Chapter 1

Introduzione

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

La tecnica forse più abusata era lo *stack smashing* [7], con la quale si compromisero centinaia di sistemi. La tecnica consisteva nello sfruttare un errato controllo dei limiti di un buffer memorizzato sullo stack per poter corrompere lo stack stesso, errore identificato come *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.

Nello stack sono presenti informazioni chiave per il funzionamento del programma. Ad esempio, al momento di una chiamata a funzione, l'indirizzo dell'istruzione successiva 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 dell'indirizzo di ritorno, controllando da dove il programma continuerà. L'attaccante quindi può, compromettendo lo stack, redirezionare il flusso del programma in modo che esegua il 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 granulità una singola pagina. Il tentativo di eseguire delle istruzioni presenti in una pagina non eseguible, così come il tentativo di scrivere su una pagina non

scrivibile causa l'arresto del programma. Il codice leggittimo viene 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 quanto le aree di memoria scrivibili (tra cui l'area di memoria riservata allo stack) non potranno essere eseguite. Un'altra misura di protezione per rendere più difficile questo tipo di attacco è la casualizzazione dell'indirizzo 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 nè del flusso del programma nè dello schema dello stack. Una delle tecniche utilizzata per ottenere il controllo di un programma nonostante lo stack non sia marcato come eseguibile si chiama *return-to-libc*[4]. In un utilizzo tipico di questa tecnica il flusso del programma non viene dirottato su un payload iniettato dall'attaccante ma su una funzione di una qualche libreria utilizzata dal programma. Su un architettura a 32bit 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 di fatto il controllo degli argomenti da passare alla funzione. Potendo, ad esempio, richiamare la funzione system della libc con argomenti arbitrari è possibile eseguire un qualiasi 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 perchè il codice ad essere eseguito risiede in un area di memoria eseguibile. Una tecnica per mitigare questo tipo di attacco è la casualizzazione dello spazio degli indirizzi (*Address Space Layout Randomization*). 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 naive non risulta possibile. Tuttavia su 32bit e sotto alcune condizioni questi indirizzi possono essere facilmente ricavati con un attacco a forza bruta[12]. Anche se su un'architettura 64bit un attacco a forza bruta non risulta fattibile è tuttavia possibile che si riesca a sfruttare qualche al-

tro errore di programmazione per ottenere l'indirizzo della libreria, riuscendo così a montare comunque l'attacco.

Una generalizzazione della return-to-libc è la tecnica che prende il nome di *Return Oriented Programming*[11, 8]. In questo caso il flusso viene redirezionato non più all'inzio 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'instruzione ret. L'istruzione ret preleva dalla memoria dello stack un indirizzo e ridireziona il flusso del programma a quel dato indirizzo. Viene utilizzata al termine dell'esecuzione di una funzione per tornare alla porzione di codice che l'ha richiamata. Potendolo controllare l'attaccante può sistemare lo schema dello stack in modo che una volta che il programma arrivi all'istruzione ret, sullo stack sia presente l'indirizzo di una seconda serie dalle caratteristiche simili alla prima. Ripetendo lo stesso discorso più volte è possibile concatenare un numero arbitrario di queste piccole serie di istruzioni (che prendono il nome di *gadget*). In [9] è stato dimostrato che, quando sono presenti opportunee tipologie di gadget, utilizzando il ROP si possono effettuare computazioni turing-complete. Utilizzando opportunee sequenze di gadget è possibile applicare tecniche per eludere varie misure di protezione [10].

Un programma può essere compilato affinchè il suo codice sia indipendente dalla posizione in cui viene mappato in memoria (*Position Indipendent Code*), in questo modo non solo le librerie, ma anche il programma stesso può essere caricato in un indirizzo casuale di memoria, che cambia tra un'esecuzione e l'altra. In questo caso non è più facilmente possibile montare un attacco che sfrutti i gadget, in quanto la posizione degli stessi non è nota. Altre tecniche sono stati sviluppate per mitigare attacchi di tipi ROP, anche se non risultano efficaci o l'overhead è troppo grande [5].

