WYDANIE II

Black Hat Python

Język Python dla hakerów i pentesterów

Justin Seitz, Tim Arnold

Helion

no starch press

Tytuł oryginału: Black Hat Python, 2nd Edition: Python Programming for Hackers and Pentesters

Tłumaczenie: Andrzej Watrak
z wykorzystaniem fragmentów poprzedniego wydania w przekładzie Łukasza Piwki

ISBN: 978-83-283-8345-6

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/bhpyth
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/bhpyth.zip

Printed in Poland.

Spis treści

11

Ofensywna analiza śledcza

Śledczy często są wzywani, gdy dojdzie do złamania zabezpieczeń, albo w celu sprawdzenia, czy pewne „zdarzenie" w ogóle miało miejsce. Najczęściej żądają zrzutu zawartości pamięci RAM komputera, aby pobrać z niego klucze kryptograficzne i inne informacje, które są przechowywane tylko w tej pamięci. Specjaliści ci są szczęściarzami, ponieważ pewien zespół utalentowanych programistów utworzył cały Pythonowy system szkieletowy o nazwie **Volatility** służący właśnie do wykonywania tego typu czynności i opisywany jako zaawansowany system do badania zawartości pamięci na potrzeby śledztw. Specjaliści od odpierania ataków, śledczy informatyczni i analitycy złośliwego oprogramowania wykorzystują Volatility do wielu różnych celów, np. badania obiektów jądra, badania i zrzucania procesów itd.

Volatility jest wprawdzie oprogramowaniem do zastosowań obronnych, ale jest na tyle wszechstronne, że może być używane zarówno ofensywnie, jak i defensywnie. Wykorzystamy je do rozpoznania docelowego systemu. Napiszemy również ofensywne wtyczki wyszukujące niedostatecznie zabezpieczone procesy uruchomione na maszynie wirtualnej.

Wyobraźmy sobie, że badamy maszynę wirtualną i odkrywamy, że użytkownik przetwarza na niej poufne dane. Prawdopodobnie na wypadek ewentualnych awarii zrobił migawkę (ang. *snapshot*) tej maszyny. Wykorzystamy narzędzie Volatility do przeanalizowania migawki i wyszukania w niej uruchomionych procesów. Sprawdzimy też, czy w zabezpieczeniach są luki, które moglibyśmy wykorzystać.

Zaczynajmy!

Instalacja

Narzędzie Volatility ma już kilka lat i niedawno zostało całkowicie odmienione. Nie tylko jego kod został przystosowany do wersji Python 3, ale została zmieniona cała jego struktura, dzięki czemu poszczególne komponenty są od siebie niezależne. Wszystkie dane niezbędne do uruchomienia wtyczki są zawarte w niej samej.

Utwórzmy środowisko wirtualne przeznaczone wyłącznie dla Volatility. W tym przykładzie będziemy używać Pythona 3 w konsoli PowerShell systemu Windows. Sprawdź, czy zainstalowany jest program `git`. Jeżeli nie, możesz go pobrać ze strony *https://git-scm.com/downloads*.

```
PS> python3 -m venv vol3 ❶
PS> vol3/Scripts/Activate.ps1
PS> cd vol3/
PS> git clone https://github.com/volatilityfoundation/volatility3.git ❷
PS> cd volatility3/
PS> python setup.py install
PS> pip install pycryptodome ❸
```

Najpierw tworzymy środowisko wirtualne o nazwie `vol3` i aktywujemy je ❶. Następnie przechodzimy do katalogu nowego środowiska, pobieramy kopię programu Volatility 3 z repozytorium GitHub i instalujemy go ❷. Na koniec instalujemy pakiet `pycryptodome`, którego użyjemy później ❸.

Aby wyświetlić dostępne wtyczki i opcje narzędzia Volatility w systemie Windows, użyj poniższego polecenia:

```
PS> vol --help
```

W systemach Linux i macOS użyj w tym celu interpretera Pythona:

```
$> python vol.py --help
```

W tym rozdziale będziemy używać narzędzia Volatility w wierszu poleceń, ale istnieją też inne sposoby. Na przykład na stronie *https://github.com/volatilityfoundation/volumetric* znajduje się projekt Volumetric zawierający przeglądarkowy interfejs graficzny narzędzia. Jeżeli chciałbyś się dowiedzieć, jak używać narzędzia Volatility we własnych programach, zajrzyj do kodu źródłowego tego projektu. Oprócz tego jest dostępny interfejs `volshell` dający dostęp do narzędzia Volatility. Można go używać tak jak zwykłej interaktywnej powłoki Pythona.

W opisanych niżej przykładach będziemy korzystać z narzędzia Volatility w wierszu poleceń. Aby oszczędzić miejsce, prezentowane są tylko opisywane fragmenty wyników. Dlatego pamiętaj, że Twoje wyniki mogą zawierać więcej wierszy i kolumn.

Teraz przejrzyjmy dokładniej kod i wnętrze narzędzia:

```
PS> cd volatility/framework/plugins/windows/
PS> ls
_init__.py     driverscan.py   memmap.py        psscan.py   vadinfo.py
bigpools.py    filescan.py     modscan.py       pstree.py   vadyarascan.py
cachedump.py   handles.py      modules.py       registry/   verinfo.py
callbacks.py   hashdump.py     mutantscan.py    ssdt.py     virtmap.py
cmdline.py     info.py         netscan.py       strings.py
dlllist.py     lsadump.py      poolscanner.py   svcscan.py
driverirp.py   malfind.py      pslist.py        symlinkscan.py
```

Powyższy wynik przedstawia pliki Pythona zapisane w katalogu *plugins/windows*. Zachęcamy Cię do poświęcenia chwili na ich przejrzenie. Jak się przekonasz, są one zbudowane według pewnego schematu, właściwego dla wtyczki Volatility. Dzięki temu lepiej poznasz strukturę narzędzia, ale przede wszystkim będziesz miał wyobrażenie o strategii i założeniach systemu ochrony. Gdy będziesz znał możliwości tego systemu i sposób, w jaki osiąga swoje cele, staniesz się skuteczniejszym hakerem i będziesz potrafił lepiej zabezpieczać swoje skrypty przed wykryciem.

