HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

SLB-2023

Sécurité logicielle bas niveau

Laboratoire 2

exploit

Département TIC - orientations ISCE/S

Professeur :

Jean-Marc Bost

Assistant :

Lucas Gianinetti

2 novembre 2023

SLB2023 – labo02 – exploit

Table des matières

1 Introduction 3

1.1 Objectifs 3

1.2 Rendu 3

1.3 Evaluation 3

1.4 Matériel nécessaire 4

2 Principe du laboratoire 5

3 Linux – 32 bits (10 pts) 6

4 Linux – 32 et 64 bits (22 pts) 8

5 Windows – 32 bits (18 points) 11

6 Défense Linux – 64 bits (20 pts) 14

7 Défense Linux – 32 bits (16 pts) 16

8 Défense Linux –32 bits (27 pts) 19

1 Introduction

1.1 Objectifs

Les objectifs de ce laboratoire sont les suivants :

- Exploiter des vulnérabilités buffer overflow sous Linux et Windows, en 32 bits et 64 bits
- Contourner les contremesures des compilateurs et systèmes d'exploitation

1.2 Rendu

Un court rapport répondant **uniquement aux questions** posées doit être rendu sur Cyberlearn avant l'échéance indiquée dans le rendu. Vous devez également y déposer vos **sources avec les noms de fichier indiqués**.

NB : Les réponses aux questions doivent être illustrées avec les **captures d'écran** de vos tests pertinents et les informations importantes doivent y être mises en évidence et référencées explicitement dans vos explications. Les codes source des scripts et programmes rédigés pour effectuer les manipulations et répondre aux questions doivent être référencés par leur nom de fichier dans le rapport et joints en annexe.

Aucune correction du laboratoire ne sera présentée en classe. Vous pouvez prendre RV avec l'équipe enseignante pour poser vos questions.

1.3 Evaluation

Ce rapport fera l'objet d'une évaluation basée sur :

- (80% de la note) exactitude, lisibilité, efficacité et exhaustivité des réponses aux questions
- (10% de la note) complétude, efficacité et lisibilité du code source joint en annexe
- (10% de la note) la qualité du document en termes de présentation, rédaction et illustrations

1.4 Matériel nécessaire

Tous les fichiers nécessaires vous sont disponibles sur Cyberlearn.

En particulier, il est indispensable de n'**utiliser que les shellcodes fournis** par ce moyen pour l'ensemble des exploitations à réaliser dans ce laboratoire.

Ce laboratoire a été testé sur les machines virtuelles Ubuntu 22.04 LTS et Windows 10 pour x86 qui vous ont été fournies pour le cours, en utilisant une version récente de peda et pwntools pour python3 sur la machine Ubuntu. Il est fortement recommandé d'utiliser ces machines pour ce laboratoire.

2 Principe du laboratoire

Le but de ce laboratoire est de mettre en pratique la théorie et les exemples vus en cours pour l'exploitation de buffer overflows.

Lqg lfhv#

Attention : l'ordre et la taille des variables locales des sources fournis ne sont pas forcément ceux du fichier binaire correspondant.

Attention : il peut être nécessaire de combiner plusieurs vulnérabilités pour résoudre le challenge qui vous est proposé.

Attention : Un CANARY commence toujours par un \0 afin d'éviter qu'on ne puisse afficher sa valeur.

Attention : L'utilisation de l'extension peda pour gdb est très utile pour visualiser la stack et le code en cours d'exécution.

Attention : L'utilisation de pwntools est très utile pour faciliter le « chaînage » des inputs/outputs successifs d'un programme interactif.

3 Linux – 32 bits (10 pts)

Binaire : *chall1*

QUESTION 3.1 (2 PTS)

Quelle fonction fait référence à la fonction *win* et comment s'y prend-elle ? Que fait cette fonction avec la référence à *win* ? Dans quelles conditions, la fonction *win* sera-t-elle exécutée ? Au final, quelle fonction fait vraiment l'appel à *win* ?

