Breaking The STACK
got (%ebp + 4)

Marco Ramilli

marco.ramilli@unibo.it

http://marcoramilli.blogspot.com

Background

- **Program** becomes **process** when it's loaded into memory being executed.
- **PID** (Process Identifier)
- **COFF** (Common Object File Format)
  - **Header.** Wraps up .text, .data, .stack and .bss. The OS loader uses the header to charge the program into memory.
  - **Payload.** Where is located the code.
- Examples of COFF:
  - **ELF** (Executable Linking File)
  - **PE** (Portable Executable)
  - **...**

Background: **The COFF Header**

- **Text AREA (.text):** the read only area which includes the program's code. If someone tries to write into .text the process terminates in "segmentation fault" exception.

- **Data AREA (.data):** includes static variables. brk(2) syscall modifies such an area.

- **Stack AREA (.stack):** includes local variables of functions, return values and parameters.

Background: Registers, Stack

- EAX, EBX, ECX, EDX, ESI, EDI.
- EBP (Base Pointer), ESP (Stack Pointer)

(General Reg)32bits

- CS (Code Segment), DS (Data Segment), EX (Extra Segment), SS (Stack Segment), …

(Segment Reg)16bits

- EIP (Instruction Pointer), CR (Control Register)

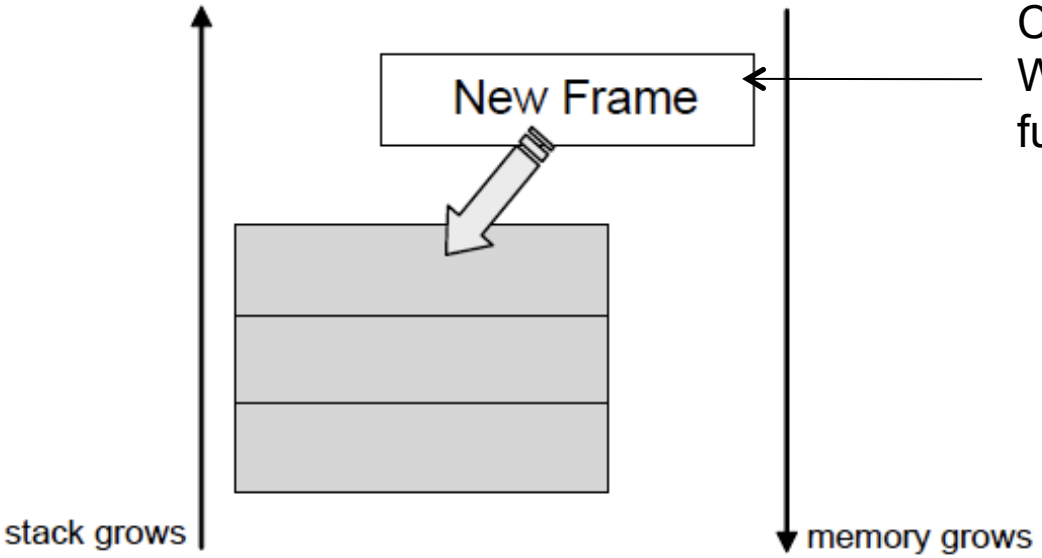
(Control Reg)32bits

- EFLAGS (Flags To Control)

(NC Reg)32bits

Stack Layout (1/3)

- For each Process a Stack, for each function call a contest.
- LIFO (Last In First Out)
- Push and Pop
- Intel architecture**: the stack grows down** !



