende dal planner che si utilizza) ottima.

Nel 1998 Drew McDermott rilascia il Planning Domain Description Language (PDDL) [2, 10], un linguaggio che ha come obiettivo quello di fornire uno strumento universale per codificare problemi di planning. In questa sezione esamineremo un primo approccio che può essere utilizzato per tradurre alcuni dei problemi affrontati da Dropper nel linguaggio PDDL, analizzandone le principali problematiche. Il linguaggio PDDL è fatto in modo da separare la descrizione del dominio del problema da quella della singola istanza.

Gli elementi che possono essere descritti all'interno della codifica del dominio del problema rilevanti al fine della nostra discussione sono:

- I diversi tipi di oggetto presenti nel nostro dominio
- *Costant*, cioè oggetti che sono presenti in tutte le istanze del problema

- *Fluent*, mapping tra una tupla di oggetti e un valore numerico che può cambiare durante un piano. La loro sintassi e semantica è stata stabilizzata nel linguaggio in [10]

- *azioni*. Ogni azione può avere dei parametri e definisce l'insieme delle possibili modifiche che possono essere effettuate nel dominio modellato. Per ogni azione si indicano le precondition (ovver condizioni necessarie per effettuare un'azione) e gli effetti che le azioni hanno sugli oggetti del dominio o su i fluent

Una singola istanza del problema invece definisce gli oggetti che le azioni possono manipolare, lo stato iniziale nel quale si trova il sistema, lo stato che vogliamo raggiungere (il nostro "goal") e la metrica con la quale valutare la bontà di un piano. Come evidenziato in [10] i fluent non possono essere utilizzati come parametri di azioni perché queste agiscono su oggetti "fisici" e quindi i parametri devono appartenere ad insiemi finiti (e non ad insiemi infiniti come i numeri).

Per ovviare a questa limitazione nella nostra modellazione utilizzeremo come stratagemma quello di creare degli oggetti di tipo `imm` e di associargli un valore tramite il fluent `val (?x - imm)`. Questo fa si che le azioni possono utilizzare parametri di tipo `imm` e che negli effetti possiamo ricavare il valore che l'oggetto di tipo `imm` rappresenta. Questo ci obbliga, nell'istanza del problema, a definire tutti i valori numerici che saranno presenti nel piano come oggetti di tipo `imm` e ad associare per ognuno di essi il relativo fluent.

I tipi di dato utilizzati nella modellazione sono `imm`, con il significato appena spiegato, e `register` che rappresenta per l'appunto un registro. I fluent utilizzati invece sono:

- `(val ?x - imm)` come appena definito

- `(reg ?r - register)` che rappresenta il valore attuale del registro ?r

- `(mem ?a - imm)` che rappresenta il valore attuale della locazione di memoria `?a`

Vengono create tante costanti quante i registri. Ogni azione invece corrisponde ad un gadget e le azioni riflettono gli effetti che il gadget ha su i registri e sulla memoria. Attualmente si è proceduto a modellare solo gadget per l'impostazione di registri o la scrittura di memoria. È da notare che per

lo stratagemma usato tutti i valori (indirizzi e valori concreti) che le nostre azioni vanno ad utilizzare devono essere esplicitati. Un gadget per l'impostazione di un registro viene tradotto in questo modo (a meno dei puntini utilizzati per generalizzare la lista dei registri):

```
1   (:action regset_gadget
2    :parameters (?vreg0 ?vreg1 ... ?vregn - imm)
3    :precondition (and)
4    :effect (and
5      (assign (reg reg0) (val ?vreg0))
6      (assign (reg reg1) (val ?vreg1))
7                  ...
8      (assign (reg regn) (val ?vregn))
9      (increase (stack-length) g.stack_offset)))
```

Come si può notare la traduzione avviene ad un livello d'astrazione più alto delle singole istruzioni del gadget, dettagli come la posizione dei valori da impostare nello stack, gli effetti secondari, vengono gestiti sempre da Dropper. La traduzione invece di un gadget per la scrittura in memoria è leggermente più complicata:

```
1 (:action memstr_gadget
2   :parameters ()
3   :precondition (and)
4   :effect (and
5            (forall (?a -imm)
6              (when (= (val ?a) (reg rdi))
7               (assign (mem ?a) (reg rsi))))
8           (increase (stack-length) 0)))
```

Questa modellazione poco elegante è dovuta al non poter utilizzare i fluent come argomenti per altri fluent.

Una volta tradotto il dominio, si deve anche tradurre un'istanza del problema. Si specificano gli oggetti che faranno parte dell'istanza del problema, cioè i valori che andremo ad utilizzare. Come stato iniziale si impostano i fluent dei registri a zero, si assegnano i rispettivi valori ai fluent di tipo `(val ?v - imm)` e si imposta anche il fluent `stack-length` a zero, quest'ultimo viene utilizzato come metrica per valutare la bontà del piano trovato. Come stato desiderato si imposta o, nel caso si voglia trovare un piano per impostare un insieme di registri, che i fluent rilevanti di tipo `(reg ?r)` abbiano i valori desiderati o, nel caso invece si voglia trovare un piano per impostare alcune specifiche locazioni di memoria, che i fluent del tipo `(mem ?a)` abbiano il valore corrispondente.

Un esempio, con solo un piccolo sottoinsieme dei gadge trovati nell'eseguibile ls è riportato nelle figure 2.1 e 2.2. L'esempio è generato tramite la classe `planner`.

La codifica appena presentata è molto banale e presenta alcuni problemi. Prima di tutto non considera i valori della memoria e dei registri come vettori di bit. Una soluzione potrebbe essere quella di modellare la granularità delle azioni (e dei fluent) al singolo byte. Anche se questo non consentirebbe di modellare azioni che effettuino operazioni di tipo bit a bit aumenterebbe di molto l'espressività della modellazione proposta.