Po przejrzeniu plików narzędzia zajmijmy się analizą pamięci. Do tego celu najprościej będzie wykorzystać migawkę maszyny wirtualnej z systemem Windows 10.

Uruchom maszynę, a w niej kilka procesów, na przykład notatnik, kalkulator i przeglądarkę (będziemy badać zawartość pamięci i przebieg uruchamiania procesów). Następnie wykonaj migawkę maszyny, wykorzystując funkcjonalność hiperwizora. W katalogu, w którym znajdują się maszyny wirtualne, powinien się pojawić nowy plik z rozszerzeniem *.vmem* lub *.mem* zawierający migawkę. Sprawdźmy, co zawiera!

Wiele obrazów pamięci jest również dostępnych w internecie. W tym rozdziale jest wykorzystany jeden z nich, udostępniany przez firmę PassMark Software na stronie *https://www.osforensics.com/tools/volatility-workbench.html*. Kilka obrazów, z którymi możesz eksperymentować, oferuje również organizacja Volatility Foundation pod adresem *https://github.com/volatilityfoundation/volatility/wiki/Memory-Samples*.

Ogólny rekonesans

Przyjrzyjmy się ogólnie analizowanej maszynie. Za pomocą wtyczki `windows.info` możemy uzyskać informacje o systemie operacyjnym i jego jądrze:

```
PS>vol -f WinDev2007Eval-Snapshot4.vmem windows.info ❶
Volatility 3 Framework 1.2.0-beta.1
Progress:   33.01               Scanning primary2 using PdbSignatureScanner
Variable        Value
```

```
Kernel Base     0xf80067a18000
DTB             0x1aa000
primary 0       WindowsIntel32e
memory_layer    1 FileLayer
KdVersionBlock  0xf800686272f0
Major/Minor     15.19041
MachineType     34404
KeNumberProcessors      1
SystemTime      2020-09-04 00:53:46
NtProductType   NtProductWinNt
NtMajorVersion  10
NtMinorVersion  0
PE MajorOperatingSystemVersion  10
PE MinorOperatingSystemVersion  0
PE Machine      34404
```

W poleceniu użyliśmy nazwy migawki, parametru `-f` oraz nazwy wtyczki `windows.info` ❶. Narzędzie Volatility odczytuje i analizuje plik migawki, po czym wyświetla ogólne informacje o maszynie wirtualnej. W tym przypadku jest to maszyna z systemem Windows 10.0, wyposażona w jeden procesor i jedną warstwę pamięci.

Kształcącym doświadczeniem może być przetestowanie pliku migawki z różnymi wtyczkami podczas przeglądania ich kodów. Porównując kod z uzyskiwanymi wynikami, będziesz mógł się dowiedzieć, jak działa wtyczka i jaka jest ogólna strategia systemu ochrony.

Teraz użyj wtyczki `registry.printkey` do wyświetlenia kluczy rejestru i ich wartości. Rejestr zawiera mnóstwo informacji, a Volatility oferuje funkcjonalność wyszukiwania potrzebnych danych. Najpierw przejrzyjmy zainstalowane usługi. Klucz */ControlSet001/Services* zawiera bazę danych menedżera usług, czyli ich listę:

```
PS>vol -f WinDev2007Eval-7d959ee5.vmem windows.registry.printkey --key
↪'ControlSet001\Services'
Volatility 3 Framework 1.2.0-beta.1
Progress:   33.01               Scanning primary2 using PdbSignatureScanner
... Key                                         Name    Data       Volatile
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services .NET CLR Data      False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services Appinfo            False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services applockerfltr      False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services AtomicAlarmClock   False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services Beep               False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services fastfat            False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services MozillaMaintenance False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services NTDS               False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services Ntfs               False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services ShellHWDetection   False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services SQLWriter          False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services Tcpip              False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services Tcpip6             False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services terminpt           False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services W32Time            False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services WaaSMedicSvc       False
```

```
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services WacomPen          False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services Winsock           False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services WinSock2          False
\REGISTRY\MACHINE\SYSTEM\ControlSet001\Services WINUSB            False
```

Uzyskany wynik przedstawia listę wszystkich zainstalowanych usług (częściową, aby oszczędzić miejsce).

Rekonesans użytkowników

Teraz przyjrzyjmy się użytkownikom maszyny. Wtyczka `cmdline` wyświetla parametry wszystkich procesów działających w chwili wykonania migawki. Procesy te dają wyobrażenie o aktywności i zamiarach użytkownika.

```
PS>vol -f WinDev2007Eval-7d959ee5.vmem windows.cmdline
Volatility 3 Framework 1.2.0-beta.1
Progress:   33.01               Scanning primary2 using PdbSignatureScanner
PID    Process         Args
72     Registry        Required memory at 0x20 is not valid (process exited?)
340    smss.exe        Required memory at 0xa5f1873020 is inaccessible (swapped)
564    lsass.exe       C:\Windows\system32\lsass.exe
624    winlogon.exe    winlogon.exe
2160   MsMpEng.exe     "C:\ProgramData\Microsoft\Windows Defender\platform\4.18.2008.9-0\MsMpEng.exe"
4732   explorer.exe    C:\Windows\Explorer.EXE
4848   svchost.exe     C:\Windows\system32\svchost.exe -k ClipboardSvcGroup -p
4920   dllhost.exe     C:\Windows\system32\DllHost.exe /Processid:{AB8902B4-09CA-4BB6-B78DA8F59079A8D5}
5084   StartMenuExper  "C:\Windows\SystemApps\Microsoft.Windows. . ."
5388   MicrosoftEdge.  "C:\Windows\SystemApps\Microsoft.MicrosoftEdge_. . ."
6452   OneDrive.exe    "C:\Users\Administrator\AppData\Local\Microsoft\OneDrive\OneDrive.exe" /background
6484   FreeDesktopClo  "C:\Program Files\Free Desktop Clock\FreeDesktopClock.exe"
7092   cmd.exe         "C:\Windows\system32\cmd.exe" ❶
3312   notepad.exe     notepad ❷
3824   powershell.exe  "C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe"
6448   Calculator.exe  "C:\Program Files\WindowsApps\Microsoft.WindowsCalculator_. . ."
6684   firefox.exe     "C:\Program Files (x86)\Mozilla Firefox\firefox.exe"
6432   PowerToys.exe   "C:\Program Files\PowerToys\PowerToys.exe"
7124   nc64.exe        Required memory at 0x2d7020 is inaccessible (swapped)
3324   smartscreen.ex  C:\Windows\System32\smartscreen.exe -Embedding
4768   ipconfig.exe    Required memory at 0x840308e020 is not valid (process exited?)
```

