

Reverse Engineering sur plateforme libre



@r00tbsd - Paul Rascagneres

malware.lu

09 Juillet 2012

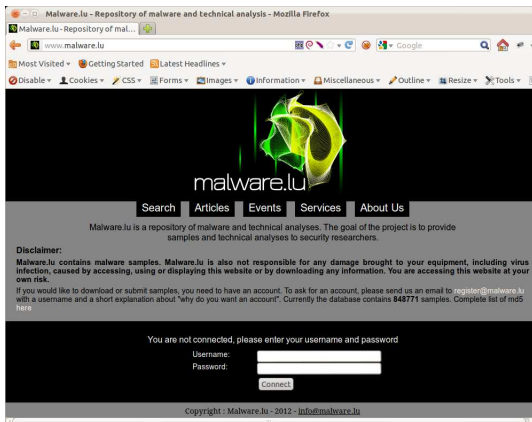
Plan

- 1 Introduction
- 2 Monitoring
- 3 Dynamique
- 4 Statique
- 5 Conclusion

Présentation du projet malware.lu.

Liste des mainteneurs :

- @r00tbsd - Paul Rascagneres (moi)
- @y0ug - Hugo Caron



Some numbers

Le projet en quelques chiffres:

- 1 197 405 Samples
- 20 articles
- 700 utilisateurs
- 659 followers sur twitter (@malwarelu)

But de la presentation:

- décrire les outils et fonctionnalités libres permettant d'analyser le fonctionnement interne du programme
- démontrer que le reverse peut être accessible
- le fun !!

Extrait du nouvel article Art. L. 335-3-1 introduit l'article 22 du DADVSI :

I. - Est puni de 3 750 EUR d'amende le fait de porter atteinte sciemment, à des fins autres que la recherche, à une mesure technique efficace telle que définie à l'article L. 331-5, afin d'altérer la protection d'une œuvre par un décodage, un décryptage ou toute autre intervention personnelle destinée à contourner, neutraliser ou supprimer un mécanisme de protection ou de contrôle, (...)

II. - Est puni de six mois d'emprisonnement et de 30 000 EUR d'amende le fait de procurer ou proposer sciemment à autrui, directement ou indirectement, des moyens conçus ou spécialement adaptés pour porter atteinte à une mesure technique efficace (...)

IV. - Ces dispositions ne sont pas applicables aux actes réalisés à des fins de recherche (...) ou de sécurité informatique, dans les limites des droits prévus par le présent code.

ltrace & strace

Ces deux outils sont des outils de debugage permettant de surveiller les appels système.

```

1 rootbsd@alien:~$ ltrace id | more
2 __libc_start_main(0x8049210, 1, 0xbfdfbd34, 0x804bf60, 0x804bfc0 <unfinished ...>
3 is_selinux_enabled(0, 0x5f85a76d, 0x9f81c255, 0, 4096) = 0
4 strrchr("id", '/') = NULL
5 setlocale(6, "") = "en_US.UTF-8"
6 bindtextdomain("coreutils", "/usr/share/locale") = "/usr/share/locale"
7 textdomain("coreutils") = "coreutils"
8 __cxa_atexit(0x804a710, 0, 0, 0xbfdfbd34, 0xbfdfbc98) = 0
9 getopt_long(1, 0xbfdfbd34, "agnruGZ", 0x0804c720, NULL) = -1
10 geteuid() = 1000
11 getuid() = 1000
12 getegid() = 1000
13 getgid() = 1000

```

ltrace

ltrace & strace

Second exemple permettant de limiter seulement à un appel système précis

```
1 rootbsd@alien:~$ ltrace -e strlen ls
2 strlen(".") = 1
3 strlen(" Conference.pdf") = 14
4 strlen(" Conference.nav") = 14
5 strlen(" Conference.ps") = 13
6 strlen(" ltrace") = 6
7 strlen(" png2eps") = 7
```

ltrace2

ptrace

ptrace() est un appel système dont l'objectif est de debugger des programmes.