QUESTION 3.2 (2 PTS)

Quelle vulnérabilité pourrait nous permettre d'assurer l'appel à la fonction *win* ?

MANIPULATION 3.1

Trouver l'adresse de la fonction *win* et les positions relatives des variables de la fonction qui référence *win* sur la pile.

QUESTION 3.3 (3 PTS)

Dessiner la stack frame de la fonction qui fait référence à *win*.

MANIPULATION 3.2

Sur la base de toutes ces informations, exploiter la vulnérabilité pour arriver dans la fonction *win*.

QUESTION 3.4 (3 PTS)

Présentez le payload que vous avez construit à l'étape précédente en séparant chaque élément qui le compose et en indiquant son rôle dans l'exploit.

4 Linux – 32 et 64 bits (22 pts)

Binaires : *chall2.32* *chall2.64*

Il s'agit de la compilation en respectivement, 32 et 64 bits, du même programme *chall2.c*.

QUESTION 4.1 (2 PTS)

Quelle fonction fait référence à la fonction *win* ? Que fait la fonction *win* ? Dans quelles conditions, la fonction *win* sera-t-elle exécutée ?

QUESTION 4.2 (2 PTS)

Quelle vulnérabilité contenue dans le programme utilisé pour générer ces binaires pourrait nous permettre d'assurer l'appel à la fonction *win* ?

MANIPULATION 4.1

Trouver l'adresse de la fonction *win* et les positions relatives des variables de la fonction qui référence *win* sur la pile.

QUESTION 4.3 (3 PTS)

Dessiner la stack frame de la fonction dont les vulnérabilités pourraient permettre d'invoquer la fonction *win*. Pour déterminer les valeurs des éléments sur la pile positionnez-vous juste avant les instructions de clôture du frame et utilisez le jeu de données fournies dans le fichier *inChall2_001.dat*.

MANIPULATION 4.2

Sur la base des informations qui précèdent, exploiter la vulnérabilité dans *chall2.32* ET *chall2.64* pour arriver dans la fonction *win*.

QUESTION 4.4 (3 PTS)

Présentez le payload que vous avez construit à l'étape précédente en séparant chaque élément qui le compose et en indiquant son rôle dans l'exploit.

QUESTION 4.5 (2 PTS)

Quel flag vous a été affiché par chacun des 2 binaires *chall2.32* ET *chall2.64* ?

QUESTION 4.6 (2 PTS)

Quelle vulnérabilité contenue dans le programme utilisé pour générer ces binaires pourrait nous permettre d'exécuter l'un des shellcodes fournis dans *bash_shellcode.c* et *lsla_shellcode.c* ? Lequel pour chacun des binaires *chall2.32* ET *chall2.64*, et pourquoi ?

MANIPULATION 4.3

Sur la base des informations qui précèdent, exploiter la vulnérabilité pour ouvrir le shellcode choisi sur chacun des binaires *chall2.32* ET *chall2.64* **via GDB**

QUESTION 4.7 (3 PTS)

Présentez les payloads que vous avez construits à l'étape précédente en séparant chaque élément qui le compose et en indiquant son rôle dans l'exploit.

QUESTION 4.8 (2 PTS)

Quelle vulnérabilité contenue dans le programme utilisé pour générer ces binaires pourrait nous permettre d'exécuter les mêmes shellcodes que précédemment, **sans passer par GDB** ?

MANIPULATION 4.4

Sur la base des informations qui précèdent, exploiter la vulnérabilité pour ouvrir le shellcode choisi sur chacun des binaires *chall2.32* ET *chall2.64* **sans avoir recours à un utilitaire d'analyse du binaire.**

AIDE DE MANIPULATION 4.4

Vous pouvez désactiver l'ASLR (la randomisation de l'espace adressable) via le script fourni pour les exercices *noaslr*.

