Flare-On 7: Challenge 11 – Rabbit Hole

Challenge Author: Sandor Nemes (@sandornemes)

“But I don’t want to go among mad people," Alice remarked.
"Oh, you can’t help that," said the Cat: "we’re all mad here. I’m mad. You’re mad."
"How do you know I’m mad?" said Alice.
"You must be," said the Cat, "or you wouldn’t have come here.”
— Lewis Carroll, Alice in Wonderland

Overview

In the past years, FLARE-On always had at least one challenge that was written in object-oriented C++ and was a nightmare to reverse engineer. It would often involve either a system of linear equations, or a Turing tarpit[1] that usually manifested itself as some sort of virtual machine using an esoteric programming language (some challenge authors even took that to an extreme level by having a virtual machine inside another virtual machine). While these are nice for the first few times, I assumed people are now a bit tired of these, so my design goal for this year was to implement a relatively complicated challenge, that does not have any of these aforementioned features. The challenge was named “Rabbit Hole” because this is a “needle in a haystack” type of challenge, as there could be many different approaches, some being dead ends, and eventually you probably won’t be able (and you are not expected to) to uncover every single detail in the code, but that is okay, and that also happens to be one of the main tenets of malware reverse engineering: "Always focus on the big picture, and do not get lost in the tiny details". The challenge is x64-based, for the simple reason that most Windows operating system installations today are 64 bits, but on most CTF games x64 code challenges are usually painfully underrepresented, and not reflecting the real-life prevalence of x64 code.

HOW THIS CHALLENGE WAS MADE

My team specializes in malware configuration extraction and network traffic emulation, so it was a natural choice to leverage this knowledge and do it the other way around this time: change a malware’s configuration and write tools that make it possible put the updated configuration back into the malware sample, and optionally add some new plugins too. Thus, I took a Gozi V3 (aka. RM3) malware sample, ran it in a VM with a live internet connection waited until it downloaded all the necessary modules for its normal operation. Then I meticulously reverse engineered the code and changed the configuration to practically defang the malware sample and turn it into a harmless executable. The module responsible for the network

[1] https://en.wikipedia.org/wiki/Turing_tarpit

communication was patched to make sure it saves the most recent data exfiltration request to the registry before the actual network communication takes place. This was needed to actually make the challenge solvable (as I had to make sure that every information that was necessary for the solution was in the registry hive file).

Now if you think this challenge was complicated, then remember that most of the things in there came from the actual, in-the-wild malware, and were already part of the code, and not something just added to hinder your progress. The only parts that were altered are the following (and I obviously do not have, and never had access to the actual malware source code for this family, so all these were performed via reverse engineering and binary patching):

1. The configuration was modified in all the modules (e.g. C2 servers, RSA public key, Serpent key, webinjects, etc.), to make sure that the sample can't call home over the network or exfiltrate any data from the computer it is running on.
2. The network module was binary patched to save the most recently exfiltrated data packet to the registry each time.
3. A very simple custom plugin was added that tries to prevent running some common debuggers and analysis tools (WinDbg, OllyDbg, x64dbg, IDA, Process Monitor, Process Explorer, Autoruns) while the malware is active and running. This custom plugin merely displays a message, and then terminates the debugger/analysis tool, so it should be fairly trivial to circumvent.

ANALYZING THE REGISTRY HIVE

This challenge consists of a single file with the name *"NTUSER.DAT"*. The file is not directly executable, but those having some deeper knowledge of Windows internals will probably recognize by its name that this file is a user registry hive, that usually resides in the *%USERPROFILE%* directory (usually *C:\Users\<username>* on Windows 10). Otherwise you can use standard tools (e.g. the Linux "file" utility) to find out the file type, or just Google the first 4 bytes of the file (that is *"regf"*):

```
$ file NTUSER.DAT
NTUSER.DAT: MS Windows registry file, NT/2000 or above
```

Figure 1 - Using the "file" utility to determine the file type

There are several free tools that you can use to open the registry hive and examine its contents. These are the ones that I have personally tested:

- NirSoft RegFileExport[2]
- Eric Zimmerman's Registry Explorer[3]

[2] https://www.nirsoft.net/utils/registry_file_offline_export.html
[3] https://ericzimmerman.github.io/

- The built-in Windows reg.exe utility

FINDING THE PERSISTENCE METHOD

Simply trying to replace the NTUSER.DAT file on a Windows installation, or trying to restore the hive using the reg.exe utility will unlikely to work out of the box. The registry hive files store permission information which will be tied to the security identifier (SID) value of the particular user for which this registry hive was created for. Your best option here is to convert the registry hive to a .reg file using the NirSoft RegFileExport tool (which will not preserve permission information), then you can import that .reg file (although you will get a warning dialog that some keys could not be imported as they are in use). Now you can use standard tools, like Sysinternals Autoruns[4] to examine the executables that are trying to automatically start in one way or another.

Note: to make this an even more challenging exercise, the persistence method was intentionally set up in a way that it is not displayed using the Autoruns default settings. You will need to uncheck the "Hide Windows Entries" menu option under "Options".

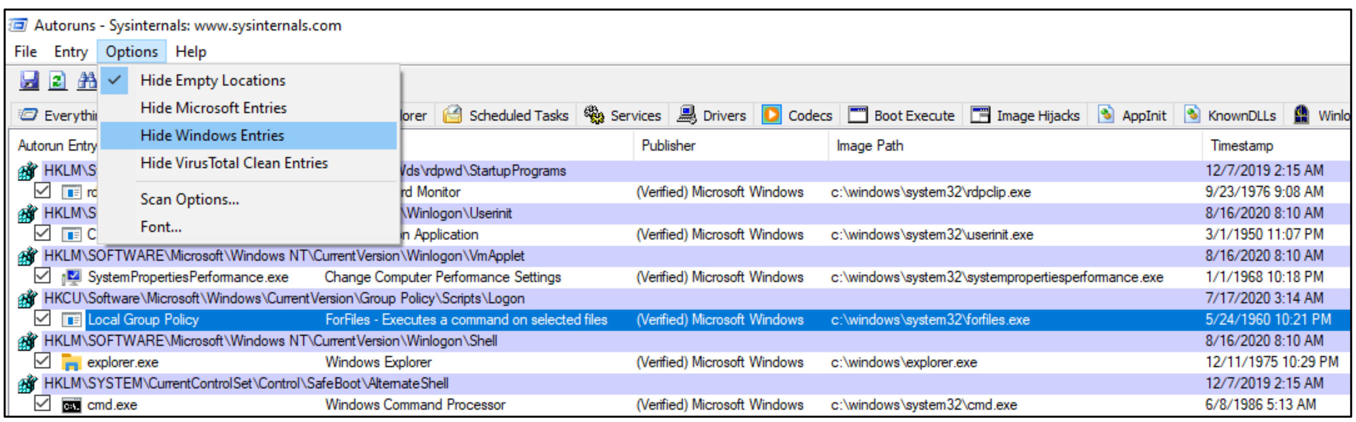

Figure 2 - Unchecking "Hide Windows Entries" will reveal the local group policy logon script