Uzyskana lista zawiera identyfikatory procesów, ich nazwy i parametry użyte w wierszu poleceń podczas uruchamiania. Jak widać, większość procesów została uruchomiona przez system operacyjny, prawdopodobnie podczas jego startu. Procesy `cmd.exe` ❶ i `notepad.exe` ❷ prawdopodobnie uruchomił użytkownik.

Zbadajmy dokładniej za pomocą wtyczki `pslist` uruchomione procesy. Wtyczka ta wyświetla listę procesów działających w chwili utworzenia migawki.

```
PS>vol -f WinDev2007Eval-7d959ee5.vmem windows.pslist
Volatility 3 Framework 1.2.0-beta.1
Progress:  33.01               Scanning primary2 using PdbSignatureScanner
PID     PPID    ImageFileName   Offset(V)    Threads Handles SessionId   Wow64
4       0       System          0xa50bb3e6d040  129     -       N/A     False
72      4       Registry        0xa50bb3fbd080  4       -       N/A     False
6452    4732    OneDrive.exe    0xa50bb4d62080  25      -       1       True
6484    4732    FreeDesktopClo  0xa50bbb847300  1       -       1       False
6212    556     SgrmBroker.exe  0xa50bbb832080  6       -       0       False
1636    556     svchost.exe     0xa50bbadbe340  8       -       0       False
7092    4732    cmd.exe         0xa50bbbc4d080  1       -       1       False
3312    7092    notepad.exe     0xa50bbb69a080  3       -       1       False
3824    4732    powershell.exe  0xa50bbb92d080  11      -       1       False
6448    704     Calculator.exe  0xa50bb4d0d0c0  21      -       1       False
4036    6684    firefox.exe     0xa50bbb178080  0       -       1       True
6432    4732    PowerToys.exe   0xa50bb4d5a2c0  14      -       1       False
4052    4700    PowerLauncher.  0xa50bb7fd3080  16      -       1       False
5340    6432    Microsoft.Powe  0xa50bb736f080  15      -       1       False
8564    4732    python-3.8.6-a  0xa50bb7bc2080  1       -       1       True
7124    7092    nc64.exe        0xa50bbab89080  1       -       1       False
3324    704     smartscreen.ex  0xa50bb4d6a080  7       -       1       False
7364    4732    cmd.exe         0xa50bbd8a8080  1       -       1       False
8916    2136    cmd.exe         0xa50bb78d9080  0       -       0       False
4768    8916    ipconfig.exe    0xa50bba7bd080  0       -       0       False
```

Powyższy wynik zawiera nazwy procesów i ich adresy w pamięci. Kilka kolumn zostało usuniętych, aby oszczędzić miejsce. Wymienionych jest kilkanaście interesujących procesów, m.in. `cmd.exe` i `notepad.exe`, które również zostały wykryte przez wtyczkę `cmdline`.

Warto byłoby znać hierarchię procesów i wiedzieć, który z nich uruchomił inne procesy. Tego rodzaju informacji dostarcza wtyczka `pstree`:

```
PS>vol -f WinDev2007Eval-7d959ee5.vmem windows.pstree
Volatility 3 Framework 1.2.0-beta.1
Progress:  33.01               Scanning primary2 using PdbSignatureScanner
PID       PPID   ImageFileName   Offset(V)    Threads Handles SessionId Wow64
4           0    System          0xa50bba7bd080 129     -       N/A     False
* 556      492   services.exe    0xa50bba7bd080  8      -       0       False
** 2176    556   wlms.exe        0xa50bba7bd080  2      -       0       False
** 1796    556   svchost.exe     0xa50bba7bd080  13     -       0       False
** 776     556   svchost.exe     0xa50bba7bd080  15     -       0       False
** 8       556   svchost.exe     0xa50bba7bd080  18     -       0       False
*** 4556    8   ctfmon.exe      0xa50bba7bd080  10     -       1       False
*** 5388   704   MicrosoftEdge.  0xa50bba7bd080  35     -       1       False
*** 6448   704   Calculator.exe  0xa50bba7bd080  21     -       1       False
*** 3324   704   smartscreen.ex  0xa50bba7bd080  7      -       1       False
** 2136    556   vmtoolsd.exe    0xa50bba7bd080  11     -       0       False
*** 8916  2136   cmd.exe         0xa50bba7bd080  0      -       0       False
**** 4768 8916   ipconfig.exe    0xa50bba7bd080  0      -       0       False
* 4704     624   userinit.exe    0xa50bba7bd080  0      -       1       False
** 4732   4704   explorer.exe    0xa50bba7bd080  92     -       1       False
*** 6432  4732   PowerToys.exe   0xa50bba7bd080  14     -       1       False
**** 5340 6432   Microsoft.Powe  0xa50bba7bd080  15     -       1       False
```

```
*** 7364  4732   cmd.exe          0xa50bba7bd080   1       -          -     False
**** 2464 7364   conhost.exe      0xa50bba7bd080   4       -          1     False
*** 7092  4732   cmd.exe          0xa50bba7bd080   1       -          -     False
**** 3312 7092   notepad.exe      0xa50bba7bd080   3       -          1     False
**** 7124 7092   nc64.exe         0xa50bba7bd080   1       -          1     False
*** 8564  4732   python-3.8.6-a  0xa50bba7bd080   1       -          1     True
**** 1036 8564   python-3.8.6-a  0xa50bba7bd080   5       -          1     True
```

