Gozi Gozi Gozi - String Decryption

By R3dy

Published: 2025-12-16 · Archived: 2026-04-05 20:49:18 UTC

Description of the Z2A challenge

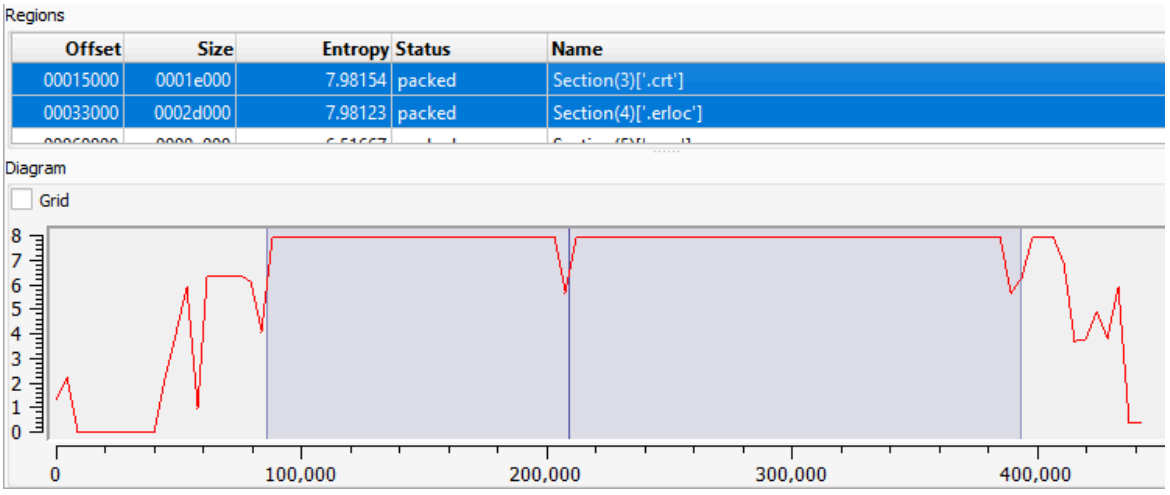
This challenge involves the ISFB malware family!

- Reverse engineer the string decryption routine
- Develop a script to automate decryption

SHA256	`0a66e8376fc6d9283e500c6e774dc0a109656fd457a0ce7dbf40419bc8d50936`
Malware Bazaar link	https://bazaar.abuse.ch/sample/0a66e8376fc6d9283e500c6e774dc0a109656fd457a0ce7dbf40419bc8d50936/
File Type	DLL

Basic Static Analysis

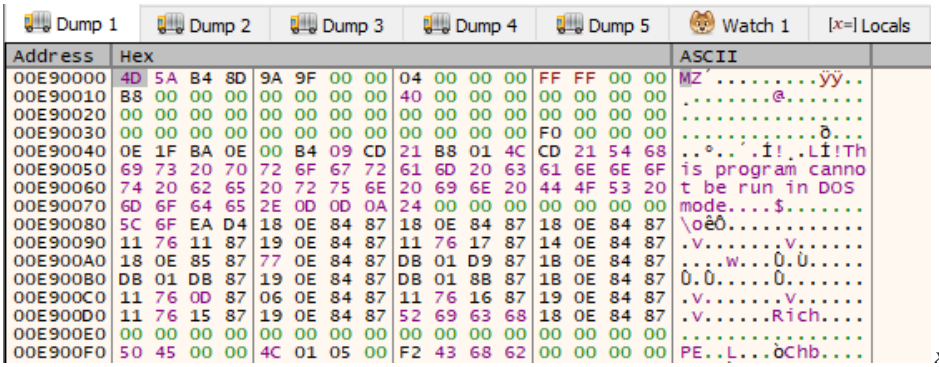
Using *Detect It Easy*, two sections of the provided file show some irregularities. The `.crt` section reaches a high entropy value of `7.98` & `.erloc` also reaches `7.98`. Moreover, the diagram shows a constant flat area on both functions with no spikes. This indicates two encrypted sections and the presence of a packer.



