WhatLicense
Extracting WinLicense main_hash using Intel PIN
https://github.com/charlesnathansmith/whatlicense

Intel PIN [1] is a powerful platform for debugging, reverse engineering, and otherwise analyzing and manipulating executables. It enables building custom tools in C++ that interface with a just-in-time execution environment, allowing fine-grained instruction-level analysis and control.

One of the major challenges that arises in working with it is that the dynamic analysis must all be performed within "analysis" callback functions set to trigger before or after instructions. Additionally, while PIN has many great built-in tools for analyzing instruction types, they are typically only available during "instrumentation" when callbacks are being inserted based on static analysis of instructions and the dynamic information available to each analysis callback must be decided.

This is all fine for simply logging instructions or other tasks where the same thing needs to happen every time the same instruction is executed. If we want to do something a bit more complicated, we will need to come up with ways to maintain state information between callbacks. More complex still is trying to do something interesting with heavily packed and obfuscated programs, where we don't know the address of specific instructions we're interested in ahead of time or even the exact form it will take.

This project looks at how we can go about implementing a fairly complex task with multiple steps in this callback-driven system, namely extracting the secret keys and checkvalues from a program protected with WinLicense without the benefit of the license file.

WinLicense [2] is an extension of the Themida virtualization engine that adds (incidentally enough) licensing support. The license file format is fairly complicated and is covered in full here [3], so that will be assumed knowledge moving forward. There are other license types that protected programs can utilize like "SmartLicense" and registry-based solutions, but we will be looking only at the default registration file (regkey.dat) verification system.

I am not trying to encourage piracy here. Our final result will still require a PIN-based launcher to hot-patch the public RSA keys in the program, so this isn't a way to produce easily-distributable cracked versions of protected programs. It will allow us to gather all of the hidden information from a protected program needed to build valid license files for it with the exception of the RSA steps. The same mechanisms we will be circumventing have been used to protect programs for over 10 years, and this may be the only remaining way to use some programs created by companies that are no longer around to buy proper licenses from. This was also just a really difficult problem to solve. A lot of lessons were learned here that extend well beyond just attacking this one particular form of DRM.

Carrying out this multi-step process while being able to rely only on information from callbacks was accomplished by creating a global state machine that can forward PIN's instruction analysis callbacks to relevant stages working on each task in turn. When active, each stage begins receiving three callbacks -- `process_ins`, which is called before every instruction executed, `process_read`, which is called before each memory read operation, and `process_write`, which is called after each write operation.

We'll try to examine how each stage uses these.
The full code for this project is available at [4].

Before we start diving in to analysis, we will need to generate a dummy license file for the protected program to process. This is a license file with the correct internal structure and checksums that a valid license file would have, but built using arbitrarily chosen keys and checkvalues that are not valid for the particular program we are analyzing.

We will follow the program as it loads and attempts to decrypt and verify various parts of the dummy license, patching in keys and bypassing checks as we go so that each step of processing can be reached, all the while collecting or solving for the real values of those keys and checkvalues. By this method we will be able to collect enough information to completely reconstruct a valid `main_hash` string which contains all of the secret information needed to build the main license body. The `main_hash` can be used to construct a license that is valid for the program in every way except for the RSA keys, which can be hot-patched with a much simpler and faster loader PIN tool than the one used for intially extracting the relevant secrets.

The `wl-lic` tool included in this project is used without arguments to generate our dummy license at "regkey.dat" and its accompanying RSA public keys at "regkey.rsa"

Once these are generated, we can launch the `wl-extract` tool via PIN to analyze our protected program.

The analysis performed by `wl-extract` is carried out in several stages which examine the following verification steps:

0 - Initial license file mapping and copying to a new buffer
1 - RSA signature verification and decryption
2 - Outer layer checkvalue verification (hash_3)
3 - Password decryption
4 - TEA decryption

If you've become familiar with the license construction, you'll probably notice some conspicuous absenses from this list. Namely, the `HWID_hash`, `hash_1`, and `hash_2` from the innermost core layer of the license.

