CS Games 2016

Reverse Engineering Competition

Participants	2
Workstations	1
Value	5%
Duration	3 hours

The Bunker

Several documents dating from the old era have been brought back to us from an expedition conducted several moons ago. On these documents were the coordinates of a place located near the Dome. Some say that in it were hidden treasures of the utmost importance for the Dome and its inhabitants.
An expedition was sent to investigate these informations, what they found was somewhat impressive...
A bunker was located at the exact spot indicated on the map, protected by legions of advanced security systems. Some research was later done on this bunker, and it appeared that the legendary General Daniel Black was in charge of this place. The General worked on radiation protection technologies within this exact bunker, but when the catastrophe stroke, he went mad and locked himself in the bunker in which he had already spent decades of his life, researching ways to avoid human extinction.
We need whatever is hidden within this bunker. We have recovered dumps of the security systems, and now need to unravel their secrets to reclaim the work done by the General.

Problem Description

This rather original competition will test your abilities to read and understand how a program works.

For this challenge, there is no need to program well, code quality is not an issue. The only matter of importance is to break through the mysteries of the programs we will give you.
A compressed archive containing all sorts of programs, some compiled, some cryptic source code and diverse specifications will be provided, from which you must extract some knowledge.

We ask of you to provide a text file summing up all this knowlege.
Each trial will provide you with a flag, at the excpetion of one, which will require you to explain the behaviour of the code provided. This will need you to be as clear and precise as possible for your readers to understand what is going on.

***Example** of a report*

```
Bin1: FLAG{123}
Bin2: FLAG{123}
Java1: FLAG{sdj}
Java2: FLAG{sdj}
ASM:
Paramètres : 1 int, 1 int, 1 int
```

Utilité: Inverse paramètres 1 et 2 et les ajoute à 3.

Note that one of the challenges requires an open answer, be as clear as possible.

Brainfuck — 2 points

A program in Brainfuck is given, your job here is to code a simple interpreter for the language. Once you have done this, you may find the hidden message in this program. Would you kindly send it to us ?

ASM — 3 points

A piece of x86-64 assembly code is given to you. Would you kindly give us as many details as you can to help us understand too. Be precise, we need to know:

- How many parameters this program expects ?
- What are their type ?
- In which order must they be passed ?
- What does this program do ?

Java 1 — 2 points

This exerpt from a Java-made security system expects a passwork to unravel is secrets. Would you kindly send us this password ?

Java 2 — 1 points

Another system is protected by a password validation system, we could extract the validation function.
Would you kindly send us the password ?

Encoding — 2 points

We found a strangely encoded file, some information is supposedly hidden within it, but can't get through to it…
The encoding algorithm was given to us, would you kindly send us the hidden information ?

Bin 1 — 2 points

A binary executable was given to us, but we can't process it, although we know something is hidden within it.
Would you kindly send us this information ?

Bin 2 — 1 points

Another binary executable was sent to us. It seems simple, but we lack the knowledge to break through its secrets.
Would you kindly send us the secrets within ?

Bin 3 — 4 points

This binary executable is quite strange, we can't seem to debug it, thus, we cannot understand how it works. Would you kindly tell us what kind of information is hidden behind this sorcery ?

Bin 4 — 6 points

This binary executable looks like something from the deepest pits of hell. An information is hidden here, this is all we know. Would you kindly send us the secret password?

Appendix A

GDB HOWTO

To use GDB, you must first learn the basic commands for the tool.
First, launch GDB on the binary you wish to debug or study.

Example: `gdb --args ./exec arg1 arg2`

Once running, GDB will prompt your for commands before executing your program.
To run it, use the **run** (or **r**) command.

If you wish to break execution at some point, you may use breakopoints.
To add a new breakpoint, use the **break** (or **b**) command, followed by the function or address at which you wish to break.

Step-by-step debugging instructions will be useful on several levels: C-instruction or ASM instruction.
To execute step-by-step on C statements, use the **step** (or **s**) command.
To execute step-by-step on ASM instructions, use the **si** (step-instruction) instruction.

Printing register values and variables is done through the **print** (or **p**) command.

To modify the state of a variable or register, use the **set** command.

Disassembly of a function is done with the **disass** command.

For more informations about GDB, you may use the **help** command anytime, or refer to the manual, given to you along with the competition.

Appendix B - Brainfuck Language

The language consists of eight commands, listed below. A brainfuck program is a sequence of these commands, possibly interspersed with other characters (which are ignored).

The commands are executed sequentially, with some exceptions: an instruction pointer begins at the first command, and each command it points to is executed, after which it normally moves forward to the next command. The program terminates when the instruction pointer moves past the last command.

The brainfuck language uses a simple machine model consisting of the program and instruction pointer, as well as an array of at least 30,000 byte cells initialized to zero; a movable data pointer (initialized to point to the leftmost byte of the array); and two streams of bytes for input and output (most often connected to a keyboard and a monitor respectively, and using the ASCII character encoding).

Commands

The eight language commands each consist of a single character :

Character	Meaning
`>`	increment the data pointer (to point to the next cell to the right).
`<`	decrement the data pointer (to point to the next cell to the left).
`+`	increment (increase by one) the byte at the data pointer.
`-`	decrement (decrease by one) the byte at the data pointer.
`.`	output the byte at the data pointer.
`,`	accept one byte of input, storing its value in the byte at the data pointer.
`[`	if the byte at the data pointer is zero, then instead of moving the instruction pointer forward to the next command, jump it *forward* to the command after the *matching* `]` command.
`]`	if the byte at the data pointer is nonzero, then instead of moving the instruction pointer forward to the next command, jump it *back* to the command after the *matching* `[` command.

`[` and `]` match as parentheses usually do : each `[` matches exactly one `]` and vice versa, the `[` comes first, and there can be no unmatched `[` or `]` between the two.