La *Satisfability Modulo Theories*[2] è un'estensione della logica del primo ordine, che viene estesa con teorie come quella dei bitvector 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 applicazione nell'ambito della sicurezza[] informatica. 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 la simulazione simbolica viene utilizzata sempre di più nell'ambito della sicurezza. Ad esempio può essere utilizzata per mettere in relazione dati in ingresso e comportamento del programma. Semplificando, una volta trovato un modo per poter simulare il programma, ad un valore concreto di ingresso viene sostituito un valore simbolico. Ogni volta che questo valore viene manipolato viene tenuta traccia delle operazioni effettuate su di esso. Le strutture dati del programma possono essere espresse in funzione di quel dato valore simbolico. È possibile poi ricavare in modo automatico una formula SMT che rappresenti queste relazioni, in modo da poter interrogare un risolutore per chiedere, ad esempio, per quali valori in ingresso il programma segua o meno una branch o per quali valori una struttura dati contenga o no un dato valore.

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 consentino ad un gadget di eseguire una determinata funzione (ad esempio i valori di registro per i quali un dato gadget scrivi in una specifica locazione di memoria un dato valore).

Lo scopo di questo lavoro di tesi è lo sviluppo di uno strumento che possa aiutare nel compito di generare una catena di gadget che sia in grado di prendere controllo di un programma, premesse alcune condizioni. In particolare lo strumento riesce a generare catene di gadgets 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* e *return to plt* [10]. Lo strumento utilizza per l'analisi del binario il recente framework di sviluppo BARF [6] e supporta l'architettura intel a 32bit e a 64bit.

Descrizioni dei capitoli...

Chapter 2

dROPper

La prima parte di questo capitolo descrive l'architettura alla quale si sta facendo riferimento per lo sviluppo di dropper mentre nella seconda si farà un'analisi dettagliata dell'attuale implementazione.

2.1 Architettura principale

L'obiettivo di dropper è quello di generare, analizzando un file oggetto eseguibile, una catena di gadget che iniettata nello stack dell'eseguibile consenta di prenderne il controllo, eseguendo un comando arbitrario sulla macchina. Lo strumento deve poter applicare tecniche note per eludere misure di protezioni esistenti come la casualizzazione degli indirizzi (ASRL), l'isolamento delle pagine (NX) o la read-only relocation. Dropper analizza il binario utilizzando alcune delle funzionalità del framework per l'analisi dei binari BARF. Il linguaggio utilizzato per la scrittura di dropper è python. I file oggetto eseguibili supportati da dropper sono ELF32 e ELF64 ed è destinato ad analisi di binari su piattaforme GNU/Linux. Dropper, nella sua fase di analisi, estrapola dal file binario i gadgets disponibli, per poi estrapolarne la semantica. Per questi compiti dropper si avvale sia dell'emulazione dei gadget, sia della loro modellizzazione in formule SMT, come descritto più in dettaglio nelle prossime sezioni. Una volta estrapolata la semantica dei gadget è possibile costruire piccole catene di uno o più gadget che consentano di eseguire operazioni particolari (come modificare il valore dello stack, chiamare una funzione, settare un registro, scrivere in memoria, ecc). Dropper assembla queste piccole catene in catene sempre più lunghe che eseguono oper-

azioni più complesse e articolate. La generazione di queste sequenze avviene, nell'implementazione attuale, in maniera programmatica, cioè la procedura per la sua costruzione è codificata in un algoritmo. È interessante esplorare metodi di generazione di sequenze in maniera dichiarativa, ad esempio utilizzando linguaggi di planning come PDDL[1]. Una possibile modellizzazione è quella in cui per ogni sequenza "elementare" vengono codificate le precondizioni e gli effetti. Viene modellato lo stato del sistema (memoria e registri) che si desidera ottenere dopo l'esecuzione di una sequenza di gadget. Una volta codificato il problema in questi termini si può chiedere ad un planner, come se fosse una sorta di oracolo, in che sequenza collegare le varie azioni (cioè le varie sequenze elementari di gadget) per ottenere quello stato. La rappresentazione delle informazioni sulla semantica dei gadget è espressa in questo modo in forma esplicita, e non più codificata in un algoritmo.