This is because as far as I can tell from all the versions I've analyzed, none of those are ever actually verified. There's no way to actually retrieve the "correct" values from the binary if it never checks them against anything. They also are not verified in the final license file we generate, so it doesn't end up mattering.

Since protected programs are so heavily virtualized, with simple actions often taking thousands of instructions, and different versions of protected binaries using different convoluted execution paths to accomplish the same thing, we're going to have to talk about what's happening at each stage mostly in general terms of data flow and a handful of instructions that are actually consistent across version I have encountered. This means I'm going to have to try to explain what's going on at each stage in the verification process without being able to provide nice assembly printouts to follow along with, but I'll try to break down everything that's happening as best as possible.

Stage 0 -- Initial license file mapping and copying

[main.cpp]

After a protected program unpacks itself, it tries to open its license file using NtCreateFile, maps it to memory with MapViewOfFile, copies it byte-by-byte to another location while scanning for some values likely related to other license file types we're not interested in here, does a sanity check on the file size, and then unmaps the temporary memory with UnmapViewOfFile.

Our primary goal during analysis here is to find the final memory location the license file gets copied to by the end of all of that, then hand that over to the next stage.

This first analysis stage is handled separately from those managed by the state manager, primarily because we don't want or need callbacks running on every instruction during the initial unpacking of the executable, and we need to hook library calls during this stage which we don't need to do anytime later. I've been calling it Stage 0, but it's really more a part of the initial setup with Stage 1 being the first proper stage after this.

We hook NtCreateFile so that whenever it tries to open "regkey.dat" at whichever default location its expecting to find it, we can redirect it to the one we just created with wl-lic. This isn't strictly necessary, as long as you copy the file we created to wherever it expects to find it, it's just a lot more convenient to redirect it.

We hook MapViewOfFile and compare the beginning of the mapped data to the beginning of our license file, so we can flag when we've opened the license file.

We start monitoring one-byte writes and log the destination address of any that are the same as the first byte of our license file. Only a handful occur between the file being mapped and unmapped, and one of them will be the start of our license data copy destination.

We also have a hook on UnmapViewOfFile so we can stop logging writes after our file is unmapped and check the ones we recorded to see which one is the start of our license data copy buffer. The hooks and write callbacks need to be set up ahead of time, but this is the order in which they'll work once the correct area is reached.

Once we have the address of the copied license file data, we can hand it off to the stage manager, which will drop us down into Stage 1.

Stage 1 - RSA signature verification and decryption

[stage1.cpp]

After copying the license file data, the protected program will RSA-decrypt it to a new buffer in 0x80-byte (1024-bit) chunks, then verify an RSA signature using a separate 2048-bit key. The public keys are embedded deep in the program somewhere, and trying to change them in the executable is a challenge for another day. We're going to log them and memory-swap in the ones used to build our dummy license file so analysis can proceed.

When a license file is being built, it's split into 0x7f-byte chunks that are individually encrypted to produce 0x80-byte chunks (since it's a 1024-bit key) that are concatenated together to form the final file, so presumably it undoes that during the decryption process. It doesn't matter for what we're doing right now, but it just struck me as odd. I don't know if that's typical during RSA encryption to avoid overflows or something, or just some unusual thing they do. It seemed worth mentioning.

WinLicense uses libtomcrypt for RSA signature verification and decryption. We can find some metadata referencing source files from the library in their web application keygen available with the demo, and luckily for us, this part of the verification process isn't virtualized for whatever reason.

The library version or compiler settings used vary between protected programs to the extent that we can't simply calculate offsets to functions once we find where RSA decryption begins, but we can reliably use some commonalities between versions to trace through to find `mp_exptmod`, which we need to hook in order to patch the hardcoded public keys with the ones used to build our dummy license.

