gitlab.enssat.fr

Alexandre DOYEN / CTF Logiciels sécurisés · GitLab

10-13 minutes

Light Victorious Machine

1) Contexte

Light Victorious Machine est un programme qui n'est utilisable qu'en entrant le bon numéro de série associé.

Pour effectuer la vérification, il se base sur une machine virtuelle codée au sein même du programme, afin de mieux obfusquer le mécanisme de vérification.

2) Analyse de la machine virtuelle

2.1) Récupération du code "compilé" exécuté par la machine virtuelle

Tout d'abord, en ouvrant le programme dans le logiciel Ghidra, on remarque qu'il y a un tableau présent dans la fonction `main()` contenant le code exécuté par la VM. Par exemple :

```
codeVM[0] = 0;
codeVM[1] = (int)*argv2[1];
codeVM[2] = 1;
codeVM[3] = 0;
local_6c4 = 0;
local_6c0 = (int)argv2[1][1];
local_6bc = 1;
local_6b8 = 1;
local_6b4 = 0;
local_6b0 = (int)argv2[1][2];
local_6ac = 1;
local_6a8 = 2;
```

Bien qu'au premier abord, il semble que ce soient des variables toutes indépendantes les unes des autres (Sauf les quatre premières), on remarque qu'en réalité l'exécution du code se fait au moyen d'une boucle parcourant l'entièreté du tableau :

```
for (i = 0; i < 0xd8; i = i + 1)
{
    vm(codeVM + i * 2);
}
```

2.2) Analyse du fonctionnement de la machine virtuelle

La machine virtuelle peut être assimilée à une machine RISC (Reduced Instruction Set Computer). En effet, en analysant le code de la fonction `vm()`, on

remarque qu'elle comporte cinq types d'instructions différentes. Ainsi, elle fonctionne avec des instructions codées sur un couple de deux variables, où la première correspond au code de l'instruction, et la deuxième, son paramètre.

2.2.1) PUSH

L'instruction "PUSH" permet de pousser le paramètre sur la pile associée à la machine virtuelle.

2.2.2) STO

L'instruction "STO" permet de mettre dépiler une valeur de la pile pour la mettre dans la mémoire à l'adresse passée en paramètre.

2.2.3) LOAD

L'instruction "LOAD" permet de charger une valeur stockée en mémoire à l'adresse passée en paramètre sur le sommet de la pile.

2.2.4) PUSH_VAL_SP_PLUS_VAL_SP_PLUS_1

L'instruction "PUSH_VAL_SP_PLUS_VAL_SP_PLUS_1" permet de dépiler deux valeurs en mémoire et d'empiler la somme des deux.

Code correspondant :

```
void vmPushPopPlusPop(void)
{
  int a;
  int b;

  a = vmPop();
  b = vmPop();
  vmPush((a + b * 2) * 2 - (a + b * 3));
  return;
}
```

Ainsi, on a $2(a+2b)-(a+3b) = 2a+4b-a-3b = a+b$, et ce, malgré la tentative d'obfuscation réalisée par le développeur !

2.2.4) PUSH_VAL_SP_MUL_VAL_SP_PLUS_1

L'instruction "PUSH_VAL_SP_MUL_VAL_SP_PLUS_1" permet de dépiler deux valeurs en mémoire et d'empiler le produit des deux.

Code correspondant :

```
void vmPushPopMulPop(void)
{
  int a;
  int b;
  int res;
  int j;
  int i;

  a = vmPop();
  b = vmPop();
  res = 0x2a;
  if (b < 0) {
    for (i = 0; b < i; i = i + -1) {
      res = res + a;
    }
```

```
    res = 0x2a - res;
  }
  else {
    for (j = 0; j < b; j = j + 1) {
      res = res + a;
    }
    res = res + -0x2a;
  }
  vmPush(res);
  return;
}
```

Ici, de même, malgré la complexité apparente du code, il se trouve qu'il ne retourne que le produit $a \times b$.