Son fonctionnement est le suivant:

- ptrace() s'attache à un processus
- si les droits sont bons, le traceur devient le père du processus
- le parent peut prendre le contrôle du fils à la demande

```
1 #include <sys/ptrace.h>
2 long ptrace(enum __ptrace_request request \
3             , pid_t pid, void *addr, void *data);
```

ptrace

ptrace

Exemple d'utilisation:

```
1 #include <sys/ptrace.h>
2 #include <sys/types.h>
3 #include <sys/wait.h>
4 #include <unistd.h>
5 #include <sys/reg.h>
6
7 int main()
8 {
9     pid_t child;
10    long orig_eax;
11    child = fork();
12    if(child == 0) {
13        ptrace(PTRACE_TRACEME, 0, NULL, NULL);
14        execl("/bin/ls", "ls", NULL);
15    }
16    else {
17        wait(NULL);
18        orig_eax = ptrace(PTRACE_PEEKUSER,
19                        child, 4 * ORIG_EAX,
20                        NULL);
21        printf("The child made a "
22              "system call %ld\n", orig_eax);
23        ptrace(PTRACE_CONT, child, NULL, NULL);
24    }
25    return 0;
26 }
```

example.c

ptrace

```
1 rootbsd@alien:~$ ./example  
2 The child made a system call 11
```

example

L'appel système 11 est `execve()`.

Cet outil a été utilisé pour reverser Skype et créer skypy.

Afin de bypasser des checksum interne à Skype.

LD PRELOAD

LD_PRELOAD est un variable particulière sous Linux, elle permet de paramétrer un chemin vers une librairie qui sera utilisée en priorité.

LD PRELOAD

Exemple d'utilisation:

```
1 # cat soft.c
2 #include <stdio.h>
3
4 int main()
5 {
6     printf(" pid : %u\n", getpid());
7     return (0);
8 }
9 # gcc soft.c -o soft
10 # ./soft
11 pid : 43562
12
13 # cat getpid.c
14 #include <sys/syscall.h>
15 #include <sys/types.h>
16 #include <unistd.h>
17 #include <stdio.h>
18
19 pid_t getpid(void)
20 {
21     printf(" Hello World!\n");
22     return syscall(SYS_getpid);
23 }
24
25 # gcc -fPIC -shared -o getpid.so getpid.c
26 # export LD_PRELOAD=./getpid.so
27 # ./soft
28 Hello World!
29 pid : 44834
```

SystemTap

SystemTap est un outil d'instrumentation de code dynamique développé par Red Hat. Basé sur un système de sonde au niveau kernel.

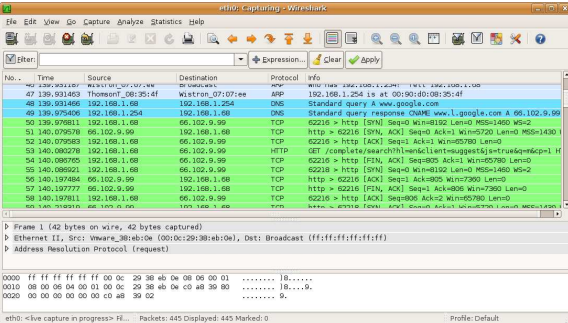
Exemple d'un script permettant de lister les connexions entrantes:

```
1 #! /usr/bin/env stap
2
3 probe begin {
4     printf("%6s %16s %6s %6s %16s\n",
5           "UID", "CMD", "PID", "PORT", "IP_SOURCE")
6 }
7
8 probe kernel.function("tcp_accept").return?,
9        kernel.function("inet_csk_accept").return? {
10     sock = $return
11     if (sock != 0)
12         printf("%6d %16s %6d %6d %16s\n", uid(), execname(), pid(),
13           inet_get_local_port(sock), inet_get_ip_source(sock))
14 }
```

stap

Analyse réseau

Généralement bien connus de tous, des outils tel que tcpdump, ethereal ou wireshark permettent de faire des captures réseaux. Ces outils sont également à classer avec les outils de monitoring de processus permettant des analyses de reverse.



The screenshot shows the Wireshark interface with a packet capture on the eth0 interface. The packet list pane displays several packets:

No.	Time	Source	Destination	Protocol	Info
46	139.931466	192.168.1.68	192.168.1.254	ARP	Who-ask 192.168.1.254 (192.168.1.0/24)
47	139.931463	ThomasT_08:35:4f	Wistron_07:07:ee	ARP	192.168.1.254 is at 00:90:0d:08:35:4f
48	139.931466	192.168.1.68	192.168.1.254	DNS	Standard query A www.google.com
49	139.935496	192.168.1.254	192.168.1.68	DNS	Standard query response QUAME www.l.google.com A 66.102.9.99
50	139.979811	192.168.1.68	66.102.9.99	TCP	62216 > http [SYN] Seq=0 Win=0 Len=0 MSS=1460 WS=2
51	140.079578	66.102.9.99	192.168.1.68	TCP	http > 62216 [SYN, ACK] Seq=0 Ack=1 Win=5720 Len=0 MSS=1430
52	140.079583	192.168.1.68	66.102.9.99	TCP	62216 > http [ACK] Seq=1 Ack=1 Win=65780 Len=0
53	140.080278	192.168.1.68	66.102.9.99	HTTP	GET /complete/search?hl=ms&client=suggest&js=true&q=6pcpl H
54	140.080705	192.168.1.68	66.102.9.99	TCP	62216 > http [FIN, ACK] Seq=805 Ack=1 Win=65780 Len=0
55	140.080921	192.168.1.68	66.102.9.99	TCP	62218 > http [SYN] Seq=0 Win=0 Len=0 MSS=1460 WS=2
56	140.197484	66.102.9.99	192.168.1.68	TCP	http > 62218 [ACK] Seq=1 Ack=805 Win=7360 Len=0
57	140.197777	66.102.9.99	192.168.1.68	TCP	http > 62218 [FIN, ACK] Seq=1 Ack=806 Win=7360 Len=0
58	140.197811	192.168.1.68	66.102.9.99	TCP	62216 > http [ACK] Seq=806 Ack=2 Win=65780 Len=0
59	140.218310	66.102.9.99	192.168.1.68	TCP	http > 62216 [ACK] Seq=806 Ack=1 Win=65780 Len=0 MSS=1460

The packet details pane shows the selected packet (No. 1) as an Ethernet II frame with source MAC VMware_38:eb:0e:00:0c:29:3b:eb and destination MAC Broadcast (ff:ff:ff:ff:ff:ff). The protocol is identified as Address Resolution Protocol (request).

The packet bytes pane shows the raw data in hexadecimal and ASCII:

```

0000 ff ff ff ff ff 00 0c 29 3b eb 0e 00 06 00 01 ..... 18.....
0010 08 00 06 04 00 01 00 0c 29 3b eb 0c a8 39 80 ..... 18....9.
0020 00 00 00 00 00 00 c0 a8 39 02 ..... 9.

```

At the bottom, the status bar indicates: eth0: <live capture in progress> FE... Packets: 445 Displayed: 445 Marked: 0 Profile: Default

Analyse des dumps memoire



Les images mémoires peuvent être utilisé pour analyser le fonctionnement d'un binaire a un instant donné. Pour faire les dumps nous pouvons utiliser VirtualBox.

```
1 rootbsd@~ $ VBoxManage --dbg --startvm <VM name>  
2  
3 .pgmphysfile filename
```

VirtualBox

Analyse des dumps memoire

Pour faire l'analyse il existe deux outils volatility (pour l'analyse de dump Windows) et volatility (pour Linux).

Demo

ATTENTION

Jusqu'à présent nous avons évité d'évoquer de l'ASM...
La suite des slides pourrait heurter la sensibilité des anti-ASM.



Reverse dynamique

L'analyse dynamique consiste à auditer le programme pendant son fonctionnement ou à utiliser les fonctions que celui-ci fournit.

Nous allons voir 4 méthodes:

- Sandbox via Cuckoo
- gdb & winegdb
- VirtualBox & gdb
- ripper metasm

Sandbox via Cuckoo



Le principe des sandboxes (ou bac a sable) est de créer un environnement virtuel où faire tourner l'application pour logger son fonctionnement. Ce type de produit est très courant dans l'analyse de malware. En effet il est pas souhaitable de faire tourner des logiciels à risques sur sa machine.

Il existe une sandbox libre appelée cuckoo.

Cuckoo utilise virtualbox pour gérer son environnement virtuel.

Sandbox via Cuckoo



Demo

Ces deux outils sont des debuggers. Ils permettent de voir l'état des registres ainsi que le cas assembleur en cours d'exécution. Ces outils permettent l'exécution des programmes instructions après instruction.

gdb cheat sheet

- help : list gdb commands
- info break : list breakpoints
- info registers : liste registers value
- break [function—address] : add a breakpoint
- disas : show asm
- continue : continue executinf until next breakpoint
- stepi : Step to next instruction (will step into function)
- nexti : Execute next instruction (will NOT enter function)

Demo

Lors de reverse d'application Windows, il est parfois nécessaire d'utiliser l'OS Windows, l'émulation ne suffit pas.
Pour illustrer notre exemple, nous allons utiliser le malware Flame.
Dispo a cette adresse:

http://www.malware.lu/_download.php?hash=bdc9e04388bda8527b398a8c34667e18

VirtualBox & gdb



Le principe est le suivant: nous allons lancer VirtualBox avec comme OS guest Windows. Dans cet OS nous installons cygwin et gdb serveur. Nous lançons le malware avec gdb serveur. Depuis la machine maitre, nous nous connectons à distance à la machine virtuelle.

