Automatically Unpacking IcedID Stage 1 with angr

Published: 2022-05-30 · Archived: 2026-04-05 13:32:49 UTC

It started with 0verfl0w posting a small challenge on the Zero 2 Automated discord server asking to automatically extract the configuration of an unpacked IcedID sample (0581f0bf260a11a5662d58b99a82ec756c9365613833bce8f102ec1235a7d4f7).

Unpacking the sample was part of the exercise but could be done manually as a one shot, however the more I looked into the stager, the more i thought an automated unpacker would be a fun thing to do.

I'll skip over some details of the stager (like API hashing and injection) to focus only on the unpacking part.

TL;DR: full code is available here: https://github.com/matthw/icedid_stage1_unpack.

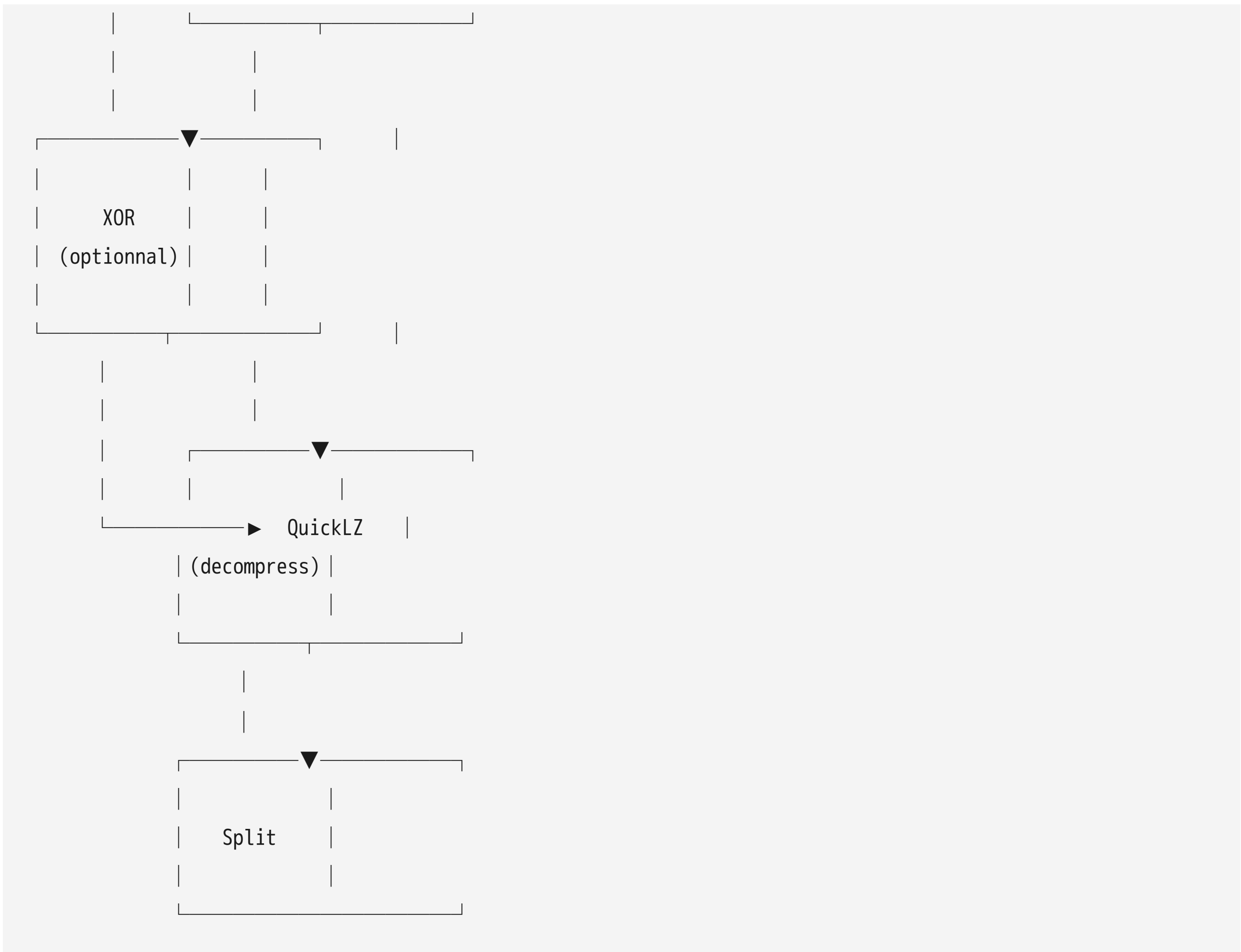
EDIT: it unpacks samples packed with SPLCrypt, including BazarLoader.

1. Structure and Flow

The packed data are really easy to identify: there's a huge hex string in the data section, and by hex string i a mean litteral string of [0-9a-f] characters.

```
 [...]
000af670  33 62 36 64 34 39 61 61  36 35 33 36 31 34 64 65  |3b6d49aa653614de|
000af680  33 31 62 32 66 64 37 31  65 64 38 66 61 30 37 63  |31b2fd71ed8fa07c|
000af690  63 34 30 39 64 64 34 38  61 65 36 35 38 39 31 61  |c409dd48ae65891a|
000af6a0  63 36 33 61 30 39 39 36  31 38 61 63 38 35 30 33  |c63a099618ac8503|
000af6b0  62 34 32 37 39 31 36 63  66 36 31 66 31 31 33 30  |b427916cf61f1130|
000af6c0  37 66 35 39 30 35 33 31  65 37 37 39 35 34 31 33  |7f590531e7795413|
000af6d0  63 64 31 62 32 30 00 00  00 00 00 00 00 00 00 00  |cd1b20..........|
```

The unpacking process is as follow:



The control flow at assembly level is very obfuscated, so the decompiler comes handy even if it doesn't produces perfect results.

```
                       LAB_1800021f2                                   XREF[1]:    180002273(j)
1800021f2 48 89 44 24 48       MOV         qword ptr [RSP + Stack[-0x30]],v1
1800021f7 e9 c6 fe ff ff       JMP         LAB_1800020c2

                       LAB_1800021fc                                   XREF[1]:    18000220a(j)
1800021fc 48 8b 84 24 80 00 00 MOV         v1,qword ptr [RSP + Stack[0x8]]
          00
180002204 eb 58                JMP         LAB_18000225e

                       LAB_180002206                                   XREF[1]:    1800022c4(j)
180002206 83 78 08 00          CMP         dword ptr [v1 + 0x8]=>DAT_56622bde,0x0
18000220a 75 f0                JNZ         LAB_1800021fc
18000220c 48 8b 84 24 80 00 00 MOV         v1,qword ptr [RSP + Stack[0x8]]
          00
180002214 eb 55                JMP         LAB_18000226b

                       LAB_180002216                                   XREF[1]:    18000213d(j)
180002216 85 c0                TEST        v1,v1
180002218 75 19                JNZ         LAB_180002233

                       LAB_18000221a                                   XREF[1]:    18000222c(j)
18000221a 41 b8 01 00 00 00    MOV         param_3,0x1
180002220 e9 5f ff ff ff       JMP         LAB_180002184

                       LAB_180002225                                   XREF[1]:    180002173(j)
180002225 e8 10 02 00 00       CALL        is_valid_hex_chr
                       not hex digit, ciao
18000222a 85 c0                TEST        v1,v1
18000222c 74 ec                JZ          LAB_18000221a
18000222e e9 01 ff ff ff       JMP         LAB_180002134

                       LAB_180002233                                   XREF[1]:    180002218(j)
180002233 8a 4c 24 20          MOV         param_1,byte ptr [RSP + Stack[0x20]]
180002237 e8 9b 03 00 00       CALL        hex_digit_to_int
18000223c e9 72 ff ff ff       JMP         LAB_1800021b3
180002241 9b                   ??          9Bh
180002242 49                   ??          49h    I
180002243 81                   ??          81h
180002244 d3                   ??          D3h
180002245 89                   ??          89h
180002246 04                   ??          04h
```

1.1. Hex decode

The first step is to decode the hex string:

```
for (i = 0; i < length; i = i + 2) {
    chr = hexencoded_data[i];
    next_chr = hexencoded_data[i + 1];

    v1 = is_valid_hex_chr(chr);

    /* not an hex digit, ciao */
    if ((v1 == 0) || (v1 = is_valid_hex_chr(next_chr), v1 == 0)) {
        memset((ulonglong)destination,0,0x10,uVar2,chr,length);
        get_TEB();
        (*RtlFreeHeap)();
        return 0;
    }

    /

    /* convert 1st ascii chr to hex value, ex: 'a' -> 0xa */
    v1 = hex_digit_to_int(chr);
```

```
            /* 0xa -> 0xa0 */
            high4 = (byte)(v1 << 4);

            /* convert 2nd ascii chr to hex value '8' -> 0x8 */
            low4 = hex_digit_to_int(next_chr);

            /* make it a byte: 0xa0 | 0x8 == 0xa8 */
            *(byte *)(*destination + (i >> 1)) = high4 | (byte)low4;
        }
```

this is a plain equivalent to python's `bytes.fromhex(...)`

1.2. RC4

The RC4 routine is easily identified:

```
76  ulonglong RC4(undefined (*param_1) [16],byte *data,ulonglong data_size,byte *key,undefined *key_len)
77
78  {
79      int j;
80      int x;
81      int n;
82      int i;
83      int ctr;
84      undefined4 uStack524;
85      byte S [256];
86      byte k [264];
87      byte tmp;
88
89      j = 0;
90      i = 0;
91                      /* Key-scheduling algorithm (KSA) */
92      for (x = 0; x < 0x100; x = x + 1) {
93          S[x] = (byte)x;
94          k[x] = key[x % (int)key_len];
95      }
96      for (n = 0; n < 0x100; n = n + 1) {
97          j = (int)(j + (uint)S[n] + (uint)k[n]) % 0x100;
98          tmp = S[j];
99          S[j] = S[n];
100         S[n] = tmp;
101     }
102     j = 0;
103                     /* Pseudo-random generation algorithm (PRGA) + XOR */
104     for (ctr = 0; ctr < (int)data_size; ctr = ctr + 1) {
105         i = (i + 1) % 0x100;
106         j = (int)(j + (uint)S[i]) % 0x100;
107         tmp = S[j];
108         S[j] = S[i];
109         S[i] = tmp;
110         data[ctr] = data[ctr] ^ S[(int)(((uint)S[i] + (uint)S[j]) % 0x100)];
111     }
112     *param_1 = CONCAT412(uStack524,CONCAT48((int)data_size,data));
113     return (ulonglong)param_1;
114 }
```

The parameters 4 and 5 are respectively a pointer to the key and the length of the key (which is always 4 apparently).

1.3. XOR

The XOR was not present in all samples i checked, but when applied, it reuses the RC4 key.

```
44 ulonglong xor(byte *data,int data_len,byte *key,byte *keylen,undefined8 param_5)
45
46 {
47     int iVar1;
48     longlong i;
49
50     for (i = 0; i <= (longlong)(ulonglong)(data_len - 1); i = i + 1) {
51         iVar1 = (int)i >> 0x1f;
52         data[SUB164(ZEXT1216(CONCAT48(iVar1,i)) % SEXT816((longlong)(ulonglong)(uint)data_len),0)] =
53              (data[SUB164(ZEXT1216(CONCAT48(iVar1,i)) % SEXT816((longlong)(ulonglong)(uint)data_len),0)] ^
54              key[SUB164(ZEXT1216(CONCAT48(iVar1,i)) % SEXT816((longlong)((ulonglong)keylen & 0xffffffff)),0)]) -
55              data[SUB164(ZEXT1216(CONCAT48((int)(i + 1) >> 0x1f,i + 1)) % SEXT816((longlong)(ulonglong)(uint)data_len),0)];
56     }
57     return (ulonglong)(data_len - 1);
58 }
```

It looks intimidating but in reality it can be translated to:

```
for x in range(len(data) - 1):
    data[x] = ((data[x] ^ key[x % len(key)]) - data[x + 1]) & 0xff
```

1.3. QuickLZ

The QuickLZ part was harder to identify. On the first sample I analyzed, there was no compression applied, so at this point the decrypted data looked OK

