Vanilla Skype part 1

Fabrice Desclaux Kostya Kortchinsky

recca(at)rstack.org - kostya.kortchinsky(at)eads.net
serpilliere(at)droids-corp.org - fabrice.desclaux(at)eads.net
EADS Corporate Research Center — DCR/STI/C
SSI Lab
Suresnes, FRANCE

Recon, June 17th, 2006

Outline

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)

⟹ Impossibility to distinguish normal behaviour from information exfiltration (encrypted traffic on strange ports, night activity)

⟹ Jams the signs of real intrusions exfiltration

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) ?! From a company not involved on Open Source ?!

$\Longrightarrow$ Is there something to hide ?

$\Longrightarrow$ Impossible to scan for trojan/backdoor/malware inclusion

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 exfiltration ?
- 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

Binary analysis: Encryption

Encryption scheme

- First, Skype allocates space to create a working space
- It will store deciphered data.

```
push    4
push    1000h
mov     eax, ds:dword_C82958 ; 3D8000h
push    eax
push    0
call    VirtualAlloc
mov     ds:allocated_memory, eax
```

This means in C:

```
LPVOID VirtualAlloc(
  LPVOID lpAddress,   // address of region to reserve or commit
  DWORD dwSize,       // size of region
  DWORD flAllocationType, // type of allocation
  DWORD flProtect     // type of access protection
);
```

Binary analysis: Encryption

Round initialization

Then, it does key initialization and decryption of the parts

```
mov     eax, offset bin_base_addr
...
add     eax, ds:start_ciphered_ptr[edx*4]
...
mov     eax, ds:start_unciphered_ptr[eax*4]
add     eax, ds:allocated_memory
...
mov     dword ptr [ebp-14h], 7077F00Fh
...
mov     eax, ds:size_ciphered[eax*4]
```

- Section information loading
- Key initialization

Binary analysis: Encryption

Information storage

- We can deduce a description of each ciphered section is stored at *start_ciphered_ptr*
- Here is the structure that describes those sections

```
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
```

Binary analysis: Encryption

Data deciphering

Skype uses its allocated memory to store deciphered areas.

```
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
```

- The data is then deciphered
- The key is updated at each round

Binary analysis: Hidden imports

Additional hidden imports

Then it loads dynamically libraries and functions. Those ones are masked to a static analysis.

- Additional imports are loaded at run time
- A generic structure is used to describe its imports

DLL and import loading

```
lea     eax, [eax+eax*2]
mov     eax, ds:dword_C82960[eax*4]
call    sub_405210
push    eax
call    j_LoadLibraryA
```

```
push    eax
mov     eax, [ebp-1Ch]
push    eax
call    j_j_GetProcAddress_0
```

Binary analysis: Hidden imports

Internal structure

- If name is set and others are null, it's a DLL to load
- 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;
}
```

Binary analysis: Hidden imports

DLL loading

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

Import by name

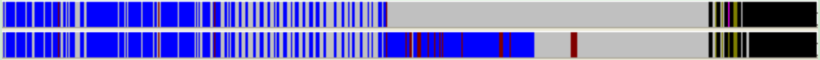
```
dd offset aWaveinreset  ; "waveInReset"
dd 0
dd 3D69D0h
```

Import by ordinal

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

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
  - ...

Final step: cleaning

Re-protection of the sections

