Cube

Dcoder
dcodr@lavabit.com

1 Introduction

Promix17's latest challenge, Cube [4], offers a little of everything. There are three main layers to penetrate until we have a working key generator:

Anti-{disassembly, debugging} Like its predecessor "Primitive Math" [3], Cube uses a few tricks to make analysis harder: searching for OllyDbg/IDA, decrypting code at runtime, etc. This incarnation brings a rudimentary version of nanomites. These, however, are fairly simple to eliminate by simple analysis of the exception handler at `1701B000`.

Encrypted image The result of a valid serial is only known once we find a valid key. This key unlocks a bitmap image that is checked for correctness using MD5. The scheme employed to encrypt the image seems to be custom, but uses SHA-1 as a building block.

Serial scheme A good old serial scheme, non-cryptographic in nature. This is the most interesting part of the challenge, but we will get into it later.

In the beginning, we are asked for a name and serial. Once a timer is activated (every 500ms), the serial is verified and, if correct, the following image is shown:

Let's see how to get there.

2 Unwrapping the cube

The original binary is packed[1] with what PEiD identifies as UPolyX. After rebuilding the binary, our task is not over yet.

There is one thread created at startup that constantly checks for OllyDbg and IDA window names, also checking for the binary's own integrity. We bypass this by passing the `CREATE_SUSPENDED` flag to `CreateThread`.

After recovering the xor-encrypted sections (very similar to Primitive Math's), there is only one thing left: handle the (poor man's) nanomites. This is the most boring task, but not very complicated. The handler function contains the hardcoded addresses of all the valid `INT 3` interrupts; there are 6 different types of stolen instructions: `JMP IMM8`, `PUSH EBP + MOV EBP, ESP`, `RETN`, `PUSH EAX`, `CALL [IMM32]`, and `CALL IMM32`. Since we have all the addresses neatly laid out for us in the exception handler, recovering the original code is straightforward.

3 Unearthing the cube

Once we get past the analysis-preventing annoyances, we hit the meat of the challenge. The name and serial are taken and used to scramble the string `0WGWb0rWbGGbbYW0rYrYrY0G`, which is then used as the key to a custom stream cipher. This cipher uses the SHA-1 compression function to generate 3 tables of 16, 256, and 512 integers each. These tables are then used to generate the keystream, as shown in Appendix A. This is not a very good cipher: the keystream quickly devolves into 0s, which in a stream cipher is quite fatal, provided one has chosen plaintext.

To verify whether it has found the correct key, Cube computes the MD5 of the decrypted image file and checks it against an hardcoded value (`134F0B02ADBAD5B7-0D78E885BEDB44EC`). Bruteforcing by this method would be very slow, as each attempt would have to decrypt and hash 225734 bytes; instead, we exploit the knowledge of the file type to speed up the search. We check for the first 6 bytes of the BMP header, which correspond to "BM" followed by the size of the file in little endian order. In other words,

```
static string Bruteforce()
{
    char KeySeed[] = "0WGWb0rWbGGbbYW0rYrYrY0G";
    u8 Buffer[16] = {0};

    for(;;)
    {
        CubeCipher CC(KeySeed);
        CC.Crypt(Buffer, ImageFile, 16);

        if( IsValidBmp( Buffer ) )
            return string(KeySeed);

        Permute(KeySeed, rand()%3);
    }
}
```

[1] "Encrypted" is a more suitable term; the loader simply xors the code section with `0x59`.

We quickly find the correct key, `WWWWrrrrGGGGbbbbYYYY0000`, and the above image.

4 Unscrambling the cube

Now that we have the image, it is time to figure out how to find a serial that yields the target key `WWWWrrrrGGGGbbbbYYYY0000` from the scrambled initial value. The process can be described as such:

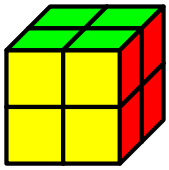
```
template<typename T>
static inline void Permute(T &buf, const size_t n)
{
        for(size_t i = 0; i < n/3 + 1; ++i)
        {
                switch(n%3)
                {
                case 0:
                        Perm1(buf);
                        break;
                case 1:
                        Perm2(buf);
                        break;
                case 2:
                        Perm3(buf);
                        break;
                }
        }
}

template<typename T>
static void MixUser(T &buf, char const *in)
{
        for(auto *i = in; *i; ++i)
                Permute(buf, *i % 9U);

        int x = 0;
        for(auto *i = in; *i; ++i)
        {
                auto const c = *i;
                x = (x + x * c) ^ (c * c);
                Permute(buf, x % 9U);
        }
}

template<typename T>
static void MixSerial(T &buf, char const *in)
{
        for(auto *i = in; *i; ++i)
        {
                auto const c = *i - 'A';
                Permute(buf, c);
        }
}
```