Detect It Easy - Two packed section

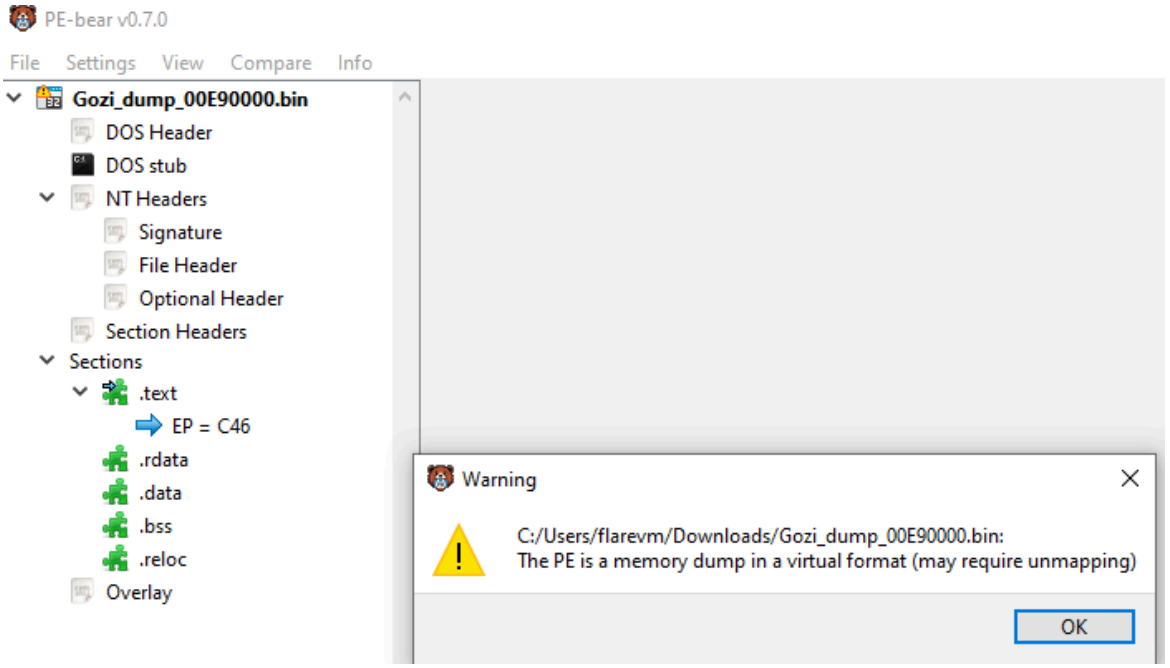
Unpacking

In *x32dbg*, it is necessary to breakpoints on the `VirtualAlloc` & `VirtualProtect` WinAPI functions. After three hits, a valid **MZ*** binary appears in the dump view.



x32dbg in the dump view of the returned value after VirtualAlloc

This binary is mapped, as confirmed by the hex view of the dump. Even if the sample is successfully dumped, it cannot be analyzed correctly at this stage.



PE-Bear Warning message

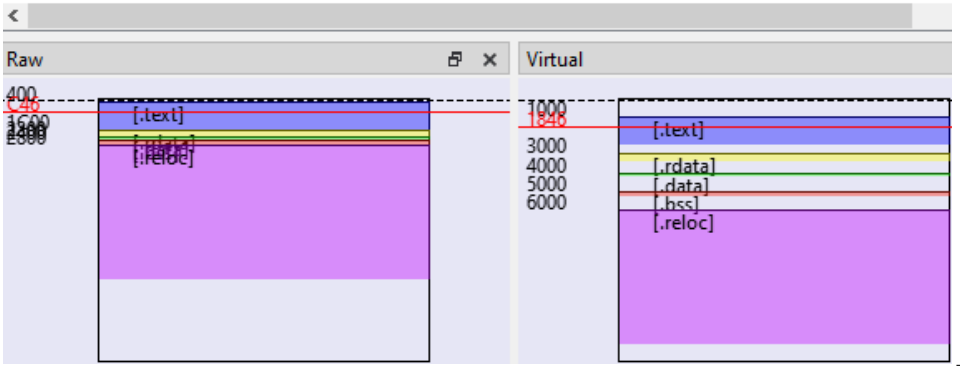
Unmapping PE file

For proper analysis, this dumped binary needs to be converted to an unmapped binary using a PE tool such as *PE-Bear*.

First, open the dumped file in PE-Bear and click on `Section Hdrs` tab.

