Seguridad - Clase Práctica

Sistemas Operativos
DC - UBA - FCEN

1er Cuatrimestre de 2023

Temario

Introducción

Problemas de seguridad clásicos (algunos ejemplos)

- 01 - Format String
- 02 - Environment Variables
- 03 - Buffer Overflow
- 04 - Buffer (Stack) Overflow
- 05 - Integer Overflow / Underflow
- 06 - Permisos
- 07 - Denial of Service
- 08 - SQL Injection

Mecanismos de Protección del SO

¿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.

Exploits

Exploit

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

Temario

Introducción

Problemas de seguridad clásicos (algunos ejemplos)

- 01 - Format String
- 02 - Environment Variables
- 03 - Buffer Overflow
- 04 - Buffer (Stack) Overflow
- 05 - Integer Overflow / Underflow
- 06 - Permisos
- 07 - Denial of Service
- 08 - SQL Injection

Mecanismos de Protección del SO

01 - Format String (código)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4
5  int BUFFERSIZE = 512;
6
7  int main(int argc, char** argv) {
8      char command[BUFFERSIZE+1];
9
10     if(setuid(0) == -1) {
11         perror("setUID ERROR");
12     }
13
14     snprintf(command, BUFFERSIZE, "ping -c 4 %s", argv[1]);
15
16     printf("Executing: '%s'\n", command);
17     system(command);
18
19     return 0;
20 }
```

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

- **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.
  - ~~whitelist~~/allowlist: validar que tenga el formato requerido (en este caso de IP o hostname).
  - ~~blacklist~~/blocklist: sanitizar caracteres peligrosos o inválidos (ejemplo: - ' "; & etc).

Temario

02 - Environment Variables (código)

```
1  int BUFFERSIZE = 512;
2
3  int main(int argc, char** argv) {
4      char command[BUFFERSIZE+1];
5      char *ipaddr_sanitized = strtok(argv[1], " ;&|()");
6      snprintf(command,
7          BUFFERSIZE,
8          "ping -c 4 %s",
9          ipaddr_sanitized);
10
11     if(setuid(0) == -1) {
12         perror("setUID ERROR");
13     }
14
15     printf("Executing: '%s'\n", command);
16     system(command);
17     return 0;
18 }
```

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
```

02 - Environment Variables

- **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

03 - Buffer Overflow

Veamos un saludador básico...

saludador.c

```
1  #include <stdio.h>
2
3  int main(int argc, char* argv[]) {
4      char nombre[80];
5
6      printf("Ingrese su nombre: ");
7      gets(nombre);
8      printf("Hola, %s!\n", nombre);
9
10     return 0;
11 }
```