Il secondo problema è che una quantità di azioni grande quanto il numero di gadget presente nell'eseguibile rende l'utilizzo di planner general-purpose inefficiente. Nei primi test effettuati la maggior parte dei planner non riesce neanche a iniziare la fase di ricerca del piano. Una soluzione potrebbe essere quella di filtrare i gadget in una prima fase, in modo da fornire una quantità più limitata di azioni nella descrizione del problema.

```
1  ;gadget:
2  ;0x412050 : pop rsi; pop r15; ret
3  ;0x412052 : pop rdi; ret
4  ;0x40ea2c : mov dword ptr [rdi], esi; ret
5
6  (define (domain dropper )
7  (:requirements :typing :fluents :conditional-effects)
8  (:types imm register)
9  (:constants rax rbx rcx rdx rdi rsi rbp rsp rip r8 r9 r10 r11
       r12 r13 r14 r15 - register)
10
11 (:functions
12   (val ?x -imm)
13   (reg ?r - register)
14   (stack-length)
15   (mem ?addres - imm))
16
17 (:action regset_412052
18    :parameters (?vrdi - imm)
19    :precondition (and)
20    :effect (and          (assign (reg rdi) (val ?vrdi))
21      (increase (stack-length) 8)))
22
23 (:action regset_412050
24    :parameters (?vr15 ?vrsi - imm)
25    :precondition (and)
```

```
26    :effect (and          (assign (reg r15) (val ?vr15))
27          (assign (reg rsi) (val ?vrsi))
28      (increase (stack-length) 16)))
29
30 (:action memstr_40ea2c
31   :parameters ()
32   :precondition (and)
33   :effect (and (forall (?a ?v -imm)
34       (when (and (= (val ?a) (reg rdi))
35       (= (val ?v) (reg rsi))) (assign (mem ?a) (val ?v))))
36      (increase (stack-length) 0)))
37 )
```

Listing 2.1: Descrizione del dominio relativo ad un piccolo sottoinsieme dei gadget presenti in ls

```
1  (define (problem memset)
2  (:domain dropper)
3  (:objects v_32896 v_2 v_33153 v_3 - imm)
4  (:init
5  (= (val v_32896) 32896)
6  (= (val v_33153) 33153)
7  (= (val v_2) 2)
8  (= (val v_3) 3)
9  (= (reg rax) 0)(= (reg rbx) 0)(= (reg rcx) 0)(= (reg rdx) 0)
      (= (reg rdi) 0)
10 (= (reg rsi) 0)(= (reg rbp) 0)(= (reg rsp) 0)(= (reg rip) 0)
      (= (reg r8) 0)
11 (= (reg r9) 0)(= (reg r10) 0)(= (reg r11) 0)(= (reg r12) 0)(=
       (reg r13) 0)
12 (= (reg r14) 0)(= (reg r15) 0)
13 (= (mem v_32896) 0)
14 (= (mem v_33153) 0)
15 (= (stack-length) 0))
16 (:goal
17 (and (= (mem v_32896) 2)
18   (= (mem v_33153) 3)))
19 (:metric minimize (stack-length))
20 )
```

Listing 2.2: Descrizione del dominio relativo ad un'istanza del problema

2.2 Modulo Gadgets

Il modulo Gadgets si occupa di ricavare i gadget presenti nel file oggetto eseguibile, di estrapolarne la semantica e della creazione di sequenze di gadget che se eseguite assolvono funzioni particolari (come scrivere in memoria, settare un registro, ecc.). Questo modulo interagisce con il modulo Payload per trasformare sequenze ordinate di gadget in un payload vero e proprio da poter iniettare sullo stack ed espone i metodi per la creazione delle catene che vengono usati dal modulo Core.

Per la ricerca dei gadget all'interno del file oggetto eseguibile viene utilizzata la classe `GadgetFinder` del framework BARF, che implementa al suo interno l'algoritmo galileo[18].

Una volta disponibile l'insieme dei gadget presenti nel file oggetto eseguibile bisogna, per ognuno di essi, estrapolarne la semantica. Per estrapolarne la semantica si intende estrapolare in maniera automatica gli effetti che una serie di istruzioni hanno sul contesto il quale, semplificando, è rappresentato dalla memoria e dai registri della macchina sulla quale saranno eseguite le istruzioni. Per questo tipo di analisi si utilizzano due strumenti principali: la simulazione e la traduzione delle istruzioni in un istanza di un problema SMT.

2.2.1 Traduzione di un gadget in formule logiche

Un'istanza di un problema SMT può essere considerata come una formula logica del primo ordine dove ad alcune funzioni e ad alcuni predicati vengono assegnati significati e interpretazioni particolari esplicitati in teorie aggiuntive [3]. Questo fa si che possano essere sviluppati metodi decisionali costruiti su misura per teorie specifiche, rendendo efficiente e possibile la codifica di problemi di alcuni domini particolari. Ad esempio, considerando una formula del tipo:

$$x < y \wedge \neg(x < y + 0)$$

spesso ci interessa conoscere se la formula sia soddisfacibile solo sotto i comuni significati delle funzione $+$, dei predicato $<$ e della costante 0, ovvero ci interessa conoscere la soddisfacibilità della formula rispetto alla teoria aritmetica degli interi. Anche se è possibile in ogni caso incorporare una serie di formule insieme a quelle che descrivono un problmea in modo da for-

zare un risolutore generale a considerare solo le interpretazioni compatibili con una data teoria, le prestazioni computazionali ottenute in questo modo non risultano spesso accettabili. Sono state sviluppate diverse procedure su misura per molte teorie tra cui quella degli array e dei bitvector, particolarmente rilevanti ai fini del nostro lavoro. Una volta che sono a disposizione queste teorie risulta naturale modellare registri e memoria rispettivamente attraverso Bitvector e Array di Bitvector.

Questi sono le motivazioni principali della scelta di una modellazione che producesse un'istanza di un problema di SMT rispetto ad altri metodi di modellazione logici come *Answer Set Programming*.

