
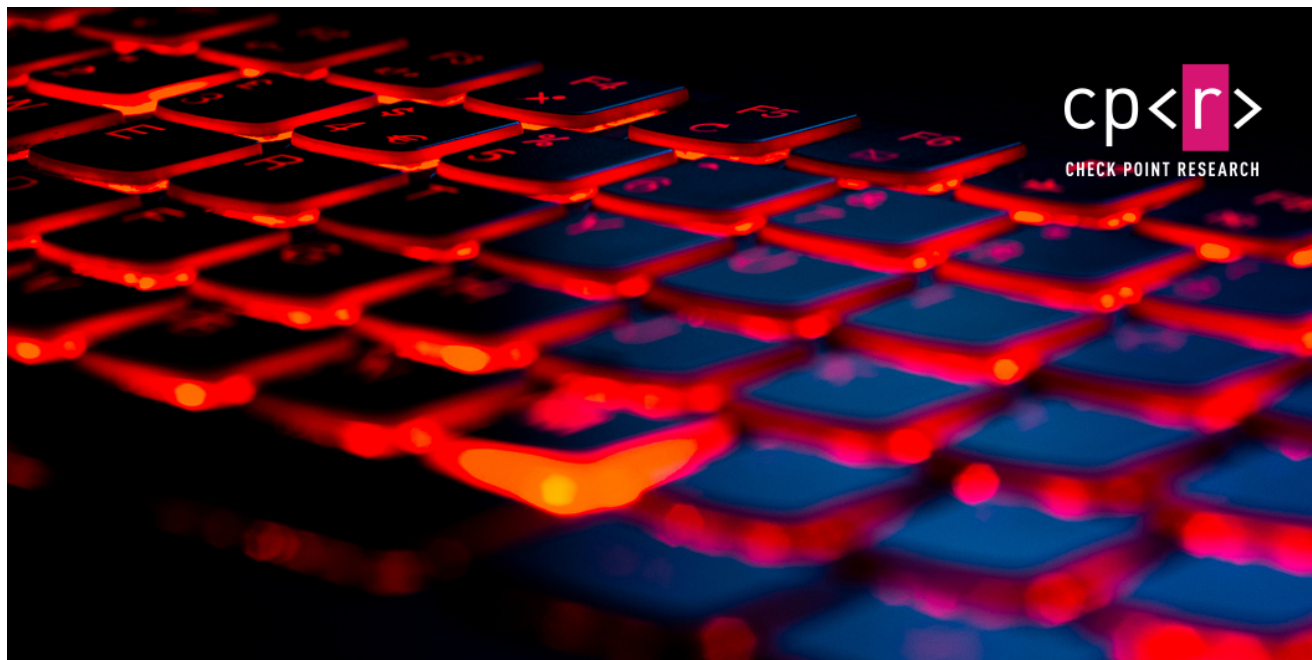


Following the Scent of TrickGate: 6-Year-Old Packer Used to Deploy the Most Wanted Malware

 research.checkpoint.com/2023/following-the-scent-of-trickgate-6-year-old-packer-used-to-deploy-the-most-wanted-malware/

January 30, 2023



January 30, 2023

Research by: [Arie Olshtein](#)

Executive summary

- Initially observed in July 2016, TrickGate is a shellcode-based packer offered as a service to hide malware from EDRs and antivirus programs.
- Over the last 6 years, TrickGate was used to deploy the top members of the “Most Wanted Malware” list, such as Cerber, Trickbot, Maze, Emotet, REvil, Cobalt Strike, AZORult, Formbook, AgentTesla and more.
- TrickGate managed to stay under the radar for years because it is transformative – it undergoes changes periodically. This characteristic caused the research community to identify it by numerous attributes and names.
- While the packer’s wrapper changed over time, the main building blocks within TrickGate shellcode are still in use today.
- Check Point [Threat Emulation](#) successfully detects and blocks the TrickGate packer.

Introduction

Cyber criminals increasingly rely on packers to carry out their malicious activities. The packer, also referred to as “Crypter” and “FUD” on hacking forums, makes it harder for antivirus programs to detect the malicious code. By using a packer, malicious actors can spread their malware more easily with fewer repercussions. One of the main characteristics of a commercial Packer-as-a-Service is that it doesn’t matter what the payload is, which means it can be used to pack many different malicious samples. Another important characteristic of the packer is that it is transformative – the packer’s wrapper is changed on a regular basis which enables it to remain invisible to security products.