There is a logon script set up in the local group policy[5], that runs the command below:

```
C:\Windows\System32\forfiles.exe /p C:\WINDOWS\system32 /s /c "cmd /c @file -ec
aQBlAHgAIAAoAGcAcAAgACcASABLAEMAVQA6AFwAUwBPAEYAVABXAEEAUgBFAFwAVABpAG0AZQByAHAAcgBvACcAKQAuAEQA"
/m p*ll.*e
```

Figure 3 - The command used in the login script

This script will enumerate files in the *C:\WINDOWS\system32* directory matching the pattern "*p*ll.*e*", then invoke that file using the specified command line arguments. The only file that should match that pattern is "powershell.exe", so basically this is just a fancy and less obvious way to invoke a PowerShell command.

---

[4] https://docs.microsoft.com/en-us/sysinternals/downloads/autoruns

[5] This is under the SOFTWARE\Microsoft\Windows\CurrentVersion\Group Policy\Scripts\Logon\0\0 registry key

The "-ec" part is an abbreviation for the "-EncodedCommand" parameter, and you can use this Python code snippet to decode it:

```
>>> import base64

>>>
print(base64.b64decode('aQBlAHgAIAAoAGcAcAAgACcASABLAEMAVQA6AFwAUwBPAEYAVABXAEEAUgBFAFwAVABpAG0AZ
QByAHAAcgBvACcAKQAuAEQA').decode('utf-16le'))

iex (gp 'HKCU:\SOFTWARE\Timerpro').D

>>>
```

Figure 4 - Decrypting the encoded PowerShell command

(**Note:** "iex" stands for "Invoke-Expression", and "gp" is an alias for "Get-ItemProperty")

This new piece of information should direct your attention to the *HKCU\SOFTWARE\Timerpro* registry key, which is the main key that stores the malware's components.

THE 1ST STAGE LOADER

You can import the content of the malware's registry key into your own HKEY_CURRENT_UESR hive using the commands below:

```
C:\Users\User\Desktop> reg load HKU\Test NTUSER.DAT
The operation completed successfully.

C:\Users\User\Desktop> reg save HKU\Test\SOFTWARE\Timerpro Timerpro.hiv
The operation completed successfully.

C:\Users\User\Desktop> reg add HKCU\SOFTWARE\Timerpro
The operation completed successfully.

C:\Users\User\Desktop> reg restore HKCU\SOFTWARE\Timerpro Timerpro.hiv
The operation completed successfully.

C:\Users\User\Desktop> reg unload HKU\Test
The operation completed successfully.
```

Figure 5 - Importing the malware's registry key to the local user hive

Note: If you have already imported the registry hive by converting it to a .reg file (as suggested in the previous section), you will still need to perform this step, because Windows won't be able to import the registry value that holds this PowerShell script.

The PowerShell script that is in the "D" registry value under the *HKCU\SOFTWARE\Timerpro* looks something like this (the full Base64 encoded parts were omitted for brevity):

```
$jjw="kcsukccudy";



function hjmk{[System.Convert]::FromBase64String($args[0]);};
```

```
[byte[]]$rpl=hjmk("6feZAAA(...)AAAAAAAA");


function
geapmkxsiw{$kjurpkot=hjmk($args[0]);[System.Text.Encoding]::ASCII.GetString($kjurpkot);};iex(geap
mkxsiw("DQokY3Fs(...)cnU7DQo="));iex(geapmkxsiw("DQoNCiRq(...)Cn0NCn=="));
```

Figure 6 - Decrypted shellcode loader PowerShell script

This script performs a simple self-injection using the *QueueUserAPC*[6] API call to invoke a new thread using the Base64 encoded loader shellcode at the beginning of the script. The easiest way to debug this is to use the age-old trick of replacing the first opcode of the payload with a self-jump (JMP $-5 aka. EB FE). Just right-click on the "D" value in the Registry Editor and select Modify Binary Data… to replace the first occurrence of "6feZ" with "6/6Z" (which is the Base64 encoded version of the JMP $-5 opcode).

