WinLicense 2.x, 3.x File Key Format Exploration

WinLicense is a commercial application developed by Oreans Technologies [1] that can be used by application developers both to obfuscate their programs and to implement various licensing schemes to enforce copy protection. While its obfuscation techniques are often examined in research on code virtualization, there is little in the way of examining the licensing implementation details, save one confusing old forum post [2].

One of the most common licensing schemes employs the use of a license file key generated by the developer of a protected program and supplied to end users to register the program on their machines. This overview will examine the internal structure of this file key and show how it is constructed from the private hashes generated during protection.

This overview is not in any way intended to encourage piracy, and indeed this information cannot be directly used to generate valid key files without the private hashes. We will look at the potential for extracting some of this private information from protected programs in a future overview, but will stop short of providing any sort of low-skill, one-click cracking solution that would enable widespread compromise of protected programs.

All information covered herein was discovered via reverse engineering of file key generation routines within the binary WinlicenseSDK libraries and standalone key generators provided as part of the web activation bundles. These are all freely available within demo versions.

Apparently source for the web activation key generator may be requested by paying customers who need to build it on hardware unsupported by the available binaries. This route was not explored due both to cost and the potential invitation of copyright infringement claims. Since I also wanted to explore how the files are validated in protected programs, it was more useful to get a sense of what the processes involved look like in assembly anyway. I haven't seen the source. I don't want to see the source. None of this comes from the source.

There was an interesting format that hadn't been thoroughly explored, so here it is being explored. Let's get to it.

General Overview

WinLicense versions 2.x and 3.x use the same file key format, usually stored in a file called "regkey.dat" in the same directory as the protected program, though this location can be changed at protection time. Version 1.x was not explored for this overview.

The key format consists of layers of nested sections (cover photo related), each encrypted via different methods with different keys, with some keys and hash values calculated from the external private hashes and some supplied by the license itself.

The private hashes generated at protection time are ASCII strings labeled `main_hash`, `rsa_private_1`, and `rsa_private_2`. The `main_hash` is used to generate keys and hash values necessary to construct the main body of the license. The `rsa_private_1` and `rsa_private_2` strings are base36-encoded RSA private keys used to sign and encrypt the main body, respectively.

`Main_hash` Format

Even though `main_hash` is an ASCII string, different parts of it are treated as numerical values stored in little-endian format, which can be easily referenced into as follows:

```
struct main_hash_bin
{
    uint32_t unused_1[2];       // Unused

    uint32_t hash_1_[2];        // Added together to get hash_1
    uint32_t hash_2_[2];        // Added together to get hash_2
    uint16_t hash_3_[2];        // Added together to get hash_3

    uint32_t tea_key_1_[4];     // tea_key[0] = tea_key_1_[0] + tea_key_2_[0]
    uint32_t tea_key_2_[4];     // etc.

    uint32_t unused_2[2];       // Unused

    uint8_t  password[32];      // Used for password encryption
    uint32_t hwid_key;          // Used for HWID hashing
};
```

So you just take your `main_hash` char pointer and cast it into a `main_hash_bin` pointer to get to the parts you need. For simplicity we're going to assume we're working on little-endian hardware for the rest of this and won't keep repeating it, but if you're not then you're going to have to convert numerical values back and forth when retrieving and storing them from and to the `main_hash` and license buffers.

A full C++ implementation is available at [3]. We won't be working through that code verbatim in this paper, but will try to look at examples that help clarify what is happening at each step.

The `unused` values are just that. Other numerical values extracted from the hash are mostly added together in pairs to form values we need. If we have `main_hash` in a c-string named `cstr_main_hash`, then we can extract all of the information we will need from it like so:

```
main_hash_bin *m = (main_hash_bin*) cstr_main_hash;

uint32_t hash_1 = m->hash_1_[0] + m->hash_1_[1];
uint32_t hash_2 = m->hash_2_[0] + m->hash_2_[1];
uint16_t hash_3 = m->hash_3_[0] + m->hash_3_[1];

uint32_t tea_key[4] = { m->tea_key_1_[0] + m->tea_key_2_[0],
                        m->tea_key_1_[1] + m->tea_key_2_[1],
                        m->tea_key_1_[2] + m->tea_key_2_[2],
                        m->tea_key_1_[3] + m->tea_key_2_[3] };

char password[33];
strncpy(password, m->password, 32);
password[32] = '\0';

uint32_t hwid_key = m->hwid_key;
```

Notice how the hash values and TEA key are constructed by just treating different parts of `main_hash` as little-endian numbers and adding them together. The `password` ASCII string we will need is just copied directly out of the middle of `main_hash`, but other characters originally followed it so we need to null-terminate it ourselves before using it later. The `hwid_key` will be used as is, and with that we have extracted everything we will need from `main_hash` and don't need to worry about it anymore.