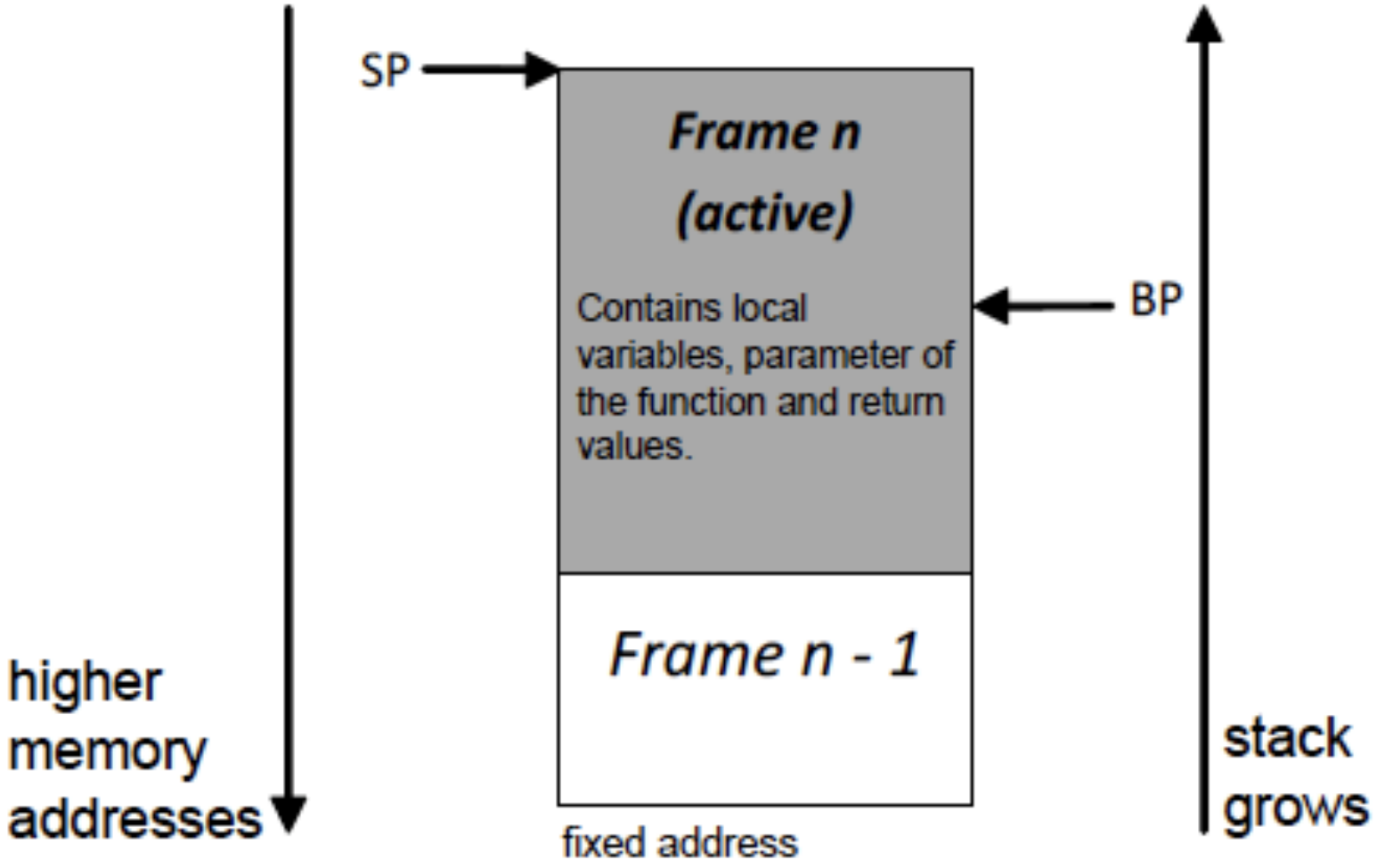
Contiguous are in the memory
Which logically belongs to a program function

Stack Layout (2/3)

- Each Frame is identified from:
  - SP. It points out to the top of the stack (lower address)
  - BP. It point to the base of the stack (higher address)
  - BP in some architectures is called FP (Frame Pointer)

- BP (or FP) is always fixed on the bottom of the stack. For such a reason is a convention to refer to each variable through BP + Offset.
  - NOTE: NOT possible through SP + Offset

Stack Layout (3/3)



Frame Change = Context Change

- From frame n-1 to frame n: **Prologue**
- From frame n to frame n-1: **Epilogue**

Prologue

- Push Parameters
- Save (call) EIP
- Save (push) EBP
- Line up EBP-ESP

Epilogue

- Line up EBP-ESP
- Restore back EBP
- Set saved EIP (ret)

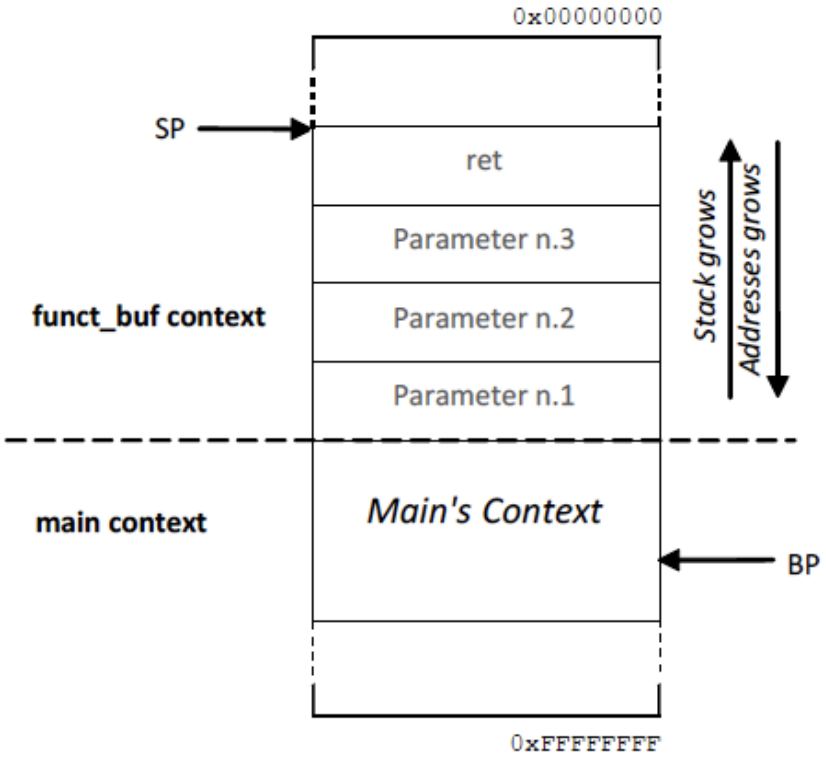
Function Call (Intel Calling Convention)