TrickGate is a good example of a strong, resilient Packer-as-a-Service, which has managed to stay under the cyber security radar for many years and continually improve itself in different ways. We managed to track TrickGate’s breadcrumb trail despite its propensity for rapidly changing its outer wrapper.

Although a lot of excellent research was conducted on the packer itself, TrickGate is a master of disguises and has been given many names based on its varied attributes. Its names include “TrickGate”, “Emotet’s packer”, “new loader”, “Loncom”, “NSIS-based crypter” and more. We connect the dots from previous researches and with high confidence point to a single operation that seems to be offered as a service.

TrickGate over the years.

We first observed TrickGate at the end of 2016. Back then, it was used to deliver Cerber ransomware. Since that time, we are continually observing TrickGate and found it is used to spread all types of malwares tools, such as ransomware, RATs, info-stealers, bankers, and miners. We noticed that many APT groups and threat actors regularly use TrickGate to wrap their malicious code to prevent detection by security products. TrickGate has been involved in wrapping some of the best-known top-distribution malware families, such as Cerber, Trickbot, Maze, Emotet, REvil, CoinMiner, Cobalt Strike, DarkVNC, BuerLoader, HawkEye, NetWire, AZORult, Formbook, Remcos, Lokibot, AgentTesla, and many more.

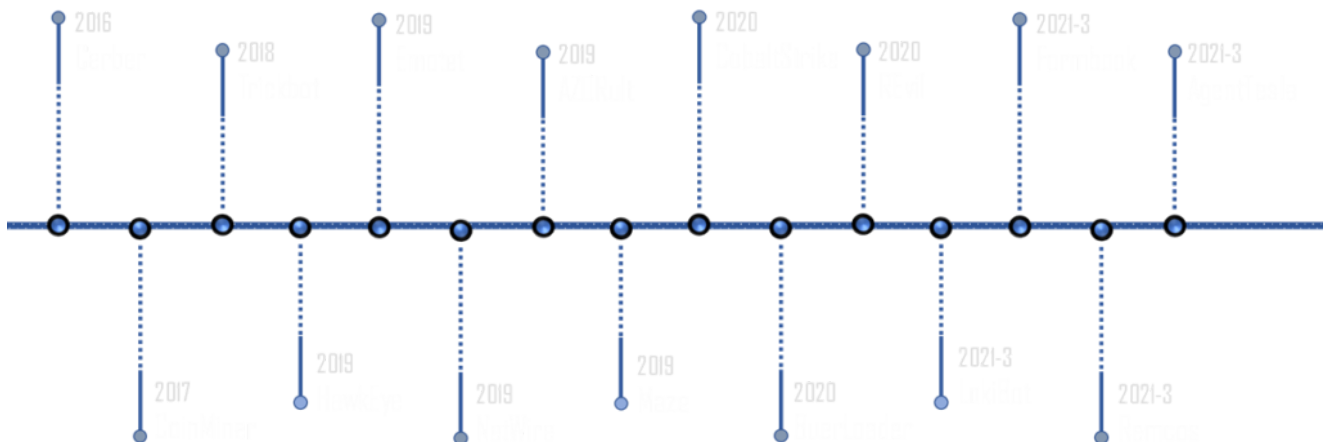


Figure 1 – TrickGate over the years.

TrickGate Distribution.

We monitored between 40 to 650 attacks per week during the last 2 years. According to our telemetry, the threat actors who use TrickGate primarily target the manufacturing sector, but also attack education facilities, healthcare, finance and business enterprises. The attacks are distributed all over the world, with an increased concentration in Taiwan and Turkey. The most popular malware family used in the last 2 months is Formbook with 42% of the total tracked distribution.