Nelle figure 2.3, 2.4 e 2.5 sono mostrate rispettivamente le istruzioni di un gadget per la scrittura in memoria e le relative traduzioni in istruzioni REIL e in formule SMT.

```
1 mov dword ptr [rdi], esi
2 ret
```

Listing 2.3: gadget per la scrittura in memoria

```
1 str   [DWORD esi, EMPTY, DWORD t3]
2 stm   [DWORD t3, EMPTY, QWORD rdi]
3
4 add   [QWORD rsp, QWORD 0x8, QWORD t2]
5 str   [QWORD t2, EMPTY, QWORD rsp]
6 ret   [EMPTY, EMPTY, EMPTY]
```

Listing 2.4: traduzione del gadget in fig. 2.3 nel linguaggio intermedio REIL

```
1
2 (= ((_ extract 31 0) rsi_0) t3_1)
3 (= MEM_1
4   (store
5   (store
6   (store
7   (store MEM_0 (bvadd rdi_0 #x0000000000000000) ((_ extract 7
        0) t3_1))
8   (bvadd rdi_0 #x0000000000000001) ((_ extract 15 8) t3_1))
9   (bvadd rdi_0 #x0000000000000002) ((_ extract 23 16) t3_1))
10  (bvadd rdi_0 #x0000000000000003) ((_ extract 31 24) t3_1)))
11 (= t2_1 (bvadd rsp_0 #x0000000000000008))
12 (= t2_1 rsp_1)
```

Listing 2.5: traduzione del gadget in fig. 2.3 in formule SMT

Un'analisi completa della semantica presenta alcuni limiti dovuti alla difficoltà di una modellazione completa del sistema nonché alla conoscenza parziale del contesto (memoria e registri) reale in cui si troveranno ad essere eseguite le istruzioni. Ad esempio utilizzando l'attuale traduzione delle istruzioni in formule SMT, è difficile porre al risolutore domande che riguardino il numero di letture effettuate, o domandare se un'area di memoria viene mai letta. Per quanto detto l'estrapolazione della semantica di un gadget avviene in tre fasi descritte nelle prossime sezioni: a) emulazione e catalogazione, b) verifica e validazione e c) verifica degli effetti secondari.

2.2.2 Emulazione e catalogazione

La classe `GadgetClassifier` del framework BARF emula il gadget tenendo traccia delle operazioni sui registri e sulla memoria. Con questi dati si verifica se il gadget appartiene ad una o più categorie indicate in [17]. Inoltre vengono identificati quali registri vengono utilizzati come "operandi" della funzione alla quale il gadget viene associato. Basandosi su una simulazione, e quindi sull'analisi degli effetti di solo un sottoinsieme dei possibili valori in ingresso, questa catalogazione deve essere poi verificata con un'analisi semantica vera e propria.

Le simulazioni vengono effettuate utilizzando dati concreti ma un'esecuzione simbolica, funzionalità ancora non disponibile all'interno del framework BARF, consentirebbe una catalogazione più accurata esplicitando le relazioni tra i valori che i registri hanno all'inizio e alla fine della simulazione o tra i valori iniziali dei registri e gli accessi alla memoria.

2.2.3 Verifica e validazione

Si deve poi verificare che, anche se il gadget esegua la funzione della categoria alla quale viene assegnato, possa essere utilizzato per i valori di tutto il "dominio" di questa funzione, cioè che non ci siano restrizioni sugli operandi del gadget. Per esempio, un gadget catalogato come `MemoryStore` ma che utilizzi lo stesso registro come operando di destinazione e operando sorgente non può essere utilizzato per scrivere in una locazione arbitraria di memoria un valore arbitrario in quanto il valore da scrivere e l'indirizzo di memoria sono dipendenti tra di loro. In questa sezione ci riferiamo a questa proprietà come alla "generalità" del gadget.

Bisogna verificare altresì che l'assegnamento dei valori appropriati ai registri "operandi" sia condizione sufficiente perché il gadget si comporti aderendo alla semantica della categoria in cui è stato catalogato. Sempre continuando con l'esempio della scrittura in memoria, questo vuol dire essere sicuri che, se i registri utilizzati come operandi sono impostati con i valori appropriati, la scrittura in memoria del valore voluto sia garantita per qualsiasi altra combinazione iniziale dei restanti registri.

Queste due proprietà vengono verificate sia attraverso delle simulazioni delle istruzioni nel linguaggio intermedio e sia attraverso la sottomissione di istanze SMT ad un risolutore. Vengono codificate per questi scopi due tipi di istanza SMT: una la quale soddisfacibilità ci garantisce che un dato effetto può essere realizzato (e viene anche fornito una configurazione del contesto che porta a quell'effetto), e un'altra la quale non soddisfacibilità ci garantisce che l'effetto del gadget aderisca alla semantica di catalogazione per qualsiasi contesto iniziale. Istanze del primo tipo possono essere utilizzate sia per ottenere la configurazione iniziale con la quale ottenere un particolare effetto sul contesto, sia per una verifica "ottimista" della generalità di un gadget (in questo caso si utilizzano dei valori casuali e si assume che, se l'istanza è soddisfacibile per alcuni valori casuali allora il gadget è generico).

Le istanze del primo tipo vengono create imponendo al modello che il contesto finale rifletti l'effetto desiderato, quelle del secondo tipo invece imponendo il complementare della condizione che rappresenta la semantica che si sta cercando di verificare. Ad esempio per un gadget utilizzato per la scrittura in memoria, nel quale sono stati individuati `rdi` come registro di destinazione e `rsi` come registro sorgente, viene imposto che la locazione di memoria individuata da `rdi` **non** abbia come valore finale il valore di `rsi`. La non soddisfacibilità di questa formula ci garantisce la validità che il gadget scriva sempre il valore presente nel registro sorgente nella locazione di memoria individuata dall'indirizzo presente nel registro di destinazione.

2.2.4 Verifica degli effetti secondari

Il punto precedente ci assicura che un gadget non abbia effetti secondari che interferiscano con gli effetti della nostra funzione. Tuttavia è possibile che il gadget in esame abbia altri effetti secondari che non lo rendano utilizzabile. Ad esempio se il gadget tenta di leggere da una locazione di memoria marcata come non leggibile causerà l'arresto del programma. La modellazione in

formule logiche utilizzata attualmente nel framework BARF non consente di porre al risolutore domande sul numero di letture in memoria o sulla lettura o meno di un'area di memoria. Per questo motivo per escludere effetti secondari di questo tipo il gadget viene emulato con i valori di ingresso ricavati nella fase di verifica e validazione, e ci si assicura che non vengano lette e scritte aree di memoria che non siano quelle relative alla funzione propria del gadget o all'area dello stack.