Disasm | General | Strings | DOS Hdr | Rich Hdr | File Hdr | Optional Hdr | Section Hdrs

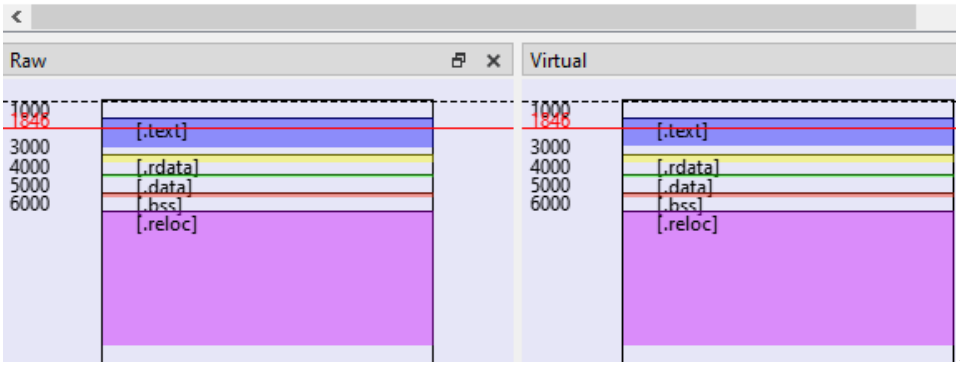
Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	Ptr to Reloc.	Num. of Reloc.
.text	400	1800	1000	16B7	60CA9620	0	0
.rdata	1C00	600	3000	59C	40656140	0	0
.data	2200	200	4000	25C	C068CA40	0	0
.bss	2400	400	5000	2DC	C0FF3840	0	0
.reloc	2800	7200	6000	8000	40F63740	0	0



_PE-Bear - Section view 1 _

Second, edit the `Raw Addr.` field to match the `Virtual Addr.`

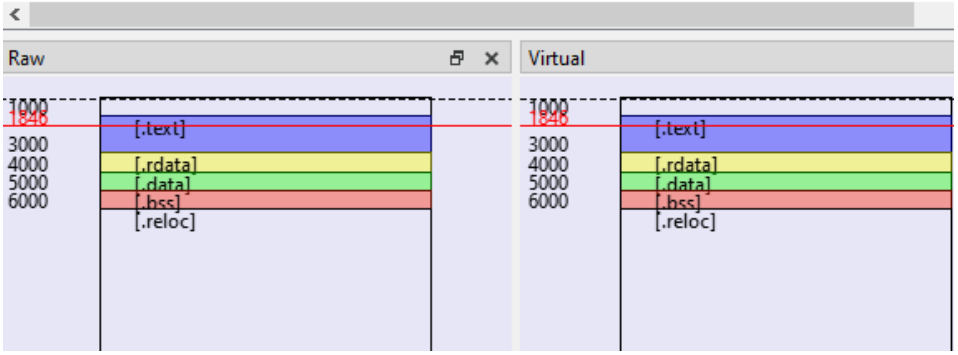
Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	Ptr to Reloc.	Num. of Reloc.
.text	1000	1800	1000	16B7	60CA9620	0	0
.rdata	3000	600	3000	59C	40656140	0	0
.data	4000	200	4000	25C	C068CA40	0	0
.bss	5000	400	5000	2DC	C0FF3840	0	0
.reloc	6000	7200	6000	8000	40F63740	0	0



PE-Bear - Section view 2

Third, use the this formula to edit the `Raw size` : `Raw size n = VA of n+1 - VA of n` . Finally, set the `Virtual Size` to match the `Raw size` .

Name	Raw Addr.	Raw size	Virtual Addr.	Virtual Size	Characteristics	Ptr to Reloc.	Num. of Reloc.
.text	1000	2000	1000	2000	60CA9620	0	0
.rdata	3000	1000	3000	1000	40656140	0	0
.data	4000	1000	4000	1000	C068CA40	0	0
.bss	5000	1000	5000	1000	C0FF3840	0	0
.reloc	6000	0	6000	0	40F63740	0	0



PE-Bear - Section view 3

After editing the `Section Hdrs` tab, go to the `Optional Hdr` tab and verify the `Image Base` value. It must match the packed malware's image base. You can then save the *Gozi* dump file to disk. To confirm that the modifications are applied successfully, inspect the different libraries listed in the `Imports` tab and launch IDA !

