CS Games 2016

Compétition de Rétro-Ingénierie

Nombre de participants	2
Nombre de postes	1
Valeur totale	5%
Durée Totale	3 heures

Le Bunker

Des documents de l'ancienne ère nous ont été rapportés d'une expédition archéologique il y a de cela plusieurs lunes. Sur ces documents, des coordonnées géographiques d'un endroit situé dans la proximité du Dôme, avec la promesse d'un trésor d'une importance capitale pour le Dôme et ses habitants. Une expédition a été envoyée pour investiguer ces informations, ce qu'ils trouvèrent est pour le moins impressionnant... En effet, un bunker était situé à cet endroit, ce dernier est protégé par de multiples systèmes de sécurité avancés... Après des recherches, il nous est apparu que ce bunker était sous le commandement du légendaire Général Daniel Black, un général des armées dont on raconte l'histoire encore aujourd'hui. Le général travaillait sur des technologies de protection contre les radiations, mais lorsque la catastrophe a frappé, il s'est enfermé dans le bunker dans lequel il avait passé des décennies de sa vie à chercher comment éviter l'éradication de l'espèce humaine. Nous avons besoin de ces renseignements, les systèmes de sécurité du Bunker ont été "dumpés", nous devons maintenant percer leurs secrets pour nous permettre de récupérer les travaux du Général.

Énoncé

Cette épreuve mettra en jeu votre capacité à comprendre le fonctionnement de différents programmes et algorithmes. Vous avez entre les mains une archive contenant des programmes compilés, du code source cryptique et d'autres spécifications. **Votre tâche consiste à extraire de ces systèmes une information clé**.

Description Technique

Vous devez remettre un rapport de votre analyse des programmes et sources fournies. Dans un fichier texte, documentez la réponse de chacune des épreuves.

***Exemple** de rapport*

```
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.

Notez qu'une des épreuves nécessite une réponse ouverte. Soyez le plus clairs et précis possible !

Brainfuck — 2 points

Un programme dans le langage Brainfuck vous est fourni. Votre tâche est de coder un interpréteur qui vous permettra de trouver le message caché dans ce programme.

ASM — 3 points

Vous avez entre les mains un fragment de code Assembleur x86-64. Votre travail consiste à documenter son fonctionnement. Précisément, nous voulons savoir :

- ❏ Combien de paramètres ce code prend-il ?
- ❏ Quels sont leur types ?
- ❏ Dans quel ordre faut-il les passer ?
- ❏ Que fait ce programme ?

Java 1 — 2 points

Cet extrait d'un système de sécurité en Java requiert un mot de passe afin d'accéder à l'information confidentielle. Pouvez-vous nous trouver ce mot de passe ?

Java 2 — 1 points

Un autre système de sécurité nécessite un mot de passe. Nous avons par contre accès à l'algorithme de validation.

Encoding — 2 points

Nous avons un fichier encodé d'une façon bizarre, une information serait supposément cachée dedans, mais nous ne pouvons pas le décoder…
L'algorithme de codage est fourni avec le fichier codé, pouvez-vous coder un programme de décodage et nous envoyer l'information cachée ?

Bin 1 — 2 points

Un fichier exécutable nous a été fourni, nous ne sommes pas capables de comprendre son fonctionnement, mais nous sommes sûrs qu'une information est cachée dedans.
Pouvez-vous comprendre son fonctionnement et nous envoyer cette information ?

Bin 2 — 1 points

Un autre fichier exécutable nous a été envoyé. Il semble assez simple, mais nos compétences ne sont pas suffisantes pour percer ses secrets.
Pouvez-vous nous envoyer l'information cachée ?

Bin 3 — 4 points

Ce fichier exécutable est pour le moins étrange, et il nous semble impossible de l'exécuter pas-à-pas ! Dites-nous ce qui est caché et vous serez récompensés adéquatement.

Bin 4 — 6 points

L'apocalypse a frappé. Une information est cachée, et c'est tout ce que nous savons. Aidez-nous, vous êtes notre dernier espoir…

Annexe A

GDB HOWTO

Pour utiliser GDB, vous devrez apprendre les commandes de base de l'outil.
Lancez tout d'abord GDB sur l'exécutable que vous voulez déboguer via une ligne de commande.

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

Une fois lancé, GDB vous offrera un prompt pour que vous puissiez exécuter votre programme.
Pour l'exécuter il faut utiliser la commande `run` (ou `r`).

Si vous souhaitez bloquer l'exécution à un point donné, vous pouvez vous servir de breakpoints.
Pour ajouter un breakpoint, utilisez la commande `break` (ou `b`), suivi de la fonction ou l'adresse à laquelle vous voulez bloquer.

Les instructions de debug pas-à-pas sont utiles à deux niveaux: instruction C ou instruction ASM.
Pour avancer pas à pas au niveau de l'instruction C, utilisez `step` (ou `s`).
Pour effectuer la manoeuvre au niveau de l'assembleur, vous devrez utiliser `si` (step-instruction).

L'affichage des registres et variables se fait à l'aide de la commande `print` (ou `p`).

Pour modifier une variable en mémoire, utilisez la commande `set`.

Le désassemblage d'une fonction se fait à l'aide de la commande `disass`.

Pour plus d'informations sur gdb, vous pourrez utiliser la commande `help` ou vous référer au manuel fourni avec la compétition.

Annexe B - Langage Brainfuck

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.