Back from the Dead

Reverse Engineering the Skype Protocol, Decades Later

[me]

Who Am I?

- Hobbyist reverse engineer and aspiring cybersecurity professional
- Fascinated with how software works (old and new)
- Technology necromancer

Just In Case You Don’t Know…

- Skype was a popular messaging app created in 2003
- Had hundreds of millions of active users at its peak in the early 2010s
- The last remaining version was shut down on May 5th, 2025
- Mostly forgotten about in modern years, but highly nostalgic for others

Why Is This Important?

- As you'll see, Skype was a great lesson in the woes of "security through obscurity"
  - Lots of protections implemented into the software and protocol, that were ultimately circumvented by reverse engineers
- Despite being fairly old, it still uses some good cryptography
  - RSA and AES, Diffie-Hellman, etc.
  - There is some weird cryptography too, but we'll get to that later
- Also a great learning experience
  - Skype's heavy obfuscation, both over the network and in the application itself, is great practice for modern reverse engineering

The Beginnings

- Using Skype 6.16.0.105 as an initial reference
- Initial encryption routine - XOR key that is rotated left 3 bits every iteration
- Skips over the initial entry point, obviously
- There's a second layer of unpacking after this that is a bit more complex

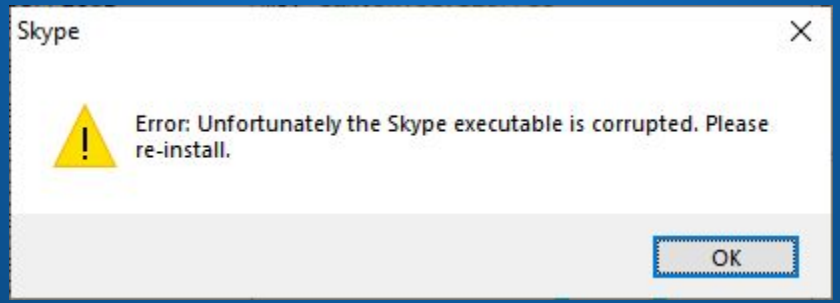
```
void entry(void)

{
  uint uVar1;
  code *pcVar2;

  uVar1 = 0x8cbd9371;
  pcVar2 = (code *)&DAT_005a3f20;
  do {
    if (pcVar2 == entry) {
      pcVar2 = (code *)0x5b9e50;
    }
    *pcVar2 = (code)((byte)*pcVar2 ^ (byte)uVar1);
    uVar1 = uVar1 << 3 | uVar1 >> 0x1d;
    pcVar2 = pcVar2 + 1;
  } while (pcVar2 != (code *)&DAT_005ba080);
  return;
}
```

Before Unpacking: Debugger Woes

- Traditional breakpoints don't work, because they change the code and can mess up decryption results or checksums
  - Solution: Hardware breakpoints instead of INT3 (x32dbg: Breakpoints -> Set Hardware on Execution)
  - Only problem: you can only have four at once
- Dead code during unpacking process (example: inc cl; dec cl)



Before Unpacking: Debugger Wins

- Hardcoded checks for the device names `\\.\NTICE`, `\\.\SICE` and `\\.\Siwvid`
- These device names belong to SoftICE
- SoftICE is an old kernel-mode debugger that hasn't been updated since the year 2000
- x32dbg FTW :)

```
5C 5C 2E 5C 53 49 43 45                                  \\.\SICE
```