- Lets analyze the following example.

```
1 void funct_buf(int a, int b, int c){
2       char buffer1[5];
3       char buffer2[10];
4 }
5 void main() {
6       funct_buf(1,2,3);
7 }
```

Function Call (Intel Calling Convention)

- Two Stack contests have been created: Step1



CODE LEVEL

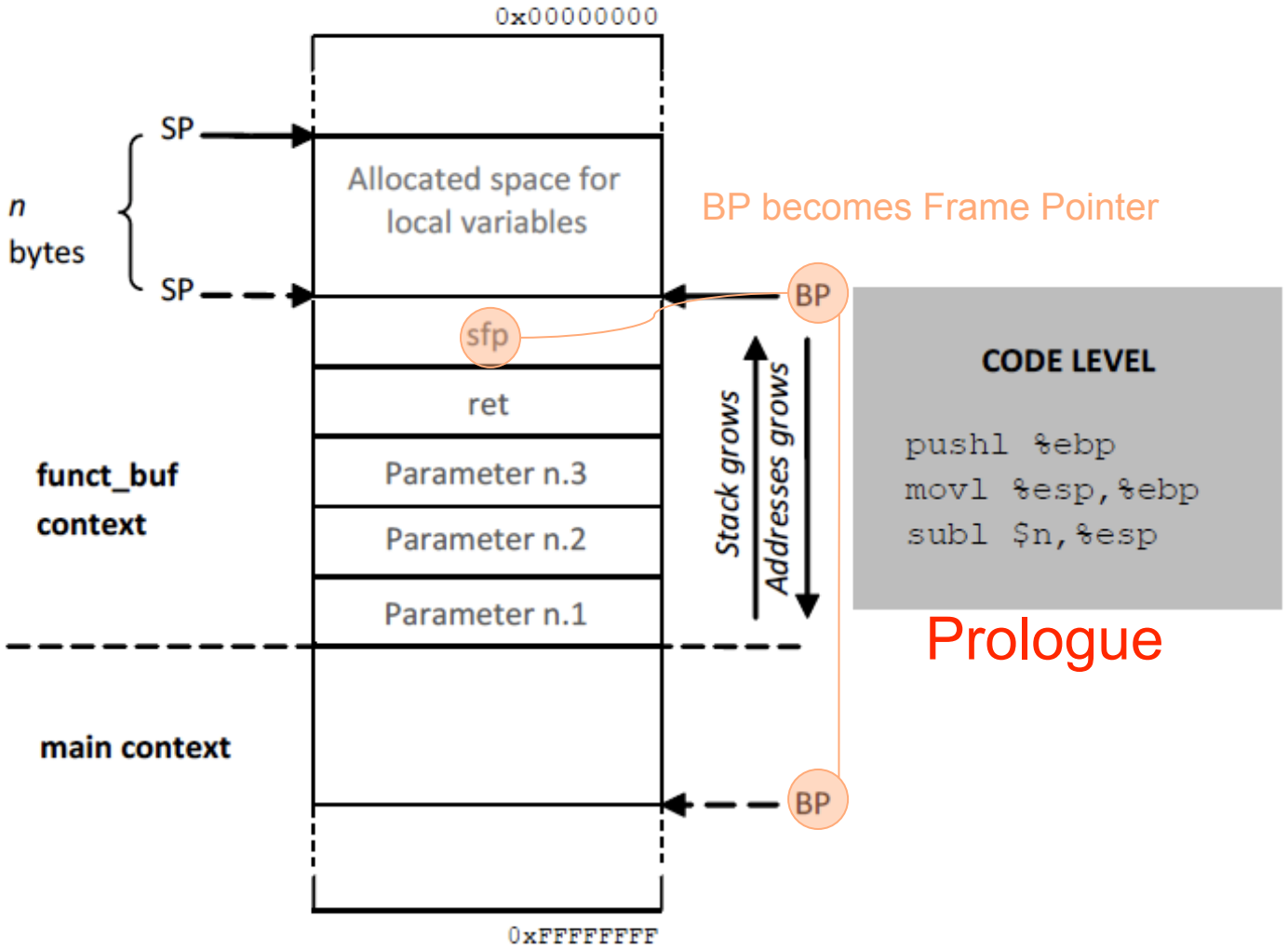
```
pushl $3
pushl $2
pushl $1
call func_buf
```