The structure of rsa_exptmod hasn't really changed over the years, and you can see the source for it here [5].

Sub-stage 1

We find the RSA code by monitoring for reads from our license buffer we found at the end of Stage 0. The first time it's read happens inside of the call to `mp_read_unsigned_bin` made at the beginning of `rsa_exptmod`. When the read occurs, we log the return address.

Sub-stage 2

We watch for execution to reach the return address we recorded, which puts us back in `rsa_exptmod` with a known memory layout (I've tested programs protected with WL versions put out 10 years apart and the layout here seems to be consistent across versions.)

The decrypted destination address is stored at [ESP + 0x8c] (we should probably reference it off of EBP since the address was an argument to `rsa_exptmod` but I'll have to go back in and see what to update it to.)

We also need to know the address EBP + 0x1c, because we can count read accesses to it to find the call to `mp_exptmod` reliably across versions.

Sub-stage 3

Counts the read accesses just mentioned

Sub-stage 4

Finally the call to `mp_exptmod` and logs its address

Sub-stage 5

This acts as a hook for `mp_exptmod`. We watch for execution to reach it and just patch in the keys needed to decrypt and verify the dummy license file we built when it does.

It will be called license size / 0x80 times with the decryption key and then once with the signature verification key.

At this point, RSA should be fully bypassed and we can move onto Stage 2, which we will pass the decrypted destination buffer found during this stage. Everything will happen in-place from here on out so our lives will get marginally simpler, but only just.

Stage 2 - Check value verification

[stage2.cpp]

After RSA decryption, a checksum on the main license body is verified, which will pass just fine because we built our dummy license to be internally consistent. A check value stored within the decrypted license buffer (a 2-byte word at position 0x33) which we are calling `hash_3` here then gets compared to a hardcoded value before verification can proceed.

The exact comparison instruction varies between versions, but will always be a word pointer comparison to some value loaded in a general register other than EBP or ESP. Since nothing else accesses the word in our license buffer before this, it's fairly straightforward to track down in a generic way, log the value it was expecting to find, and modify the flags after the comparison to tell it the comparison passed.

Sub-stage 1

Finds the first instruction that reads the `hash_3` value from our license buffer. This generally won't be the actual comparison instruction, as the virtualization engine likes to read values then push and pop them off the stack repeatedly and pass them around world before they finally get to the instruction that's actually using them, but this lets us skip over examining every single instruction up to this point, which saves some overhead and helps eliminate false positives.

Sub-stage 2

Word pointer comparisons are uncommon enough in the verification engine that the first one involving the value read from our license buffer after that read occurs is almost certainly the one we care about.

There's some logic in place to try to avoid false positives from any spurious instruction we might somehow run into here, but it doesn't seem necessary.

When the comparison is found, the EFLAGS are modified to force it to pass and we log the correct value the program was looking for. Then we can move onto Stage 3.

Stage 3 -- Password decryption

[stage3.cpp]

The verification procedure next performs a custom (as far as I can tell -- I don't recognize it as anything standard) password decryption on a section of the decrypted license buffer using a password that's hardcoded into the protected binary.

This password encryption/decryption algorithm has been fully documented in [3]

Sub-stage 1

Locates the approximate beginning of the password decryption routine, as indicated by part of the constant 0x41363233 being read in from memory somewhere (it's a constant used in the decryption algorithm.

As mentioned before, values get passed all around by the virtualization engine, so this won't be the exact beginning but we just need to narrow down our search space.

Sub-stage 2

Starts checking values being read for anything that could be a pointer to a valid password string (32 alphanumeric bytes) and logs it. In `wl-extract` this just saves the first string that meets this criteria since false positives are really unlikely here, but if it's failing verification, you can just save every potentially valid password found and verify each of them in the next step.

Sub-stage 3

Watches for the last password-decrypted dword to get written (we're decrypting in-place so we know exactly where it's going) and verifies that we're able to decrypt the first dword of our RSA-decrypted license buffer using the password we found and get the same answer as the verification engine.