Dropper è composto da tre moduli principali:

- dropper: 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 le tecniche da utilizzare per riuscire a generare la catena

- gadgetstools: È il modulo che si occupa del'estrapolazione, dell'analisi, della scelta e dell'assemblaggio dei gadget da utilizzare nella composizione della catena e dei vari sui frammenti

- payloadchunk: 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

2.1.1 Modulo dropper

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. Il modulo legge il file e instanzia un oggetto di tipo gadgets tools che si occuperà di estrapolare i gadgets dal binario e analizzarli. Per il parsing delle strutture dati presenti nell'elf dropper utilizza la libreria pyelftools [3]. Combinando le informazioni provenienti da queste sezioni dropper ricava la lista

delle funzioni che il file oggetto eseguibile utilizza e le relative entry nella *Procedure Linkage Table* (vedi TODORIF). La presenza di alcune funzioni può semplificare molto la generazione della catena, di fatto trasformandola in una serie di *return-to-plt*. Dropper analizza altre strutture chiave dell'ELF per individuare aree nella memoria dell'immagine del processo dove è possibile scrivere. L'utente, interagendo con il modulo, fornisce informazioni aggiuntive necessarie alla generazione della catena e linee guida sulla strategia da utilizzare. Tra le informazioni che il modulo non può ricavare automaticamente 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 (per la ricavazione degli offset da utilizzare nelle tecniche di sovrascrizione della GOT Table vedi TODO)
- Eventuali caratteri che non possono comparire all'interno del payload
- Eventuali limiti sulla lunghezza dei payload
- Il comando da eseguire

Con tutte queste informazioni il modulo può quindi procedere nello stabilire quale strategia utilizzare e richiedere agli altri moduli le sequenze di gadget necessarie. Lo schema principale, con alcune varianti che analizzeremo in seguito, si può dividere in due fasi: (a) preparare gli argomenti per eseguire una chiamata ad una funzione di libreria che ci consenta di eseguire un comando arbitrario e (b) eseguire quella funzione.

Alcune delle strategie che si possono utilizzare per raggiungere il primo scopo insieme alle precondizioni necessarie sono elencate di seguito:

- Utilizzo della funzione read

  Condizioni necessarie:

  - Controllo di un file descriptor da parte dell'utente
  - Utilizzo da parte del file eseguibile della funzione read o possibilità di richiamarla attraverso tecniche basate sulla ricavazione degli indirizzi partendo da indirizzi nella GOT e offset relativi

Viene richiamata la funzione read, impostando gli argomenti in modo che i dati letti dal file descriptor vengano scritti nell'area di memoria voluta

- Utilizzo di gadget per la scrittura della memoria

  Condizioni necessarie:

  – All'interno del file oggetto eseguibile deve essere presente almeno un gadget in grado di scrivere in memoria un valore arbitrario

  – I valori da scrivere nella memoria non possono presentare caratteri che non possono essere presenti nel payload, essendo essi stessi scritti nel payload della catena generata

  Solitamente una sequenza del genere consente di scrivere al massimo 8 byte per volta in architettura 64bit (4 byte per volta su un'architettura a 32bit) e sono composte da almeno due gadget (almeno un gadget serve per settare i registri utilizzati del gadget che esegue la scrittura vera e propria). Questa tecnica tende quindi a generare catene piuttosto lunghe se la dimensione dei valori da scrivere in memoria è grande

Queste appena elencate le uniche due tecniche attualmente implementate, strategie alternative potrebbero comprendere:

- Utilizzo delle funzioni strcpy o memcpy per copiare byte presenti in posizioni note all'interno dell'eseguibile nella posizione desiderata [10]. Questa tecnica, copiando verosimilmente solo un byte per volta tende anch'essa a generare payload abbastanza lunghi

- Nel caso un bug renda possibile la conoscenza dell'indirizzo dello stack gli argomenti possono essere scritti direttamente nello stack

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 funzione è utilizzata all'interno del programma, basterà ritornare alla sua entry nella Program Linkage Table, che ha una posizione nota e fissa all'interno della memoria dell'immagine del

processo. Nel caso invece che la funzione non venga utilizzata 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. Questo funziona perchè nonostante l'indirizzo inziale al quale una libreria viene caricata sia casualizzato, gli offset tra le funzioni non cambiano. Alcune considerazioni sull'utilizzo di questa tecnica:

- Devono essere noti gli offset delle funzioni all'interno del file oggetto condiviso della libreria utilizzato sulla macchina bersaglio (anche se, variando in un range relativamente piccolo, non dovrebbe essere impossibile, se le condizioni lo consentono, montare un attacco di tipo forza bruta per ricavarli)

- il binario deve presentare una sequenza di gadgets che ci consenta di eseguire un'operazione di somma (o un'operazione che ci consenta si 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 perchè in un'operazione di somma byte per byte dev'essere considerato un eventuale valore di riporto tra un byte e il successivo. Questo può essere risolto se il tipo di gadget trovato usa un'istruzione di tipo adc (add with carry), controllando che tra gli effetti secondari degli altri gadgets non venga modificato il carry flag. 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 in TODO

