Netzwerksicherheit Labor 2
Malware

Yanick Eberle
Pascal Schwarz

27. Dezember 2012

Inhaltsverzeichnis

1 Aufgabe 1 - printf

1.1 Aufgabe 1a - printf mit Strings

Das gegebene Programm ist im folgenden Listing aus Gründen der Vollständigkeit nochmals aufgeführt:

```
1 #include <stdlib.h>
2 int main(void){
3         printf("%s%s%s%s%s%s%s%s%s%s%s%s");
4         return 0;
5 }
```

Wie in der Aufgabenstellung beschrieben erzeugt der Aufruf des kompilierten Programms einen *Segmentation Fault*:

```
1 iso@iso-i7:~/docs/schule/fh/netsi/git/lab2/c_code$ ./1a
2 Segmentation fault
```

Um diesem Problem auf die Schliche zu kommen wurde das Programm zusätzlich mit dem -*S*-Flag übersetzt, so dass der Maschinencode bequem aus einem anderen File gelesen werden kann (es handelt sich um ein 64-bit System):

```
1         .file   "1a.c"
2         .section        .rodata
3 .LC0:
4         .string "%s%s%s%s%s%s%s%s%s%s%s%s"
5         .text
6         .globl  main
7         .type   main, @function
8 main:
9 .LFB2:
10        .cfi_startproc
11        pushq   %rbp
12        .cfi_def_cfa_offset 16
13        .cfi_offset 6, -16
14        movq    %rsp, %rbp
15        .cfi_def_cfa_register 6
16        movl    $.LC0, %edi
17        movl    $0, %eax
18        call    printf
19        movl    $0, %eax
20        popq    %rbp
21        .cfi_def_cfa 7, 8
22        ret
23        .cfi_endproc
24 .LFE2:
25        .size   main, .-main
26        .ident  "GCC: (GNU) 4.7.2"
27        .section        .note.GNU-stack,"",@progbits
```

In diesem Listing ist kaum etwas ersichtlich, was auf die Ursache des Problems schliessen liesse. Wir sehen wie in Zeile 16 die Adresse des Strings *%s%s%s%s...* in das Register %edi geschrieben wird.

Um das Verhalten besser zu verstehen wurde ein abgeändertes Programm erstellt, bei dem weitere Parameter übergeben werden. In diesem Programm werden korrekt zwölf weitere Strings übergeben, die printf in den ersten String einfügt.

```
1  #include <stdlib.h>
2  int main(void){
3          printf("%s%s%s%s%s%s%s%s%s%s%s%s", "01", "02", "03", "04", "05", "06",
               "07", "08", "09", "10", "11", "12");
4          return 0;
5  }
```

Auch von diesem Programm wurde wiederum die Assemblerversion generiert. Hier ist ersichtlich, dass die ersten sechs Parameter des printf-Aufrufs via Register übergeben werden und die weiteren Parameter via Stack (auf x64 Prozessoren wird das Register %rsp als Stackpointer benutzt).

```
1          .file   "1a_2.c"
2          .section        .rodata
3  .LC0:
4          .string "05"
5  .LC1:
6          .string "04"
7  .LC2:
8          .string "03"
9  .LC3:
10         .string "02"
11 .LC4:
12         .string "01"
13 .LC5:
14         .string "%s%s%s%s%s%s%s%s%s%s%s%s"
15 .LC6:
16         .string "12"
17 .LC7:
18         .string "11"
19 .LC8:
20         .string "10"
21 .LC9:
22         .string "09"
23 .LC10:
24         .string "08"
25 .LC11:
26         .string "07"
27 .LC12:
28         .string "06"
29         .text
30         .globl  main
31         .type   main, @function
32 main:
33 .LFB2:
34         .cfi_startproc
35         pushq   %rbp
36         .cfi_def_cfa_offset 16
37         .cfi_offset 6, -16
38         movq    %rsp, %rbp
39         .cfi_def_cfa_register 6
40         subq    $64, %rsp
41         movq    $.LC6, 48(%rsp)
42         movq    $.LC7, 40(%rsp)
43         movq    $.LC8, 32(%rsp)
44         movq    $.LC9, 24(%rsp)
```

```
45          movq    $.LC10, 16(%rsp)
46          movq    $.LC11, 8(%rsp)
47          movq    $.LC12, (%rsp)
48          movl    $.LC0, %r9d
49          movl    $.LC1, %r8d
50          movl    $.LC2, %ecx
51          movl    $.LC3, %edx
52          movl    $.LC4, %esi
53          movl    $.LC5, %edi
54          movl    $0, %eax
55          call    printf
56          movl    $0, %eax
57          leave
58          .cfi_def_cfa 7, 8
59          ret
60          .cfi_endproc
61  .LFE2:
62          .size   main, .-main
63          .ident  "GCC: (GNU) 4.7.2"
64          .section        .note.GNU-stack,"",@progbits
```