`Perm1`, `Perm2`, and `Perm3` are permutations that can be interpreted as rotation of the faces in a $2 \times 2 \times 2$ Rubik's cube:



Indeed, this is the key insight of this challenge, and the basis of its name! The author also forces us to find good solutions: it is known that any $2 \times 2 \times 2$ cube can be solved in at most 11 moves [5], and the serial's length is indeed limited to 11 characters (read: moves). Furthermore, unlike regular cubes, the orientation of the cube *does* matter, leaving the total number of possible states at $8!\,3^7 \approx 2^{27}$.

There are many heuristics to solve Rubik's cubes efficiently [6, 1]. Were this a computer science class, we would be trying some sort of breadth-first search variant here. However, we can look at Diffie-Hellman's meet-in-the-middle attack [2] for a quick bruteforce solution.

The first step is to precompute every sequence of 5^2 moves starting from the target, and compute the inverse sequence of moves for each one, keeping both the state of the cube and the inverse sequence stored away in a dictionary. Then, try every sequence of moves until we hit one of the states stored away in the dictionary. This reduces the bruteforce search from $9^{11} = 31381059609$ — 9 possible move types, 11 moves at the most — to $9^{11-5} = 531441$, a far more acceptable value. The rest is history.

References

[1] Demaine, Erik D., Martin L. Demaine, Sarah Eisenstat, Anna Lubiw, and Andrew Winslow: *Algorithms for solving Rubik's cubes.* In *Proceedings of the 19th European conference on Algorithms*, ESA'11, pages 689–700, Berlin, Heidelberg, 2011. Springer-Verlag, ISBN 978-3-642-23718-8. `http://dl.acm.org/citation.cfm?id=2040572.2040647`.

[2] Diffie, W. and M. E. Hellman: *Special Feature Exhaustive Cryptanalysis of the NBS Data Encryption Standard.* Computer, 10(6):74–84, June 1977, ISSN 0018-9162. `http://dx.doi.org/10.1109/C-M.1977.217750`.

[3] Promix17: *Primitive math.* `http://crackmes.de/users/promix17/primitive_math/`, March 2011.

[4] Promix17: *Cube.* `http://crackmes.de/users/promix17/cube_by_promix17/`, July 2012.

[5] Scherphuis, Jaap: *Mini cube, the* $2 \times 2 \times 2$ *rubik's cube.* `http://www.jaapsch.net/puzzles/cube2.htm`, 2012.

[2]The constant 5 was chosen somewhat arbitrarily; it seems, however, to be an adequate tradeoff between space and time.

[6] Singmaster, David: *Notes on Rubik's "Magic Cube"*. Penguin Books Ltd, 5th edition, September 1981, ISBN 0140061495. http://www.amazon.com/Notes-Rubiks-Magic-David-Singmaster/dp/0894900439.