Offset	Name	Func. Count	Bound?	OriginalFirstThun	TimeDateSta
3114	ntdll.dll	6	FALSE	3200	0
3128	KERNEL32.dll	36	FALSE	316C	0
313C	ADVAPI32.dll	1	FALSE	3164	0

ntdll.dll [6 entries]

Call via	Name	Ordinal	Original Thunk	Thunk	Forwarder
309C	_snwprintf	-	3372	77157760	-
30A0	memset	-	3368	77159580	-
30A4	NtQuerySyste...	-	321C	77153330	-
30A8	_aulldiv	-	351A	77156D00	-
30AC	RtlUnwind	-	3526	77149480	-
30B0	NtQueryVirtual...	-	3532	77153200	-

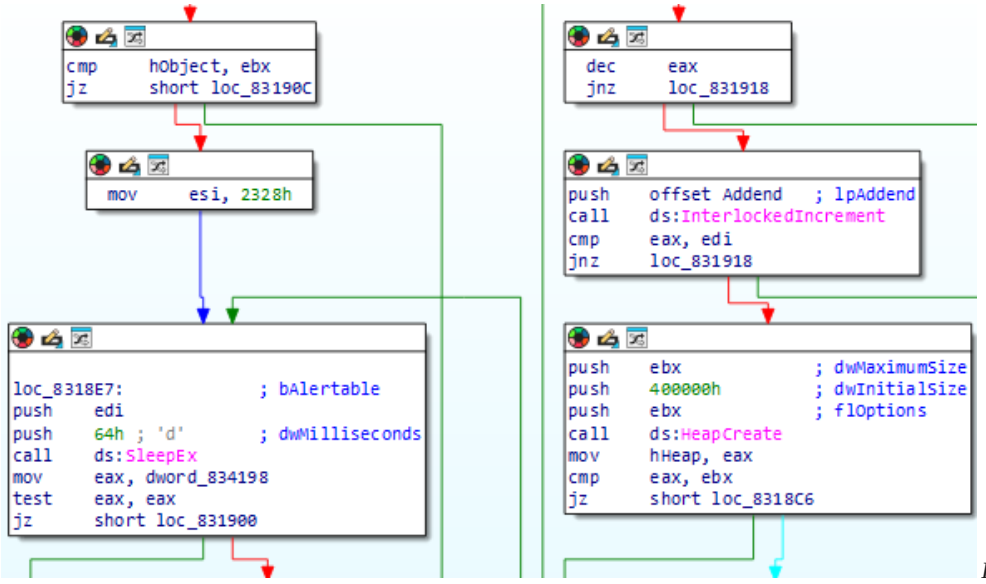
PE-Bear - Imports Tab

> Here is a comparaison between two screenshot on IDA (before & after unmapping the PE file)

```
BOOL __stdcall DllEntryPoint(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpReserved)
{
  int v3; // eax
  int i; // esi
  BOOL v6; // [esp+Ch] [ebp-4h]

  v6 = 1;
  if ( fdwReason )
  {
    if ( fdwReason == 1 && MEMORY[0xF83044](16269704) == 1 )
    {
      MEMORY[0xF84190] = MEMORY[0xF8302C](0, 0x400000, 0);
      if ( !MEMORY[0xF84190] )
        return 0;
      MEMORY[0xF841B0] = hinstDLL;
      _InterlockedExchangeAdd((volatile signed __int32 *)0xF84198, 1u);
      v3 = sub_10001EFE(lpReserved);
      MEMORY[0xF8418C] = sub_10002009(16259967, v3, &fdwReason);
      if ( !MEMORY[0xF8418C] )
      {
        _InterlockedExchangeAdd((volatile signed __int32 *)0xF84198, 0xFFFFFFFF);
        return 0;
      }
    }
  }
  else if ( !MEMORY[0xF83040](16269704) )
  {
    if ( MEMORY[0xF8418C] )
    {
      for ( i = 9000; i > 0; i -= 100 )
      {
        MEMORY[0xF83038](100, 1);
        if ( !MEMORY[0xF84198] )
          break;
      }
      MEMORY[0xF8300C](MEMORY[0xF8418C]);
    }
    MEMORY[0xF83030](MEMORY[0xF84190]);
  }
```

IDA Before Unmapping



IDA After Unmapping

Static Analysis

This first function called `sub_831EFE` checks the `MZ` signature at the beginning of the file, after, a handle to the DLL is retrieved. This handle will be used in the next function which involves **APC Injection**. This method uses 2 interesting parameters :

- A function (analyzed in *The APC function* part)
- A handle to the dll

```
possible_dll = retrieve_dll_handle(lpReserved);
hObject = APC_injection((PAPCFUNC)sub_831B7F, (ULONG_PTR)possible_dll, &fdwReason);
```

APC Injection

> This technique allows a program to execute code in a specific thread by attaching to an APC queue. The injected code will be executed by the thread when it exists of **alertable** state like `SleepEx`, `SignalObjectAndWait` or `WaitForSingleObjectsEx`.