Wir nehmen im Folgenden an, dass die printf-Methode auch im ersten Programm die Parameter, die es aufgrund des übergebenen Strings ja brauchen würde, in den Registern %edi, %esi, %edx, %ecx, %r8d, %r9d und dem Stackbereich von %rsp bis und mit 55(%rsp) erwartet. Da dieses Programm jeweils nur 32 bit lange Werte schreibt, werden hier die 32-bit Registernamen benutzt. Beim Analysieren des Programmverlaufs ist allerdings zu beachten, dass die ganzen 64 bit des Registers ausgeben werden. Für die Register %edi, %esi, %edx und %ecx sind %rdi, %rsi, %rdx und %rcx zu verwenden, bei %r8d und %r9d jeweils nur %r8 und %r9.

Da es sich bei den Parametern um Strings handeln müsste, wird die Übergabe eines char-Pointers erwartet, also einer auf diesem Testsystem 8 Byte langen Speicheradresse.

Welche Werte in diesen Registern und an diesen Adressen während der Ausführung des ersten Programms zu finden sind, können wir mit gdb ermitteln. Dazu wird das kompilierte Programm als Parameter an gdb übergeben. Als erstes wird mit dem Befehl *b 3* ein Breakpoint auf der 3. Zeile (dem Aufruf von printf) gesetzt. Der Befehl *set disassemble-next-line on* weist gdb an, bei jedem Step jeweils die nächsten Zeilen Assemblercode auszugeben.

Das Voranschreiten bis zum Aufruf von printf erzeugt somit den folgenden Output:

```
1  iso@iso-i7:~/docs/schule/fh/netsi/git/lab2/c_code$ gdb 1a
2  [some Output ommited]
3  (gdb) l
4  1       #include <stdlib.h>
5  2       int main(void){
6  3               printf("%s%s%s%s%s%s%s%s%s%s%s%s");
7  4               return 0;
8  5       }
9  (gdb) b 3
10 Breakpoint 1 at 0x4004f8: file 1a.c, line 3.
11 (gdb) set disassemble-next-line on
12 (gdb) disassemble
13 Dump of assembler code for function main:
14    0x00000000004004f4 <+0>:     push   %rbp
```

```
15     0x00000000004004f5 <+1>:     mov    %rsp,%rbp
16     0x00000000004004f8 <+4>:     mov    $0x4005fc,%edi
17     0x00000000004004fd <+9>:     mov    $0x0,%eax
18     0x0000000000400502 <+14>:    callq  0x4003f0 <printf@plt>
19     0x0000000000400507 <+19>:    mov    $0x0,%eax
20     0x000000000040050c <+24>:    pop    %rbp
21     0x000000000040050d <+25>:    retq
22 End of assembler dump.
23 (gdb) r
24 Starting program: /home/iso/docs/schule/fh/netsi/git/lab2/c_code/1a
25
26 Breakpoint 1, main () at 1a.c:3
27 3               printf("%s%s%s%s%s%s%s%s%s%s%s");
28 => 0x00000000004004f8 <main+4>: bf fc 05 40 00   mov    $0x4005fc,%edi
29     0x00000000004004fd <main+9>: b8 00 00 00 00   mov    $0x0,%eax
30     0x0000000000400502 <main+14>:        e8 e9 fe ff ff   callq  0x4003f0
           <printf@plt>
31 (gdb) si
32 0x00000000004004fd      3                printf("%s%s%s%s%s%s%s%s%s%s%s");
33     0x00000000004004f8 <main+4>: bf fc 05 40 00   mov    $0x4005fc,%edi
34 => 0x00000000004004fd <main+9>: b8 00 00 00 00   mov    $0x0,%eax
35     0x0000000000400502 <main+14>:        e8 e9 fe ff ff   callq  0x4003f0
           <printf@plt>
36 (gdb) si
37 0x0000000000400502      3                printf("%s%s%s%s%s%s%s%s%s%s%s");
38     0x00000000004004f8 <main+4>: bf fc 05 40 00   mov    $0x4005fc,%edi
39     0x00000000004004fd <main+9>: b8 00 00 00 00   mov    $0x0,%eax
40 => 0x0000000000400502 <main+14>:        e8 e9 fe ff ff   callq  0x4003f0
       <printf@plt>
41 (gdb)
```