Obraz procesów jest teraz wyraźniejszy. Gwiazdki widoczne w każdym wierszu symbolizują zależności rodzic – dziecko między procesami. Na przykład proces `userinit.exe` (PID 4704) uruchomił proces `explorer.exe`. Z kolei `explorer.exe` (PID 4732) uruchomił proces `cmd.exe` (PID 7092), a ten procesy `notepad.exe` i `nc64.exe`.

Teraz użyjmy wtyczki `hashdump` do przejrzenia haseł:

```
PS> vol -f WinDev2007Eval-7d959ee5.vmem windows.hashdump
Volatility 3 Framework 1.2.0-beta.1
Progress:   33.01               Scanning primary2 using PdbSignatureScanner
User                rid     lmhash                  nthash
Administrator       500     aad3bXXXXXXaad3bXXXXXX  fc6eb57eXXXXXXXXXXX657878
Guest               501     aad3bXXXXXXaad3bXXXXXX  1d6cfe0dXXXXXXXXXXXc089c0
DefaultAccount      503     aad3bXXXXXXaad3bXXXXXX  1d6cfe0dXXXXXXXXXXXc089c0
WDAGUtilityAccount 504      aad3bXXXXXXaad3bXXXXXX  ed66436aXXXXXXXXXX1bb50f
User               1001     aad3bXXXXXXaad3bXXXXXX  31d6cfe0XXXXXXXXXXXc089c0
tim                1002     aad3bXXXXXXaad3bXXXXXX  afc6eb57XXXXXXXXXX657878
admin              1003     aad3bXXXXXXaad3bXXXXXX  afc6eb57XXXXXXXXXX657878
```

Wynik prezentuje nazwy użytkowników oraz ich hasła zaszyfrowane za pomocą algorytmów LW i NT. Odtworzenie haseł w systemie Windows po uzyskaniu do niego dostępu jest częstym celem hakerów. Zaszyfrowane hasła można spróbować odszyfrować off-line lub wykorzystać do uzyskania dostępu do innych zasobów w sieci. Jeżeli użytkownik jest obsesyjnie ostrożny i ryzykowne operacje wykonuje tylko na maszynie wirtualnej albo firma udostępnia niektórym użytkownikom wyłącznie maszyny, to analiza migawek po uzyskaniu dostępu do hiperwizora jest doskonałą okazją, aby spróbować odtworzyć zaszyfrowane hasła. Narzędzie Volatility jest tu bardzo pomocne.

Zaszyfrowane hasła widoczne w powyższym wyniku zostały częściowo zatarte. Aby złamać hasło i uzyskać dostęp do maszyny wirtualnej, użyj własnych wyników. W internecie jest dostępnych wiele stron do odtwarzania haseł. Możesz również użyć narzędzia John the Ripper zawartego w systemie Kali.

Rekonesans słabych punktów

Użyjmy teraz narzędzia Volatility do sprawdzenia, czy maszyna wirtualna ma słabe punkty, które haker mógłby wykorzystać. Wtyczka `malfind` wyszukuje obszary pamięci zajmowanej przez procesy, które potencjalnie mogą zawierać wstrzyknięty kod. Kluczowe jest tu słowo „potencjalnie”: wtyczka wyszukuje obszary

pamięci, w których procesy mogą zapisywać dane, odczytywać je i uruchamiać kod. Warto zbadać dokładniej te procesy, ponieważ można je wykorzystać do uruchamiania wirusów, które są powszechnie dostępne. Ponadto w takich obszarach pamięci można umieszczać własne wirusy.

```
PS>vol -f WinDev2007Eval-7d959ee5.vmem windows.malfind
Volatility 3 Framework 1.2.0-beta.1
Progress:   33.01               Scanning primary2 using PdbSignatureScanner
PID  Process         Start VPN     End VPN       Tag  Protection             CommitCharge
1336 timeserv.exe    0x660000      0x660fff      VadS PAGE_EXECUTE_READWRITE 1
2160 MsMpEng.exe     0x16301690000 0x1630179cfff VadS PAGE_EXECUTE_READWRITE 269
2160 MsMpEng.exe     0x16303090000 0x1630318ffff VadS PAGE_EXECUTE_READWRITE 256
2160 MsMpEng.exe     0x16304a00000 0x16304bfffff VadS PAGE_EXECUTE_READWRITE 512
6484 FreeDesktopClo  0x2320000     0x232Offf     VadS PAGE_EXECUTE_READWRITE 1
5340 Microsoft.Powe  0x2c2502c0000 0x2c2502cffff VadS PAGE_EXECUTE_READWRITE 15
```

Powyższy wynik sygnalizuje kilka potencjalnych problemów. Proces `timeserv.exe` (PID 1336) jest częścią bezpłatnego oprogramowania FreeDesktopClock (PID 6484). Sam proces nie musi być problematyczny, jeżeli został zainstalowany w katalogu *C:\Program Files*. Jeżeli znajduje się w innym miejscu, może to być wirus ukrywający się pod nazwą legalnego programu.

Za pomocą wyszukiwarki możesz się dowiedzieć, że proces `MsMpEng.exe` (PID 2160) jest usługą antywirusową. Mimo że ma on uprawnienia do zapisywania danych w pamięci i uruchamiania kodu, nie wydaje się niebezpieczny. Może jednak stanowić zagrożenie, jeżeli w wykorzystywanych przez niego obszarach pamięci uda się umieścić własny kod. Dlatego warto przyjrzeć się mu bliżej.