Still no context change

Step n.1: the parameters and the ret are pushed into the stack. The BP at this moment, points yet to the main's ebp, while the SP point at the top of the stack

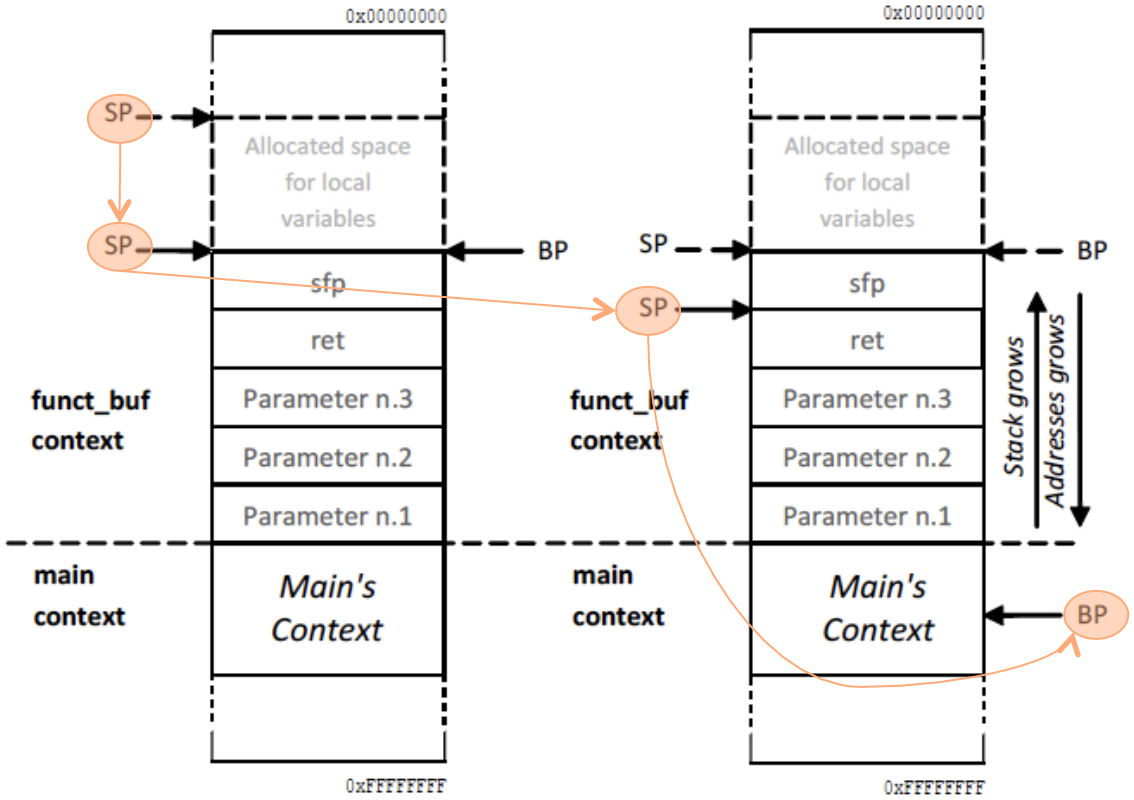
Function Call (Intel Calling Convention)

- Context Change: Step2



Function Call (Intel Calling Convention)

- Back to the main(): Step3



Epilogue

CODE LEVEL

```
mov %ebp,%esp
pop %ebp
ret
```

Step n.1: SP is moved at BP level, then sfp is popped to restore the calling function one, and a ret instruction is done to give the control to the calling function.

Buffer Overflow –BOF-

- **Stack Based BOF**
  - Overflowing the variable space. Overwriting the RET address.