I could find a valid PE file inside the decrypted data:

```
 00000db0  7c 4d 5a 90 00 03 00 00  00 04 00 00 00 ff ff 00  ||MZ............|
00000dc0  00 b8 00 00 00 00 00 00  00 40 00 00 00 00 00 00  |.........@......|
00000dd0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000de0  00 00 00 00 00 00 00 00  00 00 00 00 00 d0 00 00  |................|
00000df0  00 0e 1f ba 0e 00 b4 09  cd 21 b8 01 4c cd 21 54  |.........!..L.!T|
00000e00  68 69 73 20 70 72 6f 67  72 61 6d 20 63 61 6e 6e  |his program cann|
00000e10  6f 74 20 62 65 20 72 75  6e 20 69 6e 20 44 4f 53  |ot be run in DOS|
00000e20  20 6d 6f 64 65 2e 0d 0d  0a 24 00 00 00 00 00 00  | mode....$......|
00000e30  00 21 c9 10 93 65 a8 7e  c0 65 a8 7e c0 65 a8 7e  |.!...e.~.e.~.e.~|
```

but i still noticed some kind of header at the very beginning of the extracted data, and that the dword starting at offset 1 was actually the size of the data blob

```
                         flags
00000000 4e                         ??              4Eh    N
                         compressed size
00000001 4b c0 04 00                ddw             4C04Bh
                         decompressed size
00000005 42 c0 04 00                ddw             4C042h
00000009 48 8b c4                   MOV             RAX,RSP
0000000c 48 89 58 08                MOV             qword ptr [RAX + 0x8],RBX
00000010 4c 89 48 20                MOV             qword ptr [RAX + 0x20],R9
00000014 4c 89 40 18                MOV             qword ptr [RAX + 0x18],R8
00000018 48 89 50 10                MOV             qword ptr [RAX + 0x10],RDX

                         **************************************************************
                         *                          FUNCTION                          *
                         **************************************************************
                         undefined FUN_0000001c()
      undefined            AL:1          <RETURN>
      undefined8           Stack[0x8]:8  local_res8                          XREF[1]:
                         FUN_0000001c
0000001c 55                         PUSH            RBP
0000001d 56                         PUSH            RSI
0000001e 57                         PUSH            RDI
0000001f 41 54                      PUSH            R12
```

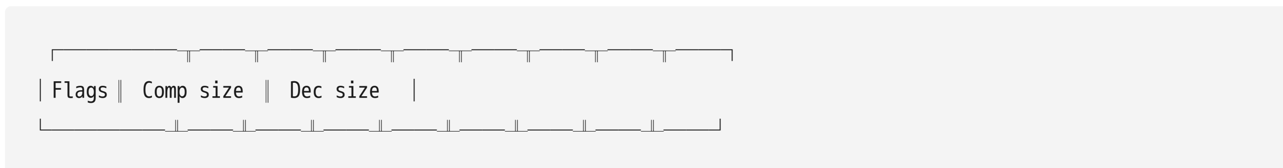
I thought it was some kind of internal structure I could just ignore, until I started having issues with some samples where the embedded PE file seemed corrupt:

```
 00000c40  3e eb df 8f 46 03 4d 5a  90 00 03 f3 03 00 80 46  |>...F.MZ.......F|
00000c50  8a 10 ff ff cd 17 20 c6  00 06 93 fb 01 00 0b f2  |...... .........|
00000c60  03 0e 1f ba 0e 00 b4 09  cd 21 b8 01 4c cd 21 54  |.........!..L.!T|
00000c70  68 69 73 20 70 72 00 00  00 80 6f 67 72 61 6d 20  |his pr....ogram |
00000c80  63 61 6e 6e 6f 74 20 62  65 20 72 75 6e 20 69 6e  |cannot be run in|
00000c90  20 44 4f 53 20 6d 6f 64  65 20 40 10 c4 2e 0d 0d  | DOS mode @.....|
00000ca0  0a 24 12 11 21 c9 10 93  65 a8 7e c0 16 01 42 6e  |.$..!...e.~...Bn|
00000cb0  05 c0 67 20 16 ca 7f c1  6e 0a 05 7f c0 4f 20 08  |..g ....n....O .|
00000cc0  0a 07 88 83 cc 7a 0a 04  83 cc 7e c1 64 0a 02 7c  |.....z....~.d..||
00000cd0  0a 02 52 69 63 68 06 0e  16 25 8b 5e 03 64 86 07  |..Rich...%.^.d..|
```

After some time staring at the code, it turned out that it's using [QuickLZ](#).

Major pointers were:

- the header format, as described [here](#):



- finding code like this

```
Decompile: quicklz_decompress - (0581f0bf260a11a5662d58b99a82ec756c9365613833bce8f102ec1235a7d4f7.bin)
49      local_58 = 2;
50      local_54 = 0;
51      local_50 = 1;
52      local_4c = 0;
53      local_48 = 3;
54      local_44 = 0;
55      local_40 = 1;
56      local_3c = 0;
57      local_38 = 2;
58      local_34 = 0;
59      local_30 = 1;
60      local_2c = 0;
61      local_80 = (undefined4 *)param_2;
62      do {
63          if (uVar3 == 1) {
64              uVar3 = thunk_FUN_1800049ce(puVar6,4);
65              puVar6 = puVar6 + 1;
66          }
67          uVar2 = thunk_FUN_1800049ce(puVar6,4);
68          if ((uVar3 & 1) == 1) {
69              uVar3 = uVar3 >> 1;
70              if ((uVar2 & 3) == 0) {
71                  local_78 = (uVar2 & 0xff) >> 2;
72                  local_88 = 3;
73                  puVar6 = (undefined4 *)((longlong)puVar6 + 1);
74              }
75              else if ((uVar2 & 2) == 0) {
76                  local_78 = (uVar2 & 0xffff) >> 2;
77                  local_88 = 3;
78                  puVar6 = (undefined4 *)((longlong)puVar6 + 2);
79              }
80              else if ((uVar2 & 1) == 0) {
81                  local_78 = (uVar2 & 0xffff) >> 6;
82                  local_88 = (uVar2 >> 2 & 0xf) + 3;
83                  puVar6 = (undefined4 *)((longlong)puVar6 + 2);
84              }
85              else if ((uVar2 & 0x7f) == 3) {
86                  local_78 = uVar2 >> 0xf;
87                  local_88 = (uVar2 >> 7 & 0xff) + 3;
88                  puVar6 = puVar6 + 1;
89              }
90              else {
91                  local_78 = uVar2 >> 7 & 0x1ffff;
92                  local_88 = (uVar2 >> 2 & 0x1f) + 2;
93                  puVar6 = (undefined4 *)((longlong)puVar6 + 3);
94              }
95              local_28 = (longlong)local_80 - (ulonglong)local_78;
96              thunk_FUN_180004459((byte *)local_80,local_28,(char)local_88,(undefined)param_4,CONCAT44(uVar3,uVar2),(int)puVar6);
97          }
```

looking very very similar to https://github.com/sergey-dryabzhinsky/python-quicklz/blob/master/quicklz.c#L630

```
630 #elif QLZ_COMPRESSION_LEVEL == 3
631                 ui32 offset;
632                 cword_val = cword_val >> 1;
633                 if ((fetch & 3) == 0)
634                 {
635                         offset = (fetch & 0xff) >> 2;
636                         matchlen = 3;
637                         src++;
638                 }
639                 else if ((fetch & 2) == 0)
640                 {
641                         offset = (fetch & 0xffff) >> 2;
642                         matchlen = 3;
643                         src += 2;
644                 }
645                 else if ((fetch & 1) == 0)
646                 {
647                         offset = (fetch & 0xffff) >> 6;
648                         matchlen = ((fetch >> 2) & 15) + 3;
649                         src += 2;
650                 }
651                 else if ((fetch & 127) != 3)
652                 {
653                         offset = (fetch >> 7) & 0x1ffff;
654                         matchlen = ((fetch >> 2) & 0x1f) + 2;
655                         src += 3;
656                 }
657                 else
658                 {
659                         offset = (fetch >> 15);
660                         matchlen = ((fetch >> 7) & 255) + 3;
661                         src += 4;
662                 }
663 
664                 offset2 = dst - offset;
665 #endif
```

Luckily for us there's python bindings for QuickLZ, which work just fine: https://pypi.org/project/pyquicklz/.

