Offensive technologies
My First Buffer Overflow: Tutorial

César Bernardini

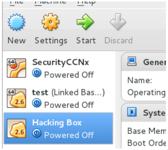
University of Trento
cesar.bernardini@unitn.it

September 25, 2015

Requirement

The Playground

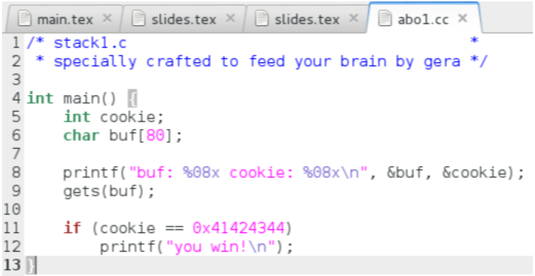
- VirtualBox or QEMU Virtual Machine
- Gentoo Linux
- No security protections; no network support
- Installed software: gcc (g++), gdb, nano, vi, python, perl
- Available in your lab!



Motivation

Insecure Programming

http://community.coresecurity.com/~gera/InsecureProgramming/



```c
/* stack1.c                      *
 * specially crafted to feed your brain by gera */

int main() {
    int cookie;
    char buf[80];

    printf("buf: %08x cookie: %08x\n", &buf, &cookie);
    gets(buf);

    if (cookie == 0x41424344)
        printf("you win!\n");
}
```

Aim

(**Hack the program**) to print **you win**!

Preliminaries

What is Hacking?

- Hacker is a term for both those who write code and those who exploit it.
- Hacking is really just the act of finding a clever and counterintuitive solution to a problem

If we want to find counterintuitive solutions...

We need to understand how technologies work **in-depth**

Preliminaries: Secure Programming

Regular programming vs Security-Flaw Exploitation

Regular Programming

- Multi-platform target
- Follow client specification (needs) – leads to many problems

Security-Flaw Exploitation

- Look for implementation-errors
- Fully-understand the environment
- Single-platform target

Programming

Big Picture: Revision on the required technologies

Requirements

- Basic knowledge on C
- Basic knowledge on gcc, gdb
- Basic Knowledge on Assembly language
- Basic Knowledge on Linux OS

Aim of today...

We revisit the basis of everyone of these technologies

The C Language

The C language

- Imperative, procedural programming language
- Developed by Dennies Ritchie between 1969 and 1973
- ISO 9899:1999

```
#include <libs>

int main(void)
{
    printf(”Hello␣World\n”);
    return 1;
}
```

The x86 Processor

8086 CPU

- First x86 Processor
- Manufactured by Intel
- Relative of 386 & i86
- Composed of many multi-purpose registers

Modern Processors

- Similar ideas, higher complexity
- i.e. AMD64, x86_64 – *uname -r*

The x86 Processor

Registers

- EAX, ECX, EDX, EBX are general purpose registers (Accumulator, Counter, Data and Base registers – temporary variables for the CPU)
- ESP, EBP, ESI, EDI are used for pointers and indexes
  - Stack Pointer and Base Pointer (delimiters (start and end) of the stack); Source Index; Destination Index
- EIP – La Vedette – is the *Instruction Pointe* register
  - Next instruction to be executed by the processor
- EFLAGS registers consists of several bit flags and are used for comparison and memory segmentations

Basic instruction

- < *operation* >< *destination* >< *source* >

Examples

- mov ebp, esp – move esp's content into ebp's content
- sub esp, 0x8 – subscract 8 to esp's content

`http://ref.x86asm.net/`

Compiler

The GNU Compiler Collection (GCC)

- GCC is a compiler system produced by the GNU Project supporting various programming languages
- GCC is a key component of the GNU toolchain

Well known features

- *-c*, *-o* – compiling c file, creating object data
- *-g*: Produce debugging information in the operating system's native format

Debugger – GDB

Basic instructions

- $breakpoint < search - tag >$ – Creates a break point into the source code.
- $next$ – Executes the following instruction
- $inforegister < register - name >$ – get register value
- $x/5i\$eip$ – Next 5 instructions to be executed.
- $list$ – list the program's source code
- $x/o < memory - value >$ – get memory-value content
- $disass < search - tag >$ – get assembler code for a search-tag function