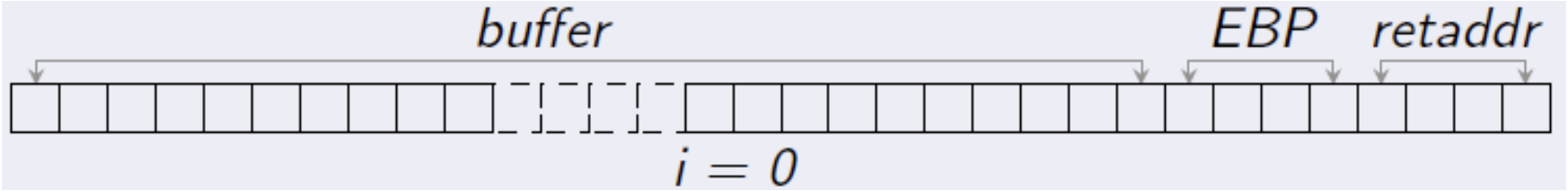
got (%ebp + 4)

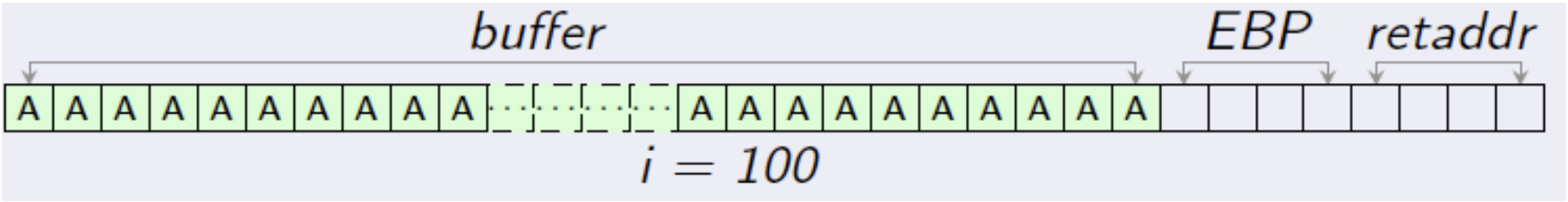
- Heap Based BOF
  - Overflowing the dynamic data space allocated for program data. Overwriting dynamic memory allocation linkage (such as “malloc” meta data) and uses the resulting pointer exchange to overwrite a program function pointer.

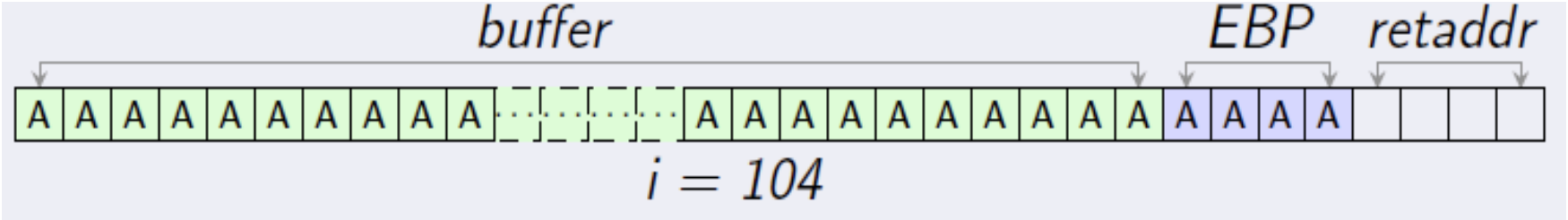
Stack Based BOF (1/2)

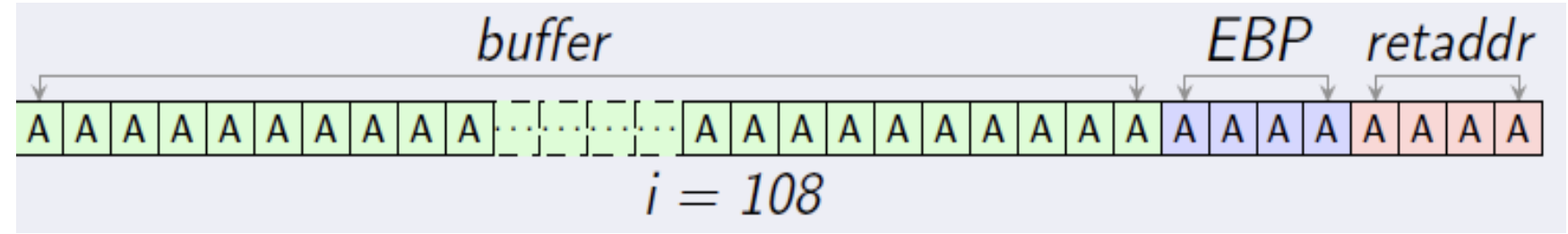
```
1  /* vuln.c */
2
3  void foobar(char *str)
4  {
5    char buffer[100];
6    strcpy(buffer, str);
7  }
8
9  int main(int argc, char **argv)
10 {
11   if(!argv[1]) return -1;
12   foobar(argv[1]);
13   exit(0);
14 }
```