La linea guida generale appena descritta va adattata alla tipologia di gadget analizzata, per le implementazioni specifiche fare riferimento al capitolo 3. Inoltre per poter utilizzare un gadget, i registri utilizzati come "operandi" devono essere "controllabili", cioè deve esistere una serie di gadget che consenta di poter caricare valori arbitrari in questi registri. Per questo prima di procedere alla verifica dei gadget si cercano all'interno dell'eseguibile tutte le sequenze di gadget che ci permettono di controllare registri. Se un gadget utilizza come operandi registri che non sono "controllabili" questo viene scartato.

Durante queste le fasi di classificazione e verifica vengono anche raccolte informazioni aggiuntive sui gadget, come di come un gadget modifica il valore del registro dello stack e la posizione dello stack dove inserire i valori utilizzati dal gadget. Queste informazioni vengono utilizzate dal modulo `Payload` per costruire il payload vero e proprio.

Per le simulazioni viene utilizzata la classe `ReilEmulator` e la traduzione in formule SMT avviene attraverso la classe `CodeAnalyzer`, entrambe del framework BARF.

2.3 Modulo Payload

Questo modulo si occupa di generare il payload vero e proprio partendo dai gadget analizzati dal modulo Gadget. Questa classe è strutturata in modo che i diversi chunk prodotti siano concatenabili, e che la loro concatenazione dia a sua una volta un frammento concatenabile. Questo ci consente di associare ad una data funzione un payload e utilizzarlo in diverse parti del programma senza dover più conoscerne i dettagli. Ci sono diverse cose da tenere in conto quando si costruisce il payload partendo dai gadget, quasi tutte legate alla manipolazione dello stack da parte del gadget stesso:

- posizioni dei valori utilizzati dal gadget in relazione alla valore puntato dallo stack al momento della sua esecuzione
- valore al quale punterà lo stack al momento dell'esecuzione dell'istruzione ret del gadget. In base alla tipologia di gadget questo valore potrebbe non essere alla fine del payload associato al gadget stesso
- valore finale dello stack dopo l'esecuzione del gadget

Le informazioni relative alle tre problematiche descritte sopra vengono estrapolate dal modulo Gadgets. La procedura di generazione del payload è specifica del tipo di gadget, i metodi `PayloadChunk.chain` e `PayloadChunk.get_general_chunk` sono invece generici, prendono come argomento una lista di chunk e restituiscono rispettivamente il payload dei chunk concatenati o un chunk che può essere nuovamente concatenato e rappresenta la concatenazione dei chunk in ingresso.

Capitolo 3

Implementazione di Dropper

In questo capitolo si farà un'analisi dell'attuale stato di sviluppo di Dropper.

3.1 Modulo Core

3.1.1 Interazione con l'utente

Attualmente Dropper si interfaccia con l'utente tramite alcune funzioni della classe `dropper`. Il costruttore di questa classe riceve come argomento il percorso sul filesystem dove risiede il file oggetto eseguibile da analizzare. Le operazioni di analisi vengono avviate tramite la funzione `analyze_all` . Una volta che l'analisi è stata effettuata si può richiamare la funzione `build_spawn_shell_payload` che si occuperà di generare la catena vera e propria. Anche se queste sono le uniche chiamate essenziali per utilizzare Dropper l'utente ha a disposizione ulteriori funzioni:

- `set_can_control_fd` setta il file descriptor controllato dall'utente (-1 se nessun file descriptor può essere controllato)
- `add_shared_object` permette di analizzare un oggetto che rappresenti una libreria dinamica. Dropper ricava da questo file gli offset relativi tra gli indirizzi delle funzioni della libreria
- `set_function_for_address_resolving` indica la funzione di riferimento da utilizzare nell'applicazione di tecniche che ricavino l'indirizzo di funzioni casualizzate in memoria partendo da valori nella GOT table (vedi sez. 1.2.3). La funzione indicata dev'essere già stata richiamata almeno una volta al momento dell'esecuzione della catena

- `set_writeable_area` imposta l'indirizzo dell'area di memoria dove effettuare operazioni di scrittura. Di default questo valore è impostato all'indirizzo di caricamento in memoria della sezione `.data`
- `set_cmd` setta il comando da eseguire sulla macchina bersaglio una volta che l'exploit ha avuto successo

Quando il file eseguibile è stato analizzato e le informazioni necessarie impostate il metodo `build_spawn_shell_payload` cerca di generare una catena che avvia il comando impostato.

3.1.2 Analisi degli import

Se una funzione esterna viene utilizzata dal programma, anche se il suo indirizzo reale viene casualizzato, sarà presente un elemento relativo a quella funzione all'interno della Procedure Linkage Table. Questo elemento, ammesso che il binario non sia stato compilato come indipendente dalla posizione in memoria (PIE), ha una posizione nota. Possiamo quindi ritornare al suo indirizzo per emulare la chiamata a quella funzione (return-to-plt). Nella GOT invece ogni elemento inizialmente, e quindi anche staticamente nel file, contiene l'indirizzo dell'istruzione successiva nell'elemento della PLT relativo alla stessa funzione. Per ogni funzione la rilocazione associata, nella sezione `.rel.plt`, contiene nel campo `r_offset` l'indirizzo dell'elemento nella GOT table. A partire da quest'ultimo, per le considerazioni appena fatte, può essere ricavato l'indirizzo dell'elemento nella PLT. Gli indirizzi del relativo elemento della GOT e di quello nella PLT vengono ricavati per ogni funzione utilizzata dal file oggetto eseguibile nel metodo `get_imports`.

3.1.3 Costruzione della catena

Attualmente il modulo Core per prima cosa cerca di scrivere in un'area di memoria scrivibile i dati ai quali punteranno gli argomenti dell'invocazione della funzione `execve`. Nel caso quest'area di memoria non viene specificata dall'utente viene utilizzato l'indirizzo di caricamento in memoria della sezione `.data`. Gli argomenti e i dati veri e propri vengono creati dal modulo partendo dal comando che si è deciso di eseguire sulla macchina (nella funzione `payload_execve_args`). Se si può controllare un file descriptor Core utilizza la funzione `read` per scrivere in memoria gli argomenti di `execve`, come descritto in 2.1. Per fare questo viene utilizzato il metodo `get_ret_func_chunk` del

modulo Gadget il quale ci consente di creare una catena che richiami una funzione di cui si conosce l'indirizzo. È compito poi dell'utente fare in modo che al momento della chiamata di questa funzione il file descriptor indicato si trovi nello stato giusto in modo da restituire i valori calcolati dal modulo. I valori da inserire nel flusso del file descriptor vengono ricavati dall'utente attraverso il parametro `fd_payload` di dropper.