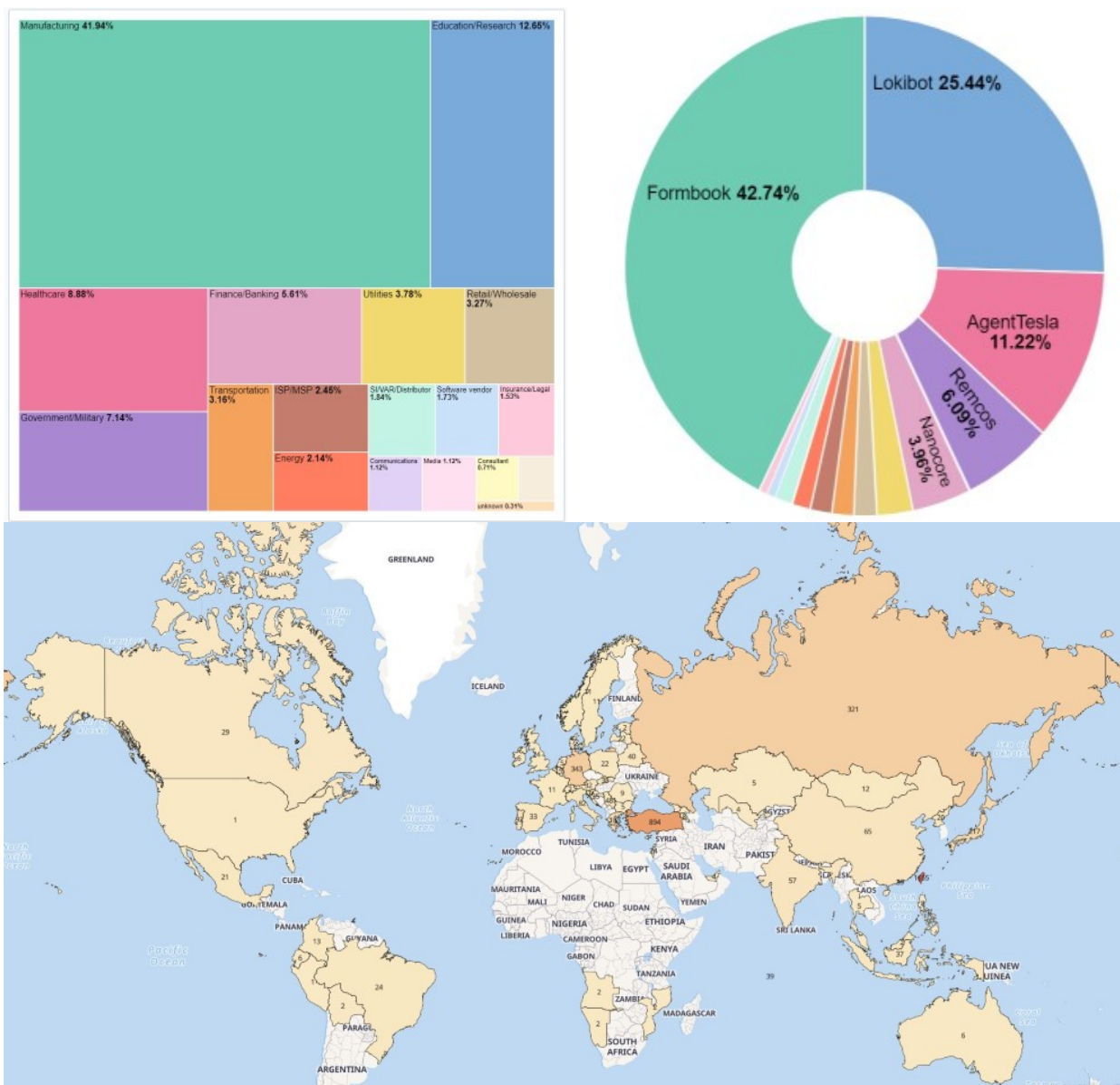


Figure 2 – TrickGate statistics during Oct-Nov 2022.

Attack flow:

Following is an overview of the attack flow that is commonly found in attacks involving TrickGate.

Initial Access

The initial access made by the packer's users can vary significantly. We monitor the packed samples spreading mainly via phishing emails with malicious attachments, but also via malicious links.

Initial Files

The first stage mainly comes in the form of an archived executable, but we monitored many file types and delivery permutations that lead to the same shellcode. We observed the following file types at the first stage:

Archive: 7Z * ACE * ARJ * BZ * BZ2 * CAB * GZ * IMG * ISO * IZH * LHA * LZ * LZH * R00 * RAR * TAR * TGZ * UU * UUE * XZ * Z * ZIP * ZIPX * ZST.