```
push    eax
...
mov     eax, ds:dword_C82904[eax*4]
...
mov     eax, ds:dword_C828F8[eax*4]
...
mov     eax, ds:start_unciphered_ptr[eax*4]
add     eax, ds:allocated_memory
push    eax
call    j_VirtualProtect
```

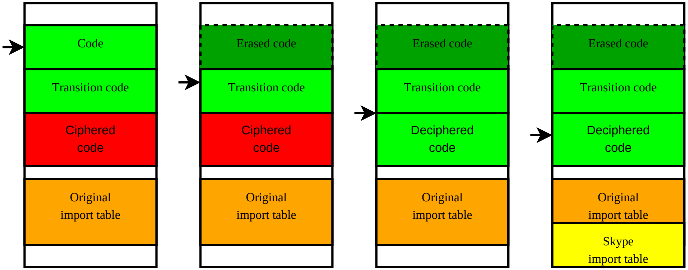
Binary smashing: tricks used

- Erase the 0xF4 first bytes located at the entry point: a memory dump won't be executable
- The addresses of additional imports replace the import table: in theory, we cannot dump both at same time

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



Conclusion

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

Oups

Humm, it seems it crashes randomly... Lets have more fun

Conclusion

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

Oups

Humm, it seems it crashes randomly... Lets have more fun

Outline

Why it crashes?

Analysis

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

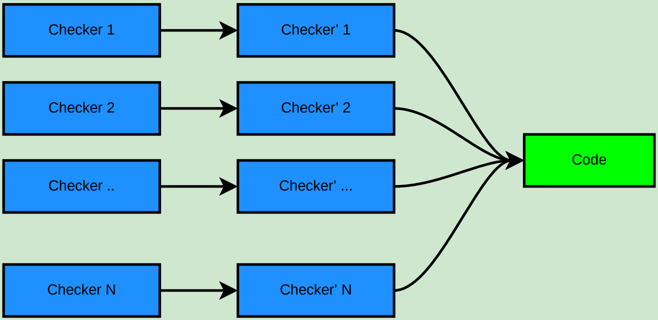
Bingo! part of the software does a checksum on the Softice detection code

Suspicious checksums

In fact, it seems the code is full of checksums! A quick search shows more than 10...

Checksum scheme in Skype

Checksum scheme



Main scheme of Skype code checkers

Why checksums?

Integrity checks

- It prevents binary modification
- If a virus infects a binary, it changes its checksum...
- If someone puts a breakpoint or removes some code parts it will be detected

The high number of checksums may mean the third reason is the good one.

How to detect them

Automatic code fingerprinting

- Find a generic way to spot checksums
- Simulate them to get the correct value
- Generate a patch
- Do that until total annihilation

Here is a code sample

```
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 checksums

Interesting characteristics

- Each checksum is a bit different: it seems to be polymorphic
- They are randomly inserted in the code so they are executed randomly
- The pointers initialization is obfuscated with calculus
- The loops steps have different values/signs
- Checksum operator is randomized (add, xor, sub, ...)
- Random code length
- Dummy mnemonic insertion
- Final test is not trivial: it can use final checksum to compute a pointer for next code part.

Semi metamorphic checksums

But...

It's composed of

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

We can build a script that spots such code

Checksum fingerprint

Invariant code

We try to spot code such as:

```
ADDR1:
    MOV     REG, [REG+XXX]
    ...
    ARIT    REG, REG
    ...
    SUB     REG, CSTE
    ...
    DEC     REG
    ...
    JCC     ADDR1
```

Code fingerprint using IDA disassembler scripting

Checksum fingerprint

Invariant code

```
        MOV     REG, CSTE | XOR   REG, REG
        ...
        ARIT    REG, *
        ...
        ARIT    REG, *
        ...
        ARIT    REG, *
ADDR1:
```

If register value is in a code segment when EIP reaches ADDR1, this is a checksum

x86emu

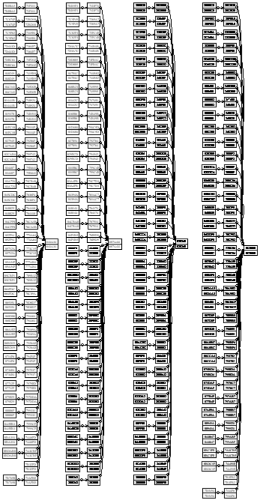
The code is emulated with *x86emu* (x86 emulator IDA plugin) to find the final value

Patch generation

Automatic patch generator

- The goal is to compute the right value of the checksum.
- $\Longrightarrow$This is done with *x86emu* again
- It detects the end of the loop (JCC)
- And stop the emulation when the JCC condition is not satisfied

Global checksum scheme



Checksum execution and patch

Solution

- Compute checksum for each one
- The script is based on a x86 emulator
- It first spots the checksum entry-point: the pointer initialization
- It detects the end of the loop
- Then, it replaces the whole loop by a simple affectation to the final checksum value
- So each checksum successes