Der einzige Parameter der Funktion wird in %edi übergeben. An der Adresse 0x4005fc finden wir erwartungsgemäss den übergebenen String:

```
1 (gdb) x/s 0x4005fc
2 0x4005fc:        "%s%s%s%s%s%s%s%s%s%s%s"
```

Im folgenden Listing werden die Inhalte der anderen Register und der Speicherstellen, die im zweiten Programm zur Parameterübergabe benutzt wurden, ausgegeben. Da die entsprechenden Stellen im Memory und die Werte der Register nicht gesetzt wurden, zeigen sie an diverse, zum Teil nicht über den Paging-Mechanismus gemappte Adressen. Dies zeigt sich beispielsweise bei der Ausgabe des „Strings“ auf welchen die Adresse auf dem untersten Stackelement zeigen sollte - die Fehlermeldung *Address 0x0 out of bounds* wird dort ausgegeben.

```
1  (gdb) info registers rcx rdx rsi rdi rsp r8 r9
2  rcx            0x400510 4195600
3  rdx            0x7fffffffe5d0   140737488348624
4  rsi            0x7fffffffe5b8   140737488348600
5  rdi            0x4005fc 4195836
6  rsp            0x7fffffffe4d0   0x7fffffffe4d0
7  r8             0x4005a0 4195744
8  r9             0x7ffff7deaf40   140737351954240
9
10 (gdb) x/s 0x7fffffffe5b8
11 0x7fffffffe5b8:  "\301\350\377\377\377\177"
12 (gdb) x/s 0x7fffffffe5d0
13 0x7fffffffe5d0:  "\365\350\377\377\377\177"
```

```
(gdb) x/s 0x400510
0x400510 <__libc_csu_init>:       "H\2111$\330L\211d$\340H\215-\003\t "
(gdb) x/s 0x4005a0
0x4005a0 <__libc_csu_fini>:
    "\363?\220\220\220\220\220\220\220\220\220\220\220\220UH\211\345SH\203\354\bH\213\005h\b
    "
(gdb) x/s 0x7ffff7deaf40
0x7ffff7deaf40:  "UH\211\345AWAVAUE1\355ATE1\344SH\203\354\070H\307E\260"
(gdb) x/a 0x7fffffffe4d0
0x7fffffffe4d0: 0x0
(gdb) x/s 0x0
0x0:     <Address 0x0 out of bounds>
(gdb) x/a 0x7fffffffe4d8
0x7fffffffe4d8: 0x7ffff7a5a30d <__libc_start_main+237>
(gdb) x/s 0x7ffff7a5a30d
0x7ffff7a5a30d <__libc_start_main+237>:  "\211\307?\001"
(gdb) x/a 0x7fffffffe4e0
0x7fffffffe4e0: 0x0
(gdb) x/s 0x0
0x0:     <Address 0x0 out of bounds>
(gdb) x/a 0x7fffffffe4e8
0x7fffffffe4e8: 0x7fffffffe5b8
(gdb) x/s 0x7fffffffe5b8
0x7fffffffe5b8:  "\301\350\377\377\377\177"
(gdb) x/a 0x7fffffffe4f0
0x7fffffffe4f0: 0x200000000
(gdb) x/s 0x200000000
0x200000000:     <Address 0x200000000 out of bounds>
(gdb) x/a 0x7fffffffe4f8
0x7fffffffe4f8: 0x4004f4 <main>
(gdb) x/s 0x4004f4
0x4004f4 <main>:          "UH\211\345\277\374\005@"
```