03 - Buffer Overflow

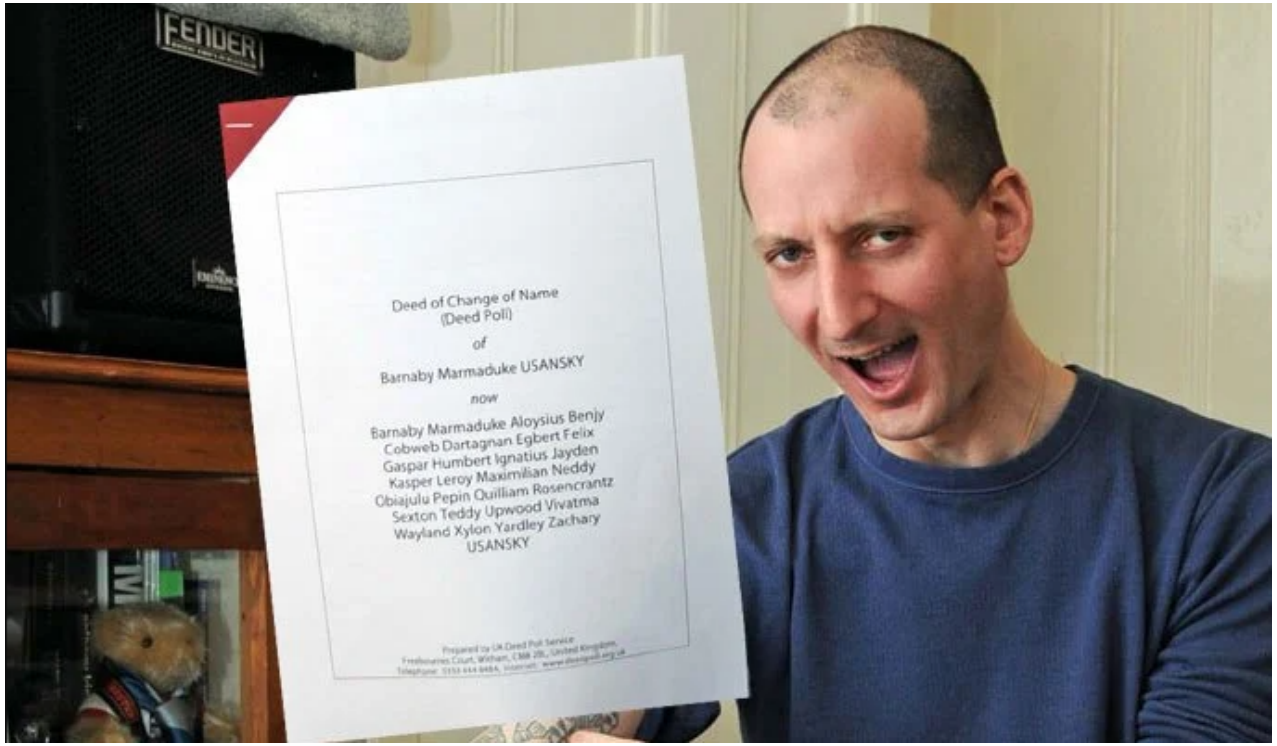
Veamos un saludador básico...

saludador.c

```
1  #include <stdio.h>
2
3  int main(int argc, char* argv[]) {
4      char nombre[80];
5
6      printf("Ingrese su nombre: ");
7      gets(nombre);
8      printf("Hola, %s!\n", nombre);
9
10     return 0;
11 }
```

¿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

login.c (parte 1)

```
1  void login_ok() {
2      printf("Login granted.\n");
3      system("/bin/sh");
4  }
5
6  void login_fail() {
7      printf("Login failed, password was not valid\n");
8  }
9
10 struct login_data_t {
11     char password[100];
12     bool valid;
13 } login_data;
```