And it's less CPU consuming :)

```
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
    nop
    nop
    nop
    nop
    jmp     lbl2
    db  0x73
lbl2:
    jmp     lbl3
    dd  0xC8528417, 0xD8FBBD1, 0xA36CFB2F, 0xE8D6E4B7, 0xC0B8797A
    db  0x61, 0xBD
lbl3:
    sub     eax, 0x4C49F346
```

Last but not least

Signature based integrity-check

- In fact our Skype version has another problem... It crashes randomly again
- There is a final check: integrity check based on RSA signature
- Moduli stored in the binary

```
lea     eax, [ebp+var_C]
mov     edx, offset a65537 ; "65537"
call    str_to_bignum
lea     eax, [ebp+var_10]
mov     edx, offset a38133593136037 ; "38133593136037677542306434298936751184 2"...
call    str_to_bignum
```

Outline

Other side of the protection

Counter measures against dynamic attack

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

Binary protection: Anti debuggers

The easy one

First Softice test

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

Another test

Hidden test: it checks if Softice is not in the drivers 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'             cmp     esi, 'trof'  ; what is that?
jnz     short next              jnz     short next
cmp     edi, 'xt.s'             cmp     edi, '2.sy'
jnz     short next              jnz     short next
cmp     eax, 'ys\x00\x00'       cmp     eax, 's\x00\x00\x00'
jnz     short next              jnz     short next
```

Timing measures

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

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

Binary protection: Anti debuggers

Counter measures

- When it detects an attack, it creates a random box in which the debugger will be trapped.
- Everything is randomized (registers, pages, ...)
- It's is difficult to trace back the detection because 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

How to protect sensitive code

Code obfuscation

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

Advantage/Disadvantage

- Slows down code study
- Avoids direct code stealing

- 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

Determined conditional jumps

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

In C, this means

Determined conditional jumps

```
...
test = 0x1337;
if (test==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_0_
rol     eax, 17h
xor     eax, 350CA27h
pop     ecx
```

- In random functions, the code raises an exception
- So an error handler is called
- Skype decides if it's a true error, or a generated one
- In the second case, Skype does calculus on memory addresses and registers
- So it comes back to the faulty code

Principle

It makes it a bit harder to understand the whole code: we have to stop the error handler and study its code.

Outline

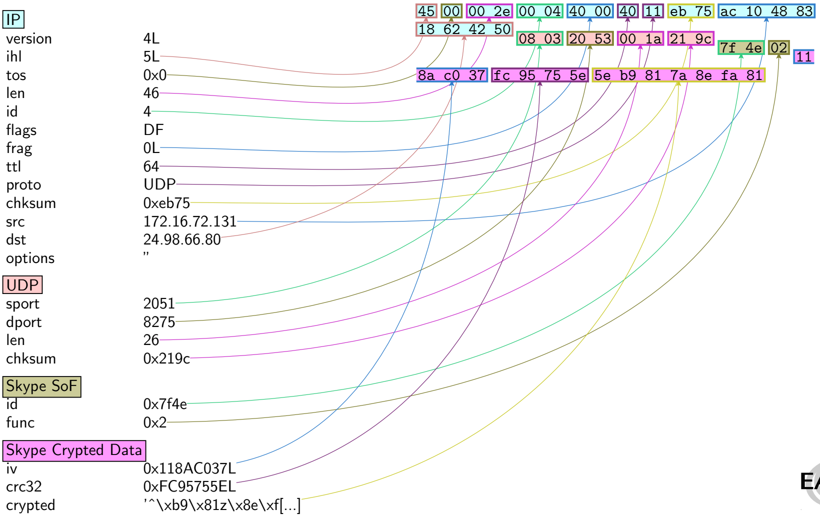
Skype on UDP

Skype UDP start of frame

Skype UDP frames begin

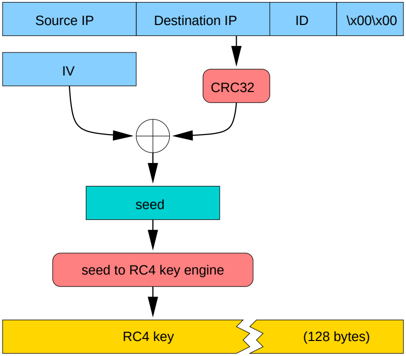
- With a 2 byte ID number
- Then one obfuscated byte that introduces the following layer:
  - Obfuscated layer
  - Ack / NAck
  - Command forwarding
  - Command resending
  - few other stuffs

Skype Network Obfuscation Layer



Skype Network Obfuscation Layer

- Packets 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

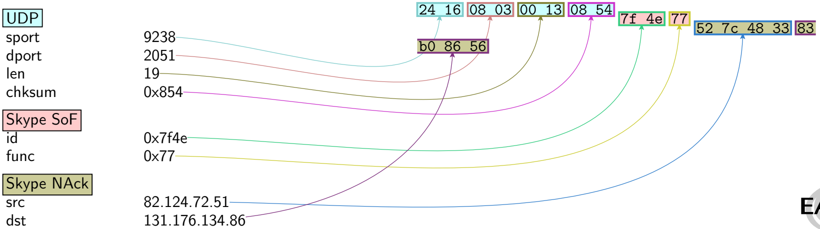


Skype Network Obfuscation Layer

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 *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

$\Longrightarrow$ first solution: we parasitized it

First solution: parasiting

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
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);
}
```