QUESTION 4.9 (3 PTS)

Présentez le payload que vous avez construit à l'étape précédente en séparant chaque élément qui le compose et en indiquant son rôle dans l'exploit.

5 Windows – 32 bits (18 points)

Binaires : *mediacoder.exe*

Il s'agit de la version 0.7.5.4796 du logiciel MediaCoder (https://www.mediacoderhq.com/). Cette version présente une vulnérabilité de type buffer overflow qui peut être exploitée comme indiqué ici : https://www.exploit-db.com/exploits/15663.

L'installeur *MediaCoder 0.7.5.4796.exe* vous est fourni. Notez qu'il peut aussi être téléchargé depuis le site d'exploit-db. MediaCoder. Il doit être installé dans une **VM Windows 10 x86**.

Le script d'exploit python fourni sur exploit-DB ne doit pas être réutilisé tel quel. En particulier, il faut **remplacer le shellcode par celui qui vous est indiqué ci-après**.

MANIPULATION 5.1

Installez MediaCoder avec l'installeur fourni, puis exécutez-le et faites-le « crasher » en générant un fichier de playlist avec le script fourni : *exploitMediacoder07_crash.py***.**

AIDE DE MANIPULATION 5.1

Il faut ignorer l'avertissement de Windows SmartScreen à l'installation.

Il faut créer un fichier *rss.xml* vide pour pouvoir passer le contrôle de mise à jour au lancement de MediaCoder :
$ type null > "%APPDATA%\Broad Intelligence\MediaCoder\rss.xml"

Une fois ouvert, MediaCoder est accessible depuis le « tray » de la barre de tâches.

Attention, après une attaque, il se peut que MediaCoder conserve votre payload dans *queue.xml*. Dans ce cas, il vous faut détruire ce fichier pour pouvoir à nouveau exécuter MediaCoder. Sinon, il recharge

automatiquement votre payload au démarrage ;(:
$ del "%APPDATA%\Broad Intelligence\MediaCoder\queue.xml"

MANIPULATION 5.2

Reproduisez et analysez le crash via Immunity.

QUESTION 5.1 (3 PTS)

Expliquez les valeurs des registres EIP et ESP. D'où viennent-elles précisément et comment se sont-elles retrouvées là ?

MANIPULATION 5.3

Concevez et passez à MediaCoder un payload qui vous permette d'écraser le contenu du registre EIP et ce vers quoi pointe le contenu du registre ESP avec respectivement "BBBB" et "CCCC".

QUESTION 5.2 (3 PTS)

Présentez le payload que vous avez construit à l'étape précédente en séparant chaque élément qui le compose et en indiquant son rôle dans l'exploit.

MANIPULATION 5.4

Fort des manipulations et réponses précédentes, exploitez maintenant la vulnérabilité de buffer overflow mise en évidence pour ouvrir *calc.exe*.

AIDE DE MANIPULATION 5.4

Utilisez le shellcode fourni dans *winexecCalcShellcode.c*.

QUESTION 5.3 (2 PTS)

Quel mécanisme utilisant la stack avez-vous abusé ? Comment utilise-t-il la stack ?

QUESTION 5.4 (10 PTS)

Présentez le payload que vous avez construit à l'étape précédente en séparant chaque élément qui le compose et en indiquant :

- à quoi il sert dans l'exploit
- sa valeur
- comment vous avez prévu/obtenu cette valeur
- comment cette valeur va être utilisée par le programme en cours d'exécution

6 Défense Linux – 64 bits (20 pts)

Binaire : *chall3*

NB : l'ASLR ne doit pas être désactivé pour cet exploit.

MANIPULATION 4.1

Analysez le binaire avec Ghidra.

QUESTION 6.1 (2 PTS)

chall3 a été compilé à partir d'une version « étendue » de *chall2.c*. Quelles sont les nouvelles fonctions ? Qu'ajoutent-elles fonctionnellement ?