Attualmente è implementata una seconda strategia nel caso non sia possibile richiamare la funzione `read` e/o non si controlli nessun file descriptor. Essa consiste nell'utilizzare una serie di gadget che possano spostare valori dallo stack in una locazione di memoria arbitraria. Per costruire una catena del genere si chiama la funzione `build_memory_write` del modulo Gadgets che prende come argomenti l'indirizzo della memoria dove scrivere i dati. Solitamente, come vedremo in 3.2.2, le catene generate in questo modo tendono ad essere piuttosto lunghe.

Per quanto riguarda la seconda fase, la chiamata della funzione, sono state implementate due strategie. La prima nel caso `execve` sia presente all'interno delle funzioni utilizzate dal programma. In questo caso la generazione della parte restante della catena è banale e, come visto nel paragrafo precedente, viene costruita dalla funzione `get_ret_func_chunk` del modulo Gadgets. Più interessante è il caso in cui il file oggetto eseguibile non utilizzi questa funzione.

Infatti in questo Dropper cerca di far costruire al modulo Dropper una catena in grado di modificare il valore della Global Offset Table in loco, in modo da effettuare una tecnica di tipo *got patching* (vedi 1.2.3). Il modulo Gadgets espone il metodo `build_mem_add` che si occupa dei dettagli implementativi di questo tipo di catena. Viene chiamato dal modulo Core con i seguenti parametri:

- l'indirizzo di memoria dove eseguire la somma: l'indirizzo dell'elemento della GOT da "patchare". La funzione viene indicata dall'utente tramite la funzione `set_function_for_address_resolving` e l'indirizzo viene ricavato durante la fase dell'analisi degli import
- l'addendo: l'offset relativo tra la funzione utilizzata come base e quella voluta, questo valore viene eventualmente ricavato dai file oggetto condivisi forniti a Dropper tramite il metodo `add_shared_object`
- il valore di base: l'offset che la funzione di base ha all'interno del file oggetto condiviso della libreria, anch'esso ottenuto analizzando un file

indicato con la funzione `add_shared_object`. Come vedremo questo rende possibile utilizzare una tecnica di tipo got patching anche quando le tipologie di gadget disponibili non sono ideali al prezzo di sacrificare un'affidabilità del 100% (vedi sez. 3.2.3)

La classe inoltre contiene alcuni metodi per il salvataggio dei dati relativi ai gadget su disco, questo perché l'analisi di eseguibili molto grandi può durare diversi minuti. Al loro interno questi metodi utilizzano la libreria `pickle` per la serializzazione degli oggetti.

3.2 Modulo Gadgets

Questo modulo, come descritto nel capitolo dedicato all'architettura, estrapola e cataloga i gadget presenti nel binario in esame ed espone i metodi di creazione delle catene per l'esecuzione di diverse funzioni. Le catene vengono costruite arrangiando i gadget disponibili e collaborando con il modulo Payload che si occupa della generazione del payload vero e proprio. Attualmente il modulo mette a disposizione i seguenti metodi per la generazione delle catene:

- `get_stack_slide_chunk`: consente di aggiungere un valore arbitrario al registro dello stack, questo tipo di catene ci consente ad esempio di "saltare" gli argomenti di una funzione sullo stack e passare al gadget successivo
- `get_ret_func_chunk`: restituisce una catena che richiami una funzione. Riceve come parametri di ingresso gli argomenti da passare alla funzione e l'indirizzo della funzione stessa
- `get_memory_add_chunk`: restituisce una catena che esegue un operazione di addizione in una locazione di memoria arbitraria
- `get_memory_store_chunk`: restituisce una catena che scrive in memoria valori arbitrari
- `get_ccf_chunk`: restituisce un chunk che azzeri il carry flag

Le tipologie di gadget utilizzate per la costruzione di queste catene sono descritte nella parte restante di questo capitolo.

Fondamentale per poter utilizzare un gadget è poter controllare i registri coinvolti nelle sue operazioni, per questo motivo la prima analisi che viene condotta una volta ottenuta la lista dei gadget è la selezione di quelli che ci consentono di controllare i registri. La ricerca dei gadget viene effettuata in tutte le sezioni che vengono caricate in memoria e marcate come eseguibili. Si è deciso di separare il modulo in diverse sottoclassi, ognuna delle quali contiene la logica per la selezione e l'utilizzo di un particolare tipo di gadget.

3.2.1 Gadget per l'impostazione dei registri

La logica e le strutture dati utilizzate per la selezione e la gestione dei gadget utilizzati per l'impostazione dei registri è contenuta nella classe `RegSet`. Il metodo `_analyze_gadget` riceve come argomento in ingresso un gadget e, nel caso esso possa essere utilizzato per l'impostazione di registri, viene aggiunto nelle strutture dati interne alla classe. La classe espone il metodo `get_chunk` che ritorna una catena che imposta i registri in modo da riflettere le coppie registro/valore del dizionario preso come argomento. Attualmente solo gadget che contengono istruzioni di tipo `pop reg` vengono considerati come potenziali candidati per l'impostazione di registri.

L'analisi avviene tramite una simulazione istruzione per istruzione. Se l'istruzione attuale è di tipo `pop reg` allora si memorizza il nome del registro e l'offset tra il valore attuale dello stack e quello all'inizio della simulazione. Questo ci consente di conoscere dove posizionare il valore da impostare nel registro al momento della generazione del payload. Durante la simulazione si controlla anche che le istruzioni che non sono del tipo `pop reg` non eseguano operazioni di lettura o scrittura sulla memoria. Questo ci consente di scartare già in questa fase i gadget che presentano effetti secondari. Dopo la fase di analisi quindi avremo, per ogni gadget, la lista dei registri che possono essere impostati insieme all'ordine e alla posizione con cui vengono prelevati dallo stack.