Second solution: recover C code

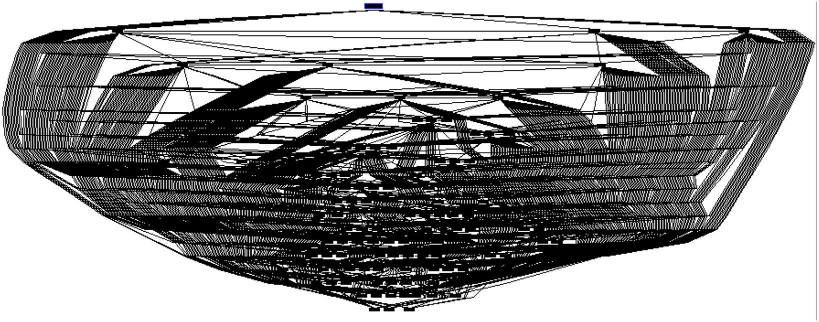
From asm to C

The goal is to recover expressions linked to known values: recover data flow

- Mark all variables (registers) as *unknown*
- Alias known values to their registers
- Find instructions linked to known values
- Update and propagate the pool of known expressions using the instruction semantic
- All memory accesses must be generated
- In case of execution flow splitting, we have to generate the code

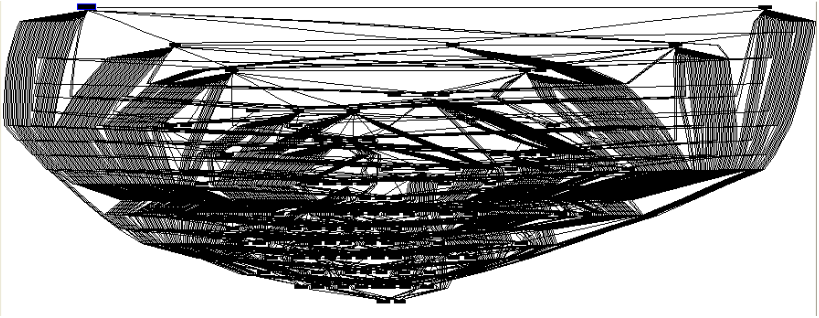
First of all, spot functions in the execution flow

Skype obfuscated function v1.0



First of all, spot functions in the execution flow

Skype obfuscated function v2.0



Value propagation

From asm to C

At this point, we need to follow the white rabbit (eax, ecx)

```
mov     eax, [KEY]
mov     ecx, loc_OUT
call    sub_007ADB80
```

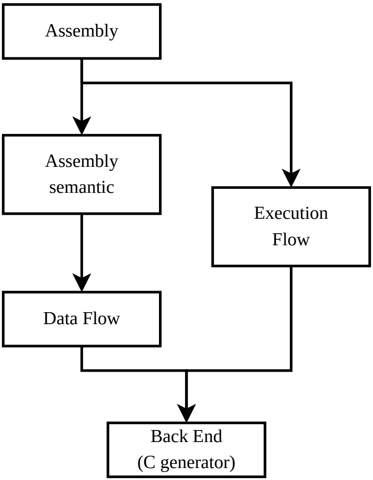
Value propagation initialization

- INPUT:
- EDX alias 'KEY'
- ECX alias 'OUT'