Wtyczka `netscan` wyświetla listę wszystkich połączeń sieciowych, jakie maszyna wirtualna utrzymywała w chwili utworzenia migawki. Poniżej jest pokazany przykład. Wszystkie nietypowe połączenia można wykorzystać do przeprowadzenia ataku.

```
PS>vol -f WinDev2007Eval-7d959ee5.vmem windows.netscan
Volatility 3 Framework 1.2.0-beta.1
Progress:   33.01               Scanning primary2 using PdbSignatureScanner
Offset         Proto LocalAddr  LocalPort ForeignAdd ForeignPort State      PID Owner
0xa50bb7a13d90 TCPv4 0.0.0.0         4444 0.0.0.0          0     LISTENING 7124 nc64.exe ❶
0xa50bb9f4c310 TCPv4 0.0.0.0         7680 0.0.0.0          0     LISTENING 1776 svchost.exe
0xa50bb9f615c0 TCPv4 0.0.0.0        49664 0.0.0.0          0     LISTENING  564 lsass.exe
0xa50bb9f62190 TCPv4 0.0.0.0        49665 0.0.0.0          0     LISTENING  492 wininit.exe
0xa50bbaa80b20 TCPv4 192.168.28.128 50948 23.40.62.19     80     CLOSED ❷
0xa50bbabd2010 TCPv4 192.168.28.128 50954 23.193.33.57   443     CLOSED
0xa50bbad8d010 TCPv4 192.168.28.128 50953 99.84.222.93   443     CLOSED
0xa50bbaef3010 TCPv4 192.168.28.128 50959 23.193.33.57   443     CLOSED
0xa50bbaff7010 TCPv4 192.168.28.128 50950 52.179.224.121 443     CLOSED
0xa50bbbd240a0 TCPv4 192.168.28.128   139 0.0.0.0          0     LISTENING
```

Widocznych jest kilka połączeń pomiędzy lokalną maszyną (192.168.28.128) a prawdopodobnie serwerami WWW ❷. Połączenia te są zamknięte. Ważniejsze są połączenia oznaczone jako `LISTENING`. Jeżeli zostały nawiązanie przez procesy

systemu Windows (`svchost`, `lsass`, `wininit`), nie trzeba się nimi przejmować. Ale proces `nc64.exe` jest nietypowy ❶. Wykorzystuje port numer 4444 w trybie do nasłuchu, dlatego warto przyjrzeć się mu bliżej lub zbadać ten port za pomocą narzędzia `netcat` opisanego w rozdziale 2.

Interfejs volshell

Narzędzia Volatility można używać nie tylko w wierszu poleceń, ale również w powłoce Pythona. Jest to możliwe dzięki poleceniu `volshell`, które pozwala wykorzystać połączone siły narzędzia Volatility i powłoki Pythona. Poniższy listing przedstawia przykład użycia za pomocą polecenia `volshell` wtyczki `pslist` do zbadania obrazu systemu Windows.

```
PS> volshell -w -f WinDev2007Eval-7d959ee5.vmem ❶
>>> from volatility.plugins.windows import pslist ❷
>>> dpo(pslist.PsList, primary=self.current_layer, nt_symbols=self.config
↪['nt_symbols']) ❸
PID     PPID    ImageFileName   Offset(V)    Threads Handles SessionId Wow64
4       0       System          0xa50bb3e6d040  129     -       N/A   False
72      4       Registry        0xa50bb3fbd080    4     -       N/A   False
6452    4732    OneDrive.exe    0xa50bb4d62080   25     -       1     True
6484    4732    FreeDesktopClo  0xa50bbb847300    1     -       1     False
...
```

W powyższym przykładzie został użyty parametr `-w` oznaczający, że użyty plik jest obrazem maszyny z systemem Windows, oraz parametr `-f` określający ten plik ❶. Interfejs `volshell` jest bardzo podobny do powłoki Pythona. Można w nim w zwykły sposób importować pakiety i definiować funkcje, a oprócz tego korzystać z funkcjonalności narzędzia Volatility. W tym przykładzie zaimportowaliśmy wtyczkę `pslist` ❷ i wyświetliliśmy uzyskany za jej pomocą wynik (funkcja `dpo()`) ❸.

Więcej informacji na temat korzystania z interfejsu `volshell` uzyskasz za pomocą polecenia `volshell --help`.

Własne wtyczki dla Volatility

Wiesz już, jak używać standardowych wtyczek narzędzia Volatility do analizowania migawek maszyn wirtualnych, wyszukiwania słabych punktów, badania profili użytkowników, wyświetlania procesów i zaszyfrowanych haseł. Oprócz tego możesz tworzyć własne wtyczki. W efekcie możliwości narzędzia Volatility będzie ograniczać jedynie Twoja wyobraźnia. Jeżeli potrzebujesz dodatkowych informacji, poza dostarczanymi przez typowe wtyczki, zacznij tworzyć własne.

Autorzy narzędzia Volatility dołożyli starań, aby tworzenie wtyczek było proste. Należy przy tym stosować określony wzorzec. Niestandardowe wtyczki mogą również uruchamiać inne wtyczki, dzięki czemu zadanie jest jeszcze prostsze.

Podstawowy szkielet wtyczki wygląda następująco:

```
import ...

class CmdLine(interfaces.plugin.PluginInterface): ❶
    @classmethod
    def get_requirements(cls): ❷
        pass

    def run(self): ❸
        pass

    def generator(self, procs): ❹
        pass
```

Przede wszystkim musisz zdefiniować klasę pochodną od `PluginInterface` ❶, zawierającą metody `get_requirements()` ❷, `run()` ❸ i `generator()` ❹. Ostatnia metoda jest opcjonalna, ale warto rozdzielić kod między metody `run()` i `generator()`, jak to ma miejsce w wielu wtyczkach. Dzięki temu wyniki uzyskuje się szybciej, a kod jest czytelniejszy.