La funzione `get_chunk` invece restituisce una catena che consente di impostare l'insieme dei registri che riceve in ingresso. Dato che un gadget potrebbe impostare più di un registro, la scelta dei gadget che andranno a comporre questa catena è riducibile ad un problema di *set covering*. È preferibile che il payload generato dalla catena sia il più piccolo possibile. La lunghezza della catena è influenzata sia dal numero di gadget utilizzati, sia dal padding che è necessario inserire nell'utilizzo di ogni gadget per via delle operazioni

che esso effettua sul registro dello stack. Attualmente per trovare la catena viene utilizzato un algoritmo greedy che scorre i registri da "coprire" e, se il registro non è già coperto da un gadget inserito nella catena, sceglie il gadget che lo copre e che necessita del minor padding possibile. Se l'aggiunta di un gadget ne rende superfluo uno già presente nei candidati quest'ultimo viene rimosso. La soluzione trovata con quest'algoritmo non è ottima. Nella sezione 2.1.1 è descritto un primo approccio per l'utilizzo di planner nella ricerca di una sequenza ottima.

3.2.2 Gadget per la scrittura in memoria

La logica e le strutture dati per gestire i gadget che ci consentono di scrivere in memoria si trovano nella classe `MemoryStore`.

Durante la prima classificazione la classe `GadgetClassifier` del framework BARF verifica se a seguito di alcune simulazioni un valore presente in un registro (denominato sorgente) viene scritto in una locazione di memoria il cui indirizzo corrisponde al valore di un registro (denominato destinazione) sommato ad un eventuale offset.

Ogni volta che questa condizione viene verificata il gadget viene aggiunto alla categoria StoreMemory. È da notare che la condizione di cui sopra potrebbe verificarsi per più coppie di registri nello stesso gadget, per ognuna di queste viene creato un oggetto gadget di tipo StoreMemory e aggiunto alla categoria. La classe parte da questi gadget per procedere alla fase di verifica e validazione.

Innanzitutto se il registro di destinazione è il registro dello stack il gadget viene scartato. Questo fa in modo che non vengano presi in considerazione i gadget con istruzioni tipo `push reg`. Questi gadget infatti, solitamente non consentono di scrivere in una regione arbitraria della memoria, oltre a complicare la gestione dello schema dello stack. Dopo di che vengono scartati anche i gadget che hanno come registro di destinazione `rip` o `eip`. Infatti essendo questo registro fisso rispetto all'istruzione che lo esegue neanche questi gadget ci consentono di scrivere in una posizione arbitraria di memoria.

Dopo di che, utilizzando le metodologie descritte in 2.2.3, il gadget viene scartato se non si possono controllare i registri sorgente e destinazione, se ne verifica e valida la generalità e ci si assicura che il gadget non presenti effetti secondari. Nella verifica degli effetti secondari si controlla che non ci siano scritture al di fuori della locazione scelta e dello stack.

Il metodo `get_chunk` invece prende come argomenti un indirizzo in memoria e una stringa e restituisce una catena che scrive la stringa in memoria partendo da quell'indirizzo. Il metodo quindi espone un livello d'astrazione più alto al modulo che la utilizza e, normalmente, vi è bisogno di più iterazioni dell'esecuzione del gadget per ottenere la scrittura desiderata. Il metodo, infatti, oltre a scegliere uno dei gadget disponibili calcola il numero di queste iterazioni necessarie per la scrittura dei valori. Per ogni iterazione poi utilizza una modellazione logica unita ad un risolutore SMT per ottenere i valori dei registri sorgente e destinazione corretti. Anche se si potrebbero utilizzare il valore di offset ottenuto durante la fase di catalogazione del gadget utilizzare il risolutore ci permette di utilizzare anche gadget i cui effetti secondari modifichino la locazione di memoria in maniera indipendente dagli altri registri. Sempre per ogni iterazione, si utilizza la classe `regset` per ottenere una catena che setti i registri a quei valori, e la si concatena a sua volta al gadget che si sta utilizzando per scrivere in memoria. Viene restituita la concatenazione delle catene create per ogni iterazione, che verosimilmente scriverà in memoria, partendo dall'indirizzo indicato i valori in ingresso.

3.2.3 Gadget per la operazioni aritmetiche in memoria

La logica e le strutture dati per la gestione dei gadget che ci consentono di scrivere in memoria risiedono nella classe `aritmeticmemorystore`.

Anche in questo caso si parte dai gadget classificati dal framework BARF, ma appartenenti, per l'appunto, alla categoria Arithmetic Memory Store. Un gadget viene inserito in questa categoria se, dopo alcune simulazioni del gadget, una locazione di memoria viene modificata e come valore finale riflette il risultato di un'operazione aritmetica binaria tra il valore precedente di quella locazione e il valore iniziale contenuto in un registro. Inoltre si individua un registro che potrebbe essere usato dal gadget come indirizzo della locazione di memoria (insieme ad un eventuale offset). Dato che la simulazione viene ripetuta più volte i falsi positivi vengono ridotti. È possibile che un gadget esibisca il comportamento di cui sopra più di una volta, anche in questo caso vengono creati diversi oggetti di tipo Arithmetic Memory Store. La verifica di questi gadget procede in maniera molto simile a quella dei gadget per la scrittura della memoria.

Tuttavia un'attenzione particolare va posta nell'analisi dei gadget che utilizzano un'operazione di addizione che considerano il valore di riporto (ad

esempio adc su architettura intel). Questi gadget possono rilevarsi molto utili. Spesso infatti sono disponibili solo gadget aritmetici che ci consentono di aggiungere un byte per volta. Effettuare un'operazione di got patching (vedi sez. 1.2.3) un byte per volta senza considerare i valori di riporto tra un byte e il successivo può portare ad un risultato errato dell'operazione che si sta effettuando. Per poter permettere la catalogazione di questo tipo di gadget, si è dovuto applicare una piccola patch al framework per permetterci di poter controllare in qualche modo il contesto dei registri che utilizza durante le simulazioni per la classificazione. Questo ci ha permesso di impostare durante le simulazioni il valore del carry flag a zero, facendo in modo che la procedura di catalogazione non scarti i gadget di questo tipo. Anche le verifiche tramite un risolutore SMT impongono che il carry flag abbia valore zero. In più viene aggiunta una seconda coppia di modellazioni SMT con le quali si verifica se il gadget in esame utilizza il carry flag nell'operazione e se a seguito di un'operazione che sicuramente generi un valore di riporto, il carry flag al termine dell'esecuzione del gadget abbia valore uno. Questo ci dà la certezza rispettivamente che il gadget va ad utilizzare il carry flag nelle operazioni di addizione e che le operazioni successive all'interno del gadget non vadano ad influenzare il carry flag.