1.4 Split

The decrypted data blob can be split at every occurences of the `|SPL|` marker, the string is build on the stack:

```
37      marker = '|';
38      uStack111 = 'S';
39      uStack110 = 'P';
40      uStack109 = 'L';
41      uStack108 = '|';
42      iStack116 = 0;
43      iStack100 = 0;
44      iStack104 = 0;
45      while ((position = find_marker(*(byte **)data,(ulonglong)*(uint *)(data + 8),(char)&marker,5,iStack116), position != -1 &&
46            (iStack104 = iStack104 + 1, iStack104 < 4))) {
47          iStack96 = position - iStack116;
48          lVar1 = (longlong)iStack100;
49          plStack88 = (longlong *)((longlong)&lStack80 + lVar1);
50          iStack100 = iStack100 + 0x10;
51          if (iStack96 != 0) {
52              *plStack88 = (longlong)iStack116 + *(longlong *)data;
53          }
54          *(int *)((longlong)aiStack72 + lVar1) = iStack96;
55          iStack116 = position + 5;
56      }
```

I kind of skipped the details in my analysis because i do not need them for now.

2. Automating

Right now we have everything we need for unpacking:

- the data blob is easy to grab from the data section, with a regex for example
- RC4 is vanilla
- the XOR is easy to implement
- QuickLZ has python bindings

The only thing we need to be able to recover is the RC4/XOR key, and that's where the fun begins.

The key is not stored as data, instead it's computed in the code and stored on the stack:

```
                     RC4_KEY                                                   XREF[1]:     180001d7e(j)
180001d8a c7 44 24 34 5e 42 c7     MOV        dword ptr [RSP + local_74],0x11c7425e
          11
180001d92 83 44 24 34 68           ADD        dword ptr [RSP + local_74],0x68
180001d97 e9 1d ff ff ff           JMP        LAB_180001cb9
```

so in this case the key is:

```
>>> p32(0x11c7425e + 0x68)
b'\xc6B\xc7\x11'
```

2.1. Failed Approach

my first approach was to match the bytes using a YARA rule like:

```
rule key {
   strings:
      // C74424 34 5E42C711       | mov dword ptr ss:[rsp+34],11C7425E
      // 834424 34 68             | add dword ptr ss:[rsp+34],68
```

```
                $instr = { C7 44 24 ?? ?? ?? ?? ?? 8? ?? 24 ?? ?? }
            condition:
                $instr
        }
```

emulate all the matches with unicorn, fetch the result from the stack and try all values as keys.