Stack Based BOF (2/2)





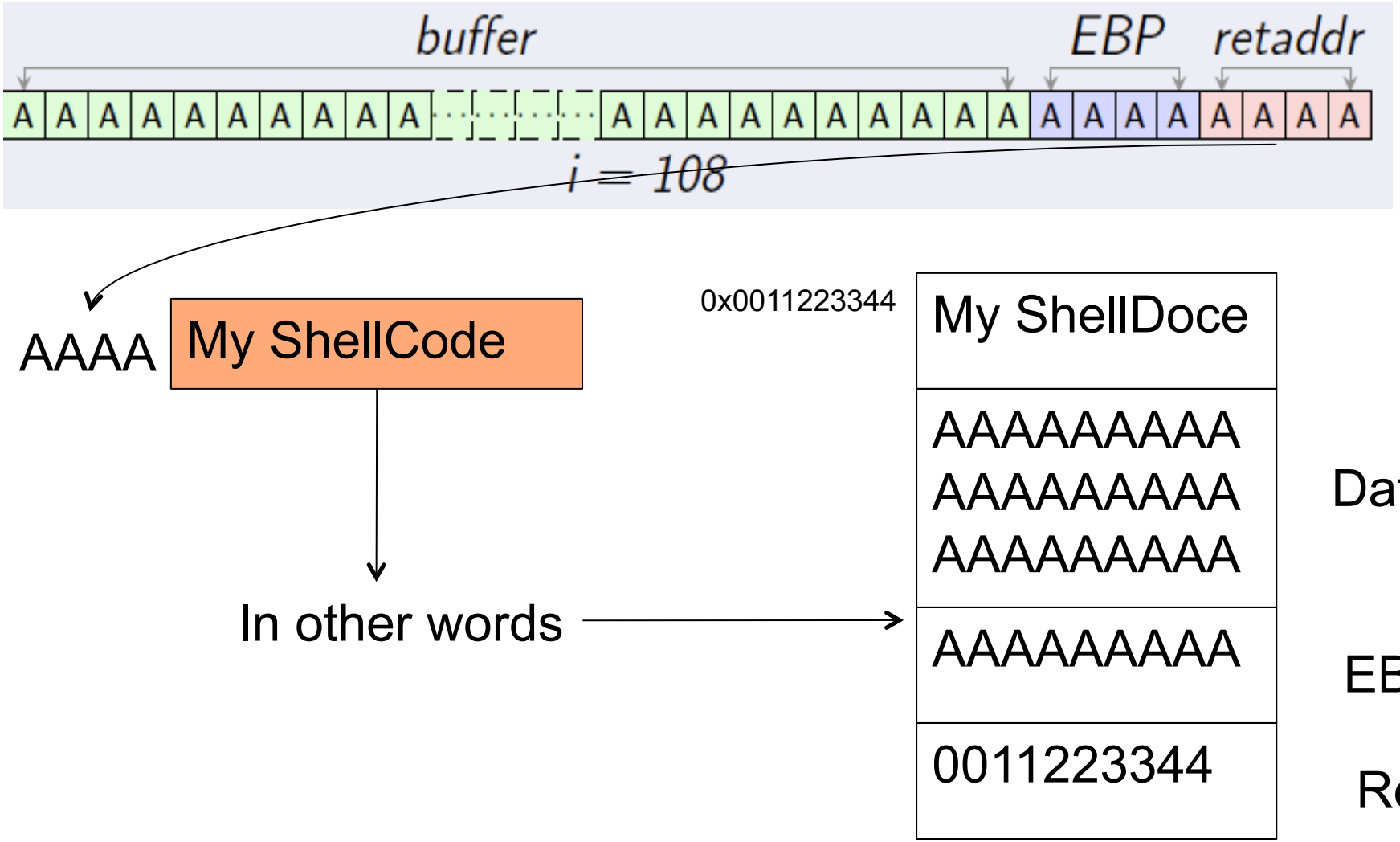




BOF Examples

- Example 1:
  - Buffer overflow Example
- Example 2:
  - Buffer overflow Control Forgin’ Example

From BOF To Own The Machine



BOF Protections

- Safe Functions:
  - For Example: strcpy VS. strncpy
- Address Space Layout Randomization (ASLR).
- Stack Execution Invalidation (NX bit).
- StackShield (compiler and linker protection).
- StackGuard.
- Stack Smashing Protector – Propolice (SSP).
- Buffer Security Check (GS Windows).
- SafeSEH (Windows)
- Data Execution Prevention (DEP Windows)