This `APC_injection` function creates a new thread with the starting routine `SleepEx` to trigger the APC queue. Then, the API `QueueUserAPC` is run with 3 parameters:

- `pfnAPC` - *The next function to be analyzed*
- `Thread` - *The thread handle*
- `dwData` - *Unique value transmitted, here the DLL handle*

```
Thread = CreateThread(0, 0, (LPTHREAD_START_ROUTINE)SleepEx, lpParameter, 0, lpThreadId);

v4 = Thread;
if ( Thread && !QueueUserAPC(pfnAPC, Thread, dwData) )
  {
    LastError = GetLastError();
    TerminateThread(v4, LastError);
    CloseHandle(v4);
    v4 = 0;
    SetLastError(LastError);
  }
return v4;
```

The APC function

Let's dive into the APC function, used as a parameter of `QueueUserAPC` above ! The thread is retrieved and pinned to the **CPU 0** with `SetThreadAffinityMask`. After, the thread priority is set to **-1** (THREAD_PRIORITY_BELOW_NORMAL).

```
CurrentThread = GetCurrentThread();
 if ( SetThreadAffinityMask(CurrentThread, 1u) )
   SetThreadPriority(CurrentThread, THREAD_PRIORITY_BELOW_NORMAL);
 v2 = query_info(Parameter);
```

The method `query_info` is the “main” of our APC function. It begins with a check function that verifies the Windows OS version. This program refuses to run on Windows <= `XP/2003`.

```
if (MajorVersion == 5 && MinorVersion == 0)
/* ... */
```

Then, a handle to the current process is collected with [several access](#) rights described in the code below.

```
current_pid = GetCurrentProcessId();
process_handle = (int)OpenProcess(0x10047Au, 0, current_pid);
/*
0x10047A =
PROCESS_QUERY_INFORMATION |
PROCESS_VM_READ |
PROCESS_VM_WRITE |
PROCESS_VM_OPERATION |
PROCESS_DUP_HANDLE |
PROCESS_SET_INFORMATION
*/
```

The following code creates a pseudo-random/magic value using the `NtQuerySystemInformation` API with `SystemProcessorPerformanceInformation` class. This parameter *returns an array of* `SYSTEM_PROCESSOR_PERFORMANCE_INFORMATION` *structures, one for each processor installed in the system*. Furthermore, the `IdleTime` value (first field of the structure) is then used modulo `0x13` (19) to compute this value (`unk_value`). As a result, `unk_value` is a number between 0 & 20 (because NT_STATUS is equal to 0 if the API works fine).

```
result = w_open_process();
  handle_selfProcess = (DWORD)result;
  if ( !result )
  {
    do
    {
      unk_arg = 0;
      ReturnLength = 0;
      SystemInformationLength = 48;
      do
      {
        SystemInformation = (_SYSTEM_PROCESSOR_PERFORMANCE_INFORMATION *)w_heap_alloc(SystemInformationLengt
        if ( SystemInformation )
        {
          ret_NtQuerySysInf = NtQuerySystemInformation(
                                SystemProcessorPerformanceInformation,
                                SystemInformation,
                                SystemInformationLength,
```

```
                                &ReturnLength);
        handle_selfProcess = (unsigned __int16)ret_NtQuerySysInf;
        if ( (unsigned __int16)ret_NtQuerySysInf == 4 )
          SystemInformationLength += 48;
        unk_value = SystemInformation->IdleTime.LowPart % 0x13 + ret_NtQuerySysInf + 1;
        w_heap_free(SystemInformation);
        /* ... */
```

This value is then used later in `sub_83197C`. The function `sub_831922` checks the presence of the `.bss` section. Iterating through all the sections in the binary using `section->Name == 'ssb.'`.

```
image_nt = (_IMAGE_NT_HEADERS *)((char *)handle_dll + handle_dll->e_lfanew);
sections_number = image_nt->FileHeader.NumberOfSections;
result = 0;
section = (_IMAGE_SECTION_HEADER *)((char *)&image_nt->OptionalHeader + image_nt->FileHeader.SizeOfOptionalHeader);
ptr_section = 0;
do
{
  if ( !*(_DWORD *)&section->Name[4] && *(_DWORD *)section->Name == 'ssb.' )
    ptr_section = section;
  ++section;
  --sections_number;
}
while ( sections_number && !ptr_section );
if ( !ptr_section )
  return 2;
bss_VA = ptr_section->VirtualAddress;
if ( bss_VA )
{
  bss_sizeRawData = ptr_section->SizeOfRawData;
  if ( bss_sizeRawData )
  {
    *ptr_bss_VA = bss_VA;
    *ptr_bssSizeRawData = bss_sizeRawData;
    return result;
  }
```

IDA - Iteration to find .bss section

Finally, this function stores the `Virtual Address` and `Size Raw Data` of the section.