The key grabbing looked like this and worked on some samples:

```
def emulate(code):
    """ emulate the potential key instruction and return
    whatever 4 byte value is on the stack (or None)
    """
    ADDR_TEXT  = 0x1000000
    ADDR_STACK = 0x7000000

    mu = Uc(UC_ARCH_X86, UC_MODE_64)
    mu.mem_map(ADDR_TEXT, 0x1000)
    mu.mem_map(ADDR_STACK, 0x1000)

    # copy code
    mu.mem_write(ADDR_TEXT, code)

    # init rsp
    mu.reg_write(UC_X86_REG_RSP, ADDR_STACK)

    # emulate
    try:
        mu.emu_start(ADDR_TEXT, ADDR_TEXT + len(code))
    except unicorn.UcError:
        pass

    # read stack
    stack = mu.mem_read(ADDR_STACK, 0x100)


    # assume there's no null byte
    for v in [stack[i:i+4] for i in range(0, len(stack), 4)]:
        if u32(v) != 0:
            return bytes(v)

    return None




def find_keys(pe):
```

```
    ''' find potential instructions setting the key
    '''
    # find .text
    data = get_section(pe, '.text')

    rule = yara.compile(source="""
            rule key {
                strings:
                    // C74424 34 5E42C711       | mov dword ptr ss:[rsp+34],11C7425E
                    // 834424 34 68             | add dword ptr ss:[rsp+34],68
                    $instr = { C7 44 24 ?? ?? ?? ?? ?? 8? ?? 24 ?? ?? }
                condition:
                    $instr
            }""")

    yara_matches = rule.match(data=data)

    if not len(yara_matches):
        return []

    # potential code snippet setting the key
    key_code = []
    for offset, _, _ in yara_matches[0].strings:
        string = data[offset:offset+16]
        if string[3] == string[11]:
            #print(string)
            key_code.append(string)



    potential_keys = []
    for code in key_code:
        print("--- emulating:")
        #disasm(code)
        key = emulate(code)

        # assume no null byte in key
        if key is not None and not b'\x00' in key:
            potential_keys.append(key)

    return potential_keys
```

```
  ./unpack2.py 0581f0bf260a11a5662d58b99a82ec756c9365613833bce8f102ec1235a7d4f7.bin
--- emulating:
0x1000:  mov      dword ptr [rsp + 0x20], 4
0x1008:  add      dword ptr [rsp + 0x20], 0
0x100d:  jmp      0x1067
```

```
--- emulating:
0x1000:  mov      dword ptr [rsp + 0x34], 0x11c7425e
0x1008:  add      dword ptr [rsp + 0x34], 0x68
--- emulating:
0x1000:  mov      dword ptr [rsp + 0x14], 0xa6
0x1008:  add      dword ptr [rsp + 0x14], 0x5a
--- emulating:
0x1000:  mov      dword ptr [rsp + 0x20], 1
0x1008:  add      dword ptr [rsp + 0x20], 0
0x100d:  jmp      0x101f
--- emulating:
0x1000:  mov      dword ptr [rsp + 0x60], 0
0x1008:  add      dword ptr [rsp + 0x60], 2
0x100d:  jmp      0xfda
--- emulating:
0x1000:  mov      dword ptr [rsp + 0x50], 2
0x1008:  add      dword ptr [rsp + 0x50], 2
0x100d:  cmp      bl, bl
--- emulating:
0x1000:  mov      dword ptr [rsp + 0x70], 0
0x1008:  add      dword ptr [rsp + 0x70], 3
0x100d:  cmp      bp, bp
--- emulating:
0x1000:  mov      dword ptr [rsp + 0x28], 0x800000e0
0x1008:  sub      dword ptr [rsp + 0x28], 0xe0
--- emulating:
0x1000:  mov      dword ptr [rsp + 0x28], 0x800000e0
0x1008:  sub      dword ptr [rsp + 0x28], 0xe0
--- emulating:
0x1000:  mov      dword ptr [rsp + 0x44], 0
0x1008:  add      dword ptr [rsp + 0x44], 3
--- emulating:
0x1000:  mov      dword ptr [rsp + 0x24], 4
0x1008:  add      dword ptr [rsp + 0x24], 0
0x100d:  jmp      0x1073
--- emulating:
0x1000:  mov      dword ptr [rsp + 0x28], 0x800000e0
0x1008:  sub      dword ptr [rsp + 0x28], 0xe0
--- emulating:
0x1000:  mov      dword ptr [rsp + 0x24], 3
0x1008:  add      dword ptr [rsp + 0x24], 1
0x100d:  jmp      0x1086
--- emulating:
0x1000:  mov      dword ptr [rsp + 0x24], 1
0x1008:  add      dword ptr [rsp + 0x24], 0
0x100d:  jmp      0x100f
--- emulating:
```

```
0x1000:  mov      dword ptr [rsp + 0x28], 1
0x1008:  add      dword ptr [rsp + 0x28], 0
0x100d:  jmp      0x1050
0x100f:  nop
--- emulating:
0x1000:  mov      dword ptr [rsp + 0x28], 3
0x1008:  add      dword ptr [rsp + 0x28], 1
0x100d:  jmp      0xff6
--- emulating:
0x1000:  mov      dword ptr [rsp + 0x28], 0x7ffffa1
0x1008:  add      dword ptr [rsp + 0x28], 0x5f
0x100d:  cmp      di, di
found 1 potential keys: [b'\xc6B\xc7\x11']
got 0x4c04b data blob
decrypted data
- dump 0581f0bf260a11a5662d58b99a82ec756c9365613833bce8f102ec1235a7d4f7.bin.dump
found 5 elements
- dumped 0581f0bf260a11a5662d58b99a82ec756c9365613833bce8f102ec1235a7d4f7.bin.extracted.0
- dumped 0581f0bf260a11a5662d58b99a82ec756c9365613833bce8f102ec1235a7d4f7.bin.extracted.1
- dumped 0581f0bf260a11a5662d58b99a82ec756c9365613833bce8f102ec1235a7d4f7.bin.extracted.2
- dumped 0581f0bf260a11a5662d58b99a82ec756c9365613833bce8f102ec1235a7d4f7.bin.extracted.3
- dumped 0581f0bf260a11a5662d58b99a82ec756c9365613833bce8f102ec1235a7d4f7.bin.extracted.4
```

until the flow obfuscation of some samples brought even more fun to the party by putting a jump right in the middle of the key setting:

```
                     RC4_KEY_PART1                                    XREF[1]:     180001b83(j)
180001b6e c7 44 24 74 c6 74 e2     MOV          dword ptr [RSP + key],0x49e274c6
          49
180001b76 e9 a9 fd ff ff           JMP          RC4_KEY_PART2
```

```
                     RC4_KEY_PART2                                    XREF[1]:     180001b76(j)
180001924 83 44 24 74 66           ADD          dword ptr [RSP + Stack[-0x104]],0x66
180001929 c7 44 24 20 02 00 00     MOV          dword ptr [RSP + Stack[-0x158]],0x2
          00
180001931 66 3b ff                 CMP          DI,DI
180001934 74 5c                    JZ           LAB_180001992
```

rendering my initial and pretty naive approach useless.

2.2. angr

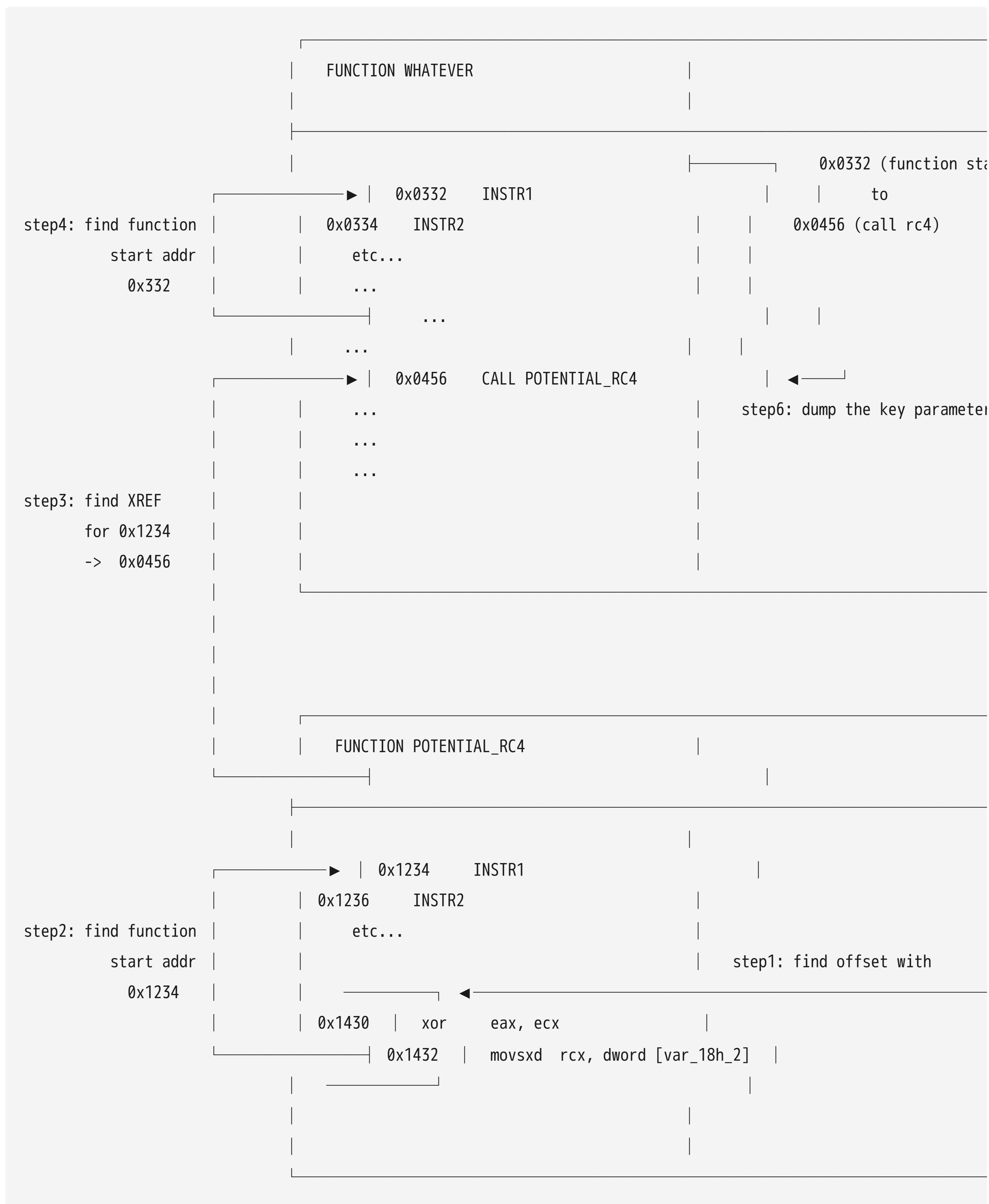
I only played with angr before to solve crackmes and CTF challenges, and I wanted to do something else with it to practice (because let's face it, i'm pretty bad with angr).