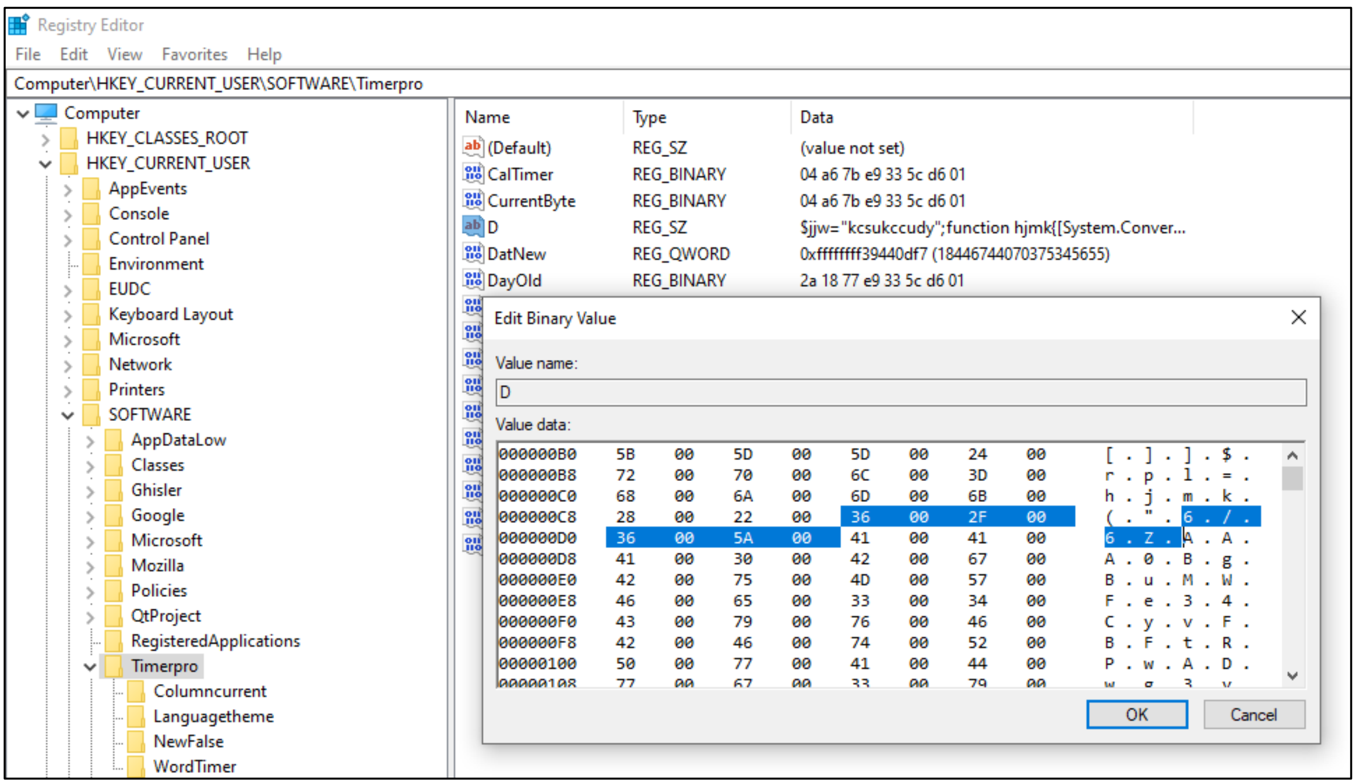

Figure 7 - Patching the first instruction of the shellcode with a self-jump

It's also important to remember the old value, because you will have to patch back the original value in the debugger once you get there.

```
>>> import base64
```

[6] https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-queueuserapc

```
>>> base64.b64decode('6/6Z').hex()
'ebfe99'
>>> base64.b64decode('6feZ').hex()
'e9f799'
>>>
```

Figure 8 - Python code snippet showing the Base64 decoded bytes

Now all you need to do is to run *powershell -Command "iex (gp 'HKCU:\\SOFTWARE\\Timerpro').D"*, and attach a debugger of your choice to the process, then restore the original starting bytes (i.e. E9 F7).

Note: you might also need to switch to the correct thread first, usually the one that has spent the longest time in user mode (this is the "User Time" column on the "Threads" tab in x64dbg).

THE 2ND STAGE LOADER

This shellcode is a simple PE loader, that processes imports and relocations, then finally jumps to the entry point. The easiest way to get past this is to set a breakpoint on the CALL R10 instruction that comes a bit above the final RET instruction.

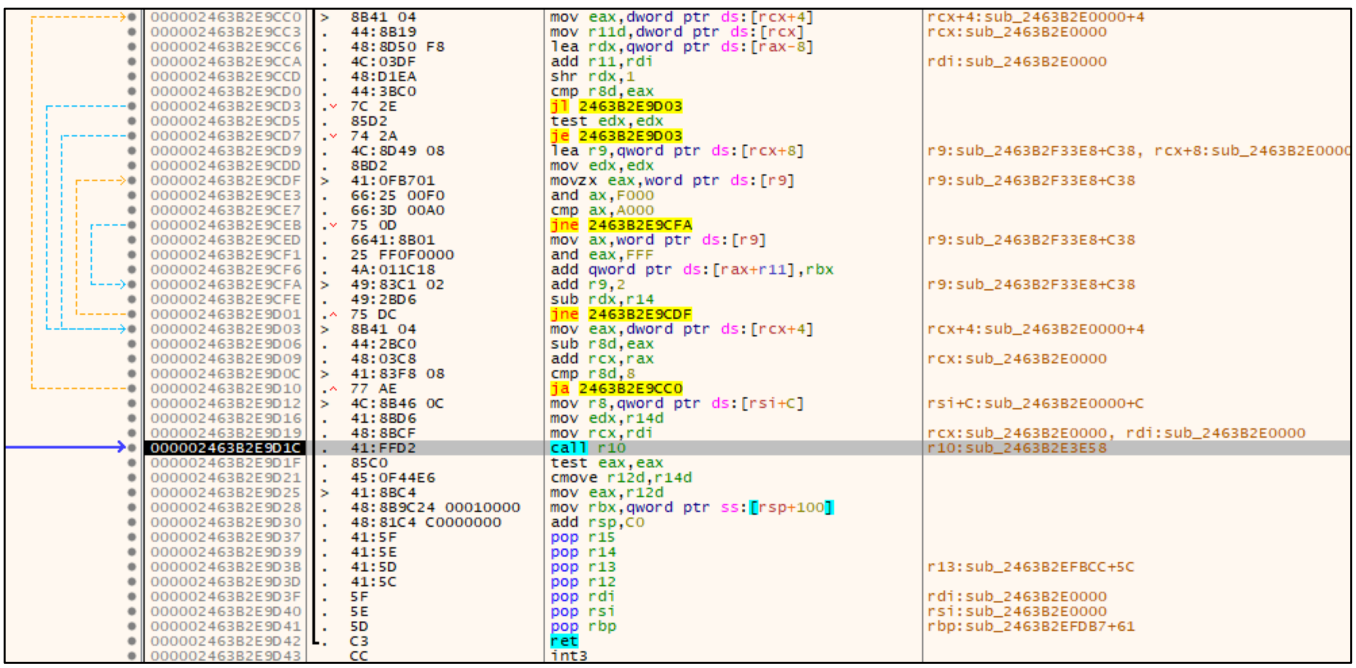

Figure 9 - PE loader shellcode jumps to the entry point

The executable then finds and decrypts the "*.bss"* PE section, initializes some global variables including a *machine ID* (take a mental note of this, because this will be important later). The *machine ID* is generated in the function at *0x18000c928* (assuming that the base address is *0x180000000*) from the machine SID, which is in turn determined by querying the process token, then getting the user SID from the token. The code then loads two embedded data blobs using the function at *0x18000b354* (let's call this function *get_joined_file*):

- an encrypted RSA public key (for decrypting/verifying configuration data in the registry),
- and a wordlist (for generating random names).

After this step a few GUIDs are generated, and a mutex is created, then finally the loader loads its main module from the registry.

Note: If you happen to recognize that this is a Gozi sample at some point during the analysis, you can leverage the leaked Gozi source code[7,8] to help getting a better understanding of what is going on in the code, however please note that Gozi V3 is considerably different in many aspects, so don't expect to find a very large amount of code overlap.

One of the biggest challenges you will face at this stage, is that all the registry key and value names are (pseudo-)randomly generated from the words of the wordlist using the *machine ID* and another seed value. Because the *machine ID* is generated based on the machine SID of the computer this was run on, it will be different on each computer. Thus, the code will very likely not find the required registry keys and will fail to proceed. In order to solve that you will either need to change your machine SID (which is probably hard) or patch the function that generates the *machine ID*, and pretend that you are running on a computer with a different machine SID that matches the machine SID of the computer the registry hive was generated on (which sounds complicated, but probably way easier than changing the machine SID in Windows). Let's see how you can possibly find that machine SID using the data that you have at your hands...