- Un tipo di attacco di questo tipo può essere mitigato se le rilocazioni vengono impostate come read-only. In questo caso infatti gli indirizzi delle funzioni vengono calcolati tutti al momento di caricamento del programma, e la GOT, una volta modificata opportunamente, viene resa non scrivibile. In questo caso è ancora possibile montare un attacco del genere, ma l'operazione aritmetica non può avvenire direttamente sulla got e non può essere utilizzato un ret-to-plt per saltare all'indirizzo calcolato. Bisogna cioè avere a disposizione i gadget nec-

essari per leggere dalla memoria, effettuare l'operazione aritmetica e eseguire un jmp al nuovo indirizzo calcolato

È interessante notare come, anche se vi è una guida nelle operazioni da seguire basata sulla strategia scelta, la seguenza 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.

2.1.2 Modulo gadgetstools

Il modulo gadgets tools si occupa di ricavare i gadgets presenti nel file oggetto eseguibile, di estrapolarne la semantica e della creazione di sequenze di gadget che se eseguite assolvino funzioni particolari (come scrivere in memoria, settare un registro, ecc.). Questo modulo interagisce con payloadchunk per trasformare seguenze ordinate di gadget in un payload vero e proprio da poter iniettare sullo stack.

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[9].

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. Per questo tipo di analisi si utilizzano due strumenti principali: la simulazione e la traduzione delle istruzioni in formule SMT. Un'analisi completa della semantica presenta alcuni limiti dovuti alla difficoltà di una modellizzazione completa del sistema nonchè alla conoscenza parziale del contesto (memoria e registri) reale in cui si troveranno ad essere esguite 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. L'estrapolazione della semantica di un gadget avviene in

tre fasi: (a) emulazione e cataloghizzazione, (b) verifica e validazione e (c) verifica degli effetti secondari.

Emulazione e cataloghizzazione

La classe GadgetClassifier 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 TODORIF. identificati quali registri vengono utilizzati come "operandi" della funzione alla quale il gadget viene associato. Basandosi su una simulazione e non su un effettiva analisi semantica questa cataloghizzazione deve essere poi verificata. Per le simulazioni vengono utilizzati dati concreti ma una simulazione simbolica, funzionalità ancora non disponibile all'interno del framework BARF, consentirebbe una cataloghizzazione più veloce e accurata esplicitando le relazioni tra i valori che i registri hanno all'inizio e alla fine della simulazione

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. Per questa verifica si scelgono casualmente gli effetti che deve avere la funzione sul contesto. Ad esempio se stiamo verificando un MemoryStore si sceglie sia una locazione casuale di memoria sia un valore casuale da scrivere in questa locazione. Si traduce il gadget in formule SMT e si impone che il contesto finale rifletta gli effetti, scelti casualmente, che la nostra funzione deve avere. Nel nostro esempio si impone che la locazione di memoria cambi al valore casuale scelto. Si chiede a questo punto al risolutore se il sistema di formule ammette una soluzione. Se il sistema non ammette soluzione il gadget può essere scartato, in quanto non essendo usabile per quei valori perde la sua generalità. Nel caso invece il sistema ammetta soluzione vengono richiesti al risolutore i valori iniziali, all'interno del modello trovato, dei registri che, secondo la cataloghizzazione al punto uno, sono gli "operandi" della funzione. Si impongono poi al risolutore nuovamente le formule SMT che rappresentano il gadget. Questa volta si impongono però

come valori iniziali per i registri operandi i valori ottenuti dalla prima verifica e, inoltre, che il valore finale sia diverso da quello aspettato. La non soddisfacibilità di questa formula ci dà la certezza che in tutti i modelli che soddisfano la semantica del gadget, che hanno come valori iniziali dei registri "operandi" quelli ottenuti dalla prima verifica, lo stato finale del contesto rispecchi i valori attesi. Questa verifica esclude che ci siano effetti secondari che possono interferire con gli effetti della nostra funzione. È da notare che queste proprietà sono verificate solo per dei valori casuali e dovrebbero essere verificate per tutti i valori. Questo non viene fatto sia per questioni di efficenza sia per non includere quantificatori universali nelle formule, complicando di molto il lavoro del risolutore e rischiando di ottenere formule non decidibili. Questo problema può essere risolto ripetendo queste verifiche al momento dell'utilizzo reale del gadget con i valori concretamente utilizzati.