Data flow analysis

We need to follow EDX and ECX

Value propagation



Example

From asm to C

EDX = 'KEY'
ECX = 'OUT'
sub_007ADB80:

```
mov   eax, edx          ; EAX = KEY
push  esi               ; None
mov   edx, [ecx+1Ch]    ;EDX = OUT[0x1C]
mov   esi, [ecx+28h]    ;ESI = OUT[0x28]
sub   edx, 354C1FF2h    ;EDX = OUT[0x1C] - 0x354C1FF2
xor   esi, edx          ; ESI = OUT[0x28]^ ( OUT[0x1C] - 0x354C1FF2 )
rol   eax, 3            ; EAX = ROL(KEY, 3)
mov   [ecx+28h], esi    ; OUT[0x28] = OUT[0x28]^ ( OUT[0x1C] - 0x354C1FF2 )
xor   eax, 22E40A3Eh    ;EAX = ROL(KEY, 3) ^ 0x22E40A3E
pop   esi               ; None
retn                    ; None
```

Example

From asm to C

```
EDX = 'KEY'
ECX = 'OUT'
sub_007ADB80:
mov   eax, edx          ; EAX = KEY
push  esi               ; None
mov   edx, [ecx+1Ch]    ;EDX = OUT[0x1C]
mov   esi, [ecx+28h]    ;ESI = OUT[0x28]
sub   edx, 354C1FF2h    ;EDX = OUT[0x1C] - 0x354C1FF2
xor   esi, edx          ; ESI = OUT[0x28]^ ( OUT[0x1C] - 0x354C1FF2 )
rol   eax, 3            ; EAX = ROL(KEY, 3)
mov   [ecx+28h], esi    ; OUT[0x28] = OUT[0x28]^ ( OUT[0x1C] - 0x354C1FF2 )
xor   eax, 22E40A3Eh    ;EAX = ROL(KEY, 3) ^ 0x22E40A3E
pop   esi               ; None
retn                    ; None
```

Example

From asm to C

```
EDX = 'KEY'
ECX = 'OUT'
sub_007ADB80:
mov   eax, edx          ; EAX = KEY
push  esi               ; None
mov   edx, [ecx+1Ch]    ;EDX = OUT[0x1C]
mov   esi, [ecx+28h]    ;ESI = OUT[0x28]
sub   edx, 354C1FF2h    ;EDX = OUT[0x1C] - 0x354C1FF2
xor   esi, edx          ; ESI = OUT[0x28]^ ( OUT[0x1C] - 0x354C1FF2 )
rol   eax, 3            ; EAX = ROL(KEY, 3)
mov   [ecx+28h], esi    ; OUT[0x28] = OUT[0x28]^ ( OUT[0x1C] - 0x354C1FF2 )
xor   eax, 22E40A3Eh    ;EAX = ROL(KEY, 3) ^ 0x22E40A3E
pop   esi               ; None
retn                    ; None
```

Example

From asm to C

```
EDX = 'KEY'
ECX = 'OUT'
sub_007ADB80:
mov   eax, edx          ; EAX = KEY
push  esi               ; None
mov   edx, [ecx+1Ch]    ;EDX = OUT[0x1C]
mov   esi, [ecx+28h]    ;ESI = OUT[0x28]
sub   edx, 354C1FF2h    ;EDX = OUT[0x1C] - 0x354C1FF2
xor   esi, edx          ; ESI = OUT[0x28]^ ( OUT[0x1C] - 0x354C1FF2 )
rol   eax, 3            ; EAX = ROL(KEY, 3)
mov   [ecx+28h], esi    ; OUT[0x28] = OUT[0x28]^ ( OUT[0x1C] - 0x354C1FF2 )
xor   eax, 22E40A3Eh    ;EAX = ROL(KEY, 3) ^ 0x22E40A3E
pop   esi               ; None
retn                    ; None
```