Executable: BAT * CMD * COM * EXE * LNK * PIF * SCR.

Document: DOC * DOCX * PDF * XLL * XLS * XLSX * RTF.

Shellcode Loader

The second stage is the shellcode loader which is responsible for decrypting and running the shellcode.

We noticed 3 different types of code language used for the shellcode loader. NSIS script, AutoIT script and C all implement similar functionality.

Shellcode

The shellcode is the core of the packer. It's responsible for decrypting the payload and stealthily injecting it into a new process.

Payload

The payload is the actual malicious code and is responsible for carrying out the intended malicious activity. The payloads differ according to the actor who used the packer.

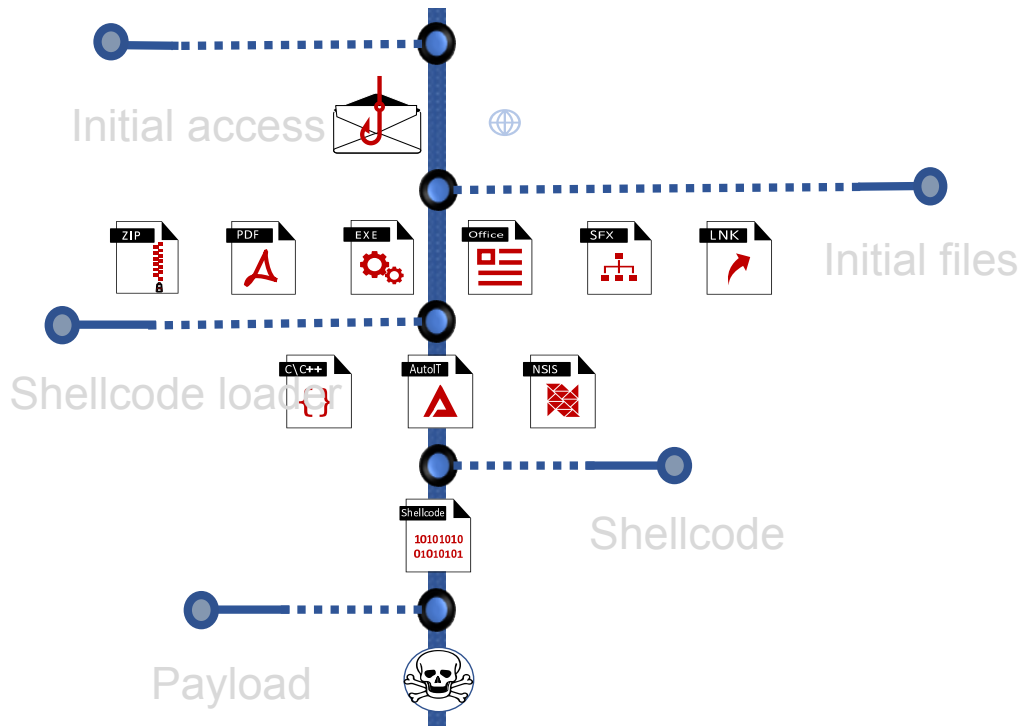


Figure 3 – Attack flow.

Examples of the different attack flows we observed in the past year:

FEB 24, 2022

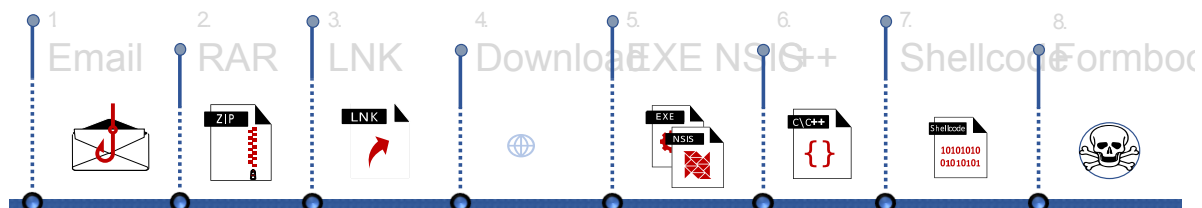


Figure 4 – LNK flow

RAR: 3f5758da2f4469810958714faed747b2309142ae

LNK: bba7c7e6b4cb113b8f8652d67ce3592901b18a74

URL: jardinaix[.]fr/w.exe

EXE 63205c7b5c84296478f1ad7d335aa06b8b7da536

Mar 10, 2022

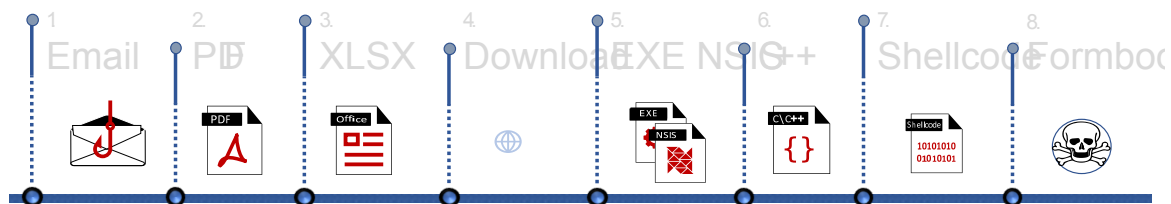


Figure 5 – PDF flow.

PDF: 08a9cf364796b483327fb76335f166fe4bf7c581

XLSX: 36b7140f0b5673d03c059a35c10e96e0ef3d429a

URL: 192.227.196[.]211/t.wirr/XLD.exe

EXE: 386e4686dd27b82e4cabca7a099fef08b000de81

Oct 3, 2022

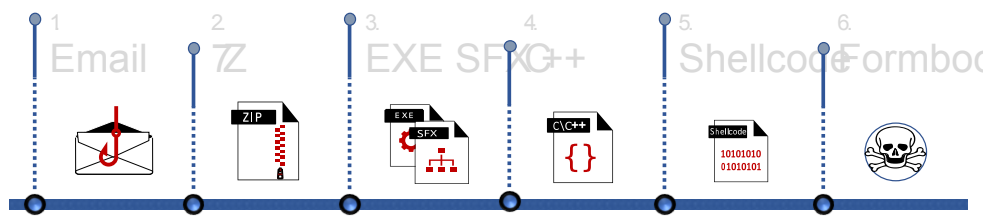


Figure 6 – SFX flow.

7Z: fac7a9d4c7d74eea7ed87d2ac5fedad08cf1d50a

EXE: 3437ea9b7592a4a05077028d54ef8ad194b45d2f

Nov 15, 2022

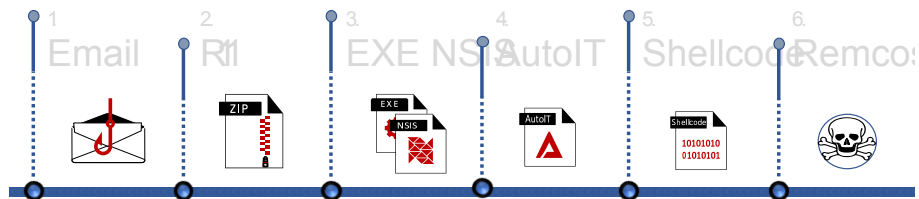


Figure 7 – AutoIT flow.

R11: 755ee43ae80421c80abfab5481d44615784e76da

EXE: 666c5b23521c1491adeeee26716a1794b09080ec

Shellcode loader

The Shellcode loader usually contains a single function which is responsible for decrypting and loading the shellcode into memory. These are the basic steps:

1. Read the encrypted shellcode. The encrypted shellcode can be stored in a file on the disc, in the ".rdata" section or as a resource.
2. Allocate memory for the shellcode, usually by calling VirtualAlloc.
3. Decrypt the shellcode.

4. Trigger the shellcode. As we explain below, this can be done using a direct call or by callback functions.

```
$shellc = FileRead(FileOpen(@TempDir & "\xksfgtax.zfu")
$shellc = StringReplace($shellc, 3366DD5502, "")
Dim $shell_aloc = DllCall(kernel32, ptr, VirtualAlloc, dword, 0, dword, $BinaryLen($shellc), dword, 0x3000, dword, 0x40)
$shell_aloc = $shell_aloc[0]
Dim $0313233qy4i = DllStructCreate(byte whndkhsrvocsa[ & BinaryLen($shellc) & ], $shell_aloc)
DllStructSetData($0313233qy4i, whndkhsrvocsa, $shellc)
DllCall(kernel32, ptr, EnumTimeFormatsA, ptr, $shell_aloc, dword, 0, dword, 0)
```