FINDING THE ORIGINAL MACHINE SID

A security Identifier (commonly abbreviated SID) is a unique, immutable identifier of a user, user group, or other security principal. For well-known SIDs this has the structure[9] below (by using the SID *S-1-5-21-1111-2222-3333-513* as an example):

- *S-1*: Indicates a revision or version 1 SID.
- *5*: SECURITY_NT_AUTHORITY, indicates it's a Windows specific SID.
- *21*: SECURITY_NT_NON_UNIQUE, indicates a Domain/Machine ID will follow.
- *1111-2222-3333*: The next three values contain 32-bit random numbers to uniquely identify the domain/machine
- *513*: RID or Relative ID, indicates a unique object ID within the domain/machine.

This has an important implication, that the machine SID (*S-1-5-21-1111-2222-3333* in the example) will be the part of every local user/group SID generated on the local machine. So simply by searching for the value "*S-1-5-21-*" in the registry data, you will get a list of SIDs, and stripping away the RID part (the last value) will result in the machine SID for the computer.

---

[7] https://github.com/gbrindisi/malware/tree/master/windows/gozi-isfb
[8] https://github.com/t3rabyt3/Gozi
[9] https://en.wikipedia.org/wiki/Security_Identifier#Machine_SIDs

```
C:\Users\User\Desktop>RegFileExport.exe NTUSER.DAT ntuser-reg.txt


C:\Users\User\Desktop>find "S-1-5-21-" ntuser-reg.txt


---------- NTUSER-REG.TXT

"Group0"="S-1-5-21-3823548243-3100178540-2044283163-513"

@="defaultroot://{S-1-5-21-3823548243-3100178540-2044283163-1006}/"

@="winrt://{S-1-5-21-3823548243-3100178540-2044283163-1006}/"

"SavePath"="C:\\Users\\Kevin\\Searches\\winrt--{S-1-5-21-3823548243-3100178540-2044283163-1006}-
.searchconnector-ms"

@="csc://{S-1-5-21-3823548243-3100178540-2044283163-1006}/"
```

Figure 10 - Finding the machine SID in the user registry hive

RANDOM STRING GENERATION

The registry key and value names are generated using a special function identified by *Ordinal #60* (in the *8576b0d0.dll/bl.dll* module at *0x18000d21c*) that generates random words using a wordlist and a seed value, and it also takes a second parameter that specifies the capitalization of the initial letters in each of the sub-words. This function uses the xorshift64* PRNG algorithm [10] for generating pseudorandom numbers. Here's a Python implementation of the string generation algorithm:

```
#!/usr/bin/env python3

XOR_KEY = 0xedb88320
MACHINE_SID = 'S-1-5-21-3823548243-3100178540-2044283163'


class XorShift64s:

    def __init__(self, seed):
        self.seed = seed

    def generate(self):
        x = self.seed
        x ^= (x >> 12) & 0xffffffffffffffff
        x ^= (x << 25) & 0xffffffffffffffff
        x ^= (x >> 27) & 0xffffffffffffffff
        self.seed = x
        x = (x * 0x2545f4914f6cdd1d) & 0xffffffffffffffff
        return x


class StringGenerator:

    def __init__(self, machine_id, wordlist):
        self.machine_id = machine_id
```

[10] https://en.wikipedia.org/wiki/Xorshift#xorshift*

```python
        self.wordlist = wordlist

    def generate(self, key, caps=0):
        results = []

        r11 = 0
        while key:
            x = XorShift64s(self.machine_id + ((key + r11) & 0xff)).generate()

            word = self.wordlist[(x & 0xffff) % len(self.wordlist)]

            rcx = x >> 0x20
            if rcx & 1:
                rdx = (rcx & 0xffff) % (len(word) - 1)
                rdx += 2
                word = word[:rdx]

            results.append(word)
            r11 += 2
            key >>= 8

        pos = 0
        while pos < len(results) and caps > 0:
            if caps & 1:
                results[pos] = results[pos].capitalize()
            pos += 1
            caps >>= 1

        return ''.join(results)


def get_machine_id(machine_sid, xor_key):
    machine_id = list(map(int, machine_sid.split('-')[4:7]))
    machine_id = sum(machine_id) + (machine_id[1] << 32)
    machine_id ^= ((xor_key << 32) | xor_key)
    return machine_id


def main():
    machine_id = get_machine_id(MACHINE_SID, XOR_KEY)
    wordlist = (
        'old', 'new', 'current', 'version', 'process', 'thread', 'id',
        'identity', 'task', 'disk', 'keyboard', 'monitor', 'class', 'archive',
        'drive', 'message', 'link', 'template', 'logic', 'protocol', 'console',
        'magic', 'system', 'software', 'word', 'byte', 'timer', 'window',
        'scale', 'info', 'char', 'calc', 'map', 'print', 'list', 'section',
        'name', 'lib', 'access', 'code', 'guid', 'build', 'warning', 'save',
        'load', 'region', 'column', 'row', 'language', 'date', 'day', 'false',
        'true', 'screen', 'net', 'info', 'web', 'server', 'client', 'search',
        'storage', 'icon', 'desktop', 'mode', 'project', 'media', 'spell',
        'work', 'security', 'explorer', 'cache', 'theme', 'solution'
    )

    strgen = StringGenerator(machine_id, wordlist)
    assert strgen.generate(0x9eff4536, 5) == 'ThemespellDaytheme'

    for i in range(1, 32):
        key = (i << 8) | i
        print(format(key, '04x'), strgen.generate(key, 3))


if __name__ == '__main__':
    main()
```

Figure 11 - Reconstruction of the random string generation algorithm in Python

You can find some of the seed values and strings used by the executable (and its plugins) in Appendix I - List of pseudorandomly generated words used in the code.