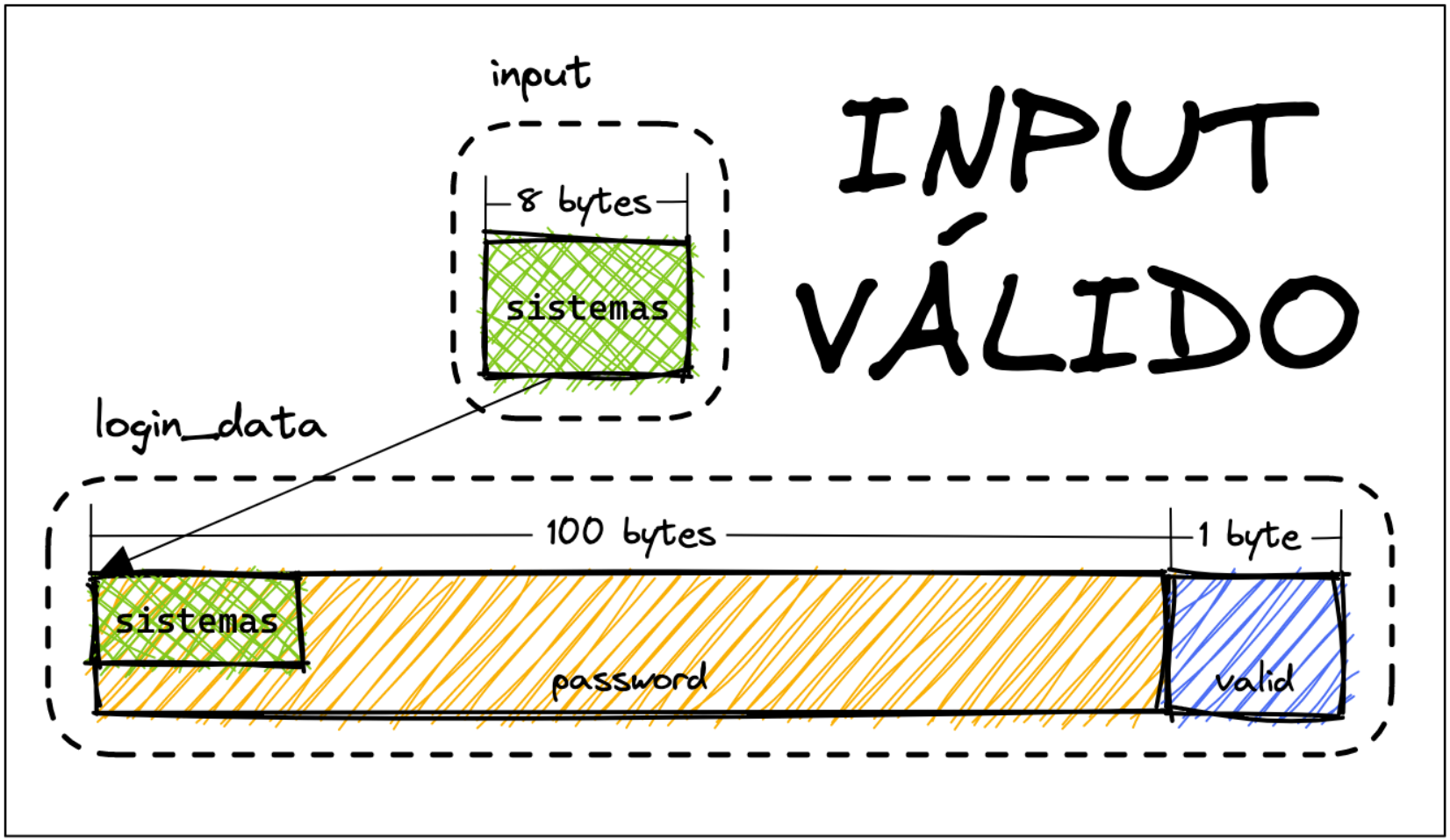
login.c (parte 2)

```
1  void validate_password() {
2      login_data.valid = false;
3
4      printf("Insert your password: ");
5      scanf("%s", login_data.password);
6
7      printf("Password is: ");
8      printf(login_data.password);
9      printf("\n");
10
11     if(strcmp(login_data.password, "porfis")==0) {
12         login_data.valid = 1;
13     }
14 }
```

