Seguridad

Explicación de práctica 5

Sistemas Operativos

Facultad de Informática
Universidad Nacional de La Plata

2025

Mecanismos de seguridad en Linux

- Permisos de acceso en el Filesystem “UGO”
  - Ver también setuid y setgid
- Permisos extendidos en el FS con ACLs
- Namespaces
  - Permite aislar procesos.
- CGroups
  - Permite limitar el uso de recursos.
- Capabilities
  - Permite limitar los privilegios de un proceso que corre con UID = 0
- SELinux: Mandatory Access Control
  - Utiliza políticas para definir a qué recursos puede acceder un proceso.
- AppArmor
  - Similar a SELinux, más simple de configurar.

Mecanismos de seguridad en Linux

Protecciones provistas por el kernel para algunos ataques específicas:

- ASLR (Address Space Layout Randomization) - Procesos de usuario
  - Randomiza la ubicación de las bibliotecas y el stack, lo que dificulta algunos ataques de stack overflow.
- KASLR (Kernel ASLR)
- NX (No eXecute) bit
  - Permite marcar áreas de memoria como no ejecutables.

Otros mecanismos

- UEFI Secure Boot + Linux Kernel firmado + Verificación de firma de módulos
- Cifrado de disco
- Firewalls
  - Herramientas como iptables y firewalld permiten configurar reglas de filtrado de paquetes.

Práctica

Nos enfocaremos en:

- ASLR (espacio de usuario) - Stack Overflow
- SystemD con Namespaces + CGroups + Capabilities
- AppArmor (el lunes que viene)

Memory Layout de un Proceso

Memory Layout de un Proceso

- Text: Instrucciones
- Data: Variables globales
- Heap: Variables alocadas dinámicamente
- Stack:
  - Variables locales
  - Argumentos de funciones (no siempre)
  - Dirección de retorno

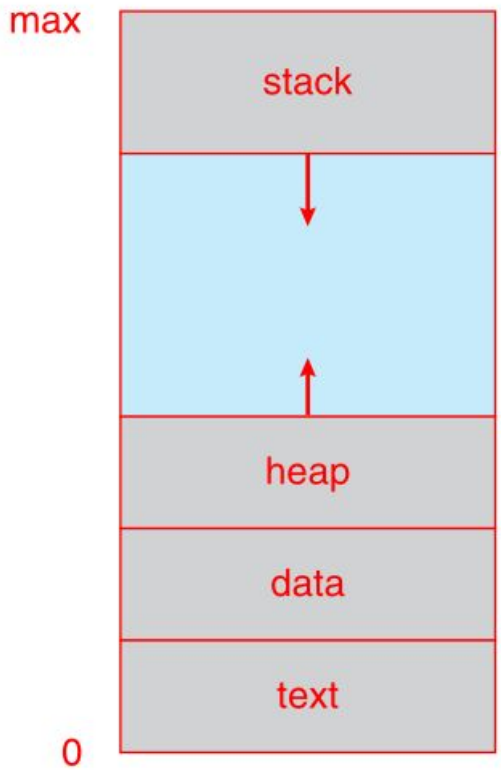
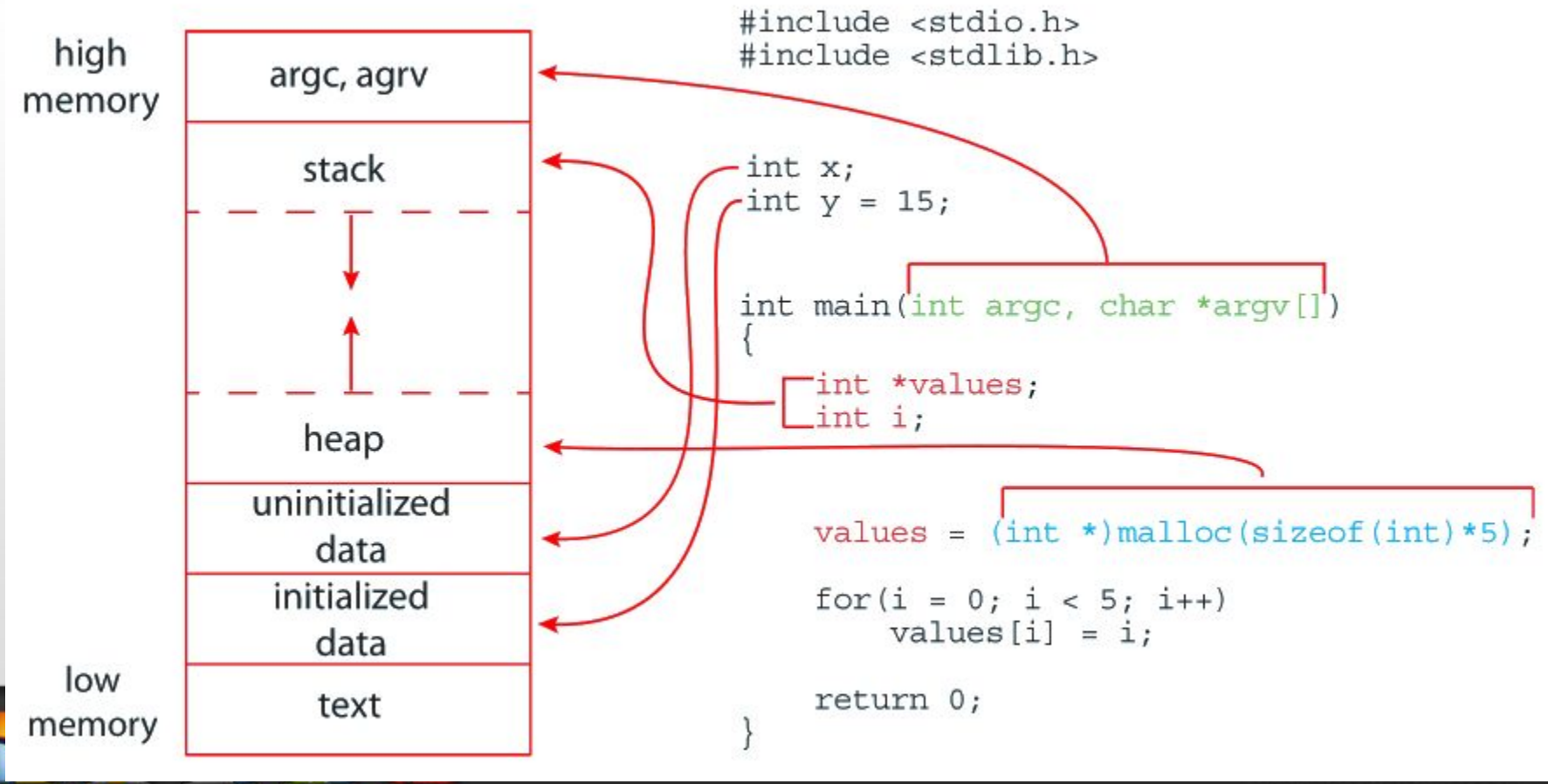


