TP 3 : Exploitation avec *Metasploit*

Romain CARRÉ
romain.carre@cea.fr

L'objectif de ce TP est de suivre pas à pas les étapes d'une **exploitation de *buffer overflow***, en s'aidant de l'outil *Metasploit*. Contrairement au TP précédent où l'exploitation était réalisée à la main (recherche de vulnérabilité, écriture de l'exploit, du payload, et exécution avec récupération éventuelle d'un shell), nous utiliserons ici un framework développé en Ruby pour gagner en temps et en modularité / réutilisabilité.

Bien que les durées indiquées dans le titre de chacune des parties soient approximatives, elles peuvent vous donner une idée du temps attendu pour réaliser les différentes tâches mentionnées. Si vous êtes totalement coincés sur une question, ou une procédure particulière du TP, inutile d'attendre 1h avant de venir me le signaler : faites moi signe !

1 Environnement de travail (∼10min)

- installez Metasploit Community à l'aide de la procédure communiquée par mail
- munissez-vous également d'un interpréteur de scripts comme `python` ou `perl`
- installez le client réseau `nc`. à quoi va-t-il nous servir ? comment l'utilise-t-on ?
- récupérez le fichier `netvuln.c` sur la clef USB de votre intervenant
- désactivez la randomisation d'adresses grâce à la commande suivante (en root) :

```
echo 0 > /proc/sys/kernel/randomize_va_space
```

2 Détection et première exploitation (∼20min)

- parcourez le code source de `netvuln.c`, et mettez en évidence la vulnérabilité
- compilez le code en lançant la commande suivante. expliquez chaque paramètre

```
gcc -m32 -fno-stack-protector -mpreferred-stack-boundary=2 -z execstack netvuln.c
```

- `gcc` vous prévient-il avec un *warning* comme dans le TP précédent ? pourquoi ?
- testez le programme normalement avec la commande suivante :

```
nc 127.0.0.1 11222
```

- vérifiez qu'il est vraiment vulnérable en provoquant une erreur de segmentation

3 Exploitation *proof-of-concept* (∼30min)

- faites crasher le programme à nouveau, avec une entrée spécialement formatée pour déterminer la longueur du *payload* nécessaire à l'écrasement d'`eip`
- que pouvez-vous, au passage, en déduire sur la manière dont le compilateur a placé les variables locales sur la pile, par rapport à l'ordre dans lequel elles sont déclarées et initialisées dans le code source ? est-ce prédictible ?
- formalisez la structure des données qu'il faudra donc envoyer au programme (on placera le shellcode à la fin du payload pour éviter les problèmes de taille)
- en vous servant du TP précédent, faites en sorte que, suite à une exploitation réussie, le programme ne plante pas et retourne 42

4 Exploitation avec *Metasploit* (∼60min)

- en vous inspirant d'un module d'exploitation déjà écrit (vous les trouverez dans `apps/pro/msf3/modules/exploits`, en particulier d'autres bof pour linux), créez un fichier `~/.msf4/modules/exploits/netvuln_buffer_overflow.rb` qui servira à exploiter le programme `netvuln.c`. il devra contenir les en-têtes habituels pour un module d'exploitation *Metasploit*, et une implémentation de la fonction `exploit` (au minimum). pourquoi ne peut-on pas écrire de fonction `check` ici ?
- lancez *Metasploit*, et chargez votre exploit avec la commande `use`
- définissez les variables nécessaires à l'exploitation avec la commande `set` (vous pouvez voir un aperçu des options avec `show options`)
- choisissez l'encodeur `generic/none`. à quoi sert un encodeur ? dans un contexte `use payload`, générez un payload avec d'autres caractères interdits. jouez avec les différentes options de la sous-commande `generate`
- exploitez le binaire en lui faisant forker un shell avec le payload `linux/x86/exec`
- exploitez le binaire en lui faisant lire et afficher le fichier de votre choix avec le payload `linux/x86/read_file`. essayez de lire le fichier `/etc/shadow`. que remarquez-vous ? comment l'expliquer ?
- exploitez le binaire en lui faisant binder un port avec un shell derrière, avec le payload `linux/x86/shell_bind_tcp`. expliquez la différence en terme d'exploitation pour un attaquant. selon vous, est-il toujours possible d'exploiter un binaire de cette façon en pratique ? comment y remédiriez-vous dans ce cas ?
- jouez un peu avec différents payloads, différents encodeurs
- faites tourner le daemon vulnérable sous l'identité d'un compte utilisateur spécifique et non privilégié, et exploitez les binaires de vos camarades à travers le réseau.

Question subsidiaire : en tant qu'administrateur éclairé, quels mécanismes mettriez-vous en place pour limiter les conséquences d'une exploitation sur votre machine ?

5 Correction et questions (∼60min)

6 Annexes

Commandes msfconsole utiles :

`use <module>`	charge le module `<module>`
`set <var> <value>`	attribue la valeur `<value>` à la variable `<var>`
`show <data>`	affiche des détails sur les données `<data>`

En Ruby pur ou API Metasploit :

`connect`	établit la connexion avec l'hôte distant
`rand_text_alphanumeric`	génère une chaîne aléatoire de taille spécifiée
`sock.put`	envoie des données sur le socket ouvert
`disconnect`	met fin à la connexion avec l'hôte distant