Debugger – GDB

Exercise

1. Create a sample program in C with one pointer and one assignation
2. Run the program with gdb
3. What is the difference between *next* and *nexti*?
4. Use *info register $eip* to understand execution of a program (before and after nexti)
5. Use x/x and x/i to retrieve the location of the pointer in the memory and its content

Debugger – Compiler

```c
int main(void)
{
    int buffer[40];

    return 0;
}
```

Exercise

1. Compile the previous program with the gcc parameter: *mpreferred-stack-boundary* equal to 2,3,4.
2. Using GDB check how the original source of the program is affected.
3. In the rest of the course, we suggest compiling every program with *mpreferred-stack-boundary=2*. Why?

Operating System

Distributions

- Debian OS, Ubuntu, LinuxMint, **Gentoo**, etc.
- Windows XP, Windows 7, 8, 9, etc.
- Mac OS X 9.3, ... Mac OS X 10.5, etc.

Differences

- Different versions, and branches, of commons apps (i.e. gcc, gdb)
- Different Ways of handling memory

Back to the Basis

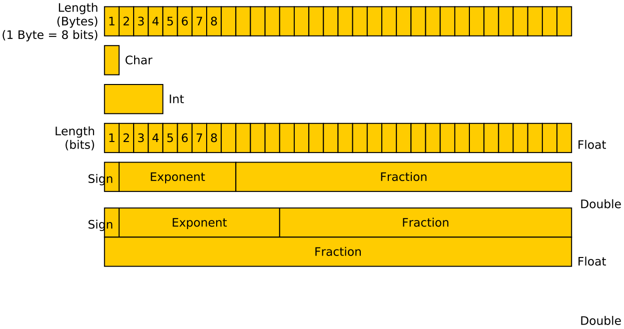
Types

- *char*: smallest addressable unit of the machine that contains a basic character set.
- *int*: basic representation of a number.
- *float*: single-precision floating-point type.
- *double*: double-precision floating-point type.

Specifier

- signed, usigned
- short, long

Back to the Basis: Types



Back to the Basis: Complex Types

Types

- Array
- Signed, Unsigned, long and short int
- Pointers
- Command-line arguments
- Variable Scoping

Back to the Basis: Arrays / Strings

Array

An array is simply a list of *n* element of a specific data type

String

Special case of Array where the data type is char and the last character is a *null byte* (\0)

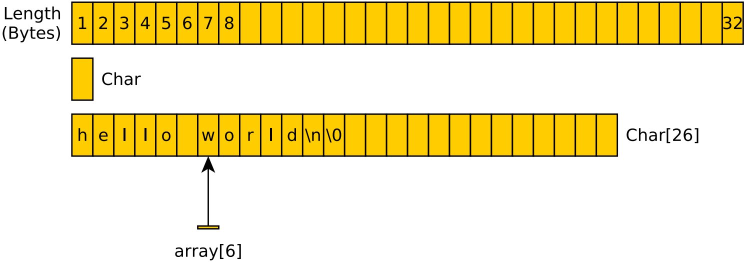
Back to the Basis: Arrays / Strings

Array

An array is simply a list of *n* element of a specific data type

String

Special case of Array where the data type is char and the last character is a *null byte* (\0)



Back to the Basis: Signed, Unsigned

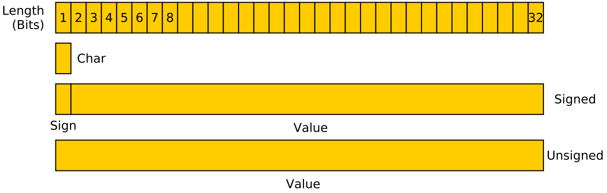
Why signs?

- Numerical values in C are signed: negative or positive
- Signed values allow positive and negative numbers
- Unsigned values only allow positive numbers.

Back to the Basis: Signed, Unsigned

Why signs?

- Numerical values in C are signed: negative or positive
- Signed values allow positive and negative numbers
- Unsigned values only allow positive numbers.