Figure 3.1 Layout of a process in memory.

Memory Layout de un Proceso vs código C



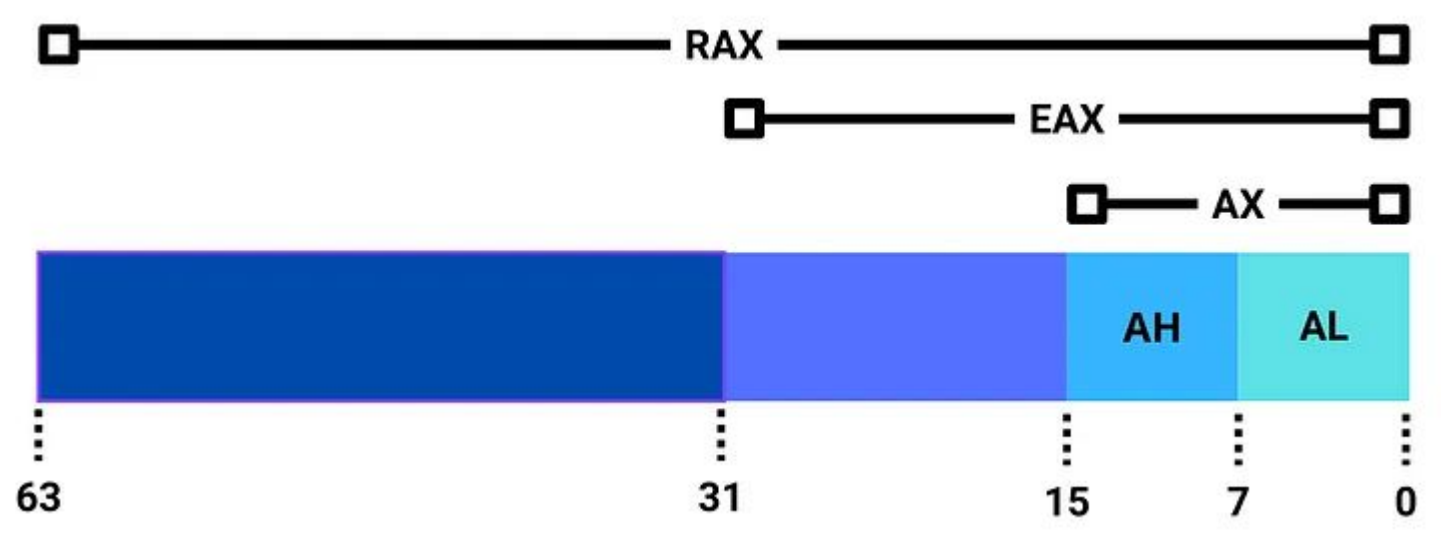
Calling conventions en x86_64

Register	Conventional use	Register	Conventional use
%rax	Return value, callee-owned	%rsp	Stack pointer, caller-owned
%rdi	1st argument, callee-owned	%rbx	Local variable, caller-owned
%rsi	2nd argument, callee-owned	%rbp	Local variable, caller-owned
%rdx	3rd argument, callee-owned	%r12	Local variable, caller-owned
%rcx	4th argument, callee-owned	%r13	Local variable, caller-owned
%r8	5th argument, callee-owned	%r14	Local variable, caller-owned
%r9	6th argument, callee-owned	%r15	Local variable, caller-owned
%r10	Scratch/temporary, callee-owned	%rip	Instruction pointer
%r11	Scratch/temporary, callee-owned	%eflags	Status/condition code bits

Registros x86_64

- En x86_64 muchos registros se pueden acceder de a partes usando otros nombres.

Por ejemplo:



La pila

Para el stack overflow nos interesa la pila y cómo manipular sus contenidos

En x86_64 hay 2 registros fundamentales para usar la pila:

- rsp: Stack Pointer. Apunta al tope de la pila
- rbp: (Stack) Base Pointer: Apunta a la base de la pila
- Un push:
  - Guarda un valor en la dirección apuntada por rsp
  - Decrementa rsp (8 bytes)
- Un pop:
  - Recupera el valor apuntado por rsp
  - Incrementa rsp (8 bytes)

Las funciones y la pila

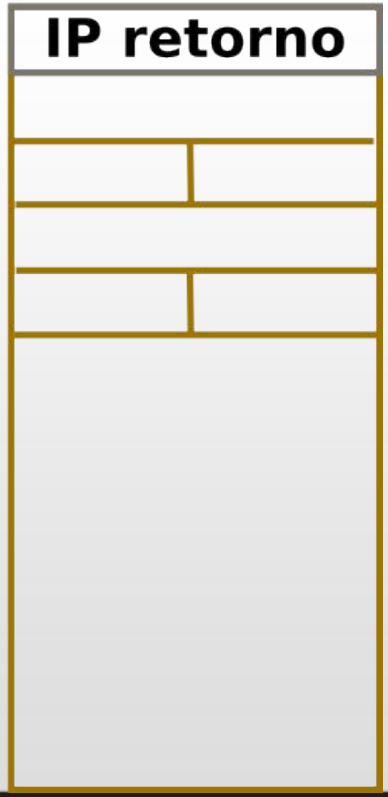
```
1   int funcion(int a, int b) {
2       int i = 15;
3       int j = 16;
4       // Code
5       return a + b + i + j;
6   }
7
8   int main() {
9       int x = 5;
10      int y = 10;
11      int result;
12  ->  result = funcion(x, y);
13  }
```

```
movl    %edx, %esi
movl    %eax, %edi
call    funcion
movl    %eax, -12(%rbp)
```

Las funciones y la pila

```
1   int funcion(int a, int b) {
2       int i = 15;
3       int j = 16;
4       // Code
5       return a + b + i + j;
6   }
7
8   int main() {
9       int x = 5;
10      int y = 10;
11      int result;
12      result = funcion(x, y);
13  }
```