CONFIGURATION STORAGE AND PLUGINS

You already know at this point that everything is stored in the registry under the "Timerpro" key, this includes various settings, webinjects, and all the plugins used by the challenge executable. A full list of the various registry keys used can be found in Appendix II - List of registry keys/values and their brief description. In order to decrypt the data from the registry, you need to find the function that is responsible for this operation. The function identified by *Ordinal #26* (in *8576b0d0.dll/bl.dll* at address *0x18000d828*) is the one that first decrypts the RSA public key using a hardcoded Serpent key (which is "*90982d21090ef347*"), then performs an RSA **encryption** (as you can only do encryption with the public key) of the last 128 bytes of the data, which will have a Serpent session key to decrypt the rest of the data. Finally, you need to decompress the aPLib compressed blob to get the data.

```
#!/usr/bin/env python3
import hashlib
import json
import struct

import aplib  # from https://github.com/snemes/aplib/blob/master/aplib.py
from malduck import serpent  # sudo -H pip3 install malduck
from Crypto.PublicKey import RSA  # sudo -H pip3 install pycrypto


ENCRYPTED_PUBLIC_KEY = bytes.fromhex(('''
    36-3C-CD-0C-BC-D0-25-A3-D7-8A-5E-A4-38-58-C1-6E-
    05-18-65-AE-EC-99-0C-70-01-E7-F2-14-94-AC-13-60-
    94-FA-A2-CC-F4-6A-DB-B1-7D-1E-EA-13-63-32-50-2D-
    25-00-16-BC-10-D4-50-E0-32-7E-C0-72-25-F9-1E-E3-
    87-40-CB-E8-7D-F8-39-E1-66-07-76-EE-EC-10-9C-90-
    7A-40-B1-4D-A2-E7-A7-34-97-03-8C-FD-B3-8E-3E-BB-
    68-0C-00-D9-56-D0-D5-DD-48-25-E1-2F-D9-57-7D-83-
    D7-FA-C0-9F-79-0E-AB-2C-4B-3F-17-7A-83-0B-C6-45-
    0B-C8-B0-35-F3-2B-07-55-BB-E6-02-C0-19-78-7B-34-
    0B-59-F9-14-59-04-2C-A0-30-E9-A3-7F-68-39-6B-FD-
    09-43-F8-BF-A1-78-5E-4E-E7-20-53-24-04-05-4B-A8-
    85-A0-4C-D1-E9-3E-1B-58-FE-1E-B6-A1-50-81-35-87-
    25-78-4B-4B-D7-21-CE-5B-65-ED-C3-28-65-95-34-49-
    59-CA-69-19-8A-CC-3B-B4-14-DF-62-71-81-30-21-BE-
    D7-97-2A-F3-F6-92-ED-59-18-EB-8C-FA-8B-D4-56-B0-
    3F-DC-58-51-0A-15-36-5F-F6-B7-81-18-E4-A0-13-5F-
    09-A7-71-75-40-43-B6-51-4D-7F-7A-D2-6E-57-89-AC
''').translate(str.maketrans('', '', '\n -')))

# from the "MonitornewWarningmap" value under "HKCU\SOFTWARE\Timerpro\Languagetheme"
REG_DATA = bytes.fromhex(('''
    06-36-3E-35-A4-CB-87-AB-6D-0F-9B-3D-19-8D-A6-D6-
    C4-E3-68-4F-52-79-4B-05-D0-C3-8A-A8-AA-B9-55-41-
    E9-0F-21-CC-37-0A-FC-62-3C-EC-C0-87-27-3E-55-21-
    73-61-FC-90-1D-45-85-B4-F4-DC-61-00-1A-E2-CD-9D-
    66-C9-76-E0-FA-E2-A0-99-58-B5-B8-A5-2C-54-39-79-
    A1-AD-E7-5A-51-B7-12-10-CD-8C-AE-72-9F-00-F4-CE-
    AA-51-68-6D-F3-82-A3-84-33-FA-E4-DD-38-6B-65-2B-
    AB-14-2E-65-03-01-22-C5-FC-77-1C-E4-F1-98-13-E4-
    CA-41-25-1A-8F-CE-E5-83-7F-A6-64-7E-24-34-AF-DA-
    2D-C8-59-7B-DA-74-24-9F-6B-51-9B-20-E0-2B-E3-F7-
    17-06-A1-F1-E1-BA-0F-9A-43-FF-01-AB-A7-19-79-3F-
```

```
    82-27-0D-61-F8-E3-17-8D-37-2B-76-CE-98-33-E2-C8-
    44-DB-49-E8-46-95-0C-DD-6C-BA-39-39-15-43-7A-4B-
    88-E3-89-21-89-38-10-5E-03-53-60-62-08-D8-25-C1-
    30-1F-B4-6F-36-CC-20-98-E1-10-23-CE-33-CB-8D-FD-
    EE-9B-81-33-78-C2-E5-09-59-80-D4-A5-71-08-F7-DC-
    71-89-D2-1D-D6-DE-AF-70-21-C2-95-02-90-3F-C5-F2-
    C3-75-D8-E7-4D-FF-66-A5-E8-AC-1F-08-E6-2F-40-51-
    93-CE-56-AF-06-87-2F-93-19-44-4B-83-F7-C4-E0-99-
    BD-46-3C-15-55-F3-DE-F4-3F-98-8D-FB-4E-FB-15-74-
    B2-78-71-D9-89-AA-BE-82-E6-CD-A2-83-63-CF-97-31-
    EF-94-A6-4A-2D-EA-85-37-3D-8E-B8-05-EE-0A-F5-97-
    5C-C4-74-B6-65-51-28-C1-87-58-16-5D-AB-D3-EB-91-
    1D-16-23-E6-3D-21-5C-CF-9A-B7-8C-79-63-4F-03-17-
    38-F2-9B-B3-BB-11-5C-17-58-E4-48-3C-02-AB-96-F5-
    24-97-08-1C-DB-95-4D-07-FA-0B-48-D3-35-32-A1-5B-
    36-FF-8F-F9-8A-99-0C-12-A6-E3-EC-B7-EC-7E-30-CF-
    71-C9-2E-97-CA-6C-4B-33-EC-C8-C8-E3-EA-AD-53-51-
    1C-BA-BA-F8-85-F8-36-0D-E7-E7-F6-F7-FF-41-9D-29-
    23-07-09-EC-7D-8A-5B-FB-EC-1A-69-1D-FF-B9-CC-32-
    45-AD-69-D5-C8-95-DB-9B-F2-DC-23-AD-31-91-78-3E-
    BE-97-3D-FC-D3-7C-FF-BD-2D-43-B2-E3-70-37-44-E0-
    F8-8E-AE-88-DD-9D-3A-22-BB-A3-76-68-58-8A-4E-92
''').translate(str.maketrans('', '', '\n -')))


def parse_rsa_key(data):
    if not data:
        return None

    if len(data) < 4:
        return None

    length = struct.unpack_from('=I', data)[0]
    if length > 0x1000:
        return None

    start = 4
    end = start + (length >> 3)
    if len(data) < end:
        return None

    N = int.from_bytes(data[start:end], 'big')

    start = end
    end = start + (length >> 3)
    if len(data) < end:
        return None

    E = int.from_bytes(data[start:end], 'big')

    return N, E


def rsa_serpent_decrypt(data, rsa_key):
    if not data:
        return None

    public_key = RSA.construct(rsa_key)
    key_size = (public_key.size() + 1) >> 3
    if len(data) < key_size:
        return None

    data, signature = data[:-key_size], data[-key_size:]
    decrypted = public_key.encrypt(signature, 0)[0]

    try:
        decrypted = decrypted.split(8 * b'\xff' + b'\x00')[1]
    except IndexError:
```

```
        return None

    md5 = decrypted[:16]
    serpent_key = decrypted[16:32]
    data_size, salt = struct.unpack_from('=II', decrypted, 32)

    try:
        decrypted = serpent.cbc.decrypt(serpent_key, data)[:data_size].ljust(data_size, b'\0')
    except Exception as e:
        print(e)
        return None

    assert hashlib.md5(decrypted).digest() == md5
    return decrypted


def parse_config(data):
    config = {}

    try:
        num = struct.unpack_from('=I', data, 0)[0]
    except struct.error as e:
        log.info(e)

    for i in range(num):
        offset = 0x8 + 0x18*i
        crc, flag, pos = struct.unpack_from('=III', data, offset)
        start = offset + pos
        end = start + data[start:].find(b'\0')
        if flag & 1:
            key = format(crc, '#010x')
            value = data[start:end].decode()
            config[key] = value

    print(json.dumps(config, indent=4))

    return config


def main():
    rsa_key_data = serpent.cbc.decrypt(b'90982d21090ef347', ENCRYPTED_PUBLIC_KEY)
    crc, size = struct.unpack_from('=II', rsa_key_data)
    rsa_key_data = rsa_key_data[8 : 8 + size]
    public_key = parse_rsa_key(rsa_key_data)

    tmp = rsa_serpent_decrypt(REG_DATA, public_key)
    confdata = aplib.decompress(tmp[20:])

    parse_config(confdata)


if __name__ == '__main__':
    main()
```