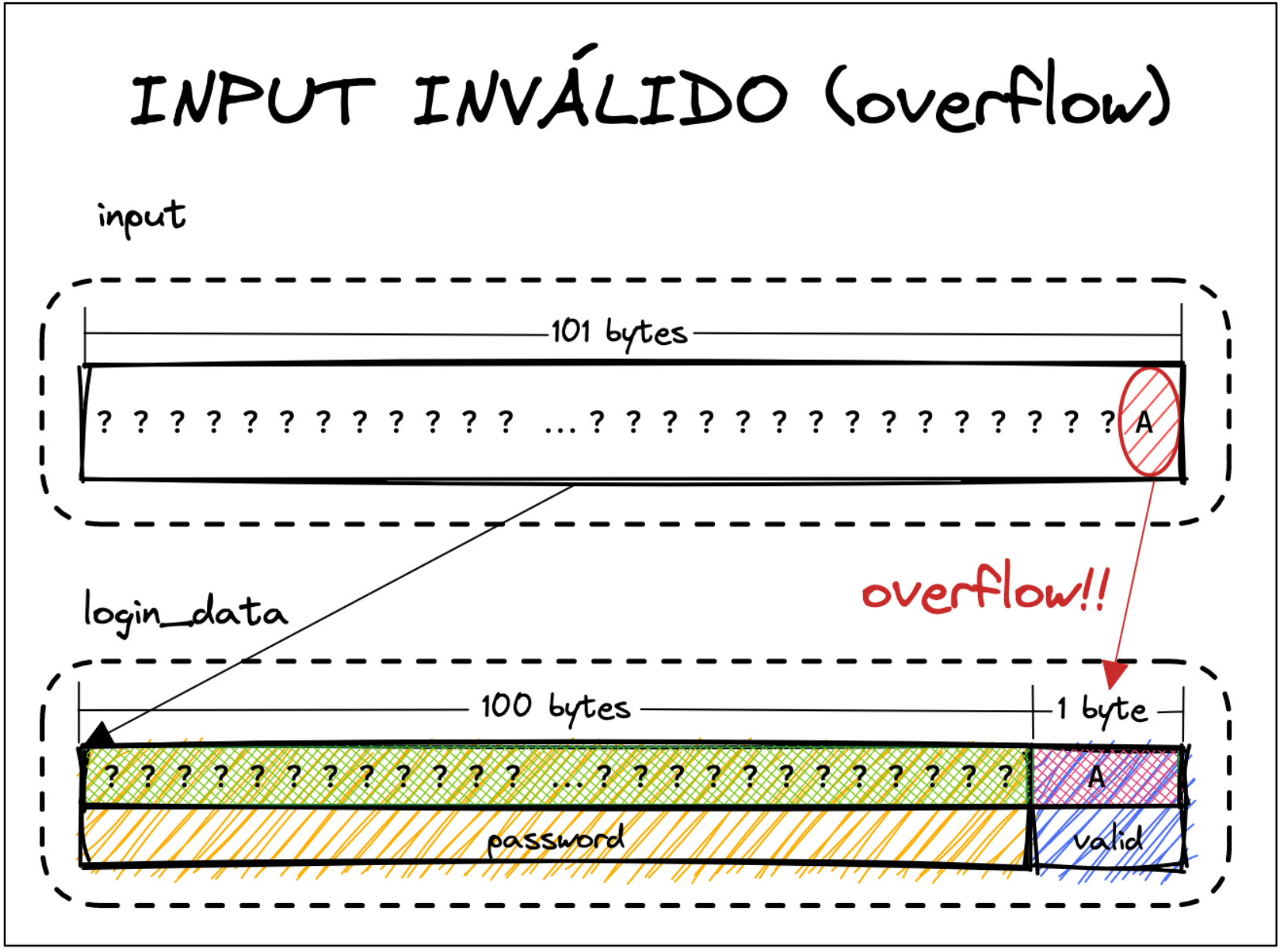
login.c (parte 3)

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

03 - Buffer Overflow (input válido)



03 - Buffer Overflow (exploit)



03 - Buffer Overflow

- **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

login.c (parte 1)

```
1  void login_ok() {
2      printf("Login granted.\n");
3      system("/bin/sh");
4  }
5
6  void login_fail() {
7      printf("Login failed, password was not valid\n");
8  }
```