Dass diese Bereiche tatsächlich nicht gemappt sind, lässt sich auch in der Datei /proc/3917/maps (wenn der Prozess die PID 3917 trägt) erkennen:

```
00400000-00401000 r-xp 00000000 08:02 524557
    /home/iso/docs/schule/fh/netsi/git/lab2/c_code/1a
00600000-00601000 r--p 00000000 08:02 524557
    /home/iso/docs/schule/fh/netsi/git/lab2/c_code/1a
00601000-00602000 rw-p 00001000 08:02 524557
    /home/iso/docs/schule/fh/netsi/git/lab2/c_code/1a
7ffff7a39000-7ffff7bd2000 r-xp 00000000 08:01 269377
    /lib/x86_64-linux-gnu/libc-2.13.so
7ffff7bd2000-7ffff7dd1000 ---p 00199000 08:01 269377
    /lib/x86_64-linux-gnu/libc-2.13.so
7ffff7dd1000-7ffff7dd5000 r--p 00198000 08:01 269377
    /lib/x86_64-linux-gnu/libc-2.13.so
7ffff7dd5000-7ffff7dd6000 rw-p 0019c000 08:01 269377
    /lib/x86_64-linux-gnu/libc-2.13.so
7ffff7dd6000-7ffff7ddc000 rw-p 00000000 00:00 0
7ffff7ddc000-7ffff7dfd000 r-xp 00000000 08:01 269357
    /lib/x86_64-linux-gnu/ld-2.13.so
7ffff7fd2000-7ffff7fd5000 rw-p 00000000 00:00 0
7ffff7ff9000-7ffff7ffb000 rw-p 00000000 00:00 0
7ffff7ffb000-7ffff7ffc000 r-xp 00000000 00:00 0                  [vdso]
7ffff7ffc000-7ffff7ffd000 r--p 00020000 08:01 269357
    /lib/x86_64-linux-gnu/ld-2.13.so
7ffff7ffd000-7ffff7fff000 rw-p 00021000 08:01 269357
    /lib/x86_64-linux-gnu/ld-2.13.so
7ffffffde000-7ffffffff000 rw-p 00000000 00:00 0                  [stack]
```

```
16 ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0
       [vsyscall]
```

Auch das Werkzeug *ltrace*, welches die von einem Programm ausgeführten Library-Calls auf der Konsole ausgibt, bestätigt unsere bisherigen Erkenntnisse:

```
1  iso@iso-i7:~/docs/schule/fh/netsi/git/lab2/c_code$ ltrace ./1a
2  iso@iso-i7:~/docs/schule/fh/netsi/git/lab2/c_code$ ltrace ./1a
3  __libc_start_main(0x4004f4, 1, 0x7fff4d305eb8, 0x400510, 0x4005a0 <unfinished ...>
4  printf("%s%s%s%s%s%s%s%s%s%s%s", "", "\005i0M\377\177",
       "H\2111$\330L\211d$\340H\215-\003\t ",
       "\363\303\220\220\220\220\220\220\220\220\220\220\220\220\220\220UH\211\345SH\203\354\bH\213\005
       ", "UH\211\345AWAVAUE1\355ATE1\344SH\203\3548H\307E\260", NULL,
       "\211\307\350\274\245\001", NULL, "",
       "\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377\377"...,
       "UH\211\345\277\374\005@", NULL <unfinished ...>
5  --- SIGSEGV (Segmentation fault) ---
6  +++ killed by SIGSEGV +++
```

1.2 Aufgabe 1b - printf mit Hex

Das Programm erzeugt jeweils ändernde Outputs, hier einige davon:

```
1  iso@iso-i7:~/docs/schule/fh/netsi/git/lab2/c_code$ ./1b
2  65aee9d8.65aee9e8.00400510.004005a0.77adcf40.
3  iso@iso-i7:~/docs/schule/fh/netsi/git/lab2/c_code$ ./1b
4  38449d58.38449d68.00400510.004005a0.91e59f40.
5  iso@iso-i7:~/docs/schule/fh/netsi/git/lab2/c_code$ ./1b
6  08f6b5f8.08f6b608.00400510.004005a0.b02b0f40.
```

Da wir auf einem 64bit-System arbeiten haben wir das Programm folgendermassen abgeändert:

```
1 #include <stdlib.h>
2 int main(void){
3         printf("%lx.%lx.%lx.%lx.%lx.\n");
4         return 0;
5 }
```

Nach dieser Änderung erhalten wir den folgenden Output:

```
1  iso@iso-i7:~/docs/schule/fh/netsi/git/lab2/c_code$ ./1b
2  7fffb4219538.7fffb4219548.400510.4005a0.7fdcf54b1f40.
3  iso@iso-i7:~/docs/schule/fh/netsi/git/lab2/c_code$ ./1b
4  7fff3a078928.7fff3a078938.400510.4005a0.7f1974dddf40.
5  iso@iso-i7:~/docs/schule/fh/netsi/git/lab2/c_code$ ./1b
6  7fffbdeb3d48.7fffbdeb3d58.400510.4005a0.7f024d6d1f40.
7  iso@iso-i7:~/docs/schule/fh/netsi/git/lab2/c_code$ ./1b
8  7fff85b52f48.7fff85b52f58.400510.4005a0.7f8ef4b15f40.
```