License Format

The main license body before RSA signing and encryption is laid out as follows:

```
<license_head>
0xffffffff, 0xffffffff, <optional UTF-16 indicator — 0x214E552A>, 'name\0'
0xffffffff, 0xffffffff, 'company\0'
0xffffffff, 0xffffffff, 'custom\0'
0xffffffff, 0xffffffff, 'hwid'
<license_tail>
```

If the `name`, `company`, or `custom` strings are not included in the license, then their terminating nulls are not either, but the 0xffffffff markers still are to delineate between entries.

The registration information strings are in ASCII format unless the UTF-16 indicator is present, in which case `name`, `company`, and `custom` should be provided in UTF-16 format (but not HWID – it is always uppercase ASCII with dashes.) We will not cover UTF-16 support in-depth.

Registered `name` and `company` are what they sound like. `custom` is data that can be accessed by the protected program if its creator desires, but is not used by WinLicense itself.

License Header

The license header is itself composed of nested layers. Here are the structures of all of them. Note that each subsequent structure has the previous one nested inside of it.

Each sub-structure will get encrypted in turn from the inside out.

```
struct lic_core
{
    uint32_t hash_1;            // Calculated from main_hash
    uint8_t  num_days;          // Num days after registration license is valid
    uint8_t  num_execs;         // Num times program can be run with this license
    uint32_t exp_date;          // Date this license expires
    uint32_t global_minutes;    // Total mins a program can run with this license
    uint32_t country_id;        // Country ID per Windows Language settings
    uint32_t runtime;           // Per-run max mins program can be open
    uint32_t hash_2;            // Calculated from main_hash
    uint8_t  hwid_hash_1;       // HWID verification - we'll cover these in depth later
    uint16_t hwid_hash_2[4];    // " "
};

struct lic_tea
{
    lic_core core;              // The above structure
    uint16_t core_xor_key;      // Xor/add key used to encrypt core
    uint16_t lic_flags;         // Dictate which license restrictions to enforce
    uint32_t random;            // Any random number
    uint16_t str_checksum;      // Registration strings checksum
    uint16_t checksum;          // Chksum on the buffer up to here after core encrypt
    uint32_t magic;             // Must be 0xe2b27878
};

struct license_head
{
    lic_tea  tea;               // The above structure
    uint16_t hash_3;            // Calculated from main_hash
    uint16_t str_checksum;      // The same registration strings checksum as in lic_tea
    uint16_t checksum;          // Checksum on up to here after encryptions
};
```

In practice, we just cast a large buffer to a `license_head` pointer and access items in deeper layers through it, but throughout this overview we will treat the structures separately to clarify what is happening in each layer.

License Core

This is the inner-most layer of the header and the one we have to build first. `hash_1` and `hash_2` are just the values we calculated from `main_hash` earlier:

```
license_core core;

core.hash_1 = hash_1;
core.hash_2 = hash_2;
```

The rest of the core contains information needed to enforce various license “features” (restrictions) if requested by the license, or are filled in with random values otherwise. Most of these are covered in-depth in the WinLicense help files so we’ll just focus on the aspects that aren’t.

Dates like `exp_date` are stored in a format that’s sensible but a bit unorthodox:

```
// Given
uint8_t  month  // Jan = 1, Feb = 2, ...
uint8_t  day    // Day of month, 1 - 31
uint16_t year   // Year as a literal number (eg. 2023)

core.exp_date = (year << 16) | (month << 8) | day;
```

The WinLicense protection mechanism generates a HWID for each machine it runs on that can be incorporated into a license to lock it to a particular machine. These take the form of ASCII strings of 4-digit hexadecimal numbers separated by dashes, such as:

```
"0123-4567-89AB-CDEF-FEDC-BA98-7654-3210"
```

If this is later stored in the strings section, it will be in this form, all uppercase with the dashes included. In order to generate the HWID hashes, it gets churned through some bit manipulation to convert each segment into the numerical word value the ASCII is spelling out:

```
uint16_t bin_hwid = { 0x0123, 0x4567, 0x89AB, 0xCDEF, 0xFEDC, 0xBA98, 0x7654, 0x3210 };
```