A Custom Cipher

```
class CubeCipher
{
    u32 mCounter;
    u32 mKeyIndex;
    u32 mArray512[512];
    u32 mArray256[256];
    u32  mArray16[ 16];

    u32 mKeyBuffer[4096];

    int mShaIdx;
    u32 ShaKey[5];
    u32 ShaBuffer[5];
    u32 ShaWorkspace[16];

    template<typename T>
    void SetKey(T const& buf)
    {
        for(size_t i = 0; i < 5; ++i)
            ShaKey[i] = ByteSwap(reinterpret_cast<const u32 *>(buf)[i]);
        fill(begin(ShaWorkspace), end(ShaWorkspace), 0);
    }

    u32 ShaIterate(const u32 n)
    {
        if( n/5 != mShaIdx )
        {
            copy(begin(ShaKey), end(ShaKey), begin(ShaBuffer));
            ShaWorkspace[0] = mShaIdx = n / 5;
            Sha1Compress(ShaBuffer, ShaWorkspace);
        }
        return ShaBuffer[n % 5];
    }

    inline u32 At(const u32 d)
    {
        return mArray512[(d&0x7FC)>>2];
    }

    inline void Mix(u32 &w0, u32 &w1, u32 &w2, u32 &w3)
    {
        u32 s0, s1, s2, s3;
        s0 = At(w0) + w1;
        s3 = Rotr32(w0, 9);
        w1 = Rotr32(s0, 9);
        s1 = At(s0) + w2;
        w2 = Rotr32(s1, 9);
```

```
    s2 = At(s1) + w3;
    w3 = Rotr32(s2, 9);
    w0 = At(s2) + s3;
}

inline void MixXor(u32 &w0, u32 &w1, u32 &w2, u32 &w3)
{
    u32 t, s;

    t = w0&0x7FC;
    w0 = Rotr32(w0, 9);
    w1 = (At(t) + w1) ^ w0;
    s = w1&0x7FC;
    w1 = Rotr32(w1, 9);
    w2 = (At(s) ^ w2) + w1;
    t += w2&0x7FC;
    w2 = Rotr32(w2, 9);
    w3 = (At(t) + w3) ^ w2;
    s += w3&0x7FC;
    w3 = Rotr32(w3, 9);
    w0 = (At(s) ^ w0) + w3;
    t += w0&0x7FC;
    w0 = Rotr32(w0, 9);
    w1 = (At(t) ^ w1);
    s += w1&0x7FC;
    w1 = Rotr32(w1, 9);
    w2 = At(s) + w2;
    t += w2&0x7FC;
    w2 = Rotr32(w2, 9);
    w3 = At(t) ^ w3;
    s += w3&0x7FC;
    w3 = Rotr32(w3, 9);
    w0 = At(s) + w0;
}

void FillKeyBuffer(const u32 Counter, u32 (&KeyBuffer)[4096])
{
    u32 *BufferPtr = KeyBuffer;
    for(size_t i = 0; i < 4; ++i)
    {
        u32 w0 = mArray16[4*i + 0] ^ Rotr32(Counter,  0);
        u32 w1 = mArray16[4*i + 1] ^ Rotr32(Counter,  8);
        u32 w2 = mArray16[4*i + 2] ^ Rotr32(Counter, 16);
        u32 w3 = mArray16[4*i + 3] ^ Rotr32(Counter, 24);

        for(size_t j = 0; j < 2; ++j)
            Mix(w0, w1, w2, w3);

        u32 u3 = w3;
        u32 u2 = w2;
        u32 u1 = w1;
        u32 u0 = w0;

        Mix(w0, w1, w2, w3);
```

```
            for(size_t k = 0; k < 64; ++k)
            {
                MixXor(w0, w1, w2, w3);
                BufferPtr[0] = ByteSwap(mArray256[4*k + 0] + w1);
                BufferPtr[1] = ByteSwap(mArray256[4*k + 1] ^ w2);
                BufferPtr[2] = ByteSwap(mArray256[4*k + 2] + w3);
                BufferPtr[3] = ByteSwap(mArray256[4*k + 3] ^ w0);
                BufferPtr += 4;
                if(k&1)
                {
                    w0 = u0 + w0;
                    w1 = u2 + w1;
                    w2 = u0 ^ w2;
                    w3 = u2 ^ w3;
                }
                else
                {
                    w0 = u3 + w0;
                    w1 = u1 + w1;
                    w2 = u3 ^ w2;
                    w3 = u1 ^ w3;
                }
            }
        }
    }

    static inline void XorBuffer(u8 *out, const u8 *in, const u8 *key, const size_t n)
    {
        for(size_t i = 0; i < n; ++i)
            out[i] = in[i] ^ key[i];
    }

    inline void Regenerate()
    {
        mKeyIndex = 0;
        mCounter += 1;
        FillKeyBuffer(mCounter, mKeyBuffer);
    }

public:

    template<typename T>
    CubeCipher(T const& buf) : mShaIdx(-1), mCounter(0), mKeyIndex(0)
    {
        SetKey(buf);

        for(size_t i = 0; i < 512; ++i)
            mArray512[i] = ShaIterate(i);
        for(size_t i = 0; i < 256; ++i)
            mArray256[i] = ShaIterate(i + 4096);
        for(size_t i = 0; i < 16; ++i)
            mArray16[i] = ShaIterate(i + 8192);

        FillKeyBuffer(mCounter, mKeyBuffer);
    }
```

```
    void Crypt(u8 *out, const u8 *in, size_t n)
    {
        while( n >= 4096 - mKeyIndex )
        {
            XorBuffer(out, in, (u8 *)(mKeyBuffer) + mKeyIndex, 4096 - mKeyIndex);
            out += 4096 - mKeyIndex;
            in += 4096 - mKeyIndex;
            n  -= 4096 - mKeyIndex;
            Regenerate();
        }
        XorBuffer(out, in, (u8 *)(mKeyBuffer) + mKeyIndex, n);
        mKeyIndex += n;
    }

    ~CubeCipher() {}
};
```