Figure 12 - Decrypting the configuration from the registry using the public key

The script above should yield results similar to this:

```
$ ./get-config.py
{
    "0xb892845a": "https://glory.to.kazohinia",
    "0x72476c70": "0",
    "0x2e58945e": "curlmyip.net",
    "0x556aed8f": "12",
    "0x4fa8693e": "GSPyrv3C79ZbR0k1",
    "0x11271c7f": "300",
```

```
    "0x31277bd5": "300",
    "0xd7a003c9": "300",
    "0x7d30ee46": "300",
    "0x955879a6": "300",
    "0x656b798a": "1000",
    "0xdeff811e": "60",
    "0x584e5925": "60",
    "0x09957591": "10",
    "0x6c451cb6": "0",
    "0x754c3c76": "0",
    "0xe3289ecb": "1",
    "0xea5946a5": "no-cache, no-store, must-revalidate",
    "0x97da04de": "300000",
    "0x8de92b0d": "30, 8, notipda",
    "0xc6c4c2fc": "480",
    "0x9571a0ff": "240",
    "0xdb80f551": "240"
}
```

Figure 13 - Decrypted and parsed bot configuration

The configuration option names are however all just CRCs instead of descriptive names, so you will need to find out where they are used in the code to be able to infer what they are for, or as an alternative, you can also just Google them, because some of these CRCs were used by older variants of Gozi too (i.e. 0xb892845a is the C2 server, 0x4fa8693e is the Serpent key used for sending data to the server, the others are really not important from a challenge perspective).

You will probably notice that the plugins stored in the registry are not using the PE file format, instead they all start with the "PX" magic bytes. By finding the function where these are loaded into memory you can write a script that converts these into a PE format executable, which will let you load these DLL files into disassembler or debugger. You can find such a script in Appendix III - PX to DLL converter script.

THE NETWORK PLUGIN PATCH

The plugin responsible for the network communication is called *45a0fcd0.dll/netwrk.dll*. This plugin was binary patched for the purposes of this challenge, so that it saves the most recently exfiltrated data packet into one of the registry values. This was obviously needed to be able to solve the challenge and retrieve the challenge flag.

The patch uses the function identified by *Ordinal #43* (in the module *d6306e08.dll/rt.dll at address 0x180002498*) to further encrypt the (already encrypted) data packet using a simple XOR-based encryption that uses the lower DWORD of the *machine ID* as the key. Then the function identified by *Ordinal #79* (in the module *8576b0d0.dll/bl.dll* at address *0x1800017fc*) is used to store the data in the registry using a randomly generated registry value name using the seed value *0x7f7f* (this is the *"DiMap"* registry value under the *HKCU\SOFTWARE\Timerpro* registry key).

DECRYPTING THE MOST RECENTLY EXFILTRATED DATA PACKET

Now that you have the following clues you can decrypt the data the malware was about to send over the network, which holds the challenge flag:

1. the fixed XOR key and the machine SID to generate the *machine ID;*
2. the Serpent decryption key -- from bot config in the registry;
3. the last exfiltrated data (encrypted) -- from the *"DiMap"* value under the *HKCU\SOFTWARE\Timerpro* key;

The data packet is encrypted twice, first using the regular Serpent encryption, then using a custom XOR based encryption that uses the lower DWORD of the *machine ID* as the key. Here's a Python script that shows how to accomplish this:

```
#!/usr/bin/env python3
import struct
from zipfile import ZipFile
from io import BytesIO

# sudo -H pip3 install malduck
from malduck import serpent

XOR_KEY = 0xedb88320
MACHINE_SID = 'S-1-5-21-3823548243-3100178540-2044283163'
SERPENT_KEY = b'GSPyrv3C79ZbR0k1'

# from the "DiMap" value under "HKCU\SOFTWARE\Timerpro"
REG_DATA = bytes.fromhex(('''
    04-0C-01-81-E1-85-1F-EF-8D-89-0F-AB-13-A6-A2-64-
    EF-F5-44-B7-10-D0-A8-F5-73-1F-9C-FF-06-9F-FC-23-
    08-4A-11-3A-92-3C-5F-51-71-70-9B-D0-76-9F-50-E7-
    11-A7-22-CE-48-C7-F3-69-78-72-1C-A2-05-B6-F2-31-
    A5-A4-BA-A6-F3-71-E0-61-4B-AD-55-66-BA-34-4F-A0-
    49-37-E6-EF-58-57-56-07-B2-FB-13-63-BC-C2-0B-E3-
    D2-91-F7-B7-1A-76-6A-42-E3-E8-2F-09-31-2F-4F-E2-
    91-44-54-EF-C7-8C-23-35-0D-25-F1-E1-38-80-14-B7-
    F2-7C-55-38-2A-9B-B4-11-D0-63-1F-24-28-90-F1-F3-
    E7-C8-74-46-02-EA-66-CE-1B-A9-71-CC-1B-12-B3-97-
    9E-05-8B-19-04-73-1F-83-E5-D7-DA-F9-05-83-F5-71-
    70-D4-59-C2-1F-D7-D4-7E-6E-77-1A-C3-58-CB-B9-34-
    1C-81-73-C9-DE-A9-64-9A-6E-FD-0F-E2-C3-3D-C3-A3
''').translate(str.maketrans('', '', '\n -')))



def get_machine_id(machine_sid, xor_key):
    machine_id = list(map(int, machine_sid.split('-')[4:7]))
    machine_id = sum(machine_id) + (machine_id[1] << 32)
    machine_id ^= ((xor_key << 32) | xor_key)
    return machine_id


def custom_decrypt(data, key):
    output = BytesIO()

    key1 = key
    key2 = 0
    for i in range(len(data) >> 2):
        x = struct.unpack_from('=I', data, 4*i)[0]
```

```
        tmp = x
        x ^= key1 ^ key2
        key2 = tmp

        rot = ((i % 2) << 2) & 0xffffffff
        x = ((x >> (32 - rot)) | (x << rot)) & 0xffffffff

        output.write(struct.pack('=I', x))

    return output.getvalue()


def main():
    custom_key = get_machine_id(MACHINE_SID, XOR_KEY) & 0xffffffff

    data = serpent.cbc.decrypt(SERPENT_KEY, custom_decrypt(REG_DATA, custom_key))

    with ZipFile(BytesIO(data), 'r') as zf:
        for zi in zf.infolist():
            with zf.open(zi.filename) as f:
                print('%s -> %r' % (zi.filename, f.read().decode().strip()))


if __name__ == '__main__':
    main()
```