ASLR

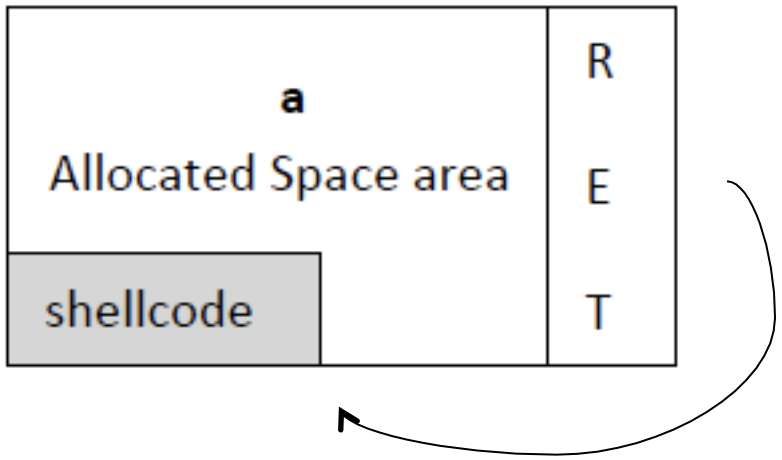
- Installed on Linux (2.6.12), MAC OS 10.4 and Windows XP sp2.
- The programs related memory address are not known a priori from attackers. Memory Space change dynamically.
- Increase **security** by increasing the **searching space**.

```
>./test_ebp
Current ebp: 0x00420058
> ./test_ebp
Current ebp: 0x00520b5a
> ./test_ebp
Current ebp: 0x0125a61f
```

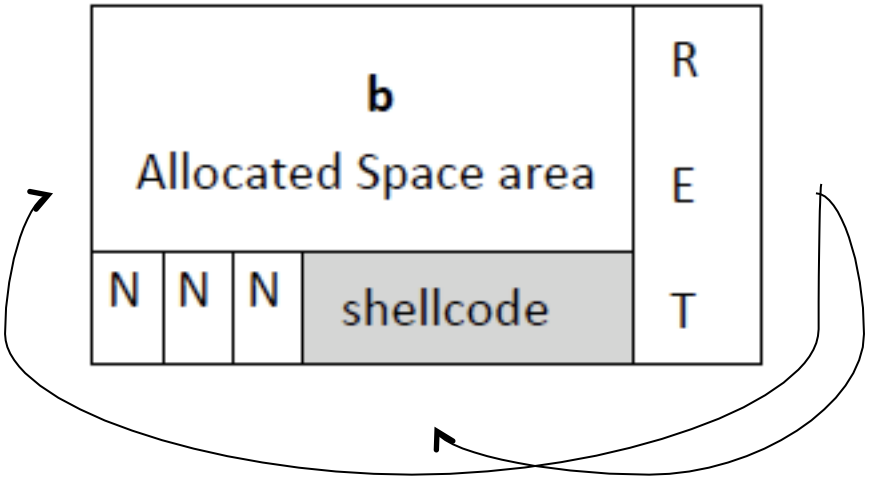
Bypassing ASLR (1/2)

- Decreasing the searching space = increasing the probability to find out **Ret**. The attacker can implement this technique by filling up memory with known content that does actually nothing: **NOP** (no operation).

Case a. 1 Shot to find “shellcode”



Case b. 4 Shots to find “shellcode”



Bypassing ASLR (1/2)

- Global variables cannot be “address randomized”. (how can you find them ?)
- Global variables (for C language or Static for Java/C# language) are stored into Block Started by Symbol (**.BSS**).
- The attacker can store his shellcode into **.BSS**, which is statically linked.
- Very easy to implement but we need 2 input variables. (.BSS and .Stack are in two different segments):
  - We need an input which is into the Stack and an input which is into the BSS.

NX bit

- It is a processor feature that can be enabled through HIGHMEM64 option (32 bit processors).
- Basically it marks some memory area non executable: for instance the stack is not executable.
- This is the last bit of the address: if 0 execution allowed, if 1 execution forbidden.
  - ExecShield and PaX are implementations that use NX processor bit into Linux environment.

Bypassing NX bit: ret2libc attack

- *libc* standard C library.
- *printf()*, *exit()*, ***system()*** are some of the basic *libc* functions.

```
#include <stdlib.h>

int system(const char *cmd);
```

This function runs the command or program specified by *cmd.*
If cmd is a null pointer, system returns non-zero only if a shell is available.

