Thomas
Caplin

01/05/2009

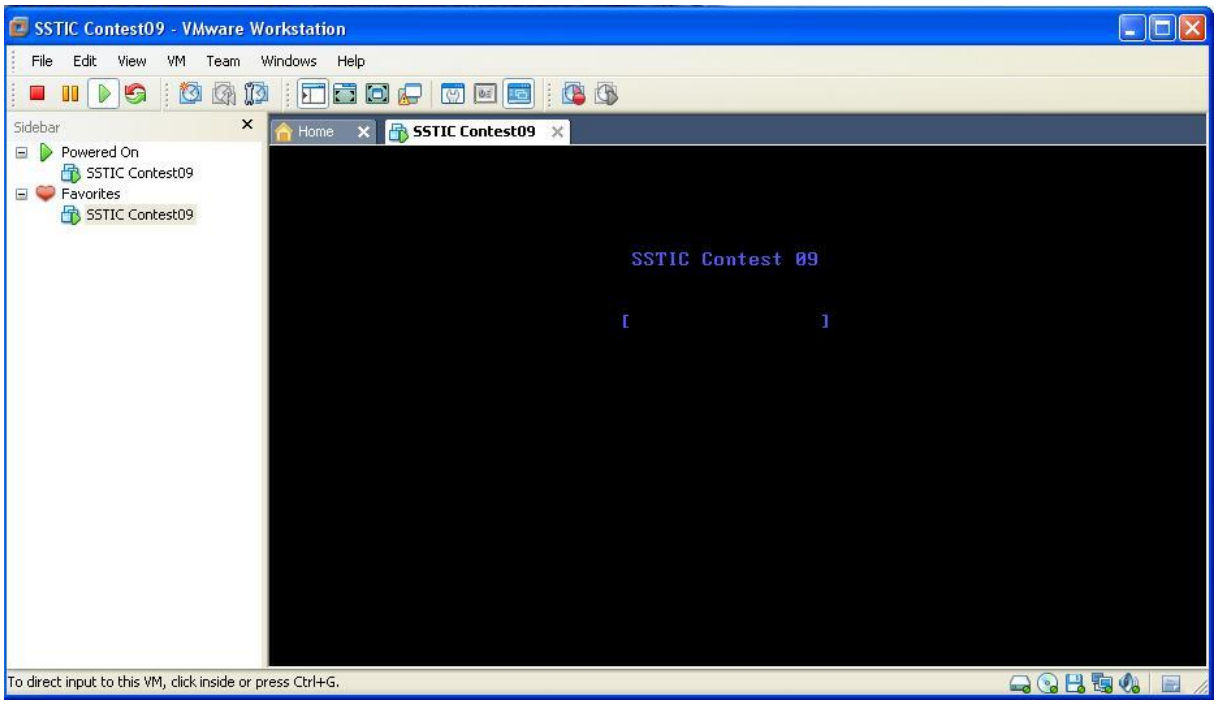
Challenge

SSTIC09

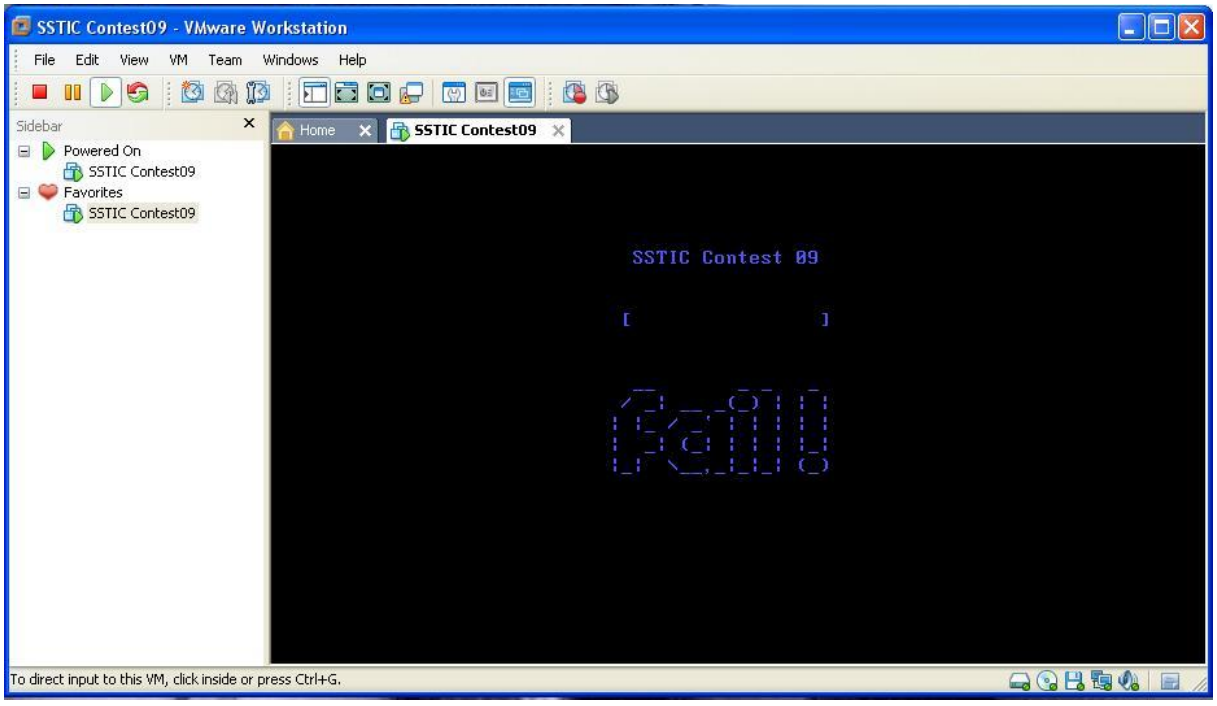
Table des Matières

1. Description du binaire

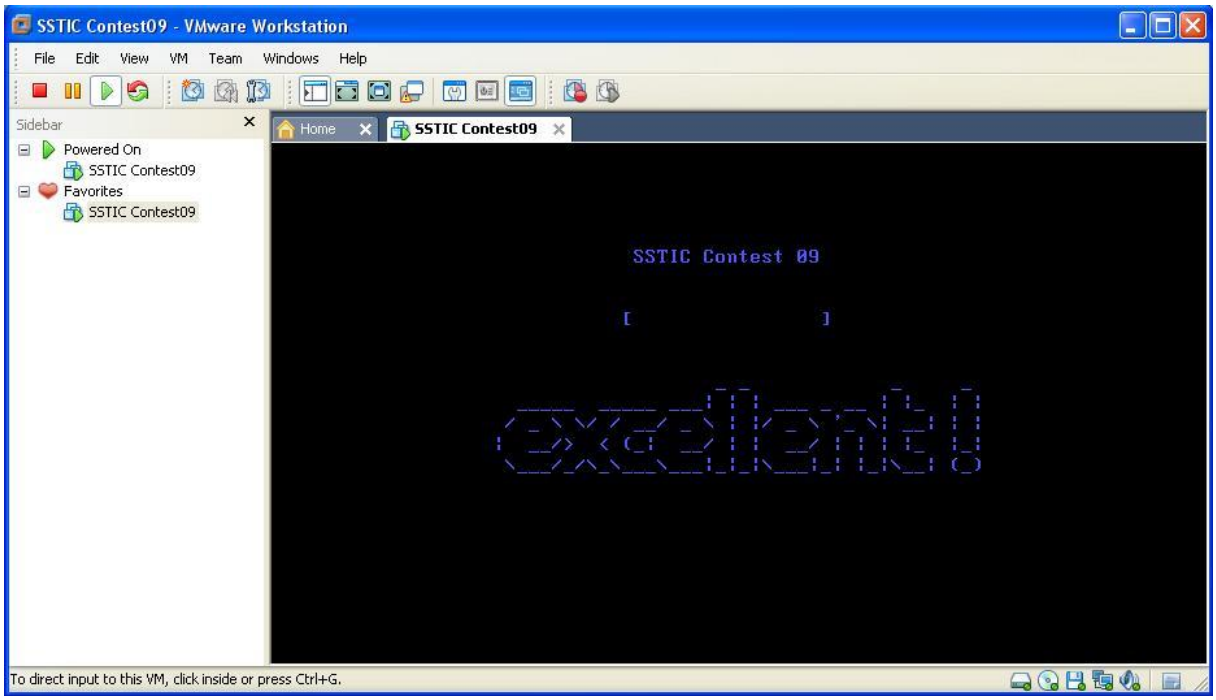
Le binaire est une image de disquette bootable. Le bootloader est grub, il démarre un noyau customisé.
Une fois le système démarré, un écran s'affiche et attend la saisie d'un password :