Figure 14 - Python script to decrypt the challenge flag from the registry

```
$ ./solve.py
C/Users/Kevin/Desktop/flag.txt -> 'r4d1x_m4l0rum_357_cup1d1745@flare-on.com'
$ 
```

Figure 15 - Running the final decryption script

The challenge flag is "`r4d1x_m4l0rum_357_cup1d1745@flare-on.com`".

APPENDIX I - LIST OF PSEUDORANDOMLY GENERATED WORDS USED IN THE CODE

Seed Value	Generated String
00000303	DayOld
00000707	SolutionDat
00000808	WordTimer
00000909	DatNew
00000a0a	TimerVersion
00000b0b	NewFalse
00000d0d	FalseLanguage
00000e0e	SoftwareColumn
00000f0f	LanguageTheme
00001010	ColumnCurrent
00001111	ThemeDay
00001212	CurrentByte
00001616	VersiScreen
00001717	ScaleThr
00001818	ScreenWeb
00001a1a	WebFalse
00001b1b	CalTimer
00001d1d	TimerPro
00007f7f	DiMap
8576b0d0	WebsoftwareProcesstemplate
d6306e08	WordlibSystemser
45a0fcd0	WebmodeThemearchive
224c6c42	RowmapGuiprotocol
e6954637	SoflogicMagiclink
5f92dac2	ScreenserProtocolacces
7f23179c	DatethrWorkscreen
309d98ff	PrintsolutSavetheme
9eff4536	ThemespellDaytheme
7b41e687	CaclibRegionmap
6bb59728	ProtocolmagicWordeskt

APPENDIX II - LIST OF REGISTRY KEYS/VALUES AND THEIR BRIEF DESCRIPTION

Registry key	**Registry value**	**Description**
Timerpro		**Main registry key**
Timerpro	D	1st stage PowerShell loader (start.ps1)
Timerpro	SolutionDat	Various bot settings
Timerpro	DayOld	Various bot settings
Timerpro	**DiMap**	**Most recently exfiltrated data packet (encrypted)**
Timerpro	ScaleThr	Various bot settings
Timerpro	SoftwareColumn	Various bot settings
Timerpro	ScreenWeb	Various bot settings
Timerpro	FalseLanguage	Various bot settings
Timerpro	ThemeDay	Various bot settings
Timerpro	CurrentByte	Various bot settings
Timerpro	TimerVersion	Various bot settings
Timerpro	DatNew	Various bot settings
Timerpro	VersiScreen	Various bot settings
Timerpro	WebFalse	Various bot settings
Timerpro\Columncurrent		**64-bit plugin storage location**
Timerpro\Columncurrent	WebsoftwareProcesstemplate	8576b0d0.dll / bl.dll
Timerpro\Columncurrent	WordlibSystemser	d6306e08.dll / rt.dll
Timerpro\Columncurrent	WebmodeThemearchive	45a0fcd0.dll / netwrk.dll
Timerpro\Columncurrent	RowmapGuiprotocol	224c6c42.dll / explorer.dll
Timerpro\Columncurrent	SoflogicMagiclink	e6954637.dll / browsers.dll
Timerpro\Columncurrent	ScreenserProtocolacces	5f92dac2.dll / iexplore.dll
Timerpro\Columncurrent	DatethrWorkscreen	7f23179c.dll / microsoftedgecp.dll
Timerpro\Columncurrent	PrintsolutSavetheme	309d98ff.dll / firefox.dll
Timerpro\Columncurrent	ThemespellDaytheme	9eff4536.dll / chrome.dll
Timerpro\Columncurrent	CaclibRegionmap	7b41e687.dll / msedge.dll
Timerpro\Columncurrent	ProtocolmagicWordeskt	6bb59728.dll / mail.dll
Timerpro\Columncurrent	*CalccalcLogicnew*	*Custom plugin for windbg.exe*
Timerpro\Columncurrent	*TimermagSelink*	*Custom plugin for x64dbg.exe*
Timerpro\Columncurrent	*InflibExplorertru*	*Custom plugin for ida64.exe*
Timerpro\Columncurrent	*DiskproIdbui*	*Custom plugin for procmon64.exe*
Timerpro\Columncurrent	*CalciconLogicthre*	*Custom plugin for procexp64.exe*
Timerpro\Columncurrent	*TasknetCharconso*	*Custom plugin for autoruns64.exe*
Timerpro\Languagetheme		**32-bit plugin storage location**
Timerpro\Languagetheme	WebsoftwareProcesstemplate	8576b0d0.dll / bl.dll
Timerpro\Languagetheme	WordlibSystemser	d6306e08.dll / rt.dll
Timerpro\Languagetheme	WebmodeThemearchive	45a0fcd0.dll / netwrk.dll
Timerpro\Languagetheme	RowmapGuiprotocol	224c6c42.dll / explorer.dll
Timerpro\Languagetheme	MonitornewWarningmap	Base configuration data (client.ini)
Timerpro\Languagetheme	SoflogicMagiclink	e6954637.dll / browsers.dll
Timerpro\Languagetheme	ScreenserProtocolacces	5f92dac2.dll / iexplore.dll
Timerpro\Languagetheme	DatethrWorkscreen	7f23179c.dll / microsoftedgecp.dll
Timerpro\Languagetheme	PrintsolutSavetheme	309d98ff.dll / firefox.dll
Timerpro\Languagetheme	ThemespellDaytheme	9eff4536.dll / chrome.dll
Timerpro\Languagetheme	CaclibRegionmap	7b41e687.dll / msedge.dll

Timerpro\Languagetheme	ProtocolmagicWordeskt	6bb59728.dll / mail.dll
Timerpro\Languagetheme	*CalccalcLogicnew*	*Custom plugin for windbg.exe*
Timerpro\Languagetheme	*TimerscreenClientsecur*	*Custom plugin for ollydbg.exe*
Timerpro\Languagetheme	*KeyboardtimerWolib*	*Custom plugin for x32dbg.exe*
Timerpro\Languagetheme	*NewinRegionsea*	*Custom plugin for ida.exe*
Timerpro\Languagetheme	*ThemewebInnet*	*Custom plugin for procmon.exe*
Timerpro\Languagetheme	*ProcesscharProtocomedia*	*Custom plugin for procexp.exe*
Timerpro\Languagetheme	*InfspellTimerver*	*Custom plugin for autoruns.exe*
Timerpro\WordTimer		**Webinjects storage location**
Timerpro\WordTimer	MAIN	Main webinject configuration