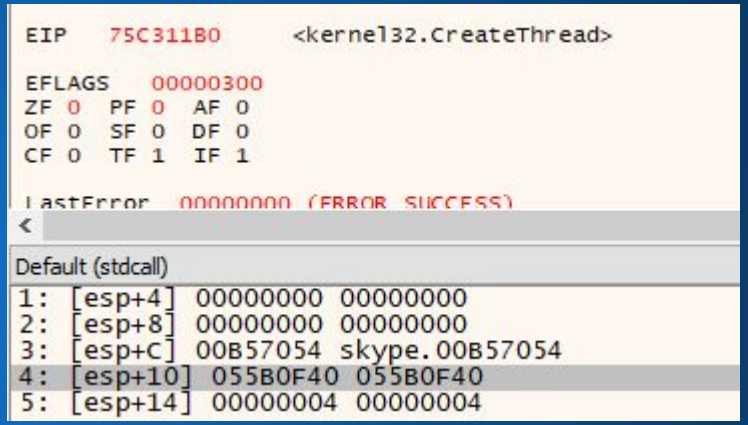
```
00F04191    C706 5C5C2E5C           mov dword ptr ds:[esi],5C2E5C5C
00F04197    C746 04 53494345        mov dword ptr ds:[esi+4],45434953
00F0419E    C646 08 00              mov byte ptr ds:[esi+8],0
00F041A2    FFD0                    call eax
00F041A4    83F8 FF                 cmp eax,FFFFFFFF
00F041A7    0F85 8D000000           jne skype.F0423A
00F041AD    C706 5C5C2E5C           mov dword ptr ds:[esi],5C2E5C5C
00F041B3    C746 04 53697776        mov dword ptr ds:[esi+4],76776953
00F041BA    66:C746 08 6964         mov word ptr ds:[esi+8],6469
00F041C0    C646 0A 00              mov byte ptr ds:[esi+A],0
```

SoftICE

Original author(s)	NuMega
Developer(s)	Compuware
Initial release	1987; 38 years ago (DOS)
Final release	v4.05 / 2000; 25 years ago[1]

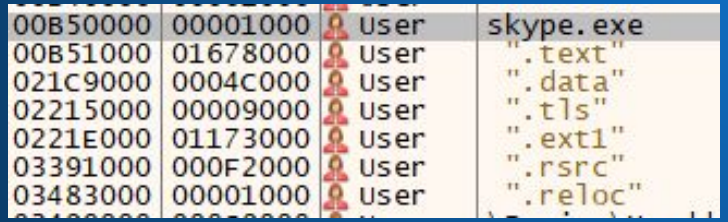
Threads, Needles, and More

- Breaking on CreateThread gets us an address within the unpacked Skype binary
- At this point, we can't be sure if it's an original entry point, but we can at least look around to try and find it

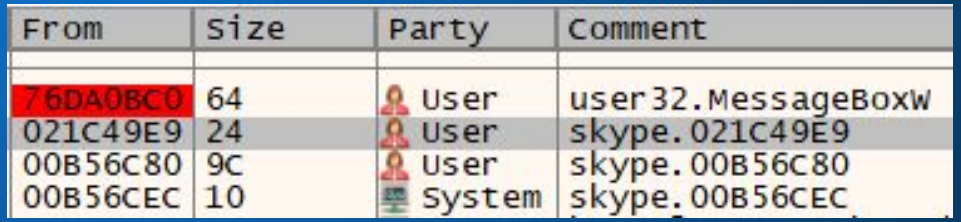


Where Is My Code???

- Single-stepping led me to that “corruption” message from earlier
- But the call is in a completely different memory space, one that x32dbg doesn’t even know about
- What is going on?



00B50000	00001000	User	skype.exe
00B51000	01678000	User	".text"
021C9000	0004C000	User	".data"
02215000	00009000	User	".tls"
0221E000	01173000	User	".ext1"
03391000	000F2000	User	".rsrc"
03483000	00001000	User	".reloc"



From	Size	Party	Comment
76DA0BC0	64	User	user32.MessageBoxW
021C49E9	24	User	skype.021C49E9
00B56C80	9C	User	skype.00B56C80
00B56CEC	10	System	skype.00B56CEC

Introducing: SkyLib

- Turns out, Skype has two different binaries embedded into the main executable
  - The Skype client itself, written in Delphi
  - “SkyLib”, the backend code handling networking, etc. written in C++
-