Example

From asm to C

```
EDX = 'KEY'
ECX = 'OUT'
sub_007ADB80:
mov   eax, edx          ; EAX = KEY
push  esi               ; None
mov   edx, [ecx+1Ch]    ;EDX = OUT[0x1C]
mov   esi, [ecx+28h]    ;ESI = OUT[0x28]
sub   edx, 354C1FF2h    ;EDX = OUT[0x1C] - 0x354C1FF2
xor   esi, edx          ; ESI = OUT[0x28]^ ( OUT[0x1C] - 0x354C1FF2 )
rol   eax, 3            ; EAX = ROL(KEY, 3)
mov   [ecx+28h], esi    ; OUT[0x28] = OUT[0x28]^ ( OUT[0x1C] - 0x354C1FF2 )
xor   eax, 22E40A3Eh    ;EAX = ROL(KEY, 3) ^ 0x22E40A3E
pop   esi               ; None
retn                    ; None
```

Example

From asm to C

```
EDX = 'KEY'
ECX = 'OUT'
sub_007ADB80:
mov     eax, edx          ; EAX = KEY
push    esi               ; None
mov     edx, [ecx+1Ch]    ;EDX = OUT[0x1C]
mov     esi , [ecx+28h]   ;ESI = OUT[0x28]
sub     edx, 354C1FF2h    ;EDX = OUT[0x1C] - 0x354C1FF2
xor     esi , edx         ; ESI = OUT[0x28]^ ( OUT[0x1C] - 0x354C1FF2 )
rol     eax, 3            ; EAX = ROL(KEY, 3)
mov     [ecx+28h], esi    ; OUT[0x28] = OUT[0x28]^ ( OUT[0x1C] - 0x354C1FF2 )
xor     eax, 22E40A3Eh    ;EAX = ROL(KEY, 3) ^ 0x22E40A3E
pop     esi               ; None
retn                      ; None
```

Example

From asm to C

```
EDX = 'KEY'
ECX = 'OUT'
sub_007ADB80:
mov     eax, edx          ; EAX = KEY
push    esi               ; None
mov     edx, [ecx+1Ch]    ;EDX = OUT[0x1C]
mov     esi, [ecx+28h]    ;ESI = OUT[0x28]
sub     edx, 354C1FF2h    ;EDX = OUT[0x1C] - 0x354C1FF2
xor     esi, edx          ; ESI = OUT[0x28]^ ( OUT[0x1C] - 0x354C1FF2 )
rol     eax, 3            ; EAX = ROL(KEY, 3)
mov     [ecx+28h], esi    ; OUT[0x28] = OUT[0x28]^ ( OUT[0x1C] - 0x354C1FF2 )
xor     eax, 22E40A3Eh    ;EAX = ROL(KEY, 3) ^ 0x22E40A3E
pop     esi               ; None
retn                      ; None
```

Example

From asm to C

```
EDX = 'KEY'
ECX = 'OUT'
sub_007ADB80:
mov     eax, edx         ; EAX = KEY
push    esi              ; None
mov     edx, [ecx+1Ch]   ;EDX = OUT[0x1C]
mov     esi, [ecx+28h]   ;ESI = OUT[0x28]
sub     edx, 354C1FF2h   ;EDX = OUT[0x1C] - 0x354C1FF2
xor     esi, edx         ; ESI = OUT[0x28]^ ( OUT[0x1C] - 0x354C1FF2 )
rol     eax, 3           ; EAX = ROL(KEY, 3)
mov     [ecx+28h], esi   ; OUT[0x28] = OUT[0x28]^ ( OUT[0x1C] - 0x354C1FF2 )
xor     eax, 22E40A3Eh   ;EAX = ROL(KEY, 3) ^ 0x22E40A3E
pop     esi              ; None
retn                     ; None
```