Zastosujemy teraz ten wzorzec do utworzenia własnej wtyczki wyszukującej procesy, które nie mają zabezpieczenia ASLR (ang. *Address Space Layout Randomization*, losowe rozmieszczanie w przestrzeni adresowej). Zabezpieczenie to polega na losowym przydzielaniu procesom przestrzeni adresowej i wpływa na położenie sterty, stosu i innych systemowych obszarów pamięci. W efekcie proces wirusa nie jest w stanie określić przestrzeni adresowej atakowanego procesu. Zabezpieczenie to wprowadzono w wersji systemu Windows Vista. W starszych wersjach, na przykład Windows XP, zabezpieczenie ASLR nie jest domyślnie włączone. W nowych systemach (m.in. Windows 10) niemal wszystkie procesy są chronione w ten sposób. Aktywne zabezpieczenie ASLR nie oznacza, że haker jest bezsilny, ale jego zadanie jest znacznie utrudnione.

Jako pierwszy krok w kierunku badania procesów utwórzmy wtyczkę, która będzie sprawdzać, czy procesy są chronione za pomocą ASLR. Utwórz katalog *plugins*, a w nim podkatalog *windows*, w którym będziesz umieszczał wtyczki przeznaczone dla systemu Windows. Dla systemu Linux lub macOS utwórz odpowiednio podkatalog *mac* lub *linux*.

Teraz w podkatalogu *plugins/windows* utwórz plik *aslrcheck.py* i wpisz w nim następujący kod:

```
# Wyszukanie wszystkich procesów i sprawdzenie zabezpieczenia ASLR
#
from typing import Callable, List

from volatility.framework import constants, exceptions, interfaces, renderers
from volatility.framework.configuration import requirements
from volatility.framework.renderers import format_hints
from volatility.framework.symbols import intermed
from volatility.framework.symbols.windows import extensions
from volatility.plugins.windows import pslist
```

```
import io
import logging
import os
import pefile

vollog = logging.getLogger(__name__)

IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE = 0x0040
IMAGE_FILE_RELOCS_STRIPPED = 0x0001
```

Na początku importujemy niezbędne pakiety oraz bibliotekę *pefile* służącą do analizowania plików PE (ang. *Portable Executable*, plik przenośny i wykonywalny).

Teraz napiszmy pomocniczą funkcję wykonującą właściwą analizę.

```
def check_aslr(pe): ❶
    pe.parse_data_directories(
           [pefile.DIRECTORY_ENTRY['IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG']]
    )
    dynamic = False
    stripped = False

    if pe.OPTIONAL_HEADER.DllCharacteristics & ❷
          IMAGE_DLL_CHARACTERISTICS_DYNAMIC_BASE:
        dynamic = True
    if pe.FILE_HEADER.Characteristics & IMAGE_FILE_RELOCS_STRIPPED: ❸
        stripped = True
    if not dynamic or (dynamic and stripped): ❹
        aslr = False
    else:
        aslr = True
    return aslr
```

Argumentem funkcji `check_aslr()` jest obiekt reprezentujący plik PE ❶. Funkcja analizuje zawartość pliku i sprawdza, czy został on skompilowany z opcją `DYNAMIC` ❷ oraz czy zostały usunięte dane relokacyjne ❸. Jeżeli powyższa opcja nie została użyta albo została, jednak brak jest danych relokacyjnych ❹, to znaczy, że dany proces nie jest chroniony zabezpieczeniem ASLR.

Teraz utwórzmy klasę `AslrCheck`:

```
class AslrCheck(interfaces.plugins.PluginInterface): ❶
    @classmethod
    def get_requirements(cls):
        return [
            requirements.TranslationLayerRequirement( ❷
                name='primary', description='Warstwa pamięci jądra',
                architectures=["Intel32", "Intel64"]),
            requirements.SymbolTableRequirement( ❸
                name="nt_symbols", description="Symbole jądra Windows"),
            requirements.PluginRequirement( ❹
                name='pslist', plugin=pslist.PsList, version=(1, 0, 0)),
```

```
            requirements.ListRequirement(name = 'pid', ❺
                element_type = int,
                description = "Identyfikatory uwzględnianych procesów (wszystkie
                ↪pozostałe są wykluczane)",
                optional = True),
            ]
```

Pierwszym krokiem w tworzeniu wtyczki jest zdefiniowanie klasy pochodnej od `PluginInterface` ❶. Następnie należy utworzyć metodę zwracającą wymagania. Dobrym sposobem określenia, co może być potrzebne, jest przejrzenie innych wtyczek. Każda wtyczka potrzebuje warstwy pamięci, dlatego ten wymóg jest zdefiniowany jako pierwszy ❷. Oprócz tego jest potrzebna tabela symboli ❸. Te dwa wymagania są stosowane niemal we wszystkich wtyczkach.

Niezbędna jest również wtyczka `pslist`, aby móc uzyskać listę wszystkich procesów umieszczonych w pamięci i odtworzyć plik PE na podstawie procesu ❹. Tak utworzony plik PE będziemy sprawdzać pod kątem zabezpieczenia ASLR.