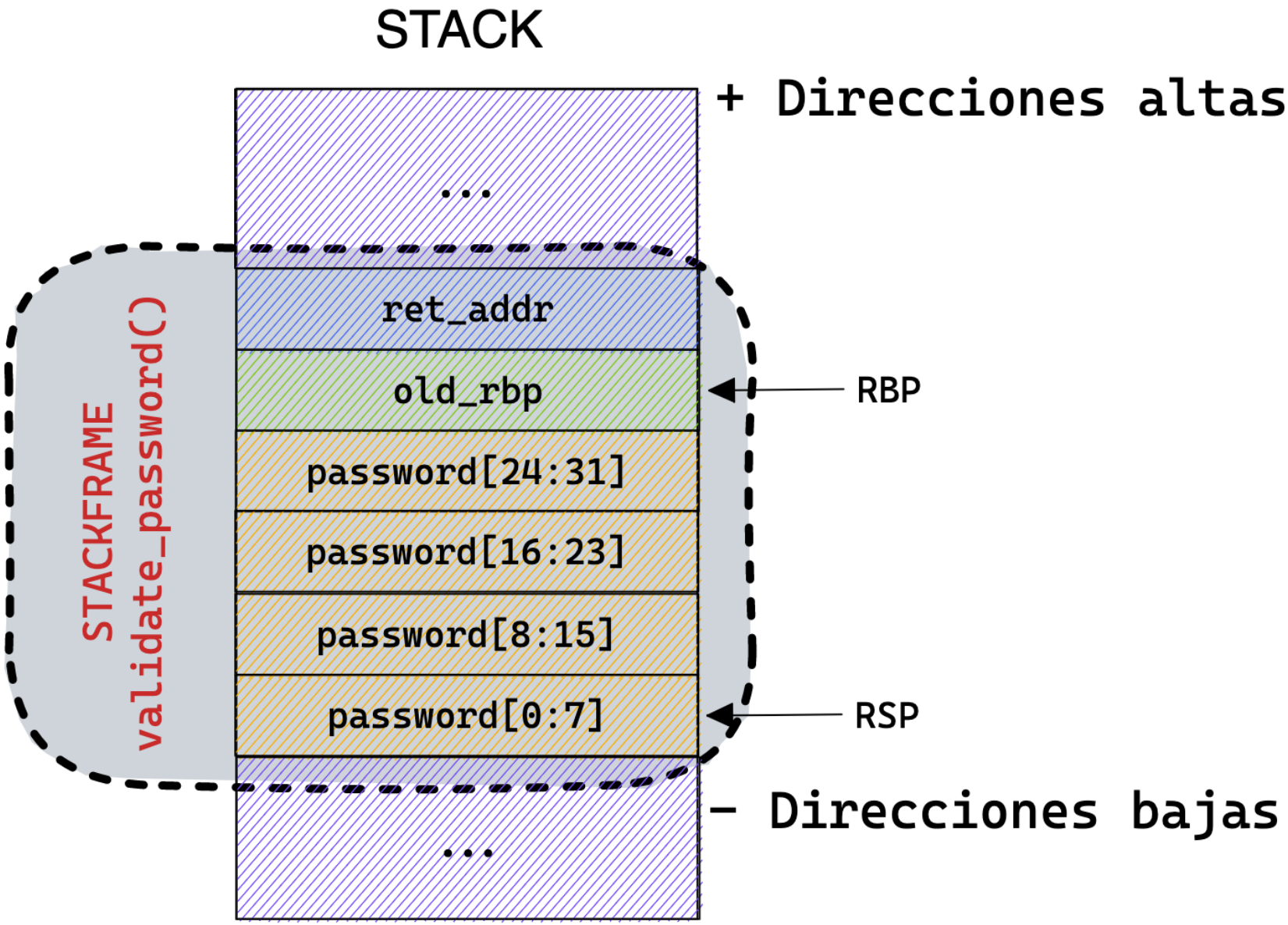
login.c (parte 2)

```
1  bool validate_password() {
2      char password[32];
3
4      printf("Insert your password: ");
5      scanf("%s", password);
6
7      printf("Password is: ");
8      printf(password);
9      printf("\n");
10
11     return strcmp(password, "porfis")==0;
12 }
```

login.c (parte 3)

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

04 - Stack Overflow (stack)



04 - Stack Overflow

- **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

05 - Integer Overflow / Underflow

¿Se cumplen las siguientes afirmaciones?:

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

05 - Integer Overflow / Underflow

¿Se cumplen las siguientes afirmaciones?:

- `a = (a * b) / b` $\forall$ `a,b` $\in$ `int`
- `a` $\leq$ `(a+b)` $\forall$ `a,b` $\in$ `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 Underflow (ejemplo real GRUB[1])

```
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 }
```

[1] 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
6  void login_fail() {
7      printf("Login failed, password was not valid\n");
8  }
```

05 - Integer Overflow

login.c (parte 2)

```
1  bool validate_password(const char *input) {
2      char password[128];
3      char input_len = strlen(input);
4
5      if(input_len<128) {
6        strcpy(password, input);
7        printf("Password is: %s\n", password);
8      } else {
9        printf("Error: password should be < 128 chars.");
10     }
11
12     return strcmp(password, "porfis")==0;
13 }
```

login.c (parte 3)

```
1  int main(int argc, char const *argv[]) {
2      if(setuid(0) == -1) {
3          perror("setUID ERROR\n");
4      }
5
6      if(argc < 2) {
7        perror("Use: ./login password\n");
8      }
9
10     if(validate_password(argv[1])) {
11         login_ok();
12     } else {
13         login_fail();
14     }
15     return 0;
16 }
```

05 - Integer Overflow / Underflow

- **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

06 - Permisos

permisos.py

```
1  #!/usr/bin/sudo python3
2  import os
3  import sys
4
5  FORBIDDEN=[";", "/", "(", ")", ">", "<", "&", "|"]
6
7  if len(sys.argv) <= 1:
8    print("Use: ping IP")
9    exit()
10
11 hostname = sys.argv[1]
12 for c in FORBIDDEN:
13   if c in hostname:
14     print("Wrong hostname!!")
15     exit()
16
17 os.system("/sbin/ping -c 1 " + hostname)
```

06 - Permisos (exploit)

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

¿Ven algo raro?

06 - Permisos (exploit)

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

¿Ven algo raro?

¡Todos los usuarios tienen permiso de escritura!

07 - Permisos

- **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.

Temario

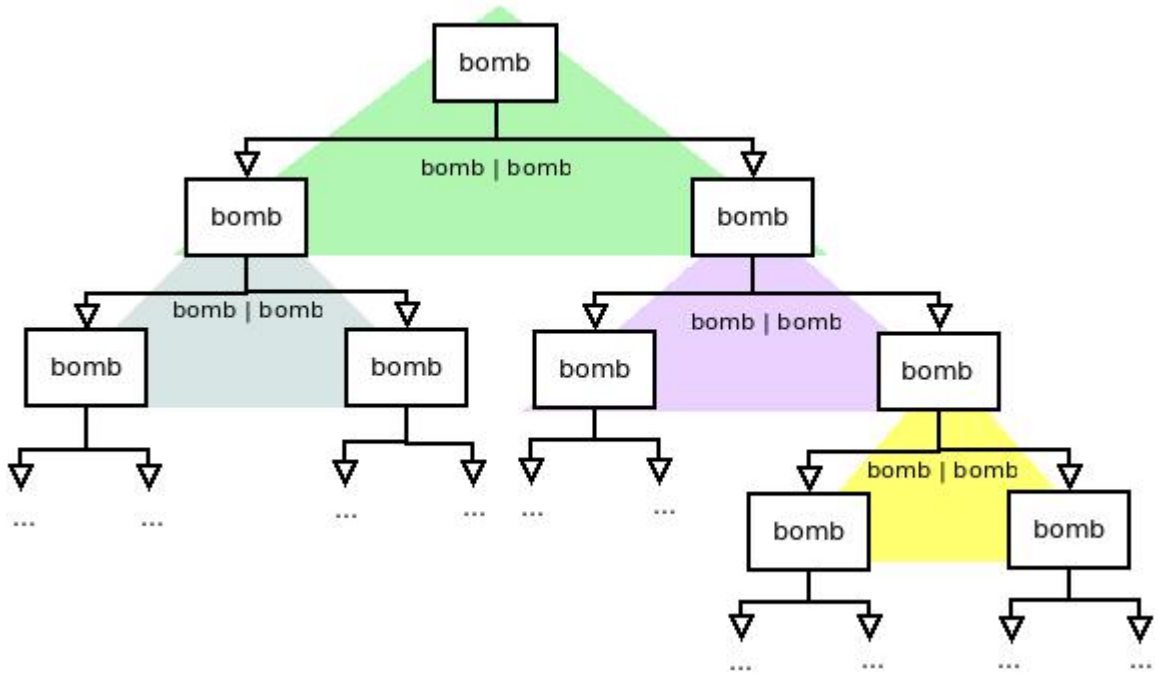
07 - Denial of Service

Fork Bomb (Bash)