- The attacker can force to return libc.system() executing the desired program out of stack.
- We have a couple of problems here: *esp lifting* and *frame faking*

StackShield

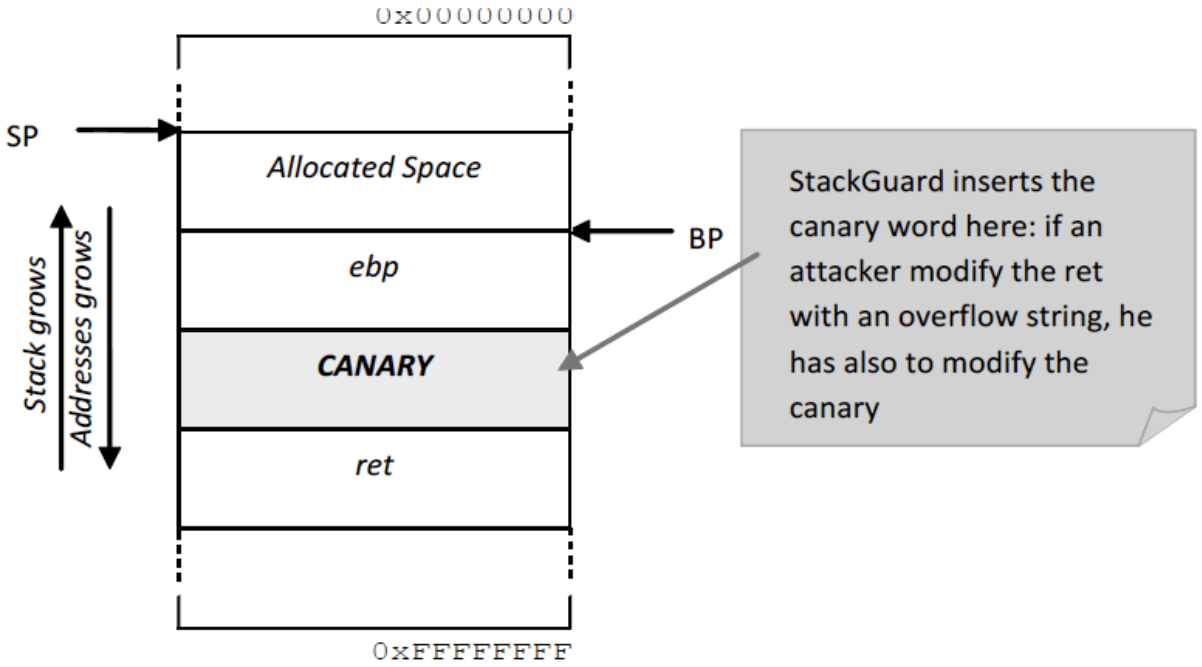
- This is a compiler module. It adds a “saving functionality”.
- When the **prologue** is completed it saves the **Ret** address into a controlled area (for instance into the data segment).
- After the called function ended (**epilogue**), StackShield controls if the current **Ret** value is equals to the stored one. In case of mismatch (current **Ret** value changed), the function is forced to end immediately.

Bypassing StackShield

- What happen if the attacker forces and error ?
- Exception handler knows how to do.
- Exception Handler Attack (EHA):
  - Exception Handler transfer the control to an “exception procedure” to manage the error.
  - This attack modifies the pointer to the “exception procedure” (or exception function).
  - From “exception procedure table” to “shellcode”

StackGuard (1/2)

- GCC module by Crispin Cowan. From 1998 has been suggested to be installed by default on all Linux platforms. Nowadays is not.
- It modifies the calling convention by adding the "**CANARY**" word before **Ret**.



StackGuard (2/2)

- 3 different types of CANARY word:
  - NULL(0x00), RC(0x0d), LF(0x0a) and EOF(0xff)
    - Are string terminate characters. (How to inject them ?)
  - RANDOM CANARY
    - The attacker cannot forge an attack vector with the perfect CANARY in it
  - Random XOR
    - Random CANARY XOR Control Data. In this way the attacker must know both: Random and Control data.

Stack Smashing Protector (SSP)

- This is the evolution of the StackGuard concept.
- CANARY words are generated from the compiler, using the same StackGuard techniques.
- Protects all the stack values (not only the **Ret**).
- Puts pointers for each buffer, making unbounded buffers bonded .
- Copies the function arguments.
- Default into gcc 4.x or above.
- To use it:
  - *-fstack-protector* (protecting strings) *–fstack-protector-all* (to protect all arguments and variables)
- To disable it:
  - *-fn0-stack-protector*

Exploiting

- Exploiting Example

Thanks

http://marcoramilli.blogspot.com

http://cesena.ing2.unibo.it

http://www.livestream.com/cesena_security