Przydatna będzie też możliwość sprawdzania wybranych procesów określonych za pomocą identyfikatorów. W tym celu zdefiniujemy opcjonalne ustawienie umożliwiające podanie listy identyfikatorów i ograniczenia liczby sprawdzanych procesów ❺.

```
    @classmethod
     def create_pid_filter(cls, pid_list: List[int] = None) ->
     ↪Callable[[interfaces.objects.ObjectInterface], bool]:
        filter_func = lambda _: False
        pid_list = pid_list or []
        filter_list = [x for x in pid_list if x is not None]
        if filter_list:
            filter_func = lambda x: x.UniqueProcessId not in filter_list
        return filter_func
```

Do przetwarzania opcjonalnego identyfikatora procesu użyjemy metody tworzącej funkcję, która będzie zwracać wartość `False`, jeżeli zadany identyfikator znajdzie się na liście. Funkcja ma informować, czy dany proces ma być wykluczony ze sprawdzania. Oznacza to, że funkcja musi zwracać wartość `True`, jeżeli danego identyfikatora na liście nie będzie.

```
def _generator(self, procs):
    pe_table_name = intermed.IntermediateSymbolTable.create( ❶
        self.context,
        self.config_path,
        "windows",
        "pe",
        class_types=extensions.pe.class_types)

    procnames = list()
    for proc in procs:
        procname = proc.ImageFileName.cast("string",
               max_length=proc.ImageFileName.vol.count, errors='replace')
```

```
if procname in procnames:
    continue
procnames.append(procname)

proc_id = "Unknown"
try:
    proc_id = proc.UniqueProcessId
    proc_layer_name = proc.add_process_layer()
except exceptions.InvalidAddressException as e:
    vollog.error(f"Proces {proc_id}: błędny adres {e} w warstwie
    ↪{e.layer_name}")
    continue

peb = self.context.object( ❷
        self.config['nt_symbols'] + constants.BANG + "_PEB",
        layer_name = proc_layer_name,
        offset = proc.Peb)

try:
    dos_header = self.context.object(
            pe_table_name + constants.BANG + "_IMAGE_DOS_HEADER",
            offset=peb.ImageBaseAddress,
            layer_name=proc_layer_name)
except Exception as e:
    continue

pe_data = io.BytesIO()
for offset, data in dos_header.reconstruct():
    pe_data.seek(offset)
    pe_data.write(data)
pe_data_raw = pe_data.getvalue() ❸
pe_data.close()

try:
    pe = pefile.PE(data=pe_data_raw) ❹
except Exception as e:
    continue

aslr = check_aslr(pe) ❺

yield (0, (proc_id, ❻
            procname,
            format_hints.Hex(pe.OPTIONAL_HEADER.ImageBase),
            aslr,
            ))
```

Definiujemy specjalną strukturę `pe_table_name` ❶, której będziemy używać podczas iterowania procesów umieszczonych w pamięci. Następnie uzyskujemy obszar pamięci PEB (ang. *Process Environment Block*, blok środowiska procesu) skojarzony z danym procesem i zapisujemy go w obiekcie ❷. Obszar PEB jest strukturą zawierającą mnóstwo informacji o procesie. Zapisujemy obszar w obiekcie plikowym `pe_data` ❸, a następnie za pomocą biblioteki *pefile* tworzymy obiekt PE ❹, który umieszczamy w argumencie metody `check_aslr()` ❺.

Na koniec generujemy krotkę zawierającą identyfikator procesu, jego nazwę, adres w pamięci i wartość logiczną informującą, czy proces jest chroniony zabezpieczeniem ASLR ❻.

Teraz utwórzmy metodę `run()`, która nie ma argumentów, ponieważ wszystkie ustawienia są zapisane w obiekcie `config`.

```
def run(self):
    procs = pslist.PsList.list_processes( ❶
            self.context,
            self.config["primary"],
            self.config["nt_symbols"],
            filter_func = self.create_pid_filter(self.config.get('pid', None)))
    return renderers.TreeGrid([ ❷
        ("PID", int),
        ("Filename", str),
        ("Base", format_hints.Hex),
        ("ASLR", bool)],
        self._generator(procs))
```

Listę procesów uzyskujemy za pomocą wtyczki `pslist` ❶. Następnie zwracamy obiekt typu `TreeGrid` zawierający dane wygenerowane przez metodę `generator()` ❷. Klasa `TreeGrid` jest stosowana w wielu wtyczkach. Dzięki niej wyniki analizy każdego procesu są umieszczane w osobnym wierszu.

Czy to w ogóle działa

Przyjrzyjmy się teraz jednemu z obrazów dostępnemu na stronie narzędzia Volatility: Cridex. Aby użyć własnej wtyczki należy podać parametr `-p` i nazwę katalogu, w którym się wtyczka znajduje:

```
PS>vol -p .\plugins\windows -f cridex.vmem aslrcheck.AslrCheck
Volatility 3 Framework 1.2.0-beta.1
Progress:    0.00               Scanning primary2 using PdbSignatureScanner
PID     Filename        Base            ASLR
368     smss.exe        0x48580000      False
584     csrss.exe       0x4a680000      False
608     winlogon.exe    0x1000000       False
652     services.exe    0x1000000       False
664     lsass.exe       0x1000000       False
824     svchost.exe     0x1000000       False
1484    explorer.exe    0x1000000       False
1512    spoolsv.exe     0x1000000       False
1640    reader_sl.exe   0x400000        False
788     alg.exe         0x1000000       False
1136    wuauclt.exe     0x400000        False
```

Jak widać, jest to obraz maszyny z systemem Windows XP i żaden proces nie ma zabezpieczenia ASLR. Poniżej jest przedstawiony wynik analizy czystego, zaktualizowanego obrazu systemu Windows 10:

```
PS>vol -p .\plugins\windows -f WinDev2007Eval-Snapshot4.vmem aslrcheck.AslrCheck
Volatility 3 Framework 1.2.0-beta.1
Progress:   33.01               Scanning primary2 using PdbSignatureScanner
PID     Filename        Base            ASLR
316     smss.exe        0x7ff668020000 True
428     csrss.exe       0x7ff796c00000 True
500     wininit.exe     0x7ff7d9bc0000 True
568     winlogon.exe    0x7ff6d7e50000 True
592     services.exe    0x7ff76d450000 True
600     lsass.exe       0x7ff6f8320000 True
696     fontdrvhost.ex  0x7ff65ce30000 True
728     svchost.exe     0x7ff78eed0000 True

Volatility was unable to read a requested page:
Page error 0x7ff65f4d0000 in layer primary2_Process928 (Page Fault at entry
↪0xd40c9d88c8a00400 in page entry)

 * Memory smear during acquisition (try re-acquiring if possible)
 * An intentionally invalid page lookup (operating system protection)
 * A bug in the plugin/volatility (re-run with -vvv and file a bug)

No further results will be produced
```

Niewiele jest tu do oglądania. Każdy proces jest zabezpieczony za pomocą ASLR. Widoczne jest natomiast **rozmazanie pamięci** (ang. *memory smear*). Oznacza to, że zawartość pamięci zmieniła się podczas wykonywania jej obrazu. W efekcie tabela deskryptorów pamięci nie jest zgodna z jej zajętością, a wskaźniki pamięci wirtualnej mogą się odnosić do niewłaściwych danych. W takim wypadku należy zgodnie z opisem ponownie uzyskać obraz (poszukać nowego lub utworzyć go).