The HWID hash values are then computed as follows:

```
// The two individual bytes of the first HWID word are added together
core.hwid_hash_1 = (bin_hwid[0] >> 8) + (bin_hwid[0] & 0xff);

// The next 6 words are added pairwise
core.hwid_hash_2[0] = bin_hwid[1] + bin_hwid[2];
core.hwid_hash_2[1] = bin_hwid[3] + bin_hwid[4];
core.hwid_hash_2[2] = bin_hwid[5] + bin_hwid[6];

// Those are all xor’d together with hwid_key, which was pulled from main_hash
core.hwid_hash_2[3] = core.hwid_hash_1    ^ core.hwid_hash_2[0]
                    ^ core.hwid_hash_2[1] ^ core.hwid_hash_2[2]
                    ^ hwid_key;
```

It isn’t a mistake that `bin_hwid[7]` never makes an appearance here. It never gets used at any point to compute the hash.

This covers all the values needed to generate the license core. We will now move up a layer and see how this one gets encrypted.

TEA Layer

The first thing we will do in this layer is generate a random 16-bit key to xor/add encrypt the core layer:

```
lic_tea tea;

// In real code we just build everything in place, but for illustrative purposes:
tea.core = core;           // Core we just created
tea.core_xor_key = rnd();  // Any random number to use as xor/add key
```

This key is then used to encrypt the core as such:

```
// Set up our key values
uint8_t key_hi = tea.core_xor_key >> 8;
uint8_t key_lo = tea.core_xor_key & 0xFF;

// Define our encryption range
uint8_t *pos = (uint8_t*) &tea.core;
uint8_t* end = pos + sizeof(tea.core);

// Encrypt each byte
while (pos != end)
{
    *pos = (*pos ^ key_lo) + key_hi;
    pos++;
}
```

Now that the core is encrypted we can fill out the rest of the values in this layer.

`lic_flags` is a bit-mask that determines which license restrictions are enforced. It is produced by bitwise or-ing together any combination of the following:

```
constexpr uint16_t LF_NUMDAYS = 1;
constexpr uint16_t LF_NUMEXECS = 2;
constexpr uint16_t LF_EXPDATE = 4;
constexpr uint16_t LF_HWID = 8;
constexpr uint16_t LF_GLOBMINS = 0x10;
constexpr uint16_t LF_RUNTIME = 0x20;
constexpr uint16_t LF_COUNTRY = 0x40;

// To enforce eg. HWID and number of days restrictions
tea.lic_flags = LF_HWID | LF_NUMDAYS;
```

As expected, `random` is just a random number.

`str_checksum` is produced by adding together the checksums for each of the registration strings that will be included later:

```
tea.str_checksum  = str_chksum(hwid)     + str_chksum(name)
                  + str_chksum(company)  + str_chksum(custom);
```

Where `str_chksum()` is defined as follows:

```
uint16_t str_chksum(const char* str)
{
    if (!str)
        return 0;

    uint8_t sum_hi = 0, sum_lo = 0;

    for (size_t i = 0; str[i] != '\0'; i++)
        if (i % 2)
            sum_lo ^= str[i];
        else
            sum_hi += str[i];

    return (sum_hi << 8) | (sum_lo & 0xff);
}
```

The next value in the TEA layer, simply labeled `checksum` here, is computed by a slightly different method on the entire buffer up to the position of this next checksum:

```
size_t chksum_size = &tea.checksum - &tea;   // Size of buffer up to checksum position
tea.checksum = bin_checksum((uint8_t*) &tea, chksum_size);
```

Where `bin_checksum()` is now defined as:

```
uint16_t bin_checksum(const uint8_t* buf, size_t size)
{
    uint8_t sum_hi = 0, sum_lo = 0;

    for (size_t i = 0; i < size; i++)
        if (i % 2)
            sum_lo += buf[i];
        else
            sum_hi ^= buf[i];

    return (sum_hi << 8) | (sum_lo & 0xFF);
}
```

Notice how the `+=` and `^=` operations have swapped in the binary version of the checksum. I'm not really sure what the reasoning is behind it other than maybe to throw off reverse engineers who aren't paying close attention. It appears we have avoided this trap for now.