L'intérêt de cette méthode est de pouvoir scripter notre analyse via le support python de gdb.

VirtualBox & gdb



Flame a une fonction d'obfuscation des chaines de caractère:
0x1000E431.

Il nous est facile d'identifier les chaines de caractere qui sont envoyées à cette fonction (via un script python et distorm3).

Une fois cette liste récupérée, nous pouvons utiliser le malware lui-même via gdb pour les décoder.

```

1 user@malware-lu ~$ gdbserver.exe host:1234 bdc9e04388bda8527b398a8c34667e18
2 Process /cygdrive/c/bdc9e04388bda8527b398a8c34667e18 created; pid = 732
3 Listening on port 1234
4
5 user@malware-lu ~$ gdb
6 (gdb) target remote 192.168.56.101:1234
7 Remote debugging using 192.168.56.101:1234
8 0x7c91120f in ?? ()
9 (gdb) source script/gdb_flame_decode1.py
10 (gdb) flameDecode1
11 Breakpoint 1 at 0x1000e442
12 Breakpoint 2 at 0x1000e476
13 (gdb) shell head resultDecode1.txt
14 0x10256c54 vsmon.exe
15 0x10256c74 zlclient.exe
16 0x10256c98 ProductName
17 0x10256cb8 ZoneAlarm
18 0x10256cd8 vsmon.exe
19 0x10256cf8 zlclient.exe
  
```

Ripper metasm



Malware.lu a développé un outil permettant de ripper du code basé sur le framework metasm.

Metasm est un framework ruby destiné à la manipulation de fichier exécutable binaire.

Il permet la compilation de fichiers sources, le désassemblage, le debuggage ou encore l'analyse de shellcode.

Ripper metasm

Nous avons utiliser ce framework afin de pouvoir ripper directement du code sans comprendre l'ASM mais uniquement le prototype.

```

1  #!/usr/bin/env ruby
2  # include the magic ripper
3  require "ripper.rb"
4  # a loop to get each encoded string
5  for a in [ 0x1AE88, 0x1AEF0, 0x1AF54, 0x1AF88, 0x1AFEC, 0x1B020, 0x1B084]
6    srcFile = File.open(ARGV[0], 'r')
7    srcFile.seek(a, IO::SEEK_SET)
8    string = srcFile.sysread(0x20)
9    # ARGV[0] in the binary to rip
10   # 0x403034 is the adress of the function use to decode string
11   # "unsigned int decode();" is the prototype of the function decode()
12   # each [], [], [], [] are not used in this example
13   # string contain the encoded string and must be store in ecx
14   specs = [Spec.new(ARGV[0], 0x403034, "unsigned int decode();", [], [], \
15                                                     [], [], string)]
16   worker = Ripper.new(specs)
17   worker.runner.decode()
18   puts string
19 end

```

to_rip.rb

Ripper metasm

L'exécution dans le cas d'un botnet (herpesnet).

```
1 rootbsd@malware.lu$ ./decode.rb db6779d497cb5e22697106e26eebfaa8
2 gpresultl
3 http://dd.zerocode.net/herpnet/
4 74978 b6ecc6c19836a17a3c2cd0840b0
5 http://www.zerocode.net/herpnet
6 ftp.zerocode.net
7 http://frk7.mine.nu/herpnet/
8 upload@zerocode.net
9 uppit
10 hwfqfqqoatstwuuhhtstshwq
11 esstttubbb
12 Nfusheapfzk
```

to_rip.bash

L'analyse statique consiste à analyser le code assembleur par simple "lecture".

Metasm est également très efficace pour de l'analyse statique.

Demo

Cette conférence a pour but de montrer les solutions open source qui permettent de faire du reverse engineering.

Cette discipline mal aimée est souvent nécessaire pour diverses raisons comme rendre les choses interopérables mais également dans l'analyse de logiciels malveillants.

L'analyse peut aller de l'utilisation simple d'outil tout fait et grand public qu'a des sessions de reverse d'ASM durant des jours ;).