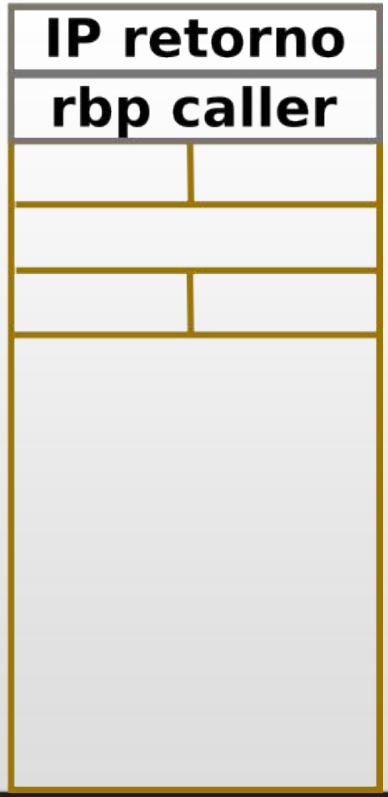
```
funcion:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -20(%rbp)
    movl    %esi, -24(%rbp)
    movl    $15, -4(%rbp)
    movl    $16, -8(%rbp)
    movl    -20(%rbp), %edx
    movl    -24(%rbp), %eax
    addl    %eax, %edx
    movl    -4(%rbp), %eax
    addl    %eax, %edx
    movl    -8(%rbp), %eax
    addl    %edx, %eax
    popq    %rbp
    ret
```



Las funciones y la pila

```
 1  int funcion(int a, int b) {
 2      int i = 15;
 3      int j = 16;
 4      // Code
 5      return a + b + i + j;
 6  }
 7
 8  int main() {
 9      int x = 5;
10      int y = 10;
11      int result;
12      result = funcion(x, y);
13  }
```

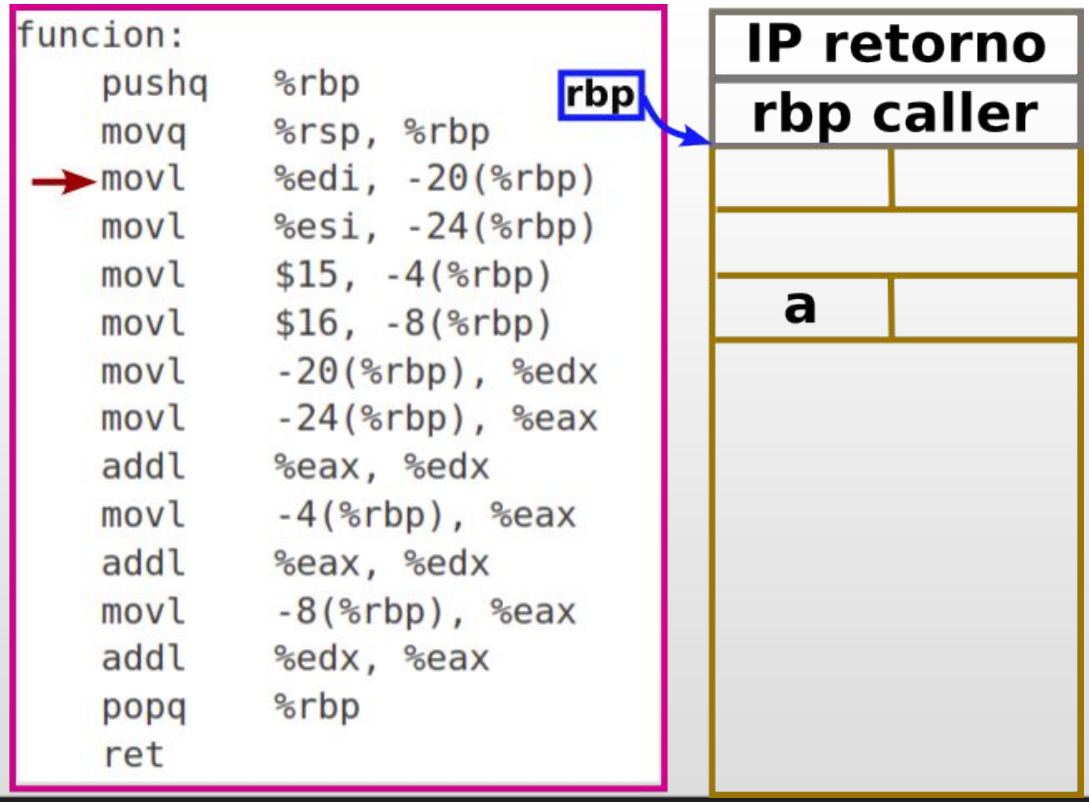
```
funcion:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -20(%rbp)
    movl    %esi, -24(%rbp)
    movl    $15, -4(%rbp)
    movl    $16, -8(%rbp)
    movl    -20(%rbp), %edx
    movl    -24(%rbp), %eax
    addl    %eax, %edx
    movl    -4(%rbp), %eax
    addl    %eax, %edx
    movl    -8(%rbp), %eax
    addl    %edx, %eax
    popq    %rbp
    ret
```