We get a reprieve with the last value of the TEA layer, `magic`, as it's just always set to `0xe2b27878`.

TEA Encryption

Before we get to move up another layer, everything we have built so far has to go through a variant of TEA encryption, and then a password encryption.

The algorithm used for "TEA" encryption is a bit unusual, to the extent that I wouldn't really call it that but it's clearly based on it. I couldn't find if there's a name for this particular variant or if it's just something they decided to do for whatever reason. Leaving `key[1]` and `key[3]` exposed the way they have simplifies a differential attack on the intermediate values when recovering the keys from protected binaries, but that's a different paper for a different time:

```
void tea_encrypt(uint32_t* buf, size_t size, uint32_t* key)
{
    size_t num_blocks = size / 8;

    for (size_t cur_block = 0; cur_block < num_blocks; cur_block++)
    {
        uint32_t *b = &buf[cur_block * 2];
        uint32_t sum = 0;

        for (size_t i = 0; i < 32; i++)
        {
            sum += 0x9e3779b9;  //delta
            b[0] += (b[1] << 4) + (b[1] ^ key[0]) + ((b[1] >> 5) ^ sum) + key[1];
            b[1] += (b[0] << 4) + (b[0] ^ key[2]) + ((b[0] >> 5) ^ sum) + key[3];
        }
    }
}
```

The TEA layer is then encrypted with this as follows:

```
tea_encrypt((uint32_t*) tea, sizeof(lic_tea), tea_key);
```

Where `tea_key` was calculated from `main_hash` earlier. Note that the size provided is the total size of the buffer to encrypt, not the number of `uint32_t` blocks that make it up. The C standard library goes back and forth with that kind of thing so much that it seems worth clarifying here.

After this it immediately undergoes password encryption, which is a bit convoluted so bare with me through this next section.

Password Encryption

The password encryption algorithm will be a lot easier to follow if we take a top-down approach.

It is invoked similar to the TEA encryption:

```
pass_encrypt((uint32_t*) tea, sizeof(lic_tea), password);
```

Where password was extracted from `main_hash`. Our `pass_encrypt()` routine looks like this:

```
void pass_encrypt(uint32_t* buf, size_t size, const char* key)
{
    size_t num_blocks = size / 4;

    // Initialize shift registers
    uint32_t shift_a = 0x41363233 ^ key[0];
    uint32_t shift_b = shift_a = codex(key, shift_a);

    shift_a ^= ((uint32_t) key[0]) << 8; //Better safe than sorry with the type promote
    shift_a  = codex(key, shift_a);

    // Encryption loop
    for (size_t i = 0; i < num_blocks; i++)
    {
        buf[i] ^= shift_a;

        shift_b  = rol32(shift_b, shift_a & 0xff);
        shift_a ^= shift_b;

        shift_a  = ror32(shift_a, (shift_b >> 8) & 0xff);
        shift_b += shift_a;
    }
}
```

Yeah, I know. Let's break this down.

`rol32(number, amount)` bitwise rotates a 32-bit `number` to the left by `amount` bits, exactly like the `rol` assembly instruction, but C++ doesn't have any native implementation of that so it has to be handled manually. Equivalently, `ror32()` does the exact opposite. You can look at the demo code for implementations of these, but I'm pretty sure I just copied them from some CS101 article as they're standard fare.

Our password encryption routine sets up some starting values for the shift registers, then just goes through the main loop encrypting 4-byte blocks and convoluting the registers after each block.

This just leaves the question as to what the hell codex does, which we'll look at now.

Our enigmatic codex routine:

```
uint32_t codex(const char* key, uint32_t shift)
{
    uint16_t counter = 0;

    do {
        // Again, this works without type promotion, but I don't trust it
        shift ^= ((uint32_t) *key) << 8;

        do {
            shift  = (shift ^ (counter & 0xFF)) + 0x7034616b;
            shift  = ror32(shift, shift & 0xFF);
            shift ^= 0x8372a5a7;
            counter--;
        } while (counter != 0);
    } while (*key++ != '\0');

    return shift;
}
```

It takes an ASCII key and a 32-bit `shift` value, then goes char by char through the `key`, xor'ing the second-least significant byte of `shift` with the current `key` char, running `shift` through an insane number of convolutions, then does it with the next `key` char and so on until it uses the entire key and terminating null.