.
Si le password est incorrect, un message d'erreur est affiché :



Si le password est correct, un message de félicitations s'affiche :



Le **but** du challenge est donc de trouver le bon password !

2. Résolution du challenge

a) Comment gagner ?

Tout d'abord, nous avons cherché à savoir comment obtenir le message signifiant que le mot de passe entré était correct.
La démarche fut la suivante :

- rechercher où se trouve la fonction permettant d'afficher un caractère à l'écran : pour cela nous avons *reversé* le code en partant de l'affichage de la chaîne « SSTIC Contest 09 ».
- rechercher où se trouve le code qui affiche le message d'erreur en recherchant donc toutes les parties du code qui utilisent la routine affichant un caractère.
- Une fois trouvé, on cherche quelle condition fait afficher ce message d'erreur plutôt que l'autre.
- On étudie cette condition : le programme prend les 12 mots à partir de l'adresse 0x70000 (donc 0x70000, 0x70004, etc…), réalise pour chaque, un XOR avec une valeur fixe : 0x0A5CC1A27, et une comparaison avec des valeurs fixes dans le code, exemple :
  - 0x70000 XOR 0x0A5CC1A27 comparé à 0x0CB3C6DC1
  - 0x70004 XOR 0x0A5CC1A27 comparé à 0x7D35CCCD
  - Etc...

- A la moindre différence d'un des mots, la boucle de test s'arrête et le message d'échec est affiché.
- Nous calculons donc calculer quelles valeurs doivent se trouver dans 0x70000 pour que le bon message soit affiché :
  - 0x70000 : 0x6EF077E6
  - 0x70004 : 0xD8F9D6EA
  - 0x70008 : 0x9BF64FF8
  - 0x7000c : 0x4AFEB1FC
  - 0x70010 : 0x9AE5EBC4
  - 0x70014 : 0xDECFBCE0
  - 0x70018 : 0x56F4D5FD
  - 0x7001c : 0xA7FC8DDA
  - 0x70020 : 0x2CFD1BC7
  - 0x70024 : 0x98C364F9
  - 0x70028 : 0x72CE63FF
  - 0x7002c : 0xA3FFAFF9

b) Et notre chaîne dans tout ça ?

Nous allons maintenant chercher ce que fait le programme avec notre chaîne.

- nous trouvons son adresse toujours grâce à l'utilisation de la routine qui affiche un caractère.
- Nous étudions le code afin d'en dégager l'algorithme principal :
    - Tant que 1 :
        - Afficher la boite de saisie
        - Attendre la saisie d'un caractère
        - Si ce caractère est le retour chariot on passe à la suite
        - Sinon, on l'affiche et on le rentre dans notre buffer, sauf si on a déjà saisit 16 caractères.
        - Une fois le retour chariot détecté on passe à la routine qui vérifie le password
    - Fin Tant que
- Notre chaîne n'est pas rangée en 0x70000, ce n'est donc pas si simple !
- Nous déroulons l'exécution jusqu'à l'utilisation par le programme de la chaîne et nous nous rendons compte ainsi qu'il la copie en 0x60000 :
    - 0x60000 = 4 premiers caractères
    - 0x60004 = 2ème paquet de 4 caractères
    - 0x60008 = 3ème paquet de 4 caractères
    - 0x6000c = dernier paquet de 4 caractères
- Nous continuons l'exécution du programme en nous arrêtant juste avant la boucle de comparaison vue en A), et nous nous apercevons que les valeurs des 4 mots à partir de 0x60000 ont été modifiées.

c) Relation entre 0x60000 et 0x70000 ?