Nach der Bearbeitung der Aufgabe 1a ist hier eigentlich schon recht klar was passiert. Printf erwartete im ersten Programm die Übergabe von Pointern, während es nun die Übergabe von Werten erwartet. Im Folgenden wird die selbe Vorgehensweise wie in Aufgabe 1a gewählt:

```
(gdb) info registers rdi rcx rdx rsi r8 r9
rdi            0x4005fc 4195836
rcx            0x400510 4195600
rdx            0x7fffffffe5c8   140737488348616
rsi            0x7fffffffe5b8   140737488348600
r8             0x4005a0 4195744
r9             0x7ffff7deaf40   140737351954240
(gdb) x/s 0x4005fc
0x4005fc:       "%lx.%lx.%lx.%lx.%lx.\n"
(gdb) c
Continuing.
7fffffffe5b8.7fffffffe5c8.400510.4005a0.7ffff7deaf40.
[Inferior 1 (process 5277) exited normally]
```

Wie wir sehen, werden die Inhalte der einzelnen Register jetzt direkt ausgegeben.

2 Aufgabe 2 - Virus

2.1 Variablen main-Methode, globale Variablen

V_OFFSET Länge des Virus-Codes in Bytes. Je nach Compiler und Architektur wird das Virus-Binary unterschiedlich gross ausfallen, daher muss dies jeweils angepasst werden.

len Anzahl gelesene resp. zu schreibende Bytes während Kopiervorgang.

rval Exit-Code, den das infizierte Binary zurückgibt.

fd_r Filedescriptor, aus dem gelesen wird. Zeigt auf die gerade ausgeführte Datei.

fd_w Filedescriptor, in den geschrieben wird (für tempfile).

tmp String, der den Namen eines temp-Files enthält. („The tmpnam() function returns a pointer to a unique temporary filename, or NULL if a unique name cannot be generated.“)

pid, status Enthalten Informationen zum Zustand des Child-Prozesses.

buf Zwischenspeicher für Kopiervorgang.

2.2 Variablen do_infect-Methode

fd_r Filedescriptor, aus dem gelesen wird. Zeigt auf die gerade ausgeführte Datei.

fd_t Filedescriptor, der aufs Target-Binary zeigt.

target, i, done, bytes, length Zähl-Variablen

map Zeigt auf allozierten Speicherbereich, in dem während der Infektion der Original-Code des Target-Binaries zwischengespeichert wird.

stat Speichert Fileinformationen, die von fstat geliefert werden.

buf Zwischenspeicher für Kopiervorgang.

2.3 Ablauf des Programms

Die Programmdatei, welche im „Normalbetrieb“ des Virus ausgeführt wird, beinhaltet zwei Binaries. Vom Start des Files bis zu Byte V_OFFSET (exkl.) befindet sich der Code des Virus, an den Bytes V_OFFSET (inkl.) bis zum Ende des Files das ursprüngliche Programm.

Wird ein derartig infiziertes Binary gestartet, werden die folgenden Schritte ausgeführt:

- Der Teil, der das Original-Programm enthält, wird in ein tmp-File kopiert.
- In den Argumenten, mit denen das infizierte Programm aufgerufen wurde, werden beschreib- und ausführbare Dateien gesucht.
- Falls Dateien mit den entsprechenden Rechten gefunden werden, werden diese infiziert. Dazu kopiert der Virus den Virus-Code-Teil in die Datei und hängt danach den ursprünglichen Inhalt an.
- Das im ersten Schritt erstellte tmp-File (welches nur noch den ursprünglichen Inhalt hat) wird als Kindprozess ausgeführt.
- Sobald dieses Kindprozess beendet wurde, wird der Returncode gespeichert.
- Das tmp-File wird gelöscht.
- Der Returncode des originalen Programms wird zurückgegeben.

2.4 Ausführung des Programms

Die folgenden Befehle wurden unter einer Ubuntu 12.10 Live-CD-Umgebung ausgeführt.