Las funciones y la pila

```
 1  int funcion(int a, int b) {
 2      int i = 15;
 3      int j = 16;
 4      // Code
 5      return a + b + i + j;
 6  }
 7
 8  int main() {
 9      int x = 5;
10      int y = 10;
11      int result;
12      result = funcion(x, y);
13  }
```

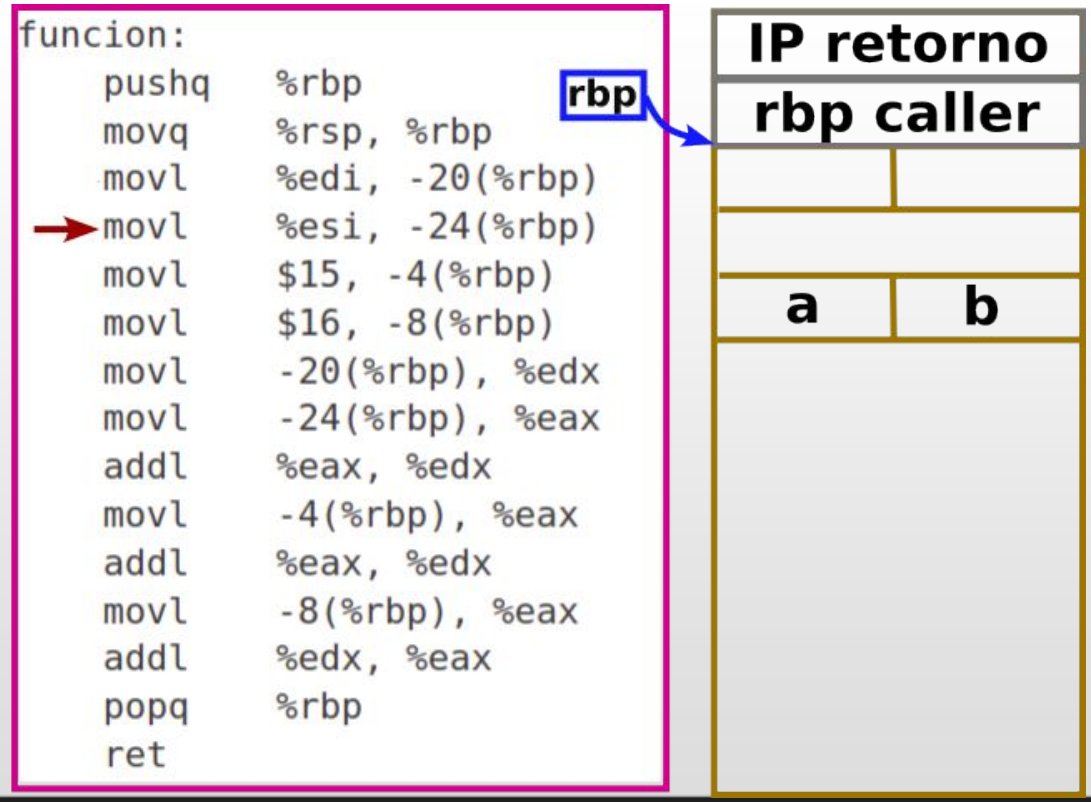
```
funcion:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -20(%rbp)
    movl    %esi, -24(%rbp)
    movl    $15, -4(%rbp)
    movl    $16, -8(%rbp)
    movl    -20(%rbp), %edx
    movl    -24(%rbp), %eax
    addl    %eax, %edx
    movl    -4(%rbp), %eax
    addl    %eax, %edx
    movl    -8(%rbp), %eax
    addl    %edx, %eax
    popq    %rbp
    ret
```



Las funciones y la pila

```
 1  int funcion(int a, int b) {
 2      int i = 15;
 3      int j = 16;
 4      // Code
 5      return a + b + i + j;
 6  }
 7
 8  int main() {
 9      int x = 5;
10      int y = 10;
11      int result;
12      result = funcion(x, y);
13  }
```

