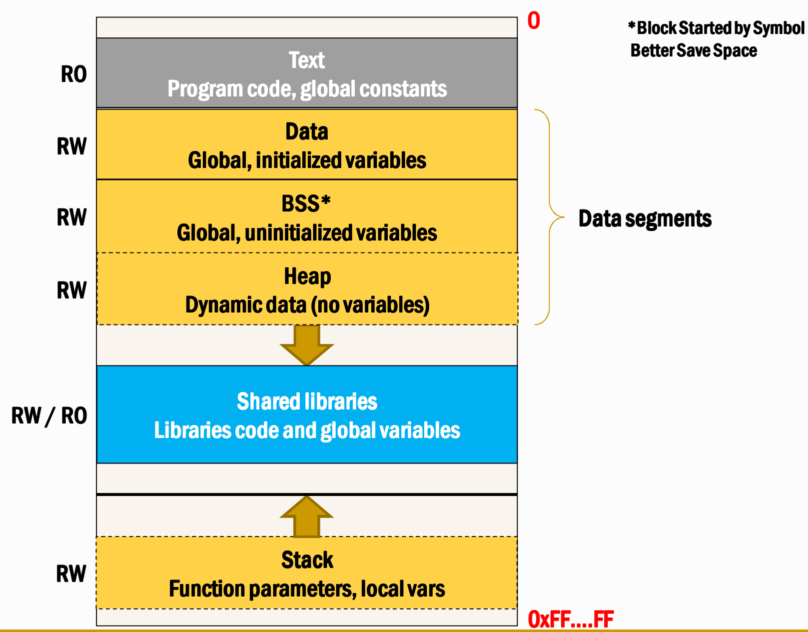
Buffer Overflows

Memory organization topics

- ▷ Kernel organizes memory in pages
  - Typically 4 kB
- ▷ Processes operate in a virtual memory space
  - Mapped to real 4k pages
  - Could live in RAM, be file-mapped or be swapped out
- ▷ Kernel groups pages in several segments
  - Increases security
    - Segment-based permissions (RO, RW)
  - Increases performance
    - Some are dynamic: discarded when program terminates
    - Some are static: can be retained, speeding up reuses



mem.c

```c
//CONST
const char cntvar[]="constant";

//BSS
static char bssvar[4];

int main(int argc, void** argv)
{
      void * dynmem = malloc(1);
…
}
```

```
&main  = 0804865c -> text  = 08048000
cntvar = 08048920 -> const = 08048000
bssvar = 0804a034 -> bss   = 0804a000
&argc  = bfeb8590 -> stack = bfeb8000
dynmem = 08435008 -> heap  = 08435000
```

mem.c

```
Content of /proc/self/maps
08048000-08049000 r-xp 00000000 08:01 26845750   /home/s/seguranca/mem
08049000-0804a000 r--p 00000000 08:01 26845750   /home/s/seguranca/mem
0804a000-0804b000 rw-p 00001000 08:01 26845750   /home/s/mem
08435000-08456000 rw-p 00000000 00:00 0          [heap]
b7616000-b7617000 rw-p 00000000 00:00 0
b7617000-b776a000 r-xp 00000000 08:01 1574823    /lib/tls/i686/cmov/libc-2.11.1.so
b776a000-b776b000 ---p 00153000 08:01 1574823    /lib/tls/i686/cmov/libc-2.11.1.so
b776b000-b776d000 r--p 00153000 08:01 1574823    /lib/tls/i686/cmov/libc-2.11.1.so
b776d000-b776e000 rw-p 00155000 08:01 1574823    /lib/tls/i686/cmov/libc-2.11.1.so
b776e000-b7771000 rw-p 00000000 00:00 0
b777e000-b7782000 rw-p 00000000 00:00 0
b7782000-b7783000 r-xp 00000000 00:00 0          [vdso]
b7783000-b779e000 r-xp 00000000 08:01 1565567    /lib/ld-2.11.1.so
b779e000-b779f000 r--p 0001a000 08:01 1565567    /lib/ld-2.11.1.so
b779f000-b77a0000 rw-p 0001b000 08:01 1565567    /lib/ld-2.11.1.so
bfe99000-bfeba000 rw-p 00000000 00:00 0          [stack]
```