To quote their website:

```
 angr is an open-source binary analysis platform for Python.
It combines both static and dynamic symbolic ("concolic") analysis, providing tools to solve a variety of tasks.
```

My new goal was:

- Identify the RC4 function using some heuristics
- Walk the control-flow graph (CFG) up to find where it’s called from
- Execute the function(s) calling the RC4 function up to the actual call
- Get the key



Luckily `angr` can provide a CFG and has a sense of “*function*”.

We can define a project and get the CFG:

```
 self.prj = angr.Project(filename, load_options={'auto_load_libs': False})
self.cfg = self.prj.analyses.CFGFast()
```

2.2.1. Finding the RC4 Function

I used some loosy heuristic to find the RC4 but it seems to work well:

```
# find .text
section = get_section(self.pe, '.text')
data = section.get_data()

# oddly enough this seems to match the rc4 function
# fairly accurately
# like
# 0x1800027c2      33c1                   xor     eax, ecx
# 0x1800027c4      48634c2418             movsxd  rcx, dword [var_18h_2]
# or
# 0x180004242      0fb68c0cd0000000       movzx   ecx, byte [rsp + rcx + 0xd0]
# 0x18000424a      33c1                   xor     eax, ecx
# 0x18000424c      e974feffff             jmp     0x1800040c5 ; fcn.180003bbf+0x506
rule = yara.compile(source="""
    rule rc4 {
        strings:
            //$s1 = { 33 c1 }
            $s2 = { 33 c1 48 63 4c 24 ?? }
            $s3 = { 33 c1 (e9 | 3a) }
        condition:
            $s2 or $s3
    }""")



# get matching offsets
yara_matches = rule.match(data=data)
```

Then we just need to fix the offsets - which are relative to the start of `.text`, so they match the virtual address:

```
 offsets = []
for offset, _, _ in yara_matches[0].strings:
    # offset are relative to .text, rebase them
```

```
            off = self.pe.OPTIONAL_HEADER.ImageBase + section.VirtualAddress + offset
            offsets.append(off)
```

Using these offsets, we can find the start of the function they live in:

```
        for offset in offsets:
            # find function containg the offset
            func = self.find_func_addr(offset)

            if func is None:
                print("skip 0x%x: not part of a func..."%offset)
                continue

            if not len(func.predecessors):
                print("skip 0x%x: no predecessor..."%offset)
                continue

            if len(func.predecessors) > 2:
                print("skip 0x%s: too many predecessors (%d)"%(func.addr, len(func.predecessors)))
                continue


            print("found potential rc4 code: 0x%x"%func.addr)
```

We discard an offset if:

- it does not belong to a function
- it belongs to a function with no predecessors (meaning it's not called)
- it belongs to a function with more than 2 predecessors (called from more than 2 different places)
    - *the RC4 function should only be called from one place - 2 is being conservative*

Now that we have the start address of a *potential* RC4 function, we need to:

- find the XREF (the `CALL rc4`)
- find the calling function start address

Which is just repeating what we just did:

```
             # list of (start_addr, stop_addr) to emulate
            explorer = []



            for pred in func.predecessors:
                caller = self.find_func_addr(pred.addr)

                # skip some cases where start_addr == stop_addr
```

```
                if caller is not None and caller.addr != pred.addr:
                    explorer.append((caller.addr, pred.addr))
                    print(" * found caller (0x%x -> 0x%x)"%(caller.addr, pred.addr))
```

At this stage, we should have a list in the form of:

```
explorer = [
    (addr_start_func1, addr_call_rc4_in_func1),
    (addr_start_func2, addr_call_rc4_in_func2),
    ...
]
```

We can just emulate from the `function start address` to the `call rc4 function address` and dump the key from register `r9` (according to the [x64 fastcall convention](), r9 holds the 4th parameter - the pointer to the key in our case).

2.2.2. Emulation

The emulation goes as follow:

- create an initial state simulating a function call at our start address
    - *we use the `CALLLESS` option to skip over function calls as they are not related to the key computation*

```
state = self.prj.factory.call_state(addr=start_addr)
state.options.add(angr.options.CALLLESS)
```

- create a `Simulation Manager` and then `step` until one of the state reachs our destination address

```
simgr = self.prj.factory.simulation_manager(state)

while True:
    simgr.step()
    for state in simgr.active:
        key = self.check_state(state, stop_addr)
        if key is not None:
            return key
```

with a few extra conditions to avoid looping when there’s no active path left or when the path gets too complex (arbitrary pick), it looks like this:

```
    def emulate(self, start_addr, stop_addr, max_iter=3000):
       """ symbolic execution from start_addr to stop_addr.
       max_iter is the maximum number of instructions
```

```
        return None if failed, or [r9]
        """
        print("emulating from 0x%x to 0x%x (max iter = %s)"%(start_addr, stop_addr, max_iter))
        state = self.prj.factory.call_state(addr=start_addr)

        # no function call
        state.options.add(angr.options.CALLLESS)

        simgr = self.prj.factory.simulation_manager(state)

        while True:
            # advance all states by one basic block
            simgr.step()
            max_iter -= 1

            # very arbitrary picks
            # we shouldnt run into too complex paths
            if not max_iter or len(simgr.active) > 10 or not len(simgr.active):
                return None

            # check each active
            for state in simgr.active:
                key = self.check_state(state, stop_addr)
                if key is not None:
                    return key

        return None
```

The `check_state` function will check if the current state address is the destination address, and if so, will dereference the value of the R9 register (4th parameter) and read a DWORD in there.