Come detto sopra, per costruire una catena che ci consente di aggiungere un valore arbitrario ad una locazione di memoria arbitraria, bisogna avere a disposizione o gadget aritmetici la cui dimensione dell'operazione sia di almeno un byte più grande dell'addendo o un gadget che, per l'appunto, consenta di eseguire operazioni utilizzando i valori di ritorno. In questo secondo caso però bisogna anche concatenare, se è disponibile, un gadget che abbia come effetto quello di azzerare il carry flag, da inserire prima dell'operazione. La logica per la ricerca di questo tipo di gadget è presente nella classe `ClearCarryFlag`. È da notare che se questo tipo di gadget non fosse presente la catena generata potrebbe non avere l'esito sperato, tuttavia è presumibile che in molti tipi di exploit il carry flag abbia sempre lo stesso valore all'inizio dell'esecuzione della catena. La sottrazione quindi di un'unità al primo byte dell'addendo dovrebbe essere sufficiente per generare una catena corretta.

Se non sono disponibili né un un gadget che consenta di aggiungere abbastanza byte per volta né un gadget che effettua operazioni considerando il valore di riporto è tuttavia possibile generare una catena con alte probabilità di successo. Infatti l'indirizzo di memoria nel quale viene caricato il file oggetto condiviso della libreria, nonostante sia casualizzato, dev'essere

allineato in memoria al valore specificato nel parametro `p_align` all'interno del Program Header. Questo comporta che un certo numero di bit meno significativi dell'indirizzo di una funzione non cambino da un'esecuzione all'altra. Solitamente sono i 21bis meno significativi, e l'offset tra due funzioni nella libc solitamente è tre byte. Questo fa si che si può calcolare il valore di riporto per i primi due byte. Solo il valore di riporto dell'ultimo byte (di cui conosciamo solo la parte meno significativa) risulterebbe casuale, facendo si che si possano generare catene con probabilità di successo pari almeno al 50%.

3.3 Modulo Payload

Questo modulo si occupa di generare a partire da una lista ordinata di uno o più gadget una stringa binaria di dati da iniettare nello stack del file oggetto eseguibile esaminato, denominata Payload. La preparazione del payload varia rispetto al gadget per il quale si sta creando il chunk e può consistere nel solo indirizzo del gadget, o in schemi più complicati come indirizzi di funzioni più relativi argomenti o a valori da trasferire dallo stack ai registri. Ogni classe che si occupa della generazione di un particolare tipo di chunk ha le informazioni necessarie per permettere al chunk stesso di essere concatenato ad altri chunk per generarne un altro, anch'esso a sua volta concatenabile, che rappresenti i chunk di partenza. Per permettere questa concatenazione ogni chunk deve presentare le seguenti informazioni:

- L'indirizzo che deve trovarsi sullo stack al momento dell'esecuzione dell'istruzione ret del gadget precedente
- In quale posizione del payload inserire l'indirizzo del gadget successivo
- La dimensione degli indirizzi nell'architettura considerata

Le classi che modellano i vari tipi di chunk hanno tutti un metodo, `get_payload`, che restituisce il payload del chunk di riferimento (escluso l'indirizzo del gadget stesso). Quando due gadget vengono concatenati per generare un chunk più generico (nel metodo `PayloadChunk.get_general_chunk`), l'indirizzo di ogni chunk viene inserito nella posizione indicata dal chunk precedente. Nel momento che si richiede il payload vero e proprio (attraverso il metodo `PayloadChunk.chain`) oltre all'operazione di concatenazione viene anche inserito l'indirizzo del primo gadget all'inizio del payload.

Capitolo 4

Conclusioni

Lo sviluppo di uno strumento come Dropper mi ha permesso di studiare ed esplorare problematiche interessanti e complesse. Uno strumento che riesce infatti ad analizzare un file binario e da esso estrarne in maniera automatica le informazioni necessarie per poter costruire un payload funzionante implica la risoluzione di diverse problematiche che abbracciano più aree dell'informatica. In particolare in questo lavoro alcuni strumenti propri dell'intelligenza artificiale hanno giocato un ruolo cruciale rendendo evidente l'apporto che questa disciplina può dare all'area della sicurezza informatica in generale. Tra le problematiche più rilevanti nello sviluppo di uno strumento come Dropper elenchiamo:

1. L'estrapolazione della semantica dei singoli gadget
2. L'individuazione di una sequenza di gadget che ci consente di eseguire una data operazione. Questo punto è complicato dal fatto che gli effetti di un gadget possono interferire con quelli di un altro nonché con l'esecuzione stessa del programma in esame
3. Scegliere, in base alle operazioni disponibili, la strategia da utilizzare per eseguire l'exploit
4. Individuare una sequenza di operazioni che eseguono con successo una data strategia

Il primo punto è stato affrontato, come descritto nei capitoli 2 e 3, attraverso sia tecniche di emulazione che tecniche di verifica basate sulla rappresentazione in formule logiche delle istruzioni dei gadget. Dropper può

estrapolare la semantica di diverse tipologie di gadget che ci permettono di eseguire operazioni, come il caricamento di costanti nei registri o l'effettuare operazioni aritmetiche in locazioni di memoria arbitrarie, e ed è facilmente estendibile per incorporare ulteriori analisi semantiche che consentano di eseguire operazioni sempre più complesse.

La scelta di una sequenza di gadget che ci permetta di eseguire un'operazione articolata è un'altra problematica interessante. Infatti, oltre ad essere sicuri che la catena abbia gli effetti desiderati, è interessante cercare, tra le sequenze di gadget che ci permettano una particolare operazione, quella che produca un payload di minore lunghezza. Infatti, anche se questo dipende molto dalla tipologia di errore che si sta sfruttando, spesso si è limitati nella quantità di dati da poter iniettare.