mem.c

```
Stack evolution:
foo [000]: &argc  = bfeb8140 -> stack = bfeb8000
foo [001]: &argc  = bfdb8110 -> stack = bfdb8000
foo [002]: &argc  = bfcb80e0 -> stack = bfcb8000
foo [003]: &argc  = bfbb80b0 -> stack = bfbb8000
foo [004]: &argc  = bfab8080 -> stack = bfab8000
foo [005]: &argc  = bf9b8050 -> stack = bf9b8000
foo [006]: &argc  = bf8b8020 -> stack = bf8b8000
foo [007]: &argc  = bf7b7ff0 -> stack = bf7b7000
foo [008]: &argc  = bf6b7fc0 -> stack = bf6b7000
Segmentation fault
```

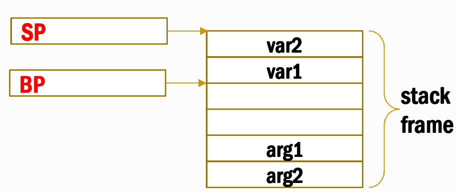
Some x86 CPU registers

- General Purpose: A, B, C, D
  - A: 8bits, AX: 16bits, **EAX: 32bits**, RAX: 64bits
- BP: Base Pointer (EBP if w/ 32 bits)
  - Base address of the current function stack frame
  - A function stack frame is where we have
    - The function parameters
    - The local function variables
- SP: Stack Pointer (ESP if w/ 32 bits)
  - Points to end of stack (last value pushed)
- IP: Instruction Pointer (EIP if w/ 32 bits)
  - Points to current instruction

Stack segment

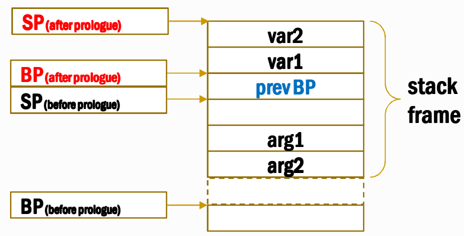
```
function ( int arg1, int arg2 )
{
    int var1 = arg1;
    int var2;
}
```

- Stack is used to
  - Pass parameters to functions (eg. arg1)
  - Store local variables (eg. var1)
- Values are PUSHed or POPed from stack
  - eg: `push eax`, `pop eax`
- Allocation of local variables in space
  - `int var1;` → `sub esp,4`
- Accessing variables in the stack
  - A parameter:
    - arg1 → `ebp + 8`
    - arg2 → `ebp + 12`
  - A local variable:
    - var1 → `ebp - 4`
    - var2 → `ebp - 8`



Initialization of a stack frame

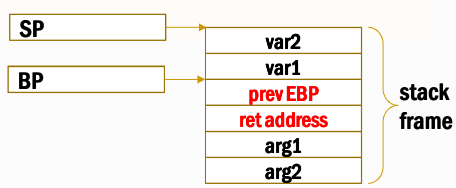
- This is done in the prologue of a function
  - And is undone at its epilogue
- The prologue consists in:
  - Saving the base address of the previous stack frame and setting the new one
    - `push ebp`
    - `mov ebp, esp`
  - Allocate space for local variables
    - `sub esp, Imm`



- The epilogue is
  - `mov esp, ebp`
  - `pop ebp`

  leave

Function call and return

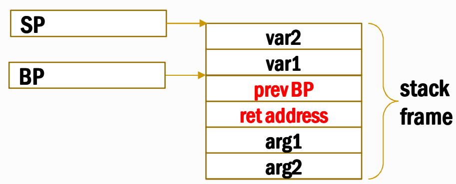


- Call steps
  - Put arguments in stack
    - Usually with PUSH
  - Call the function address
    - Pushes the IP to the stack (return address)
    - IP has the next instruction address
  - Release stack space
    - Usually increasing ESP
- Returning from a function
  - The RET instruction pops the saved IP (return address)

Function foo()

```
void foo ( int arg1, int arg2 )
{
    int var1 = arg1;
    int var2;
}
```

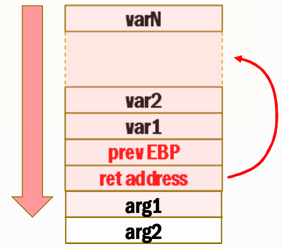
```
foo:
    push ebp
    mov ebp, esp
    sub esp, 8
    mov eax, DWORD PTR [ebp+8]
    mov DWORD PTR [ebp-4], eax
    leave
    ret
```



Buffer overflow

- Write beyond the boundaries of a buffer
- Consequences
  - Write over other values located next to the buffer
  - Write over special values co-located (saved registers)
    - Saved BP
      - Damages the base address of the previous stack frame
    - Saved IP (return address)
      - Jump to any address on return!