Figure 8 – Shellcode loader – deobfuscated AutoIT version.

```
hCypherT_shell = CreateFileW((LPCWSTR)argv[1], 0x80000000, 1u, 0, 3u, 0x80u, 0);
dwSize = GetFileSize(hCypherT_shell, 0);
shellcode = (CODEPAGE_ENUMPROCW)VirtualAlloc(0, dwSize, 0x3000u, 0x40u);
ReadFile(hCypherT_shell, shellcode, dwSize, &NumberOfBytesRead, 0);
do
{
  *((_BYTE *)shellcode + len) -= 0x59;
  --*((_BYTE *)shellcode + len);
  *((_BYTE *)shellcode + len) += 0x3C;
  *((_BYTE *)shellcode + len) ^= 0xC3u;
  ++*((_BYTE *)shellcode + len);
  ++*((_BYTE *)shellcode + len);
  *((_BYTE *)shellcode + len) -= 0x3A;
  --*((_BYTE *)shellcode + len);
  *((_BYTE *)shellcode + len) ^= 0xAEu;
  *((_BYTE *)shellcode + len) += 0xF;
  *((_BYTE *)shellcode + len) -= 0x61;
  *((_BYTE *)shellcode + len) += 0x65;
  ++*((_BYTE *)shellcode + len);
  *((_BYTE *)shellcode + len) ^= 0x8Eu;
  --*((_BYTE *)shellcode + len);
  *((_BYTE *)shellcode + len) += 0x5E;
  *((_BYTE *)shellcode + len) -= 0x67;
  *((_BYTE *)shellcode + len) += 0x31;
  ++*((_BYTE *)shellcode + len);
  --*((_BYTE *)shellcode + len++);
}
while ( len < dwSize );
EnumSystemCodePagesW(shellcode, 0);
```

Figure 9 – Shellcode loader C version.

In the more recent versions of TrickGate, the shellcode loader abuses the “[Callback Functions](#)” mechanism. The loader utilizes many native API calls which take a memory address as an argument of a callback function. Instead of the Callback Function, the loader passes on the address of the newly allocated memory which holds the shellcode. When Windows reaches the point of the registered events, the [DriverCallback](#) executes the shellcode. This technique breaks the flow of the behavior we’re monitoring by having Windows OS run the shellcode at an unknown time. In the shellcode loader above, you can see two examples of this in the images “EnumTimeFormatsA” and “EnumSystemCodePagesW”.

Shellcode similarity and TrickGate vacation

Usually, when we find code similarity between unrelated malware families, it is more likely that the actors copied from a mutual resource or shared some pieces of code. For a long time, we noticed a unique injection technique that incorporated the use of direct kernel syscalls, but we didn't realize the significance, thinking it was probably a fragment of shared code. What caused us to suspect that this unique injection may be controlled solely by one actor is the fact that we saw an occasional "time-off" in operation, and it is very unlikely that several different groups will take a break at exactly the same time. The last break, which was more than 3 months long (from June 13, 2022 to September 26, 2022) was an opportunity for us to verify our suspicion, and dive into the shellcode.

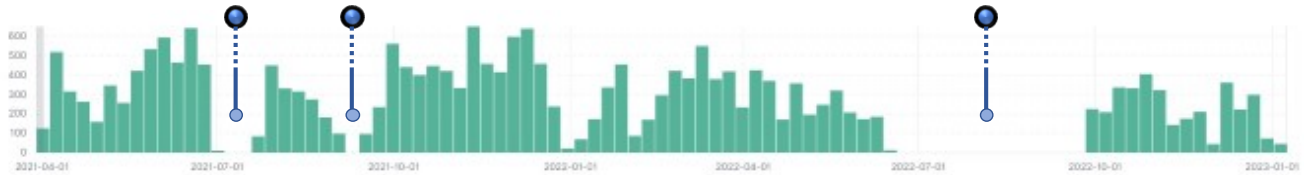


Figure 10 – TrickGate in the last 2 years.

To verify our suspicion, we started to analyze samples across the timeline.