If the middle loop doesn't look quite right, that's because it purposely goes through the first pass with `counter = 0`, subtracts to get 0xffff, then starts comparing it to 0, so the inner loop actually executes 65536 times for each `key` char that gets incorporated. I'm not entirely convinced this isn't reducible, but haven't analyzed it.

Now we get to move up.

License Head

This is the top layer of the license header and nothing particularly interesting happens here.

`tea` is of course the part we just built and encrypted twice.
`hash_3` is just the value we calculated from `main_hash`.

`str_checksum` is exactly the same as we calculated for `tea.str_checksum`.
(Be sure not to try to access that value from the `tea` structure to copy here now, as it's been encrypted into the ground at this point)

`checksum` is another binary checksum on everything up to its location like we did in the TEA layer:

```
size_t chksum_size = &head.checksum - &head;   // Buffer size up to checksum position
head.checksum = bin_checksum((uint8_t*) &head, chksum_size);
```

That completes the header, and now we get to tack on some registration strings.

Registration Strings

Immediately following the license header are tacked on several null-terminated strings representing the registration information `name`, `company`, `custom`, and `hwid`, each preceded by two 32-bit end markers.

We're not going to get into working with UTF-16 strings here. It's not particularly complicated, it's just not particularly interesting either, and I haven't really dug into verifying that they work the way I think they do yet. My apologies to international readers, but you only see your info in a message box the first time you use a license anyway. I may add an addendum with more on them sometime later.

The strings get appended directly after `license_head` as such (in this exact order):

```
0xffffffff, 0xffffffff, 'name\0'
0xffffffff, 0xffffffff, 'company\0'
0xffffffff, 0xffffffff, 'custom\0'
0xffffffff, 0xffffffff, 'hwid'
```

These need to be the same strings you used to calculate the `str_checksum` value in `license_tea` and `license_head` structures earlier.

Notice that `hwid` isn't null-terminated.

If you omit a string, then you also omit its null terminator, but not it's beginning markers. It must also be omitted from the `str_checksum` calculations earlier. So if you don't use `custom`, the string section will look like:

```
0xffffffff, 0xffffffff, 'name\0',
0xffffffff, 0xffffffff, 'company\0'
0xffffffff, 0xffffffff,
0xffffffff, 0xffffffff, 'hwid'
```

This is based entirely on how the SDK key generators seem to handle omissions and hasn't been fully tested yet. It doesn't hurt anything just to include values for all of them, so there's no particular reason not to just do that, but this is being mentioned for the sake of completeness.

I haven't really experimented to see which if any of these are really mandatory for a functional license. `custom` is definitely optional as WinLicense only uses it to embed information the protected program author might want to access. I suspect they all may be. This is about as interesting as the UTF-16 support, though.

Next comes the `license_tail`, whose structure we haven't actually looked at yet but it's fairly straightforward.

License Tail

Here is the tail structure that immediately follows the string section:

```
struct license_tail
{
    uint16_t double_null;       // Always 0x0000
    uint32_t beg_marker[2];     // Always 0xfffffffd, 0xfffffffd
    uint32_t magic[2];          // Always 0x83a9b0f1, 0x1C
    uint32_t random;            // Pick a number, any number

    uint32_t install_by;        // First license use must be before this date
    uint32_t net_instances;     // Total instances that may run on a network
    uint32_t creation_date;     // When the license was made
    uint32_t unknown;           // Probably used by SmartActivate licenses

    uint32_t end_marker[4];     // 0xfffffffd, 0xfffffffd, 0xfffffffe, 0xfffffffe
    uint32_t checksum;          // Final checksum on main body
};
```

We can fill in most of these except for `checksum`, which comes after a final encryption step for the main license body.

`double_null`, `beg_marker`, `magic`, and `end_marker` are hard-coded as indicated.

`double_null` and `beg_marker` are probably technically part of the strings section, but they are always going to be the same values and it works out cleaner just to deal with them here. If `hwid` string isn't used, the `double_null` will still always be present, which is different from how the terminators work for the other strings and why I've just separated it out.

`install_by` and `creation_date` use the same date format discussed earlier for `exp_date`.

If `install_by` is anything other than 0, the license must be used to register the protected program before that date. Unlike with `exp_date`, it can still be used past that date, but the first run with this license must occur before `install_by` if set.

`creation_date` is the license creation date. It's not really clear to me why this matters, there aren't supposed to be any protections built around it, but some protected programs will refuse the license if this isn't set to something sensible.