Example

From asm to C

```
EDX = 'KEY'
ECX = 'OUT'
sub_007ADB80:
mov   eax, edx          ; EAX = KEY
push  esi               ; None
mov   edx, [ecx+1Ch]    ;EDX = OUT[0x1C]
mov   esi, [ecx+28h]    ;ESI = OUT[0x28]
sub   edx, 354C1FF2h    ;EDX = OUT[0x1C] - 0x354C1FF2
xor   esi, edx          ; ESI = OUT[0x28]^ ( OUT[0x1C] - 0x354C1FF2 )
rol   eax, 3            ; EAX = ROL(KEY, 3)
mov   [ecx+28h], esi    ; OUT[0x28] = OUT[0x28]^ ( OUT[0x1C] - 0x354C1FF2 )
xor   eax, 22E40A3Eh    ;EAX = ROL(KEY, 3) ^ 0x22E40A3E
pop   esi               ; None
retn                    ; None
```

Example

From asm to C

```
EDX = 'KEY'
ECX = 'OUT'
sub_007ADB80:
mov   eax, edx          ; EAX = KEY
push  esi               ; None
mov   edx, [ecx+1Ch]    ;EDX = OUT[0x1C]
mov   esi, [ecx+28h]    ;ESI = OUT[0x28]
sub   edx, 354C1FF2h    ;EDX = OUT[0x1C] - 0x354C1FF2
xor   esi, edx          ; ESI = OUT[0x28]^ ( OUT[0x1C] - 0x354C1FF2 )
rol   eax, 3            ; EAX = ROL(KEY, 3)
mov   [ecx+28h], esi    ; OUT[0x28] = OUT[0x28]^ ( OUT[0x1C] - 0x354C1FF2 )
xor   eax, 22E40A3Eh    ;EAX = ROL(KEY, 3) ^ 0x22E40A3E
pop   esi               ; None
retn                    ; None
```

Example

From asm to C

```
EDX = 'KEY'
ECX = 'OUT'
sub_007ADB80:
mov     eax, edx          ; EAX = KEY
push    esi               ; None
mov     edx, [ecx+1Ch]    ;EDX = OUT[0x1C]
mov     esi, [ecx+28h]    ;ESI = OUT[0x28]
sub     edx, 354C1FF2h    ;EDX = OUT[0x1C] - 0x354C1FF2
xor     esi, edx          ; ESI = OUT[0x28]^ ( OUT[0x1C] - 0x354C1FF2 )
rol     eax, 3            ; EAX = ROL(KEY, 3)
mov     [ecx+28h], esi    ; OUT[0x28] = OUT[0x28]^ ( OUT[0x1C] - 0x354C1FF2 )
xor     eax, 22E40A3Eh    ;EAX = ROL(KEY, 3) ^ 0x22E40A3E
pop     esi               ; None
retn                      ; None
```

Value propagation

C back-end

We can re-generate C code

- Only generate expressions that write output variables
- Discard all other intermediate expressions
- Some execution flow informations are needed to correctly update variables

Example

Keep only out values writing

```
EDX = 'KEY'
ECX = 'OUT'
sub_007ADB80:
mov eax, edx ; EAX = KEY
push esi ; None
mov edx, [ecx+1Ch] ;EDX = OUT[0x1C]
mov esi, [ecx+28h] ;ESI = OUT[0x28]
sub edx, 354C1FF2h ;EDX = OUT[0x1C] - 0x354C1FF2
xor esi, edx ; ESI = OUT[0x28]^ ( OUT[0x1C] - 0x354C1FF2 )
rol eax, 3 ; EAX = ROL(KEY, 3)
mov     [ecx+28h], esi ; OUT[0x28] = OUT[0x28]^ ( OUT[0x1C] - 0x354C1FF2 )
xor     eax, 22E40A3Eh ;EAX = ROL(KEY, 3) ^ 0x22E40A3E
pop esi ; None
retn ; None
```

Example

Keep only out values writing

```
EDX = 'KEY'
ECX = 'OUT'
sub_007ADB80:
mov eax, edx ; EAX = KEY
push esi ; None
mov edx, [ecx+1Ch] ;EDX = OUT[0x1C]
mov esi, [ecx+28h] ;ESI = OUT[0x28]
sub edx, 354C1FF2h ;EDX = OUT[0x1C] - 0x354C1FF2
xor esi , edx ; ESI = OUT[0x28]^ ( OUT[0x1C] - 0x354C1FF2 )
rol eax, 3 ; EAX = ROL(KEY, 3)
mov    [ecx+28h], esi ; OUT[0x28] = OUT[0x28]^ ( OUT[0x1C] - 0x354C1FF2 )
xor    eax, 22E40A3Eh ;EAX = ROL(KEY, 3) ^ 0x22E40A3E
pop esi ; None
retn ; None
```