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 legge da una locazione di memoria non leggibile causerà l'arresto del programma. La modellizzazione in SMT 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 al punto 2, e ci si assicura che non vengano lette e scritte aree di memoria che non siano quelle relative alla funzione propria del gadget o allo stack.

La linea guida generale appena descritta va adattata alla tipologia di gadget analizzata, per le implementazioni specifiche fare riferimento alla seconda parte di questo capitolo. Inoltre per poter essere utilizzato 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.

Inoltre è da notare come la verifica potrebbe avvenire in modo più effi-

cente, utilizzando delle simulazioni mirate a scartare i gadget che sicuramente non ci consentono di utilizzare valori arbitrari. In questa fase dello sviluppo del progetto però l'efficenza non viene considerata come prioritaria. 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 PayloadChunk 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.

2.1.3 Payload Chunk

Questo modulo si occupa di generare il payload vero e proprio partendo dai gadget analizzati dal modulo gadgetstools. 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 gadgets, 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 stesso. In base alla tipologia di gadget questo valore potrebbe non essere alla fine del payload relativo al gadget stesso
- valore finale dello stack dopo l'esecuzione del gadget

Le informazioni relative alle tre problematiche descritte sopra vengono estrapolate dal gadgetstools. 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 restiuiscono rispettivamente il payload dei chunk concatenati o un chunk che può essere nuovamente concatenato e rappresenta la concatenazione dei chunks in ingresso.

2.2 Implementazione attuale

Chapter 3

Exploit

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 questo capitolo tratterà eslusivamente di come queste tecniche sono implementate in ambiente GNU/Linux, su archiettura 8086 e amd64. Nella prima parte di questo capitolo 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*. Una volta creata l'immagine del processo il controllo viene ceduto al programma che inizia la sua esecuzione. La prossima sezione si soffermerà solo sulle parti rilevanti per il lavoro svolto.

3.1 ELF, loading

ELF (Executable and Linkable Format) viene utilizzato per 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 *file oggetto condiviso*
- *file oggetto eseguibile*: contiene un programma adatto per la sua esecuzione, questo file ha le informazioni necessarie affinchè 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 linking statico insieme ad altri file oggetto rilocabili o file oggetto condivisi per creare altri file oggetto, oppure può essere collegato in fase di creazione dell'immagine del processo di un file oggetto eseguibile.

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 responsabili di questi due "punti di vista" sono diffrerenti, 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 solo sulle principali strutture dati all'interno dell'ELF rilevanti al fine di creare l'immagine di un processo.

3.1.1 Program header table

L'ELF header (che è l'unica struttra dati che ha una posizione fissa all'interno del file) funge da una sorta di mappa, e ci consente di localizzare tutte le altre stuture 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. Una entry all'interno della program header table è rappresentata dalla seguente struttura C:

L'attributo ptype indica il tipo di entry. Per il nostro scopo ci limiteremo a descrivere le entry di tipo PTLOAD, PTINTERPETER e PTDYNAMIC. Le entry di tipo ptload rappresentano informazioni su un segmento da caricare in memoria. Il campo offset e filesz indicano rispettivamente l'offset e la sua grandezza all'interno del file. pvaddr e pmemsz 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 filesz). pflags 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 affichè risulti indipendente dal indirizzo base nel quale risiede in memoria, il campo pvaddr potrebbe essere nullo e la posizione del segmento in memoria potrebbe essere casualizzata (vedi TODO:INSRIF).

La voce all'interno della program header table con il tipo PTINTERP 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 l'interprete, dandogli il controllo e la possibilità di accedere alle informazioni presenti nel file oggetto principale. Sarà compito poi dell'interprete creare l'ambiente necessario all'esecuzione dell' eseguibile. 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 dinamice, il linker aggiunge alla program header teble un elemento di tipo tinterpreter indicando 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. Questo modo di risolvere i simboli viene denominato *lazy binding*. 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.

- Una .dynamic section che contiene gli inidirizzi di altre strutture necessarie al processo di dynamic linking e la lisa delqle librerie necessarie all'esecuzione del file oggetto eseguibile
- Una .hash section contiene una tabella dei simboli
- La .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 utilizzate dal'eseguibile

Bibliography

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

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

[3] Eli Bendersky. pyelftools, 2014.

[4] c0ntex. Bypassing non-executable-stack during exploitation using return-to-libc, 96.

[5] 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.

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

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

[8] 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.

[9] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Info. & System Security*, 15(1), March 2012.

[10] 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.

[11] 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.

[12] 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.