APPENDIX III - PX TO DLL CONVERTER SCRIPT

(**Note:** @hasherezade also has her own Gozi config parser toolkit[11], which includes a PX to DLL converter feature, so that could work too, but to be fully honest I have not tested it.)

```
#!/usr/bin/env python3
import os
import argparse
import struct
import logging

__author__ = "Sandor Nemes"

log = logging.getLogger(__name__)

MZ_HEADER = bytes.fromhex(
    '4d5a90000300000004000000ffff0000'
    'b8000000000000004000000000000000'
    '00000000000000000000000000000000'
    '00000000000000000000000010000000'
    '0e1fba0e00b409cd21b8014ccd215468'
    '69732070726f6772616d2063616e6e6f'
    '742062652072756e20696e20444f5320'
    '6d6f64652e0d0d0a2400000000000000'
    '00000000000000000000000000000000'
    '00000000000000000000000000000000'
    '00000000000000000000000000000000'
    '00000000000000000000000000000000'
    '00000000000000000000000000000000'
    '00000000000000000000000000000000'
    '00000000000000000000000000000000'
    '00000000000000000000000000000000'
)


def offset_from_rva(rva, sections):
    for section in sections:
        if section['virtual_address'] <= rva < section['virtual_address'] +
section['virtual_size']:
            return rva - section['virtual_address'] + section['physical_offset']
    return 0


def process_px_file(data):
    magic, = struct.unpack_from('=I', data, 0)
    log.debug("Magic: %08x", magic)
    if magic != 0x5850: return None

    checksum, = struct.unpack_from('=I', data, 4)
    log.debug("Checksum: %08x", checksum)

    raw_data_size, raw_data_offset = struct.unpack_from('=II', data, 8)
    log.debug("Raw data size: %08x", raw_data_size)
    log.debug("Raw data offset: %08x", raw_data_offset)

    size_of_image, size_of_headers = struct.unpack_from('=II', data, 0x10)
    log.debug("Size of image: %08x", size_of_image)
    log.debug("Size of headers: %08x", size_of_headers)

    lfanew = len(MZ_HEADER)
    buffer = bytearray(size_of_image)
    buffer[:lfanew] = MZ_HEADER
    buffer[lfanew:lfanew + size_of_headers] = data[raw_data_offset:raw_data_offset +
```

[11] https://github.com/hasherezade/funky_malware_formats/tree/master/isfb_parser

```
size_of_headers]
    buffer[lfanew:lfanew + 4] = b'PE\0\0'

    machine_arch, number_of_sections, entry_point = struct.unpack_from('=HHI', data, 0x60)
    log.debug("Number of sections: %#x", number_of_sections)
    sections = []
    for i in range(number_of_sections):
        virtual_address, virtual_size, physical_offset, physical_size, section_flags =
struct.unpack_from('=IIIII', data, 0x68 + 0x14 * i)
        log.debug("- section: %x %x %x %x %x", virtual_address, virtual_size, physical_offset,
physical_size, section_flags)
        section_data = data[physical_offset:physical_offset + physical_size]
        section_name, virtual_size, virtual_address, physical_size, physical_offset, _, _, _, _,
section_flags = struct.unpack_from('=8sIIIIIIHHI', data, raw_data_offset + size_of_headers + 0x28
* (i - number_of_sections))
        buffer[physical_offset:physical_offset + len(section_data)] = section_data
        sections.append({
            'name': section_name.rstrip(b'\0'),
            'virtual_size': virtual_size,
            'virtual_address': virtual_address,
            'physical_size': physical_size,
            'physical_offset': physical_offset,
            'flags': section_flags
        })

    log.debug('Directories:')
    directories = {}
    for i, name in enumerate(('import', 'export', 'iat', 'security', 'exception', 'fixups')):
        rva, size, offset = struct.unpack_from('=III', data, 0x18 + 0x0c * i)
        if not rva or not size: continue
        directory_data = data[offset:offset + size]
        offset = offset_from_rva(rva, sections)
        directories[name] = {
            'rva': rva,  # if name != 'security' else offset,
            'offset': offset,
            'size': size,
            'data': directory_data
        }

    size_of_optional_header, = struct.unpack_from('=H', buffer, lfanew + 0x14)
    directory_offset = lfanew + size_of_optional_header - 0x68

    # fix directories
    for i, name in enumerate(('export', 'import', 'resource', 'exception', 'security', 'fixups',
'debug', 'description', 'mips_gp', 'tls', 'load_config', 'bound_import', 'iat', 'delay_import',
'com_runtime', 'reserved')):
        directory = directories.get(name)
        if not directory: continue
        if not directory['rva']: continue
        if not directory['offset']: continue
        if not directory['size']: continue
        struct.pack_into('=II', buffer, directory_offset + 8 * i, directory['rva'],
directory['size'])
        log.debug('- directory: %s %08x %08x', name, directory['offset'], directory['size'])
        buffer[directory['offset']:directory['offset'] + directory['size']] = directory['data']

    struct.pack_into('=H', buffer, lfanew + 0x04, machine_arch)
    struct.pack_into('=I', buffer, lfanew + 0x0c, 0)  # pointer to symbol table
    struct.pack_into('=I', buffer, lfanew + 0x10, 0)  # number of symbols
    struct.pack_into('=H', buffer, lfanew + 0x16, {0x14c: 0x230e, 0x8664:
0x222e}.get(machine_arch))
    struct.pack_into('=H', buffer, lfanew + 0x18, {0x14c: 0x010b, 0x8664:
0x020b}.get(machine_arch))
    struct.pack_into('=I', buffer, lfanew + 0x28, entry_point)
    struct.pack_into('=I', buffer, lfanew + 0x50, size_of_image)

    file_alignment, = struct.unpack_from('=I', data, raw_data_offset + 0x3c)
    overlay_offset = max(section['physical_offset'] + section['physical_size'] for section in
```

```
sections)
    overlay_offset = (overlay_offset + file_alignment - 1) // file_alignment * file_alignment
    buffer = buffer[:overlay_offset]

    return buffer


def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('sample')
    args = parser.parse_args()

    logging.basicConfig(format='%(message)s', level=logging.DEBUG)

    with open(args.sample, 'rb') as f:
        data = f.read()

    with open(os.path.splitext(os.path.basename(args.sample))[0] + os.extsep + 'dll', 'wb') as f:
        f.write(process_px_file(data))


if __name__ == '__main__':
    main()
```