If you're really getting stuck recovering the password here for some reason, you don't actually need it. You can grab the first 3 or 4 dwords from the buffer before and after password decryption, and calculate the shift registers needed for encryption using the known plain-text attack outlined here [6]. It's just a lot neater to have the actual password and usually trivial to obtain the way we just did.

That does it for password decryption. Since we built our dummy license with a random password, the data decrypted to complete garbage. There aren't any integrity checks before going straight into TEA decryption though, so we don't need copy in what it should've decrypted to or anything before proceeding. The next stage doesn't care if it's working on garbage.

Stage 4 - TEA Decryption

[stage4.cpp]

Ok, it isn't really TEA decryption. It's some sort of in-house variant that swaps XOR and ADD operations around for whatever reason, but it looks an awful lot like TEA, so that seems like the best way to refer to it. As with the rest of these steps, a full breakdown of the algorithm used is avalaible within [3] and a full breakdown of the process used to solve for the keys here is given at [7].

One weird note is that somehow it seems to know if you built your license with the correct TEA key even before it decrypts anything. If you used the correct key, then it tries to decrypt it using 32 rounds like it's supposed to, but if you didn't it tries to decrypt it using 12 rounds. We can still get the data we need to solve the key, we just have to be aware that it's based on 12 decryption rounds for some reason and adjust our `sum` values accordingly.

Just know that if you try to run `wl-extract` in extract mode using a regkey.dat file that is already built with the correct `main_hash` for your program, it's gonna give an error during TEA key solving because it's assuming the wrong `sum` values.

We can't directly extract the keys

A lot more effort seems to have been put into obfuscating this step and obscuring the keys. Operations involving the keys are performed in a masked way that can give mathematically equivalent results without ever storing the keys or having them present in memory at any point the same way the password was, so we can't just directly extract them.

The only values we can reliably find are certain intermediate results from the decryption rounds, which can be used to calculate the keys.

The solution example in [7] focused only on solving one half-round given a set of known intermediate values, but in reality two different half-rounds involving different key pairs are being alternatively calculated during decryption, with results of one being fed into the other in turn.

This doesn't make solving each half-round using the intermediates collected for it more complicated. Geting those values requires thoroughly understanding what we need and what we have access to.

What's happening during the decryption routine works out to something like this:

```
void tea_decrypt(uint32_t* buf, uint32_t* key, uint32_t sum)
{
    uint32_t a = buf[0], b = buf[1];

    do
    {
        b -= ((a << 4) + (key[2] ^ a) + ((a >> 5) ^ sum) + key[3]);
        a -= ((b << 4) + (key[0] ^ b) + ((b >> 5) ^ sum) + key[1]);
        sum += 0x61c88647;
    } while (sum);

    buf[0] = a, buf[1] = b;
}
```

We would like to find the XOR and ADD instructions directly involving the keys, but because of the way these operations are masked, this proves to be quite difficult if not impossible.

If we call each pass through the do / while loop a “round,” then each of the two large equations inside is a “half-round.” They are the exact same equation, the `a` and `b` values are just swapped. Looking at the first equation, if we can collect the values of `a`, the initial `b` input, `sum`, and the answer that gets subtracted from `b` from enough different passes, then we can solve for key[2] and key[3].

The only instruction providing valuable data that we can reliably catch is “shr [reg], 5” where [reg] points to the `a` value about to be used in the first equation if that’s next, or the `b` value going into the second equation if that’s the next to get calculated. (We can catch “shr [reg], 4” too but it gives the same information)

It turns out this is enough to find the rest

Let’s step through a few rounds.

For the first equation of the first round, `a` and `b` are just the first two dwords of our encrypted buffer, and `sum` is -0x61c88647 * 12 since we know it’s going to do 12 rounds of decryption. We know everything except the solution that gets calculated.