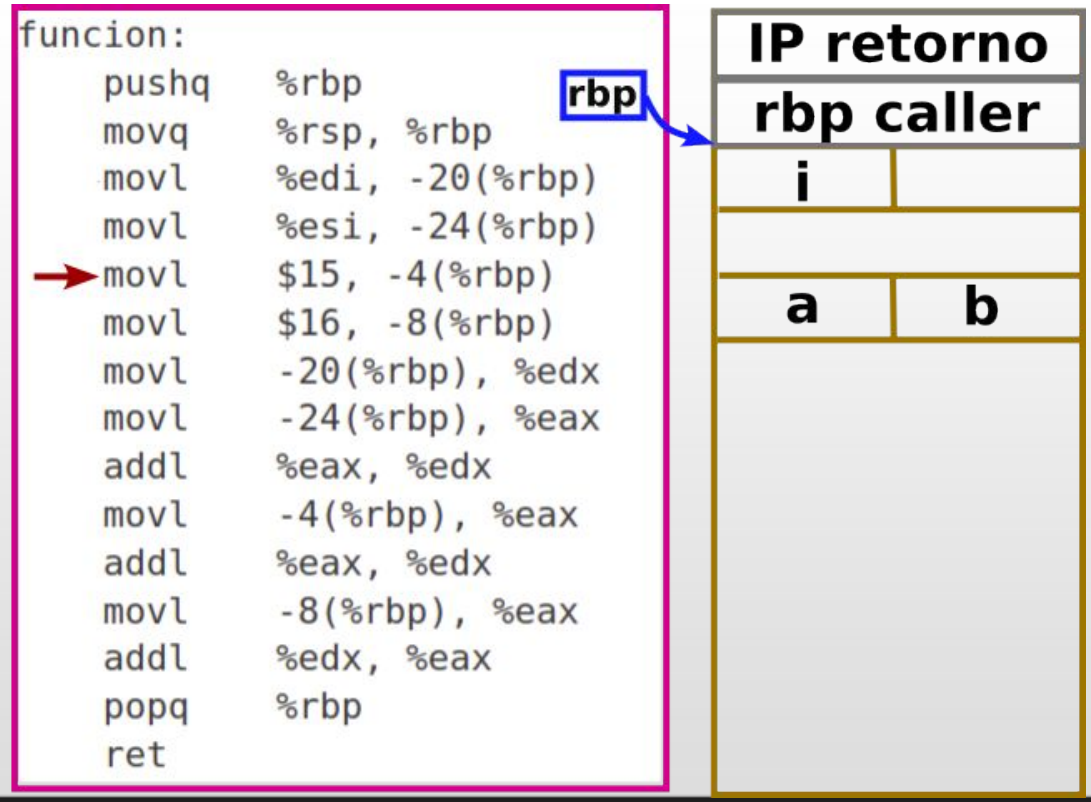
```
funcion:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -20(%rbp)
    movl    %esi, -24(%rbp)
    movl    $15, -4(%rbp)
    movl    $16, -8(%rbp)
    movl    -20(%rbp), %edx
    movl    -24(%rbp), %eax
    addl    %eax, %edx
    movl    -4(%rbp), %eax
    addl    %eax, %edx
    movl    -8(%rbp), %eax
    addl    %edx, %eax
    popq    %rbp
    ret
```



Las funciones y la pila

```
 1  int funcion(int a, int b) {
 2      int i = 15;
 3      int j = 16;
 4      // Code
 5      return a + b + i + j;
 6  }
 7
 8  int main() {
 9      int x = 5;
10      int y = 10;
11      int result;
12      result = funcion(x, y);
13  }
```