Etant donné que le but est de rentrer un bon password, il est logique de se dire que les 12 mots qui se trouvent à partir de l'adresse 0x70000 ont été construits à partir de ceux qui sont à partir de l'adresse 0x60000 (eux-mêmes construits à partir de notre chaîne).
Nous déroulons donc l'exécution du programme jusqu'à la première modification de la valeur de 0x70000. Puis nous *reversons* le code afin de voir comment elle est construite. Nous trouvons qu'il y a, pour chacun des 12 mots, 2 transformations. Les voici pour le premier mot en 0x70000 :

- 1ère transformation :
    - Le 1er demi-mot est le reste de la division du 1er demi-mot en 0x60000 par la constante 0x23BF
    - Le 2ème demi-mot est le reste de la division du 2ème demi-mot en 0x60000 par la constante 0x34F6
- 2ème transformation :
    - Le 1er demi-mot devient : not(inv(1er demi-mot)), ex : si le demi-mot vaut 0x1234, alors on lui applique not(0x3412)
    - Le 2ème demi-mot devient : not(inv(2ème demi-mot)

Pour certains demi-mots, la 1ère transformation est la suivante :

Demi-mot = demi-mot 0x6000x correspondant – A
Avec, A = [[[(demi-mot(0x6000x)*constante)>>0x10]>>constante]*constante]
Ceci est fait pour chaque mot, seule la constante change et le mot pour la division :

- 0x70004 utilise le mot en 0x60004
- 0x70008 utilise le mot en 0x60008
- 0x7000c utilise le mot en 0x6000c
- 0x70010 utilise le mot en 0x60000
- Etc…

d) Comment gagner avec 0x60000

Nous connaissons le rapport entre les mots en 0x60000 et en 0x70000. Nous connaissons les valeurs finales des mots en 0x70000 pour gagner. Il est donc facile maintenant de trouver les valeurs finales des mots en 0x60000 pour gagner.
Cherchons par exemple le mot en 0x60000 :

- il est utilisé dans le calcul de 0x70000, 0x70010 et 0x70020 qui doivent au final valoir respectivement : 0x6EF077E6, 0x9AE5EBC4, 0x2CFD1BC7. Pour chacun de ces 3 mots, on *reverse* l'algorithme afin de voir combien doit valoir chaque demi-mot de 0x60000 pour donner les bons résultats.
- On obtient les systèmes suivant :
  - X / 0x23BF reste **0x0F91**
  - X – ((((X * 0xF7F3)>>0x10)>>0xE) * 0x4214) = 0x1A65
  - X – ((((X * 0x349B)>>0x10)>>0xC) * 0x4DDD) = 0x02D3

  - Y / 0x34F6 reste 0x1988
  - Y – ((((Y * 0x82B9)>>0x10)>>0xD) * 0x3EAB) = 0x3B14
  - Y / 0x3FC3 reste 0x38E4

  X et Y étant chacun les demi-mots correspondant à la valeur en 0x60000. Les restes des divisions, et les résultats des soustractions sont obtenus simplement en *reversant* la 2ème transformation sur chaque mot 0x70000, 0x70010, 0x70020. Par exemple :
  - 0x70000 final doit valoir 0x6EF077E6
  - Not(0x6EF0) = 0x910F => inv : **0x0F91**

Nous résolvons ces systèmes pour 0x60000, 0x60004, 0x60008 et 0x6000c. Nous trouvons qu'ils doivent avoir comme valeur finale :

- 0x60000 : 0x9E8DB86A
- 0x60004 : 0xCE6D6C69
- 0x60008 : 0xBCC7253C
- 0x6000c : 0x196CD5F6

e) Oui mais ma chaîne dans tout ça ?

Nous en arrivons maintenant au point crucial : trouver comment sont modifiés les 4 mots en 0x60000 après leur initialisation par la valeur de notre chaîne.