`net_instances` is the number of instances that may concurrently run on a network. One machine has to be setup as a server and the others coordinate with it. It seems like a huge mess. 0 = unlimited and avoids all of that.

`unknown` gets set to an uninitialized local variable which just so happens to be 0. This is probably the field to indicate if "Dynamic SmartActivate" is being used, and should be set to 0 for our purposes. It kind of looks like dumb luck that their key generator accidentally sets it to the correct value, but I might just be missing something.

There's one more simple encryption round, then the checksum, then we can move on to RSA.

String and Tail Encryption

This routine uses the first 32-bit value from the completed main license buffer we just built as a key to encrypt the buffer from the start of the strings section up to right before the last two end markers in the `license_tail`.

For this example, assume everything we have made up to this point is stored starting at `buf`:

```
// Macro to simplify accessing 32-bit values from the buffer
#define pdword(x) *((uint32_t*)(x))

void str_tail_encrypt(uint8_t* buf)
{
    // Uses the first dword of the license buffer as a key
    uint32_t shift = pdword(buf);

    // Start at the beginning of the strings section
    uint8_t *pos = buf + sizeof(license_head);

    // Encrypt up to last license_tail end markers
    while (!((pdword(pos) == 0xfffffffe) && (pdword(pos + 4) == 0xfffffffe)))
    {
        *pos ^= shift;
        *pos += (shift >> 8);
        shift = ror32(shift, 1);

        pos++;
    }
}
```

The final checksum routine gets its own page since it takes up quite a bit of room.

Final Checksum

Again assuming our license buffer begins at `buf`:

```
size_t final_checksum(uint8_t* buf)
{
    uint8_t sum[4] = { 0 };
    size_t state = 0;
    uint8_t *pos = buf;

    while (!((pdword(pos) == 0xffffffffe) && (pdword(pos + 4) == 0xffffffffe)))
    {
        size_t i = state % 4;

        switch (i)
        {
            case 0:
            case 1:
                sum[i] += *pos;
                break;
            case 2:
            case 3:
                sum[i] ^= *pos;
                break;
        }
        pos++;
        state++;
    }

    // pos = &tail.checksum;
    pos += 8;

    // Write checksum;
    pdword(pos) = (sum[1] << 24) | (sum[0] << 16) | (sum[3] << 8) | sum[2];

    // Return total buffer size (one past end of checksum) - (start of license)
    return (pos + 4) - buf;
}
```

This will calculate the correct place to write the checksum to `license_tail` and also return to us the total size of our main license body.

We are now ready to RSA sign and encrypt our license.

RSA Signing

`rsa_private_key1` is a DER-encoded 2048-bit RSAPrivateKey [4][5] that has been base36 encoded to yield the ASCII string provided when protecting a program.

The main license body we just created is hashed using SHA-1, which is encoded in a signature message per EMSA-PKCS1-v1_5 [6] with padding PS = 0xff

This is then RSA encrypted the way you would expect (eg. with `rsa_exptmod()` in libtomcrypt) using `rsa_private_key1`, and the encrypted signature message is appended directly to the end of the main license body.

RSA Encryption

`rsa_private_key2` is a 1024-bit RSAPrivateKey encoded as above.

The main license body and signature are RSA encrypted with `rsa_private_key2` in sequential blocks of size 0x7f. Since the key is 1024-bits, the result of each encryption will be 0x80 bytes long, so the license cannot be encrypted in place. The resulting 0x80-byte results are written sequentially into a new buffer to build the final license.

Any remaining bytes of the main license body and signature left over
(ie. the last `(main body size + sig size) mod 0x7f` bytes)
are just copied directly into the final license immediately following the last encryption result.

This final license is written out to regkey.dat or whichever license file the protected program will use.

There's another paper coming that looks at how to divine the relevant hashes and keys from protected programs and deal with the RSA. It's a bit complex so I still need to clean up the code for it and figure out how to explain it all in a sensible way, so this is it for now.

References

[1] https://www.oreans.com/WinLicense.php
[2] https://bbs.kanxue.com/thread-96931.htm
[3] https://github.com/charlesnathansmith/wl_regkey
[4] https://www.rfc-editor.org/rfc/rfc3447#page-44
[5] https://learn.microsoft.com/en-us/windows/win32/seccertenroll/about-der-transfer-syntax
[6] https://www.rfc-editor.org/rfc/rfc3447#page-41