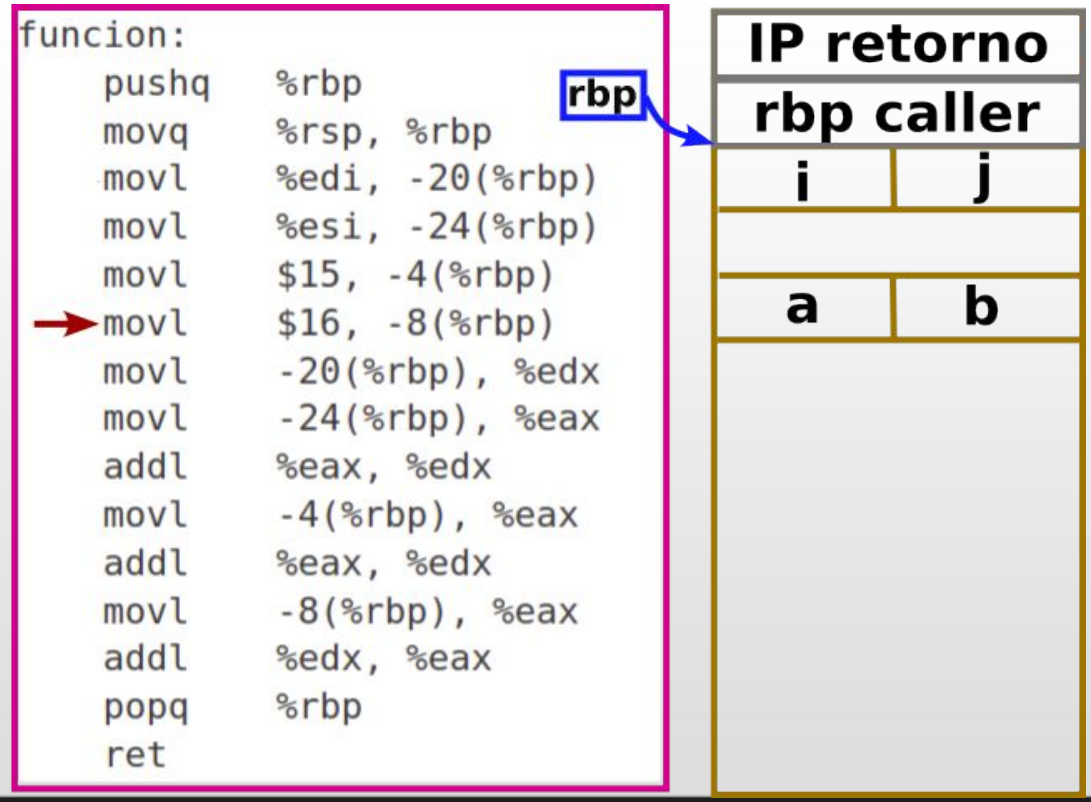
```
funcion:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -20(%rbp)
    movl    %esi, -24(%rbp)
    movl    $15, -4(%rbp)
    movl    $16, -8(%rbp)
    movl    -20(%rbp), %edx
    movl    -24(%rbp), %eax
    addl    %eax, %edx
    movl    -4(%rbp), %eax
    addl    %eax, %edx
    movl    -8(%rbp), %eax
    addl    %edx, %eax
    popq    %rbp
    ret
```



Las funciones y la pila

```
 1  int funcion(int a, int b) {
 2      int i = 15;
 3      int j = 16;
 4      // Code
 5      return a + b + i + j;
 6  }
 7
 8  int main() {
 9      int x = 5;
10      int y = 10;
11      int result;
12      result = funcion(x, y);
13  }
```

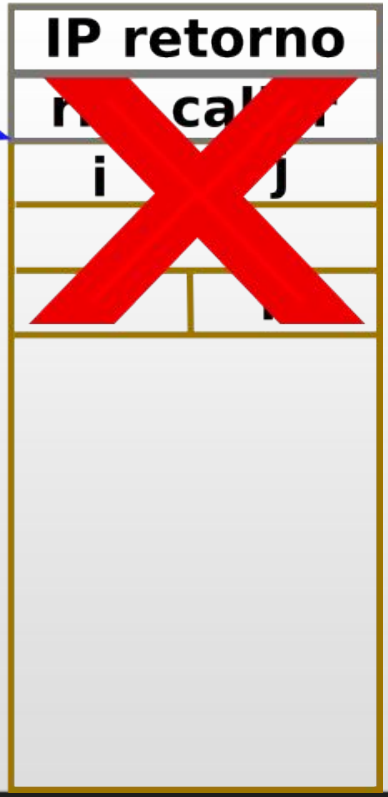
```
funcion:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -20(%rbp)
    movl    %esi, -24(%rbp)
    movl    $15, -4(%rbp)
    movl    $16, -8(%rbp)
    movl    -20(%rbp), %edx
    movl    -24(%rbp), %eax
    addl    %eax, %edx
    movl    -4(%rbp), %eax
    addl    %eax, %edx
    movl    -8(%rbp), %eax
    addl    %edx, %eax
    popq    %rbp
    ret
```



Las funciones y la pila

```
 1  int funcion(int a, int b) {
 2      int i = 15;
 3      int j = 16;
 4      // Code
 5      return a + b + i + j;
 6  }
 7
 8  int main() {
 9      int x = 5;
10      int y = 10;
11      int result;
12      result = funcion(x, y);
13  }
```

```
funcion:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -20(%rbp)
    movl    %esi, -24(%rbp)
    movl    $15, -4(%rbp)
    movl    $16, -8(%rbp)
    movl    -20(%rbp), %edx
    movl    -24(%rbp), %eax
    addl    %eax, %edx
    movl    -4(%rbp), %eax
    addl    %eax, %edx
    movl    -8(%rbp), %eax
    addl    %edx, %eax
    popq    %rbp
    ret
```