2.2.5) VERIF_ARG

L'instruction "VERIF_ARG" permet de vérifier si la valeur étant au sommet de la pile est égale à l'argument. Si ce n'est pas le cas, alors la valeur étant au sommet de la pile n'est pas correcte, et in fine, le numéro de série non plus.

2.3) Analyse du programme exécuté par la machine virtuelle

Ainsi, connaissant ces informations, il devient facile de créer un désassembleur permettant de comprendre le code exécuté par la machine virtuelle.

2.3.1) Chargement en mémoire du numéro de série entré par l'utilisateur

Dans un premier temps, la machine virtuelle va charger le numéro de série entré par l'utilisateur dans sa mémoire interne.

```
PUSH                                                    argv[1]
[0]
STO                                                     0x0
(argv[0x0])
PUSH                                                    argv[1]
[1]
STO                                                     0x1
(argv[0x1])
PUSH                                                    argv[1]
[2]
STO                                                     0x2
(argv[0x2])
```

Par la suite, on appellera `argv[x]` le x-ième caractère (En partant de 0) du numéro de série.

2.3.2) Vérification du numéro de série

Avant d'exécuter le code de la machine virtuelle, le programme vérifie s'il a bien une taille de 18 caractères.

Ainsi, il effectue 18 verifications du type :

```
PUSH                                                    0x7
LOAD                                                    0xe
(argv[0xe])
PUSH_VAL_SP_MUL_VAL_SP_PLUS_1                           0x0
STO                                                     0x12
PUSH
0xfffffffe
LOAD                                                    0xf
```

```
(argv[0xf])
PUSH_VAL_SP_MUL_VAL_SP_PLUS_1                    0x0
LOAD                                             0x12
PUSH_VAL_SP_PLUS_VAL_SP_PLUS_1                   0x0
VERIF_ARG                                        0xf7
```

Donc, on se rend compte que, dans ce cas, il vérifie que $247 = 7 \times \text{argv}[14] - 2 \times \text{arg}[15]$.

En réalité, il effectue 18 vérifications du type $z = a \times x + b \times y$, où z est le résultat, a et b les coefficients de l'équation, et x et y les catactères du numéro de série placés aux indices i et j. En effet, ce bloc d'instructions peut être traduit ainsi :

```
a = 0x7 7
i = 14
b = -0x2 -2
j = 15
z = 0xf7 247
```

L'enjeu est donc de trouver les solutions arg[i] et arg[j] pour toutes les équations diophantiennes telles qu'elles soient comprises dans l'intervalle $[\![33; 126]\!]$. Il correspond à l'intervalle des valeurs décimales correspondant à des caractères ASCII affichables.

3) Décodage du numéro de série

Ainsi, à l'aide d'un solveur d'équations diophantiennes, on trouve qu'il faut résoudre les trois systèmes d'équations suivants avec leurs solutions associées étant fonction d'un nombre $k \in \mathbb{Z}$. L'enjeu est donc de trouver la bonne valeur de k associée à chaque équation permettant d'avoir des valeurs de $\text{arg}[i]$ satisfaisant la contrainte du caractère ASCII affichable, et ce, pour tous les $\text{arg}[i]$ du même système.

$$\begin{cases} -1 \times \text{arg}[0] + 3 \times \text{arg}[1] = 253 \\ -2 \times \text{arg}[2] + 3 \times \text{arg}[1] = 195 \\ 3 \times \text{arg}[2] - 2 \times \text{arg}[3] = 24 \\ 0 \times \text{arg}[4] + 1 \times \text{arg}[3] = 114 \\ 3 \times \text{arg}[4] - 5 \times \text{arg}[5] = 7 \\ 6 \times \text{arg}[0] - 9 \times \text{arg}[5] = 30 \end{cases} \iff \begin{cases} \text{arg}[0] = -253 + 3k; \text{arg}[1] = 0 + 1k \\ \text{arg}[2] = 195 + 3k; \text{arg}[1] = 195 + 2k \\ \text{arg}[2] = 24 - 2k; \text{arg}[3] = 24 - 3k \\ \text{arg}[4] = 0 + 1k; \text{arg}[3] = 114 + 0k \\ \text{arg}[4] = 14 - 5k; \text{arg}[5] = 7 - 3k \\ \text{arg}[0] = -10 - 3k; \text{arg}[5] = -10 - 2k \end{cases}$$