Signed: +/- 2^{31} ($-2^{31}+1$ to $+2^{31}-1$)

Unsigned: 2^{32} (0 to $+2^{32}-1$)

Back to the Basis: Long and Short

Short

- Restraint to *int* data type with only 2 bytes (16 bits)

Long

- Extension of *int* data type with 8 bytes (16 bits)

Back to the Basis: Pointers

Pointer

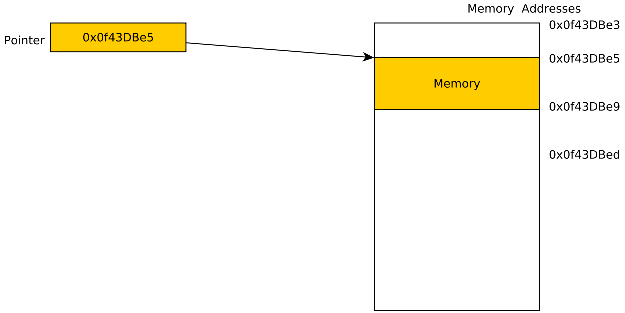
- A pointer is a programming language object whose value refers directly to another value stored elsewhere in the computer memory using its address.
- Useful to avoid copying large bulks of memory.
- Instead of copying, we simply pass the address of a block.

C implementation

- Pointers are defined with an *integer* data type (4 bytes)
- Pointers are defined with a prefix (*)
- Memory management is in charge of malloc/calloc/free instructions

Back to the Basis: Pointers

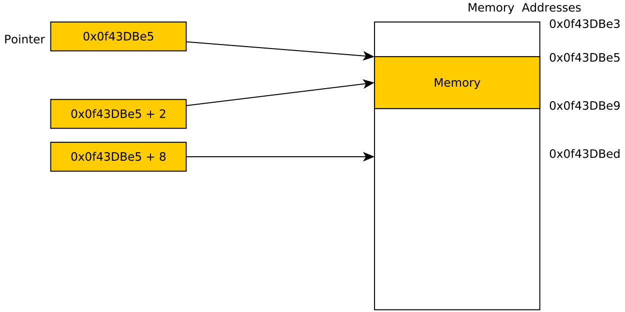
Structure of Pointers



Back to the Basis: Pointers

Operations on Pointers

- Pointers are memory addresses, which are numbers, as such math operations apply



Back to the Basis: Command Line Arguments

Command Line Args in C

- Sent through main function with two arguments (argc and argv)
- *argc*: argument counter, number of arguments
- *argv*: arguments values, contain each of the arguments

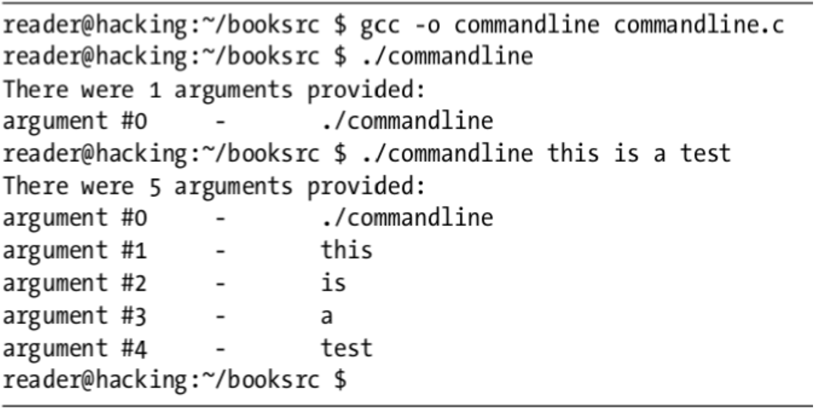
Back to the Basis: Command Line Arguments

```
#include <stdio.h>

int main(int argc, char *argv)
{
  int i;
  printf("%d␣args:\n", argc);

  for (i=0; i< argc; i++)
  {
    printf("arg␣#%d:%s\n", i, argv[i]);
  }
  return 0;
}
```

Back to the Basis: Command Line Arguments



```
reader@hacking:~/booksrc $ gcc -o commandline commandline.c
reader@hacking:~/booksrc $ ./commandline
There were 1 arguments provided:
argument #0     -       ./commandline
reader@hacking:~/booksrc $ ./commandline this is a test
There were 5 arguments provided:
argument #0     -       ./commandline
argument #1     -       this
argument #2     -       is
argument #3     -       a
argument #4     -       test
reader@hacking:~/booksrc $
```