```
1  #!/usr/bin/env bash
2  :(){ :|: & };:
```

Fork Bomb (C)

```
1  int main() {
2      while(1) fork();
3      return 0;
4  }
```



Fork Bomb (ASM)

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

Temario

08 - SQL Injection

- Ejemplo de consulta a una base de datos para la validación de un usuario:

```
SELECT *                                                          1
FROM   usuarios                                                   2
WHERE  login=''$USER'' AND pass=''$PASS''                         3
```

  donde $USER y $PASS son controlados por el usuario.
- Si hay una entrada en la base de datos con esas credenciales, el SELECT lo devuelve.
- **¿Seguro?**

08 - SQL Injection

- Ejemplo de consulta a una base de datos para la validación de un usuario:

```
SELECT *                                                          1
FROM   usuarios                                                   2
WHERE  login=''$USER'' AND pass=''$PASS''                         3
```

  donde `$USER` y `$PASS` son controlados por el usuario.

- Si hay una entrada en la base de datos con esas credenciales, el `SELECT` lo devuelve.
- **¿Seguro?**
- ¿Qué pasaría si el usuario ingresara lo siguiente?

  USER: `admin`
  PASS: `nose'' OR ''1''=''1`

08 - SQL Injection

La consulta queda conformada de la siguiente manera:

```
SELECT *                                                                    1
FROM   usuarios                                                             2
WHERE  login=''admin'' AND pass=''nose'' OR ''1''=''1''                     3
```

- El reemplazo se realiza en el programa y la consulta ya reemplazada se envía a la base de datos.
- El `SELECT` devuelve la entrada de la base de datos aún cuando el *password* ingresado no es el correcto.
- El programa asume que el *password* es correcto porque el `SELECT` devolvió la entrada.

08 - SQL Injection

El ataque básicamente es una variante de format string.

- **Problema**: Se utiliza input provisto por el usuario directo en una consulta SQL.
- **Impacto**: Cualquier posibilidad derivada de poder manipular una base de datos
  - Escalado de privilegio o login indebido.
  - Destrucción de datos, por ejemplo, PASS: "`'; DROP TABLE usuarios; --`"
  - Denegación de servicio.
  - Obtención de datos privados.
- **Solución**: limpiar/sanitizar los datos antes de usarlos en una consulta.

08 - SQL Injection (xkcd.com - traducido)

XKCD: "`Robert'); DROP TABLE Students; --`" alias "Little Bobby Tables".

08 - SQL Injection

Mecanismos de Protección del SO

Algunos sistemas operativos implementan uno o más mecanísmos 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 mecanísmos que se utilizan en conjunto para intentar mitigar diferentes clases de ataques.

DEP

- 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.

DEP

- 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?

DEP

- 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 "bypassear" esta protección: ROP (Return-Oriented Programming)

ASLR

- 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 "bypasseable" (aunque puede ser más difícil)

Stack Canaries

- 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 "bypasseable".

Todas estas técinas 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.