Le catene che attualmente dropper riesce a generare ci consentono di avere una visione ad un livello di astrazione più alto delle operazioni che possiamo eseguire con i singoli gadget. Per generare catene di queste operazioni, come descritto nei capitoli precedenti, si utilizza un approccio per lo più programmatico, usando quando possibile algoritmi greedy per la generazione di soluzioni di una certa qualità. Tuttavia il problema di generazione di una catena, definito come l'ordine nel quale effettuare alcune azioni (i gadget) per raggiungere un dato stato del sistema, si presta naturalmente ad una traduzione in un problema di planning.

Alcune considerazioni su questo approccio, insieme ad una prima possibile modellazione che utilizzi il linguaggio PDDL 2.1 [2, 10] sono riportate nella sez. 2.1.1.

Le ultime problematiche che sono state affrontate, nella nostra panoramica dal basso verso l'alto, sono la scelta di una tecnica di exploiting e di una sequenza di operazioni per applicare tale tecniche. Tra le strategie applicate con successo nei primi testcase troviamo 1) l'utilizzo della funzione "read" per la scrittura in memoria 2) utilizzo di sequenza di gadget sempre per la scrittura in memoria 3) got patching e 4) return-to-plt

Trovare le sequenze per portare a termine queste tecniche è un problema per certi versi simile a trovare le catene di gadget per eseguire le operazioni "elementari", ma da un punto di astrazione più alto. Vi è qui, tra le difficoltà più rilevanti, quella di esaminare tutte le possibili sequenze di operazioni per applicare quella tecnica. In questa fase i vantaggi dell'utilizzo di un planner sembrano ancora più rilevanti.

I test durante lo sviluppo sono stati effettuati prendendo in esame alcuni livelli di exploit-exercise[1] e alcuni eseguibili contenuti nelle bin-utils (ls, echo, mv).

4.1 Sviluppi futuri

Di seguito sono elencate alcune idee per lo sviluppo futuro di Dropper:

- Utilizzo di un'esecuzione simbolica, che semplificherebbe di molto l'analisi semantica, potendo verificare direttamente le relazioni tra valori in ingresso ed effetti dei gadget
- Ulteriori indagini sull'utilizzo di una modellazione in problemi di planning
- Estrapolazione di gadget non "classici". Ad esempio un'estensione della ROP consiste nell'utilizzare sequenze di istruzioni che terminano con un'istruzione di tipo `jmp` invece che con una di tipo `ret` (infatti questa tecnica prende il nome di *Jump Oriented Programming* [6]). Un altro esempio è quello di ricercare gadget di lunghezza maggiore ma con effetti secondari "controllabili", ad esempio obbligando il gadget a seguire un flusso di esecuzione piuttosto che un altro. Quest'ultima tipologia di gadget è necessaria per eludere alcune tecniche di mitigazione che si basano sul monitoraggio di un numero limitato di istruzioni
- Aumentare le architetture supportate
- Implementazione di nuove strategie di exploit e rendere più semplice l'incorporazione di strategie definite dall'utente

Ringraziamenti

I miei ringraziamenti vanno a un “contenitore” di persone, idee e processi che è l’aula p2 occupata. Ringraziamenti dovuti non solo per la crescita morale, le amicizie e le belle giornate passate là dentro, ma anche perché buona parte della mia formazione “informatica” è dovuta alle discussione, ai progetti e alle iniziative tenute in quel luogo. Un posto dove la mia passione per l’informatica ha trovato modo di crescere e esprimersi svincolata dalle logiche di competizione e mercato che pian piano stanno filtrando a tutti i livelli della formazione universitaria. Un posto che per me e per molti rappresenta gli unici pochi metri liberati in un ponte totalmente occupato.

Un altro ringraziamento va a Spike, per avermi trascinato in qualche modo giù nel low-level e avermi convinto ad esplorare le schizofrenie della Return Oriented Programming.

Bibliografia

[1] exploit-exercise. https://exploit-exercises.com/fusion/.

[2] Pddl - the planning domain definition language, 1997.

[3] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In Edmund Clarke, Tom Henzinger, and Helmut Veith, editors, *Handbook of Model Checking*. Springer, 2014.

[4] Eli Bendersky. pyelftools, 2014.

[5] c0ntex. Bypassing non-executable-stack during exploitation using return-to-libc, 96. http://www.infosecwriters.com/text_resources/pdf/return-to-libc.pdf.

[6] Stephen Checkoway and Hovav Shacham. Escape from return-oriented programming: Returnoriented programming without returns (on the x86. Technical report, 2010.

[7] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 401–416, San Diego, CA, August 2014. USENIX Association.

[8] Solar Designer. return-to-libc, August 1997. Bugtraq.

[9] Solar Designer. stack-patch, August 1997. http://www.openwall.com/linux.

[10] Maria Fox and Derek Long. Pddl2.1: An extension to pddl for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:2003, 2003.

[11] Christian Heitman and Iván Arce. Barf: A multiplatform open source binary analysis and reverse engineering framework, 2014.

[12] ELF Kickers, 1997. `http://www.muppetlabs.com/~breadbox/software/elfkickers.html`.

[13] Long Le. Payload already inside: data reuse for rop exploits. 2010.

[14] Aleph One. Smashing the stack for fun and profit. *phrack*, 96.

[15] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur.*, 15(1):2:1–2:34, March 2012.

[16] Giampaolo Fresi Roglia, Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. Surgically returning to randomized lib (c). In *Computer Security Applications Conference, 2009. ACSAC'09. Annual*, pages 60–69. IEEE, 2009.

[17] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit hardening made easy. In *Proceedings of the USENIX Security Symposium*, 2011.

[18] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 552–561, New York, NY, USA, 2007. ACM.

[19] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM.

[20] Sebastian Porst Thomas Dullien. Reil: A platform-independent intermediate representation of disassembled code for static code analysis, 2009.

[21] Rolf Rolles ulien Vanegue, Sean Heelan. Smt solvers in software security. In *Presented as part of the 6th USENIX Workshop on Offensive Technologies*, Berkeley, CA, 2012. USENIX.