Value propagation

Final code

```
unsigned int sub_007ADB80(unsigned char* OUT, unsigned int KEY)
{
OUT[0x28] = OUT[0x28]^ ( OUT[0x1C] - 0x354C1FF2 );
return ROL(KEY, 3) ^ 0x22E40A3E;
}
```

Sub-functions

Generate function calls

To generate calls to sub-functions, we need to know arguments form

```
AND     R0, R0, #0xFF
MOV     LR, PC
MOV     PC, R2
LDR     R3, =sin
LDR     R2, [R3]
MOV     LR, PC
MOV     PC, R2      ; sin ( r0 )
```

Function description

```
func_imported ={
                '__addd':['R0', 'R2', 'R3'],
                'sqrt':['R0'],
                'cos':['R0'],
                'sin':['R0'],
                '__ltd':['R0', 'R2', 'R3'],
        ...
```

Example

```
void sub_16957C(unsigned int *TAB, unsigned int IN_KEY){
    unsigned int tmp_var_507;
    unsigned int tmp_var_0;
    unsigned int tmp_var_6;
    unsigned int tmp_var_224;
    unsigned int tmp_var_96;
    unsigned int tmp_var_522;
    unsigned int tmp_var_408;
    unsigned int tmp_var_62;
    unsigned int tmp_var_31;

    try{
        tmp_var_0 = ( ( ( TAB [ 48 ] ^ TAB [ 28 ] ) ^ IN_KEY ) - \
        ( LSR ( mul_64_h ( ( ( TAB [ 48 ] ^ TAB [ 28 ] ) ^ IN_KEY ) , 0x38E38E39 ) , 1 )
        + LSL ( LSR ( mul_64_h ( ( ( TAB [ 48 ] ^ TAB [ 28 ] ) ^ IN_KEY ) , 0x38E38E39 )
        if (!( ( tmp_var_0 != 8 ) ))
        {
            sub_16254C ( TAB , 0xBC04BB40 );
            sub_165880 ( TAB , 0x141586A );
            sub_1645CC ( TAB , TAB [ 60 ] );
        }

        tmp_var_6 = ( ( LSL ( TAB [ 64 ] , 4 ) - TAB [ 64 ] ) + TAB [ 16 ] ) ;
        TAB [ 16 ] = tmp_var_6 ;
        if (!( ( tmp_var_0 != 0 ) ))
        {
            sub_1656B0 ( TAB , 0x1CB835FD );
            sub_166D34 ( TAB , 0x835400E0 );
            sub_164374 ( TAB , TAB [ 64 ] );
```

Demo

☣ Warning!

This is *not* generic asm2c program

- It can only recover simple expressions
- It doesn't support complex flow graph
- Don't think about taking an OS and recovering it's source code ☺

Outline

Conclusion

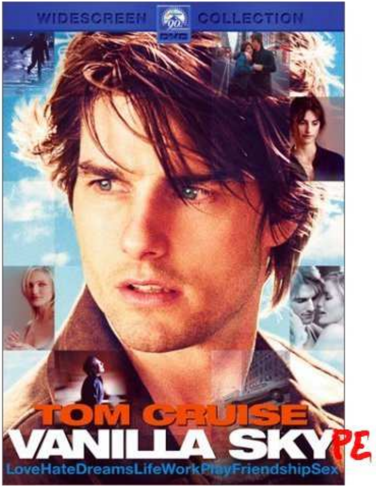
Automated reversing

- Using simple hypothesis, it works
- But it can be improved...

Future work

- Better execution flow analyzer
- Detection of trivial dead-code (or opaque conditions)
- Automatic variable finding (IN/OUT variables)
- Stack analyzer for local variable manipulation
- Python Back-end ☺

Conclusion
Questions?



Outline

8 References

References

- P. Biondi, *Scapy*
  http://www.secdev.org/projects/scapy/
- F. Desclaux, *RR0D: the Rasta Ring 0 Debugger*
  http://rr0d.droids-corp.org/