I assume the key is always 4 bytes long, however should it not be the case, its length can be read from the stack (5th function parameter).

```
    def check_state(self, state, stop_addr):
        """ check if a state reached the expected address
            hook potential calls with unconstrained destinations

            returns the key if arrived at destination
        """

        # final destination
        #if state.addr in range(stop_addr, stop_addr+8):
        if state.addr == stop_addr:
            # dereference r9 register and read a DWORD
```

```
            # we assume the key is 4 bytes, we could read its size off the stack
            return p32(state.solver.eval(state.mem[state.regs.r9].uint32_t.resolved))
```

There’s an extra twist in the `check_state` function.

The sample uses API hashing to resolve api proc addresses, and we explicitly told angr to skip function calls (*CALLLESS*). The effect of the CALLLESS flag is that the return value of all function calls will be unconstrained (symbolic).

So what should have been:

```
 address = get_proc_address(0x12345678) // returns 0x18032323
(*0x18032323)(arg1, arg2)
```

becomes:

```
 address = get_proc_address(0x12345678) // returns some symbolic constant
(*????????)(arg1, arg2)
```

and basically angr stops because there’s too many possible paths (for some reason, even with the `CALLLESS` flag).

```
76    pcVar4 = (code *)get_proc_by_hash(param_1,0x726774c,(ulonglong)param_3,param_4,(ulonglong)param_5);
77    pcVar2 = (code *)get_proc_by_hash(CONCAT44(extraout_XMM0_Db,extraout_XMM0_Da),0x7802f749,(ulonglong)param_3,param_4,
78                                      (ulonglong)param_5);
79    lVar3 = (*pcVar4)();
80    (*pcVar2)();
81    (*pcVar2)();
82    puVar9 = &stack0xfffffffffffffee8;
83    lVar8 = (*pcVar2)();
```

call to resolved function (call to a stack address):

```
                         LAB_180001846                                          XREF[1]:    180001818(j)
   180001846 48 8d 4c 24 30          LEA             param_2=>Stack[-0x148],[RSP + 0x30]
   18000184b ff 94 24 90 00 00 00    CALL            qword ptr [RSP + Stack[-0xe8]]
   180001852 e9 5f 03 00 00          JMP             LAB_180001bb6
```

My work around to that was to hook all `CALL` instructions to a temp address by:

- checking if the node successor is reached via a call:

```
if block.vex.jumpkind == 'Ijk_Call':
```

- then checking if the call uses a temp value:

```
if block.vex.next.tag == 'Iex_RdTmp':
```

- looping over the block instructions to find the actual call and hook it with something of ours:

```
            for insn in self.prj.factory.block(block.addr).capstone.insns:
                if insn.mnemonic == 'call':
                    if insn.address not in self.hooks:
                        print("hooking addr=0x%x size=%s"%(insn.address, insn.size))
                        self.prj.hook(insn.address, hook_api_hash, length=insn.size)
                        # in order to avoid hook twice // angr would warn anyway
                        self.hooks.append(insn.address)
```

The hook is very simple and looks like that:

```
def hook_api_hash(state):
    """ hook register calls with this
    """
    # symbolize return value
    state.regs.rax = claripy.BVS('ret', 64)
```

The full `check_state` function looks like this:

```
    def check_state(self, state, stop_addr):
        """ check if a state reached the expected address)
            hook potential call with unconstrained destinatinations

            returns the key if arrived at destination
        """

        # final destination
        #if state.addr in range(stop_addr, stop_addr+8):
        if state.addr == stop_addr:
            # dereference r9 register and read a DWORD
            # we assume the key is 4 bytes, we could read its size off the stack
            return p32(state.solver.eval(state.mem[state.regs.r9].uint32_t.resolved))



        #
        # hook registers calls (api hashing)
        # we want to hook all "call $tmp", otherwise angr gets lost
        # even with angr.options.CALLLESS
        #
        try:
            block = state.block()
        except angr.errors.SimEngineError:
            return None

        # verify that the block ends with a call
        if block.vex.jumpkind == 'Ijk_Call':
```

```
            # the next block is based on tmp value
            if block.vex.next.tag == 'Iex_RdTmp':
                # iterates over block instructions to find the call addr and size
                for insn in self.prj.factory.block(block.addr).capstone.insns:
                    if insn.mnemonic == 'call':
                        if insn.address not in self.hooks:
                            print("hooking addr=0x%x size=%s"%(insn.address, insn.size))
                            self.prj.hook(insn.address, hook_api_hash, length=insn.size)
                            # in order to avoid hook twice // angr would warn anyway
                            self.hooks.append(insn.address)

        return None
```

in the end we, we can just loop over the `(start_addr, stop_addr)` tupples, to get a list of potential RC4 keys:

```
         # emulate all potential calls
        potential_keys = []

        for start, stop in explorer:
            # emulate
            key = self.emulate(start, stop)
            if key:
                potential_keys.append(key)

        return potential_keys
```

2.3. Decrypting

now that we constructed a list of potential keys, we can just try them all. using the QuickLZ header, we can know that we found a correct one by matching the size of the data with what's in the header:

```
def try_to_decrypt(data, potential_keys):
    ''' try all keys with xor and without
        it seems the xor is not always applied
    '''
    for key in potential_keys:
        for apply_xor in [True, False]:
            print("trying key %r / xor=%r"%(key, apply_xor))
            dec = decrypt(data, key, apply_xor)
            if dec is not None:
                return dec

    return None
```

```
def decrypt(data, key, apply_xor):
    ''' decrypt + decompress data
    '''

    # RC4 decrypt
    cipher = ARC4(key)
    dec = bytearray(cipher.decrypt(data))

    # dexor
    if apply_xor:
        for x in range(len(dec) - 1):
            dec[x] = ((dec[x] ^ key[x % len(key)]) - dec[x + 1]) & 0xff

    # Quick check we got valid data
    # ref: quicklz format: https://github.com/ReSpeak/quicklz/blob/master/Format.md
    # DWORD at decrypted data+1 should be the length
    if u32(dec[1:5]) == len(data):
        return quicklz.decompress(bytes(dec))
```

The XOR pass doesn't seem to always be applied, so we try with and without.