QUESTION 6.2 (2 PTS)

Quelles protections contre les attaques par buffer overflow sont actives dans *chall3* ? Lesquelles sont inactives ? Expliquez l'effet des protections actives sur le code généré de la fonction vulnérable identifiée pour *chall2*.

QUESTION 6.3 (2 PTS)

Comment les nouvelles fonctionnalités vont-elles vous permettre de tenter de déjouer l'ASLR ainsi que les protections du binaire ? De quel outil de scripting vu en cours allez-vous avoir besoin et pourquoi ?

MANIPULATION 6.2

Sur la base des informations qui précèdent, exploiter la vulnérabilité pour ouvrir un shell avec le shellcode *bash_shellcode64.c* fourni.

QUESTION 6.4 (8 PTS)

Présentez les différents payloads que vous avez construits à l'étape précédente. Pour chacun, expliquez l'objectif poursuivi ainsi que les éléments qui les compose en indiquant leur rôle dans l'exploit.

QUESTION 6.5 (6 PTS)

Présentez le/s script/s réalisé/s pour être en mesure de construire et passer les payloads précédemment décrits à *chall3* jusqu'à obtenir un shell fonctionnel. Numérotez les lignes de votre code et indiquez en légende ce que font les lignes essentielles du script.

7 Défense Linux – 32 bits (16 pts)

Binaire : *chall4*

Ce binaire est largement repris de l'exercice *buffer2* vu en cours. Nous allons l'utiliser comme décrit dans le tutoriel https://www.0x0ff.info/2021/advanced-buffer-overflow-bypass-aslr-32-bits/ (dont le langage « fleuri » n'engage que son auteur). L'objectif consiste à implémenter la démarche décrite dans ce lien de manière à vérifier la faible entropie de la randomisation en 32 bits.

MANIPULATION 7.1

Analysez *chall4* avec Ghidra et adaptez l'exploit présenté en cours pour *buffer2* sans l'ASLR. Utilisez le shellcode fourni dans *bash_shellcode32.c*.

QUESTION 7.1 (1 PT)

Quelle différence principale avez-vous notée entre les binaires *chall4* et *buffer2* ?

QUESTION 7.2 (1 PT)

Expliquez la modification que vous avez faite sur le payload prévu pour *buffer2* afin d'exploiter *chall4*.

MANIPULATION 7.2

A l'aide des informations fournies par le tutoriel, concevez un script capable de calculer « l'adresse **médiane** » du buffer vulnérable de *chall4*.

QUESTION 7.3 (2 PTS)

Quelle valeur médiane avez-vous trouvé ? Justifiez rapidement cette valeur au vu de l'échantillon collecté.

QUESTION 7.4 (2 PTS)

Présentez le/s script/s réalisé/s. Numérotez les lignes de votre code et indiquez en légende ce que font les lignes essentielles du script.

MANIPULATION 7.3

Ecrivez maintenant un script qui exploite en boucle *chall4* en espérant que le buffer vulnérable finira bien par apparaître à une adresse proche de la médiane précédemment calculée, cette fois avec l'ASLR.

Aide de manipulation 7.3

Votre payload doit minimiser les risques de tomber à une adresse trop éloignée.

Soyez un peu patient…

QUESTION 7.5 (2 PTS)

Votre attaque a-t-elle finalement réussie ? Combien de fois et pendant combien de temps avez-vous dû envoyer votre payload.

QUESTION 7.6 (2 PTS)

Comment avez-vous optimisé votre payload pour cette attaque ? Estimez empiriquement l'ordre de grandeur de réduction de l'espace à bruteforcer grâce à cette optimistation.

MANIPULATION 7.4

Utilisez maintenant l'astuce mentionnée (sans explication) dans le tutoriel pour mettre votre payload ailleurs en mémoire, là où vous serez plus libre de l'optimiser pour le bruteforce. Adaptez votre script qui exploite en boucle *chall4* afin de vérifier l'efficacité de l'astuce.