Stack smashing attack



- Roadmap
  - Overflow a local variable
  - Extend the overflow to the return address
  - Change the return address in order to jump to the injected data
    - Which should be executable code
  - Wait for the return of the function
- Difficulty
  - A return using a saved address is an absolute jump
  - The attacker needs to know the absolute address of the vulnerable variable
    - Given the source code, knowing the machine and the initial stack address, this is feasible

bo.c

```
int foo()
{
  char a[4];
  scanf("%s", a);
}
```

```
.LC0:
  .string "%s"
  .text

foo:
    push  ebp
    mov ebp, esp
    sub esp, 40
    mov eax, OFFSET FLAT:.LC0
    lea edx, [ebp-12]
    mov DWORD PTR [esp+4], edx
    mov DWORD PTR [esp], eax
    call  __isoc99_scanf
    leave
    ret
```

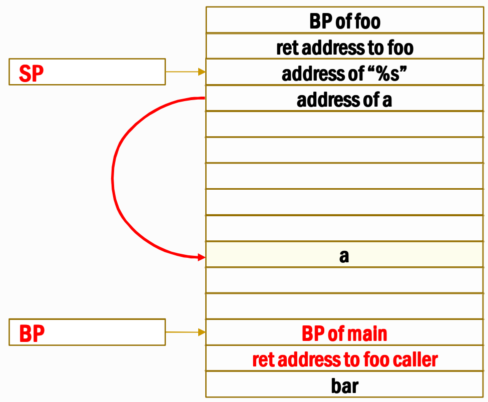
Pre-allocation of space for function call parameters in advance (and excess)

Allows function calls without pushing/poping values to/from the stack

bo.s

```
.LC0:
  .string "%s"
  .text

foo:
push  ebp
mov ebp, esp
sub esp, 40
mov eax, OFFSET FLAT:.LC0
lea edx, [ebp-12]
mov DWORD PTR [esp+4], edx
mov DWORD PTR [esp], eax
call  __isoc99_scanf
leave
ret
```



Buffer overflow

```
[jpbarraca@atnog: seguranca]$ ./bo
a
[jpbarraca@atnog: seguranca]$ ./bo
aa
[jpbarraca@atnog: seguranca]$ ./bo
aaaaaaaaaaa
[jpbarraca@atnog: seguranca]$ ./bo
aaaaaaaaaaaa
Segmentation fault
```

Write inside a

Write inside a

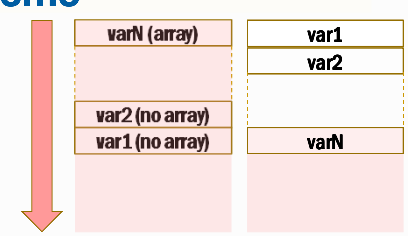
Write outside a

Write over stored BP

Mitigation: Prevention mechanisms

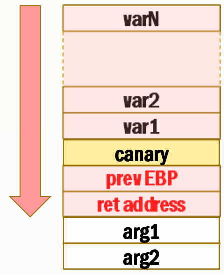
- ▷ Avoid execution of injected instructions
  - In segments/pages that usually have no code
  - Prevents the execution of code injected as data
- ▷ Randomize the address space
  - ADLR (Address Space Layout Randomization)
  - Segments do not start in fixed positions on each run of the same application
    - But segments keep their relative position
  - Prevents jumps to well-known code locations

Mitigation: Prevention mechanisms



- ▷ Variable reordering
  - Usually the vulnerable variables are arrays
  - To protect other kinds of local variables (in the same stack frame), arrays are moved closer to the saved registers
  - This reduces the set of variables that may be affected by a buffer overrun

Mitigation:
Detection mechanisms



▷ Stack canaries

- A value unknown to attackers (canary) is stored next to saved registers
  - Saved BP and return address
- Stack smashing attacks usually cannot affect saved registers with running over a canary
  - Because they are usually based on string overruns
- The canary is checked before the function's epilogue
  - If different from the original value, an exception is raised