$$\begin{cases} 9 \times \text{arg}[6] - 8 \times \text{arg}[7] = 209 \\ -7 \times \text{arg}[9] + 7 \times \text{arg}[6] = 77 \\ 1 \times \text{arg}[8] + 1 \times \text{arg}[9] = 231 \\ -7 \times \text{arg}[11] + 8 \times \text{arg}[8] = 191 \\ -6 \times \text{arg}[10] + 8 \times \text{arg}[11] = 186 \\ -3 \times \text{arg}[7] + 4 \times \text{arg}[10] = 138 \end{cases} \iff \begin{cases} \text{arg}[6] = 209 - 8k; \text{arg}[7] = 209 - 9k \\ \text{arg}[9] = -11 + 1k; \text{arg}[6] = 0 + 1k \\ \text{arg}[8] = 231 + 1k; \text{arg}[9] = 0 - 1k \\ \text{arg}[11] = 191 + 8k; \text{arg}[8] = 191 + 7k \\ \text{arg}[10] = 93 + 4k; \text{arg}[11] = 93 + 3k \\ \text{arg}[7] = 138 + 4k; \text{arg}[10] = 138 + 3k \end{cases}$$

$$\begin{cases} 0 \times \arg[13] + 2 \times \arg[16] = 128 \\ 5 \times \arg[17] - 7 \times \arg[14] = 129 \\ 8 \times \arg[15] - 9 \times \arg[12] = 56 \\ -1 \times \arg[16] + 2 \times \arg[17] = 164 \\ 7 \times \arg[14] - 2 \times \arg[15] = 247 \\ 9 \times \arg[12] - 7 \times \arg[13] = 55 \end{cases} \iff \begin{cases} \arg[13] = 0 + 1k; \arg[16] = 64 + 0k \\ \arg[17] = 387 - 7k; \arg[14] = 258 - 5k \\ \arg[15] = -56 - 9k; \arg[12] = -56 - 8k \\ \arg[16] = -164 + 2k; \arg[17] = 0 + 1k \\ \arg[14] = 247 - 2k; \arg[15] = 741 - 7k \\ \arg[12] = -165 - 7k; \arg[13] = -220 - 9k \end{cases}$$

Ainsi, le premier et le dernier système ont chacun une valeur triviale (Trouvable grâce à une équation de la forme $0 \times \arg[i] + a \times \arg[j] = z$). On remarque donc très vite que $\arg[3] = 114$ (Caractère `r`) et $\arg[16] = 64$ (Caractère `@`). Donc, connaissant cela, on peut en déduire les autres inconnues pour ces deux systèmes.

Concernant le deuxième système, étant donné qu'il n'a pas de valeur triviale, il a fallu passer par une étape de force brute, en fixant $\arg[6]$ de 33 à 126 (Valeurs ASCII convenables). Ainsi, il est possible de calculer les autres valeurs manquantes, et vérifier si les valeurs calculées conviennent. Sinon, on incrémente $\arg[6]$[1].

Finalement, les valeurs trouvées sont les suivantes :

```
arg[0] = n
arg[1] = y
arg[2] = T
arg[3] = r
arg[4] = w
arg[5] = F
arg[6] = y
arg[7] = n
arg[8] = y
arg[9] = n
arg[10] = u
arg[11] = o
arg[12] = P
arg[13] = _
arg[14] = ?
arg[15] = a
arg[16] = @
arg[17] = r
```

4) Notes

1. Le script permettant de résoudre les systèmes est fourni en annexe de ce Writeup. ↩