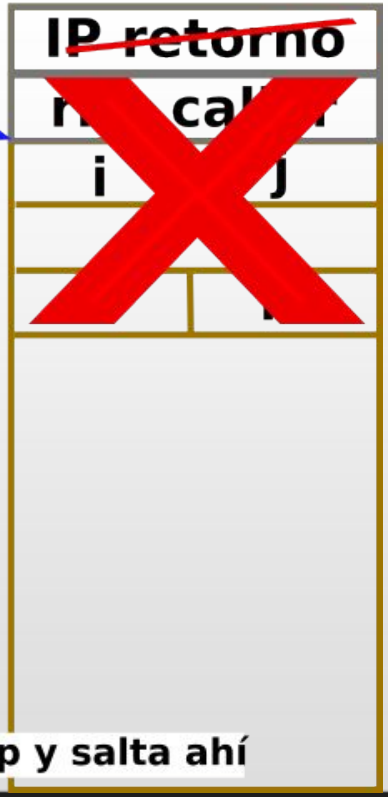
Las funciones y la pila

```
 1  int funcion(int a, int b) {
 2      int i = 15;
 3      int j = 16;
 4      // Code
 5      return a + b + i + j;
 6  }
 7
 8  int main() {
 9      int x = 5;
10      int y = 10;
11      int result;
12      result = funcion(x, y);
13  }
```

```
funcion:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -20(%rbp)
    movl    %esi, -24(%rbp)
    movl    $15, -4(%rbp)
    movl    $16, -8(%rbp)
    movl    -20(%rbp), %edx
    movl    -24(%rbp), %eax
    addl    %eax, %edx
    movl    -4(%rbp), %eax
    addl    %eax, %edx
    movl    -8(%rbp), %eax
    addl    %edx, %eax
    popq    %rbp
    ret
```

rbp

saca de la pila al viejo rip y salta ahí



Como lograr el overflow

- Funciones inseguras que leen un string sin verificar límites:
  - gets()
  - scanf()
  - strcpy()
- Funciones seguras pero mal usadas:
  - fgets()
  - fread()
  - memcpy() - strncpy()
- Leemos de teclado hasta llenar el string y luego seguimos leyendo hasta pisar la posición de la pila que nos interesa.
- Podemos pisar otra variable.
- O pisar la dirección de retorno con otro puntero. Func. interna o de LibC

Demo simple + ret fallido

Address Space Layout Randomization

ASLR

- Dificulta los ataques de stack buffer overflow
- En cada ejecución del proceso cambian las direcciones del stack, heap, data, text y bibliotecas.
- Si conseguimos un puntero útil en una ejecución en la siguiente ya no nos sirve.
- No es infalible pero ayuda.
- cat /proc/sys/kernel/randomize_va_space
  - 0. Deshabilitado
  - 1. Randomizar stack, virtual dynamic shared object page, memoria compartida. Data se ubica al final de text.
  - 2. Randomizar stack, VDSO page, memoria compartida y data.

https://docs.oracle.com/en/operating-systems/oracle-linux/6/security/ol_aslr_sec.html

ASLR

- ¿Por qué alguien la deshabilitaría?
- Experimentación
- Increíblemente hay software que necesita que esté apagada (es muy raro):
  - https://www.ibm.com/docs/en/storage-protect/8.1.24?topic=systems-suggested-settings
  - The kernel.randomize_va_space parameter configures the use of memory ASLR for the kernel. Disable ASLR because it can cause errors for the Db2 software. To learn more details about the Linux® ASLR and Db2, see the technote at: technote 384757.

Demo ret con debugger

SystemD

SystemD

- SystemD puede restringir los privilegios de un servicio.
- Utiliza una combinación de técnicas
  - CGroups
  - Namespaces
  - Ulimit
  - Capabilities
  - System call filtering
  - Ejecución en un CHROOT
  - Proteger /proc /tmp /home
  - Filtrar IPs
  - DynamicUser

https://wiki.debian.org/systemd/Services
https://www.freedesktop.org/software/systemd/man/latest/systemd.exec.html
https://linux-audit.com/systemd/settings/units/

AppArmor (la próxima clase)
Explicación + Consulta

Fuentes

- Operating System Concepts 10º ed. - Silberschatz
- https://docs.redhat.com/en/documentation/red_hat_enterprise_linux_atomic_host/7/html/container_security_guide/linux_capabilities_and_seccomp
- https://www.redhat.com/en/blog/systemd-secure-services
- https://www.freedesktop.org/software/systemd/man/latest/systemd.exec.html
- https://web.stanford.edu/class/archive/cs/cs107/cs107.1206/guide/x86-64.html
- https://medium.com/@pierre.ansar/the-accumulator-register-in-x86-64-assembly-understanding-rax-eax-ax-ah-and-al-0deb18032778

¿Preguntas?