Pour cela, nous nous plaçons au moment où la chaîne est entièrement copiée en 0x60000, et nous exécutons le programme jusqu'à ce que la valeur en 0x60000 ait changée. Ensuite nous *reversons* le code afin de trouver l'algorithme utilisé. Nous faisons la même chose pour 0x60004, 0x60008 et 0x6000c.

Nous nous plaçons maintenant juste avant que la valeur en 0x60000 obtienne sa valeur finale, nous *reversons* le code afin de voir si le même algorithme est utilisé : c'est le cas, seul les constantes changent. Nous faisons pareil pour 0x60004, 0x60008 et 0x6000c.

Nous comptons combien de fois chaque mot est modifié jusqu'à sa valeur finale : 19 fois !

Voici donc les 4 algorithmes qui sont appliqués 19 fois sur chaque mot :

- Pour 0x60000, on note i la valeur du mot au tour i :

Nouvelle valeur = ancienne valeur + A
Avec A = [(cste1 XOR 0x6000c(i-1)) + (cste2 XOR 0x60004(i-1))]
XOR
[(0x60004(i-1)<<2 XOR 0x6000c(i-1)>>5) + (0x60004(i-1)>>3 XOR 0x6000c(i-1)<<4)]

- Pour 0x60004 :

Nouvelle valeur = ancienne valeur + A
Avec A = [(cste1 XOR 0x60000(i)) + (cste2 XOR 0x60008(i-1))]
XOR
[(0x60008(i-1)<<2 XOR 0x60000(i)>>5) + (0x60008(i-1)>>3 XOR 0x60000(i)<<4)]

- Pour 0x60008 :

Nouvelle valeur = ancienne valeur + A
Avec A = [(cste1 XOR 0x60004(i)) + (cste2 XOR 0x6000c(i-1))]
XOR
[(0x6000c(i-1)<<2 XOR 0x60004(i)>>5) + (0x6000c(i-1)>>3 XOR 0x60004(i)<<4)]

- Pour 0x6000c :

Nouvelle valeur = ancienne valeur + A
Avec A = [(cste1 XOR 0x60008(i)) + (cste2 XOR 0x60000(i))]
XOR
[(0x60000(i)<<2 XOR 0x60008(i)>>5) + (0x60000(i)>>3 XOR 0x60008(i)<<4)]

Les constantes cste1 et cste2 varient pour chaque tour de boucle.
La constante cste2 est la même à chaque tour de boucle pour chaque mot, en revanche cste1 non.

Nous *reversons* le code afin de trouver la valeur de chaque constante.

f) On y est !

Nous avons les valeurs finales en 0x60000, 0x60004, 0x60008 et 0x6000c. Nous avons aussi les algorithmes pour les modifications successives de chaque mot. Nous avons également les constantes utilisées dans chaque tour de boucle. Nous pouvons donc *reverser* chaque tour de boucle afin de trouver les bonnes valeurs initiales en 0x60000, 0x60004, 0x60008 et 0x6000c.

Les algorithmes *reversés* sont triviaux :