Extracting the C2 address and campain ID from the unpacked PE is pretty straight forward.

We just need to XOR 2 32 bytes data blob (from the `.d` section) with each other:

```
2  void decrypt_config(longlong param_1)
3
4  {
5      ulonglong n;
6      ulonglong uVar1;
7
8      n = 0;
9      do {
10         uVar1 = n + 1;
11         (CONFIG_DATA_OR_KEY2 + n)[param_1 + -0x180007fc0] = CONFIG_DATA_OR_KEY[n] ^ CONFIG_DATA_OR_KEY2[n];
12         n = uVar1;
13     } while (uVar1 < 32);
14     return;
15 }
16
```

```
def extract_c2(filename):
    pe = pefile.PE(filename)

    data = get_section(pe, ".d").get_data()
    key = data[:0x20]
    conf = data[0x40:0x40+0x20]
    data = xor(key, conf)

    camp = u32(data[:4])
    c2 = data[4:].split(b'\x00')[0]
```

```
    return {'campaign_id': camp, 'c2': c2}
```

4. Showcase

```
 % ./icedid_stage1_unpack.py 0581f0bf260a11a5662d58b99a82ec756c9365613833bce8f102ec1235a7d4f7.bin
got data blob: 0x4c04b bytes
found potential rc4 code: 0x1800026b3
 * found caller (0x180001b77 -> 0x180001bc0)
emulating from 0x180001b77 to 0x180001bc0 (max iter = 3000)
found 1 potential keys: [b'\xc6B\xc7\x11']
trying key b'\xc6B\xc7\x11' / xor=True
decrypted data: 0x4c042 bytes
found 5 elements
- dumped 0581f0bf260a11a5662d58b99a82ec756c9365613833bce8f102ec1235a7d4f7.bin.extracted.0
- dumped 0581f0bf260a11a5662d58b99a82ec756c9365613833bce8f102ec1235a7d4f7.bin.extracted.1
- dumped 0581f0bf260a11a5662d58b99a82ec756c9365613833bce8f102ec1235a7d4f7.bin.extracted.2
    looks like a PE... {'campaign_id': 109932505, 'c2': b'ilekvoyn.com'}
- dumped 0581f0bf260a11a5662d58b99a82ec756c9365613833bce8f102ec1235a7d4f7.bin.extracted.3
- dumped 0581f0bf260a11a5662d58b99a82ec756c9365613833bce8f102ec1235a7d4f7.bin.extracted.4
```

```
 % ./icedid_stage1_unpack.py samples/17aeebe6c1098a312074b0fdeae6f97339f2d64d66a2b07496bfc1373694a4e3.bin
got data blob: 0x3820 bytes
found potential rc4 code: 0x180003fc1
 * found caller (0x1800011c3 -> 0x180001507)
emulating from 0x1800011c3 to 0x180001507 (max iter = 3000)
found 1 potential keys: [b'k\xfe\xfa\x8b']
trying key b'k\xfe\xfa\x8b' / xor=True
trying key b'k\xfe\xfa\x8b' / xor=False
decrypted data: 0x5714 bytes
found 4 elements
- dumped samples/17aeebe6c1098a312074b0fdeae6f97339f2d64d66a2b07496bfc1373694a4e3.bin.extracted.0
- dumped samples/17aeebe6c1098a312074b0fdeae6f97339f2d64d66a2b07496bfc1373694a4e3.bin.extracted.1
- dumped samples/17aeebe6c1098a312074b0fdeae6f97339f2d64d66a2b07496bfc1373694a4e3.bin.extracted.2
    looks like a PE... {'campaign_id': 429479428, 'c2': b'arelyevennot.top'}
- dumped samples/17aeebe6c1098a312074b0fdeae6f97339f2d64d66a2b07496bfc1373694a4e3.bin.extracted.3
```

```
 % ./icedid_stage1_unpack.py samples/12a692718d21b8dc3a8d5a2715688f533f1a978ee825163d41de11847039393d.bin
got data blob: 0x16064 bytes
skip 0x6442458550: too many predecessors (4)
found potential rc4 code: 0x180003bbf
 * found caller (0x1800016bf -> 0x180001980)
emulating from 0x1800016bf to 0x180001980 (max iter = 3000)
hooking addr=0x18000184b size=7
```

```
hooking addr=0x180001c19 size=7
hooking addr=0x180001c09 size=7
hooking addr=0x180001bdc size=7
found 1 potential keys: [b',u\xe2I']
trying key b',u\xe2I' / xor=True
decrypted data: 0x179f7 bytes
found 5 elements
- dumped samples/12a692718d21b8dc3a8d5a2715688f533f1a978ee825163d41de11847039393d.bin.extracted.0
- dumped samples/12a692718d21b8dc3a8d5a2715688f533f1a978ee825163d41de11847039393d.bin.extracted.1
- dumped samples/12a692718d21b8dc3a8d5a2715688f533f1a978ee825163d41de11847039393d.bin.extracted.2
    looks like a PE... {'campaign_id': 3068011852, 'c2': b'yolneanz.com'}
- dumped samples/12a692718d21b8dc3a8d5a2715688f533f1a978ee825163d41de11847039393d.bin.extracted.3
- dumped samples/12a692718d21b8dc3a8d5a2715688f533f1a978ee825163d41de11847039393d.bin.extracted.4
```

The extracted data blobs are:

- 2 shellcodes
- 1 DLL
- 1 or 2 images:

```
 % file 0581f0bf260a11a5662d58b99a82ec756c9365613833bce8f102ec1235a7d4f7.bin.extracted.*
0581f0bf260a11a5662d58b99a82ec756c9365613833bce8f102ec1235a7d4f7.bin.extracted.0: data
0581f0bf260a11a5662d58b99a82ec756c9365613833bce8f102ec1235a7d4f7.bin.extracted.1: data
0581f0bf260a11a5662d58b99a82ec756c9365613833bce8f102ec1235a7d4f7.bin.extracted.2: PE32+ executable (DLL) (GUI) x
0581f0bf260a11a5662d58b99a82ec756c9365613833bce8f102ec1235a7d4f7.bin.extracted.3: JPEG image data, JFIF standard
0581f0bf260a11a5662d58b99a82ec756c9365613833bce8f102ec1235a7d4f7.bin.extracted.4: JPEG image data, Exif Standard
```

5. Conclusion

While it is certainly not the most optimal method to unpack the samples, it was a fun exercise to do.

The full code is available here: https://github.com/matthw/icedid_stage1_unpack.

Source: https://matth.dmz42.org/posts/2022/automatically-unpacking-icedid-stage1-with-angr/