Zunächst haben wir das Virusprogramm kompiliert, um zu sehen, wie gross das Binary wird (in unserem Fall 8127 Bytes). Nach der Anpassung der Variable V_OFFSET und einer Neukompilation konnte die Funktionsweise des Virus gezeigt werden (Dateigrössen beachten):

```
1  ubuntu@ubuntu:~/netsi$ cp /bin/echo .
2  ubuntu@ubuntu:~/netsi$ cp /bin/echo echo2
3  ubuntu@ubuntu:~/netsi$ ls -l
4  total 72
5  -rwxr-xr-x 1 ubuntu ubuntu 26172 Dec 27 14:43 echo
6  -rwxr-xr-x 1 ubuntu ubuntu 26172 Dec 27 14:43 echo2
7  -rwxrwxr-x 1 ubuntu ubuntu  8127 Dec 27 14:37 virus
8  -rw-rw-r-- 1 ubuntu ubuntu  2445 Dec 27 14:37 virus.c
9  -rw-rw-r-- 1 ubuntu ubuntu  2447 Dec 27 14:36 virus.c~
10 ubuntu@ubuntu:~/netsi$ ./virus echo
11 ubuntu@ubuntu:~/netsi$ ls -l virus echo*
```

```
-rwxr-xr-x 1 ubuntu ubuntu 34299 Dec 27 14:46 echo
-rwxr-xr-x 1 ubuntu ubuntu 26172 Dec 27 14:43 echo2
-rwxrwxr-x 1 ubuntu ubuntu  8127 Dec 27 14:37 virus
ubuntu@ubuntu:~/netsi$ ./echo foo
foo
ubuntu@ubuntu:~/netsi$ ./echo scheint sich ganz normal zu verhalten
scheint sich ganz normal zu verhalten
ubuntu@ubuntu:~/netsi$ ./echo echo2
echo2
ubuntu@ubuntu:~/netsi$ ls -l virus echo*
-rwxr-xr-x 1 ubuntu ubuntu 34299 Dec 27 14:46 echo
-rwxr-xr-x 1 ubuntu ubuntu 34299 Dec 27 14:47 echo2
-rwxrwxr-x 1 ubuntu ubuntu  8127 Dec 27 14:37 virus
```

Dass beim Aufruf des verseuchten echo-Binaries einiges mehr passiert, kann auch wieder mit dem Werkzeug *ltrace* verdeutlicht werden. Zunächst der Aufruf des unmodifizierten Binaries:

```
ubuntu@ubuntu:~/netsi$ ltrace -c echo echo2
echo2
% time     seconds  usecs/call     calls      function
------ ----------- ----------- --------- --------------------
 22.44    0.002768        1384         2 fclose
 17.44    0.002151        2151         1 setlocale
 13.51    0.001667         416         4 __freading
  7.49    0.000924         462         2 fflush
  7.33    0.000904         452         2 fileno
  6.54    0.000807         403         2 __fpending
  5.53    0.000682         682         1 fputs_unlocked
  4.11    0.000507         507         1 __overflow
  3.92    0.000483         241         2 strcmp
  2.70    0.000333         333         1 getenv
  2.37    0.000292         292         1 strrchr
  2.35    0.000290         290         1 textdomain
  2.27    0.000280         280         1 __cxa_atexit
  2.00    0.000247         247         1 bindtextdomain
------ ----------- ----------- --------- --------------------
100.00    0.012335                    22 total
```

Und des infizierten Binaries:

```
ubuntu@ubuntu:~/netsi$ ltrace -c ./echo echo2
echo2
% time     seconds  usecs/call     calls      function
------ ----------- ----------- --------- --------------------
 57.60   56.683053        1652     34304 read
 42.39   41.713511        1216     34303 write
  0.00    0.002293        1146         2 waitpid
  0.00    0.001885         628         3 close
  0.00    0.001716         572         3 open
  0.00    0.001375        1375         1 tmpnam
  0.00    0.001186         395         3 lseek
  0.00    0.000962         962         1 fork
  0.00    0.000856         856         1 malloc
  0.00    0.000594         594         1 free
  0.00    0.000570         570         1 __fxstat
  0.00    0.000556         556         1 access
  0.00    0.000549         549         1 unlink
  0.00    0.000420         420         1 __errno_location
  0.00    0.000368         368         1 ftruncate
------ ----------- ----------- --------- --------------------
```

```
21  100.00    98.409894                 68627 total
```

Die Laufzeit wurde aufgrund des ltrace-Aufrufs erheblich langsamer. Dies ist auch darin begründet, dass der Virus praktisch alle Kopiervorgänge Byte für Byte ausführt.

Es ist ebenfalls möglich, dass ein Binary mehrmals infiziert wird. Wenn ein Binary z.B. drei mal infiziert ist, wird auch der „Virus-Header“ dreimal entfernt (und dabei immer temporäre Files angelegt), es wird dreimal geforkt, etc.