- Ancienne valeur = nouvelle valeur – A

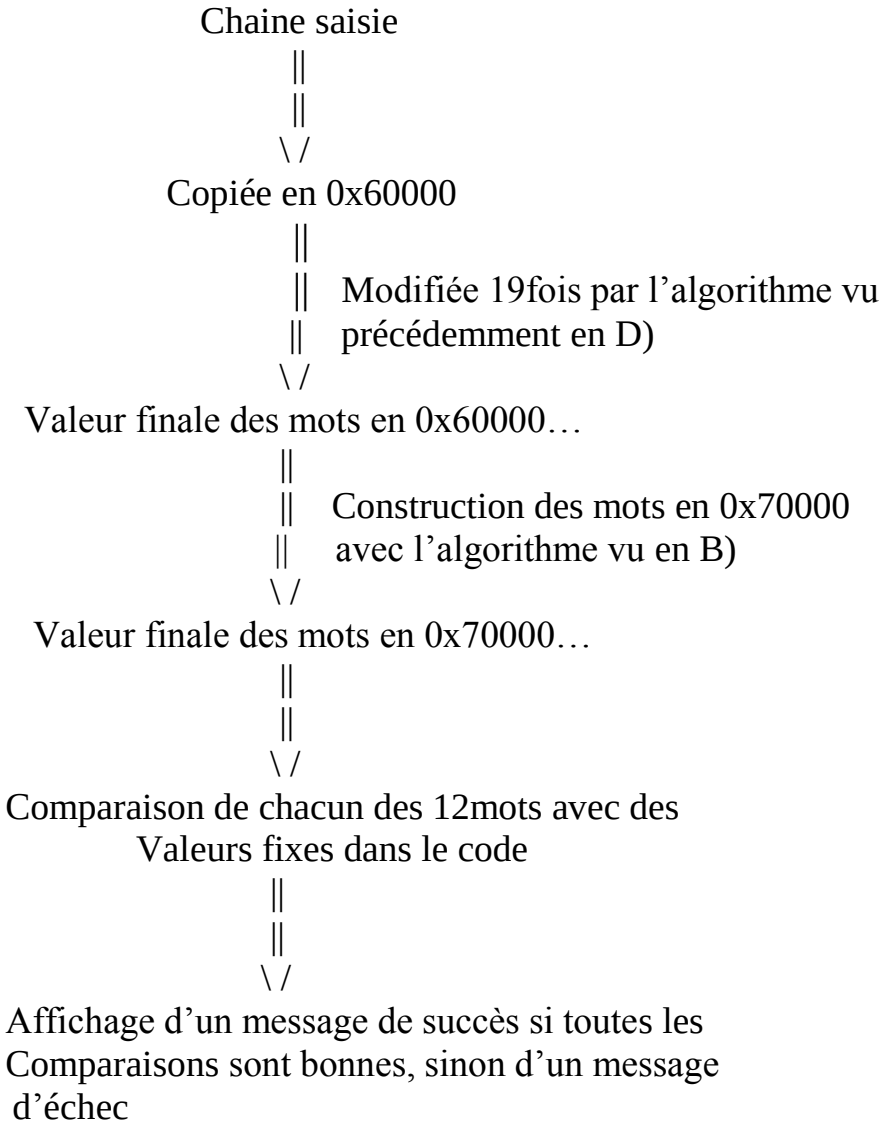
Attention, il faut commencer par *reverser* 0x6000c, puis 0x60008, puis 0x60004 et enfin 0x60000 au risque de se retrouver avec trop d'inconnues ;)

Nous trouvons donc les valeurs initiales conduisant à un succès suivantes :

- 0x60000 : 0x7761375A
- 0x60004 : 0x5C57673F
- 0x60008 : 0x7C4C483D
- 0x6000c : 0x395F704E

Et comme nous le savons déjà, c'est notre chaîne qui se trouve ici ! Donc le password correct est : **Z7aw?gW\=HL|Np_9**

3. Schéma récapitulatif des mécanismes du challenge



4. Outils utilisés

- VMWare : exécution du binaire dans une machine virtuelle ayant un processeur 32bits et un lecteur disquette sur lequel elle boot (l'image de la disquette étant le binaire).
- IDA Pro : désassemblage du binaire, étude du code, des chaînes, etc…
- GDB connecté en remote à VMWare : étude du code, étude de l'exécution du programme.
- Scripts GDB : afin de gérer l'exécution pas à pas. Le flot d'exécution étant obfusqué et énorme, une exécution pas à pas à la main n'est pas envisageable.
- Scripts python : afin de résoudre les systèmes d'équations vus en C), et de reverser automatiquement les algorithmes vus en D).

5. Conclusion

Ce challenge m'a permis d'apprendre à *reverser* un binaire ayant un énorme flot d'exécution. J'ai dû pour cela apprendre à manier les scripts GDB.

Même sans entrer dans les détails de l'obfuscation il nous a été possible de réussir ce challenge. La méthode globale fut de se positionner à la fin d'une partie de code intéressante (modification d'une valeur à une adresse intéressante) puis de remonter le code jusqu'à obtenir l'expression de cette valeur uniquement en fonction de constantes ou de valeurs connues (valeurs précédentes, saisies, etc..).

Merci à l'auteur du challenge pour ce beau travail !