Aide de manipulation 7.4

Les variables d'environnement sont adressables ainsi dans GDB :
*(gdb) x/100s *((char **)environ)*

QUESTION 7.7 (2 PTS)

Votre attaque a-t-elle finalement réussie ? Combien de fois et pendant combien de temps avez-vous dû envoyer votre payload.

QUESTION 7.8 (2 PTS)

Comment avez-vous optimisé votre payload pour cette attaque ? Estimez empiriquement l'ordre de grandeur de réduction de l'espace à bruteforcer.

QUESTION 7.9 (2 PTS)

En conclusion de ce qui précède, dans quels cas jugez-vous la protection apportée par l'ASLR en 32 bits insuffisante et pourquoi ?

8 Défense Linux –32 bits (27 pts)

Binaire : *chall5*

MANIPULATION 8.1

Afficher les protections sur ce binaire à l'aide de l'utilitaire *checksec.sh* (http://www.trapkit.de/tools/checksec.html).

QUESTION 8.1 (1 PT)

Quelles protections vues en cours contre les attaques par buffer overflow sont actives dans *chall5* ?

QUESTION 8.2 (3 PTS)

Dessiner la stackframe de la fonction *secure_login*.

QUESTION 8.3 (1 PT)

Que veut-on écraser avec l'adresse de la fonction *win* ? Pour obtenir quel effet ? Quelle vulnérabilité permet de le faire ?

MANIPULATION 8.2

Lancez une attaque de type stack smashing pour démontrer l'efficacité de la protection discutée en 4.1.

QUESTION 8.4 (2 PTS)

Présentez le payload que vous avez construit à l'étape précédente. Expliquez l'objectif poursuivi ainsi que les éléments qui les compose en indiquant leur rôle dans l'exploit.

QUESTION 8.5 (2 PTS)

Quelle autre valeur que l'adresse de la fonction *win* doit-on connaitre afin d'effectuer l'attaque ? Quelle(s) vulnérabilité(s) permet(tent) d'obtenir cette valeur sans besoin d'avoir recours à un debugger ?

MANIPULATION 8.3

Trouver la valeur en question lors d'une première exécution du programme, puis utilisez-la pour essayer d'accéder à la fonction *win* lors d'une seconde exécution.

QUESTION 8.6 (4 PTS)

Présentez les différents payloads que vous avez construits à l'étape précédente. Pour chacun, expliquez l'objectif poursuivi ainsi que les éléments qui les compose en indiquant leur rôle dans l'exploit.

QUESTION 8.7 (2 PTS)

La manipulation précédente vous a-t-elle permis d'exécuter la fonction *win* ? Pourquoi ?

MANIPULATION 8.4

Sur la base de toutes ces informations, exploiter la vulnérabilité pour exécuter la fonction *win*.

QUESTION 8.8 (2 PTS)

Présentez les modifications que vous avez faites sur vos différents payloads. Pour chacune, expliquez l'objectif poursuivi ainsi que l'effet de la modification sur l'exploit.

QUESTION 8.9 (6 PTS)

Présentez le/s script/s réalisé/s pour être en mesure de construire et passer les payloads précédemment décrits à *chall5* jusqu'à pouvoir exécuter la fonction *win*. Numérotez les lignes de votre code et indiquez en légende ce que font les lignes essentielles du script.

QUESTION 8.10 (2 PTS)

Si vous n'avez pas anticipé sur cette question, vous ne devriez pas avoir réussir à faire afficher son flag par la fonction *win* ? Pourquoi ?

MANIPULATION 8.5

Sur la base de cette nouvelle constatation, exploiter la vulnérabilité pour exécuter la fonction *win* lui faire afficher son flag.

QUESTION 8.11 (2 PTS)

Expliquez les modifications réalisées sur les différents payloads concernés. Pour chacune, expliquez l'objectif poursuivi ainsi que l'effet de la modification sur l'exploit.