Sprawdźmy jeszcze obraz pamięci systemu Windows 10 uzyskanego ze strony PassMark:

```
PS>vol -p .\plugins\windows -f WinDump.mem aslrcheck.AslrCheck
Volatility 3 Framework 1.2.0-beta.1
Progress:    0.00               Scanning primary2 using PdbSignatureScanner
PID     Filename        Base            ASLR
356     smss.exe        0x7ff6abfc0000 True
2688    MsMpEng.exe     0x7ff799490000 True
2800    SecurityHealth  0x7ff6ef1e0000 True
5932    GoogleCrashHan  0xed0000       True
5380    SearchIndexer.  0x7ff6756e0000 True
3376    winlogon.exe    0x7ff65ec50000 True
6976    dwm.exe         0x7ff6ddc80000 True
9336    atieclxx.exe    0x7ff7bbc30000 True
9932    remsh.exe       0x7ff736d40000 True
2192    SynTPEnh.exe    0x140000000    False
7688    explorer.exe    0x7ff7e7050000 True
7736    SynTPHelper.ex  0x7ff7782e0000 True
```

Chronione są wszystkie procesy oprócz jednego: `SynTPEnh.exe`. W internecie można znaleźć informację, że jest to oprogramowanie Synaptics Pointing Device, prawdopodobnie obsługujące panel dotykowy. Jeżeli jest zainstalowanie w katalogu *C:\Program Files*, to wszystko jest w porządku. Niemniej jednak można później spróbować się do niego dobrać.

W tym rozdziale nauczyłeś się wykorzystywać moc narzędzia Volatility do badania zachowania użytkownika, uzyskiwania informacji o połączeniach i analizowania danych zapisanych w pamięci oraz uruchomionych procesów. Informacje te pomogą Ci lepiej zrozumieć aktywność użytkownika i działanie docelowej maszyny, jak również strategię systemu ochrony.

Idź dalej!

Zapewne przekonałeś się, że Python doskonale nadaje się do zastosowań hakerskich, tym bardziej, że jest wyposażony w wiele bibliotek i platform. Hakerzy mają wprawdzie mnóstwo narzędzi do dyspozycji, ale w rzeczywistości nic nie zastąpi samodzielnie napisanych programów. W ten sposób można też lepiej poznać działanie innych narzędzi.

Zacznij od razu kodować narzędzia spełniające Twoje specjalne wymagania. Niezależnie od tego, czy będzie to klient SSH dla Windows, scraper WWW, czy system zarządzania trojanami, Python okaże się niezastąpiony.


PROGRAM
PARTNERSKI
GRUPY HELION
1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ
Zmień swoją stronę WWW w działający bankomat!
Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl
GRUPA
Helion

Python: niezawodny kod może służyć także ciemnej stronie mocy!

Język Python jest znany jako wszechstronny, elastyczny i łatwy do nauczenia. Te zalety doceniają naukowcy, programiści i oczywiście hakerzy. Testowanie penetracyjne bowiem wymaga umiejętności szybkiego tworzenia skutecznych narzędzi, a do tego Python nadaje się znakomicie. Jednak wiedza o mrocznej stronie Pythona przydaje się nie tylko pentesterom i napastnikom. Świadomość możliwości kodu Pythona jest pomocna również podczas pracy administratorów i programistów.

To drugie wydanie bestsellerowego przewodnika obrazującego hakerskie możliwości Pythona. Opisano w nim, jak tworzyć narzędzia do podsłuchiwania ruchu sieciowego, wykradania poświadczeń, prowadzenia włamań siłowych, a także jak pisać fuzzery i trojany. Książkę zaktualizowano do Pythona 3 i wzbogacono o informacje dotyczące przesuwania bitów, utrzymywania higieny kodu, korzystania z narzędzia Volatility i bibliotek: *ctypes*, *struct*, *lxml* i BeautifulSoup. Opisano tu również ofensywne strategie hakerskie, takie jak dzielenie bajtów, stosowanie bibliotek do widzenia komputerowego czy przeszukiwanie stron internetowych. To zbiór nie tylko ważnych informacji, ale i inspiracji do realizowania własnych pomysłów.

Dzięki książce nauczysz się:

- wykradać dane z sieci bez pozostawiania śladów
- stosować ofensywne techniki analizy pamięci
- pisać złośliwy kod, taki jak trojany
- rozszerzać możliwości pakietu Burp Suite
- wykorzystywać niektóre potencjalne podatności systemu Windows

Justin Seitz jest ekspertem cyberbezpieczeństwa i białego wywiadu. Jest też autorem branżowych publikacji, a także członkiem International Criminal Court's Technical Advisory Board i Center for Advanced Defense Studies w Waszyngtonie w Stanach Zjednoczonych.

Tim Arnold jest zawodowym programistą Pythona i statystykiem. Był międzynarodowym wykładowcą i uznanym dydaktykiem. Był też członkiem zarządu stowarzyszenia Raleigh ISSA, a także konsultantem Międzynarodowego Instytutu Statystycznego.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

ISBN 978-83-283-8345-6
9 788328 383456

Cena: 59,00 zł