Once the malware get these values, it converts the size of the `.bss` section to a number of memory pages using the following formula :

```
pages = (bss_sizeRawData >> 12) + ((bss_sizeRawData & 0xFFF) != 0);
```

> `X >> 12` = `X / 4096` = np `X & 0xFFF` = `X % 4096` = `nb != 0 -> pn++` np (number pages) - nb (number bytes) So if `X = 6500` -> `np = 2` —-> **2 memory pages needed to hold 6500 bytes**

Using the number of pages, it allocates a memory area using `VirtualAlloc` :

```
v5 = VirtualAlloc(
        0,
        pages << 12, // pn * 4096 = bytes
        MEM_COMMIT | MEM_RESERVE, //0x3000
        PAGE_READWRITE // 0x4
);
```

Then, the `decrypt` function is called, using the pseudo-code shows a lot of information that can mislead the analyst. The code below displays the string used by the malware : “ `Apr 26 2022` ”, surely the campaign date of *gozi*.

```
strcpy((char *)weird_str, "26 2022");
decrypt(
        (_DWORD *)((char *)cpy_bss_offset + delta),
        (int)cpy_bss_offset,
        bss_VA + first_dword_date[0] + *(_DWORD *)second_dword_date - counter + unk_value - 1,
        1024);
```

To put it in a nutshell, the `decrypt` function iterates block by block over the `.bss` section. Each block is decrypted by subtracting the `key` from the previous cyphertext. This key is a value that depends on the value derived from the `NtQuerySystemInformation` API (`unk_arg`).

The most important line is `16` , where `current_block += prev_block - key;` is equivalent to `plaintext = ciphertext - key + prev_ciphertext;`

```
int __fastcall decrypt(_DWORD *dest, int source, int key, int counte
{
  int prev_block; // esi
  int addr; // edx
  int current_block; // eax
  int cpy_current_block; // edi

  prev_block = 0;
  addr = source - (_DWORD)dest;
  do
  {
    current_block = *(_DWORD *)((char *)dest + addr);
    cpy_current_block = current_block;
    if ( current_block )
    {
      current_block += prev_block - key;
      *dest = current_block;
      prev_block = cpy_current_block;
      ++dest;
    }
    else
    {
      counter = 1;
    }
    --counter;
  }
  while ( counter );
  return current_block;
}
```

Using all the information we gather while analyzing this sample, a string decryption script can be code.

Strings Decryption Code

The behavior observed above is translated into the following Python script:

```
import pefile
import struct

path = "////////////"
date = b"Apr 26 2022" # Date of campaign

def get_bss_section(pe):
    for section in pe.sections:
        if b".bss" in section.Name:
            data = file[section.PointerToRawData:section.PointerToRawData+section.SizeOfRawData]
            return section.VirtualAddress, data

def retrieve_key(bss_va, date, index):
    date_first_part = struct.unpack("<I", date[0:4])[0]
    date_second_part = struct.unpack("<I", date[4:8])[0]
    return date_first_part + date_second_part + bss_va + index

def decrypt(data, key):
    ct = 0
    final = b""
    for i in range (0, len(data), 4):
        encoded = struct.unpack("I", data[i:i+4])[0]
        if encoded:

            final += struct.pack("I", (ct - key + encoded) & 0xffffffff)
            ct = encoded

        else:
            break
    return final

found = False
pe = pefile.PE(path)
file = open(path, "rb").read()
bss_va, data = get_bss_section(pe)

for i in range (0,20):
    key = retrieve_key(bss_va, date, i)
    decrypted = decrypt(data, key)
    if b"NTDLL.DLL" in decrypted:
        found = True
        break

if found:
    print(decrypted)
    print("-> Decrypted strings above !")
    print("-> Magic value : " + hex(i))
    print("-> Key : " + hex(key))
```

In the next article on the Gozi sample, I will continue the analysis and explore the next steps. See you next time !

R3dy ——————-

“Gouzi-gouzi is a French onomatopoeic expression used in infant-directed speech to amuse babies”