In comes “shr [reg], 5”. This is the final `b` after the subtraction from the first equation right-shifted by 5. Whatever value is being shifted here is the starting value of `b` minus the solution, so we can easily calculate the solution to the first equation and have all the data we need for it.

The value from this “shr [reg], 5” is also the input `b` value for the second equation, the input `a` is the same as for the first equation because the first one didn’t alter it, and `sum` is the same as for the first because we are still in the same round. We again have every variable except for the final solution to the second equation, so when we reach the next “shr [reg], 5” going into the beginning of the next ound, we can repeat this process, and keep repeating it until we have a full set of data for both equations that we know how to solve.

If your eyes are glossing over, that’s the correct response. It’s convoluted, but it means we can fully solve the TEA key given the really limited data we can reliably access.

Stage 5 -- Reconstructing a valid main_hash string

[stage5.cpp]

At this point, we have all of the information we need to construct the main body (ie. before RSA encryption) of a valid license. As stated in the beginning, there are two more secret check values and a HWID hash that involves a secret value that ought to be verified after the TEA decryption -- but they just aren’t for whatever reason. We can just use random values for them when constructing a new license. If you do happen to find a protected program that seems to care about these, let me know, as I’m curious as to what exactly happened here for every version I’ve tested to seemingly skip the last few obvious verification steps.

Since there is no way to deduce RSA private keys from public keys, we will still always need to hot patch the public keys at startup, but we are now capable of building a license that will otherwise pass

all validation checks after that point. The obvious next area of research would be working out how to patch the hardcoded public keys in the executable to avoid this step, though that will require quite a bit of work understanding the unpacking and self-integrity routines.

We could write a license file generator that just uses the values we've collected as inputs, but there's a cleaner way to represent it. When protecting a program, WinLicense stores all of these in an alphanumeric string called `main_hash` and it would be interesting to reconstruct a valid one for our protected program.

Most secret values are calculated from the alphanumeric hash by treating specific substrings from it as little-endian numbers and adding them together. For example, if a 32-bit key value is being calculated from a substring "ABCD" and a substring "1234", the byte-level representations of these are hex 41 42 43 44 and 31 32 33 34, respectively. Interpeted as little-endian numbers, these are 0x44434241 and 0x34333231, which would then be added together to arrive at the key value they encode: 0x78767472.

If we want to generate a valid `main_hash`, we have to do this in reverse and generate pairs of strings that can be summed in this manner to get the desired key values. This actually turns out to be fairly straightforward, since the sum of the two largest alphanumeric characters ('z' + 'z' = 0x7a + 0x7a) is 0xf4, which means we don't have to worry about carries and can solve each byte individually, and can do so with a simple lookup table. (This also means that key bytes can only range from 0x60 = '0' + '0' to 0x7a, though this information isn't particularly helpful other than maybe as a sanity check on the values we discover.)

The exception is the password, which is just copied directly into the middle of the `main_hash` string.

The hash we generate will certainly not be the one originally used to protect the program, since there are multiple character pairs that can sum to the same value and part of the `main_hash` is actually random garbage that doesn't get used when building a license file, but we can generate hash strings that will produce equivalent keys and can even be used with the license generators supplied by WinLicense.

Conclusion

This was a really long project, but a rewarding one. The virtualization was definitely painful to wade through. The license file format is really haphazard, which always seemed like a weird contrast, but it may be on purpose because it forces you to solve one problem after another to break the whole thing.

Neither I nor this project are endorsed or affiliated with Oreans or Intel in anyway.

References

[1] https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html
[2] https://www.oreans.com/WinLicense.php
[3] https://github.com/charlesnathansmith/wl_regkey/blob/main/regkey_format.pdf
[4] https://github.com/charlesnathansmith/whatlicense
[5] https://github.com/libtom/libtomcrypt/blob/develop/src/pk/rsa/rsa_exptmod.c
[6] https://github.com/charlesnathansmith/pwcrack
[7] https://github.com/charlesnathansmith/teasolver