We started our analysis by comparing a fresh sample to an older one. For this test we used

2022-12_Remcos: a1f73365b88872de170e69ed2150c6df7adcdc9c

compared to

2017-10_CoinMiner: 1a455baf4ce680d74af964ea6f5253bbbeeacb3de

We know from the behavioral analysis that a similarity exists in the shellcode, so we ran the samples till the point the shellcode is decrypted in memory and then we dumped the shellcode to the disk. Next, we used the Zynamics [BinDiff](#) tool (owned by Google) to check similarities in both shellcodes. The results showed a 50% similarity between the tested shellcodes. Fifty percent over a long period of time – more than five years – for quite a large piece of shellcode (~5kb) is unexpected. This automatically raised suspicions that this might be a maintained shellcode, but we needed further evidence in the form of similarity analysis over shorter periods of times to see if it had changed gradually.

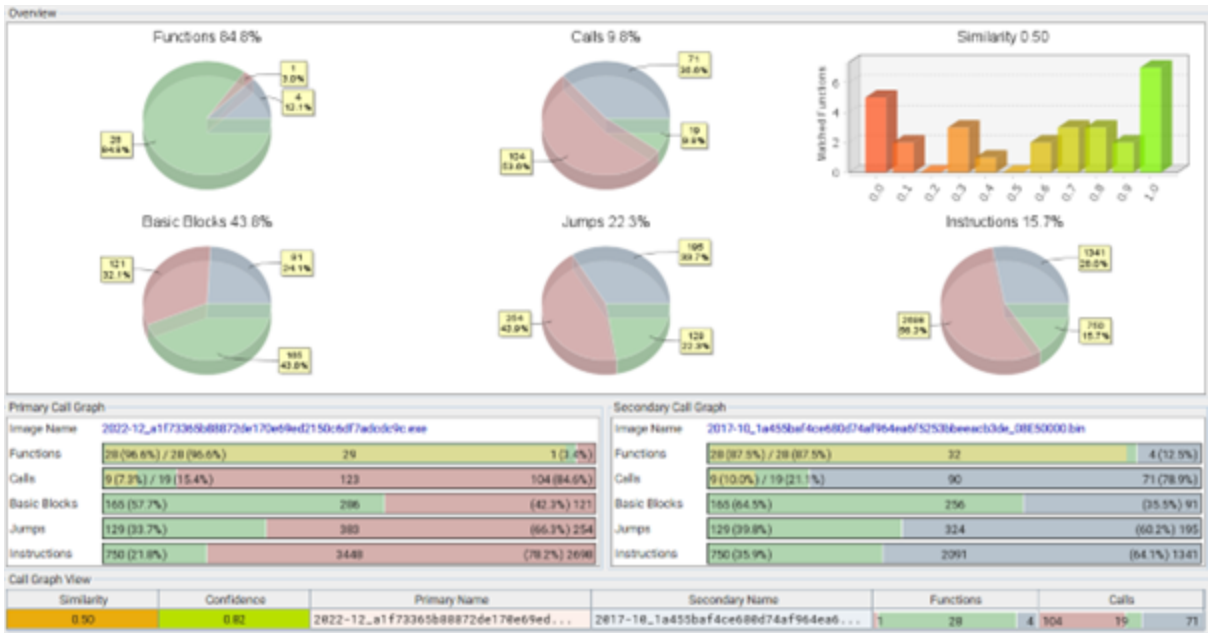


Figure 11 – BinDiff result on shellcode extracted 2022-12_Remcos: a1f73365b88872de170e69ed2150c6df7adcdc9c VS 2017-10_CoinMiner: 1a455baf4ce680d74af964ea6f5253bbeacb3de.

For further analysis, we took random samples from the past 6 years. For each sample, we dumped the shellcode and checked the similarity of the result over time. As you can see in the following graph, the results point to small changes made over time. On the left side we see samples dating from 2016 till 2020 showing about 90% similarity. On the right side, we see a forked version showing a high similarity within itself, but lower similarity with the original version on the left.



Figure 12 – Bindiff result on extracted shellcodes.

We then dived into the gap between the shellcodes to see the impact caused by:

- Different compilers
- Obfuscations
- Evasion modules
- Persistence modules (run the packet payload at the next login)
- Function order
- Local variables vs structures

After we cleaned the gap noise, we got the core functionality of the packer. The author constantly maintained the shellcode but used “building blocks” as described in the next section.

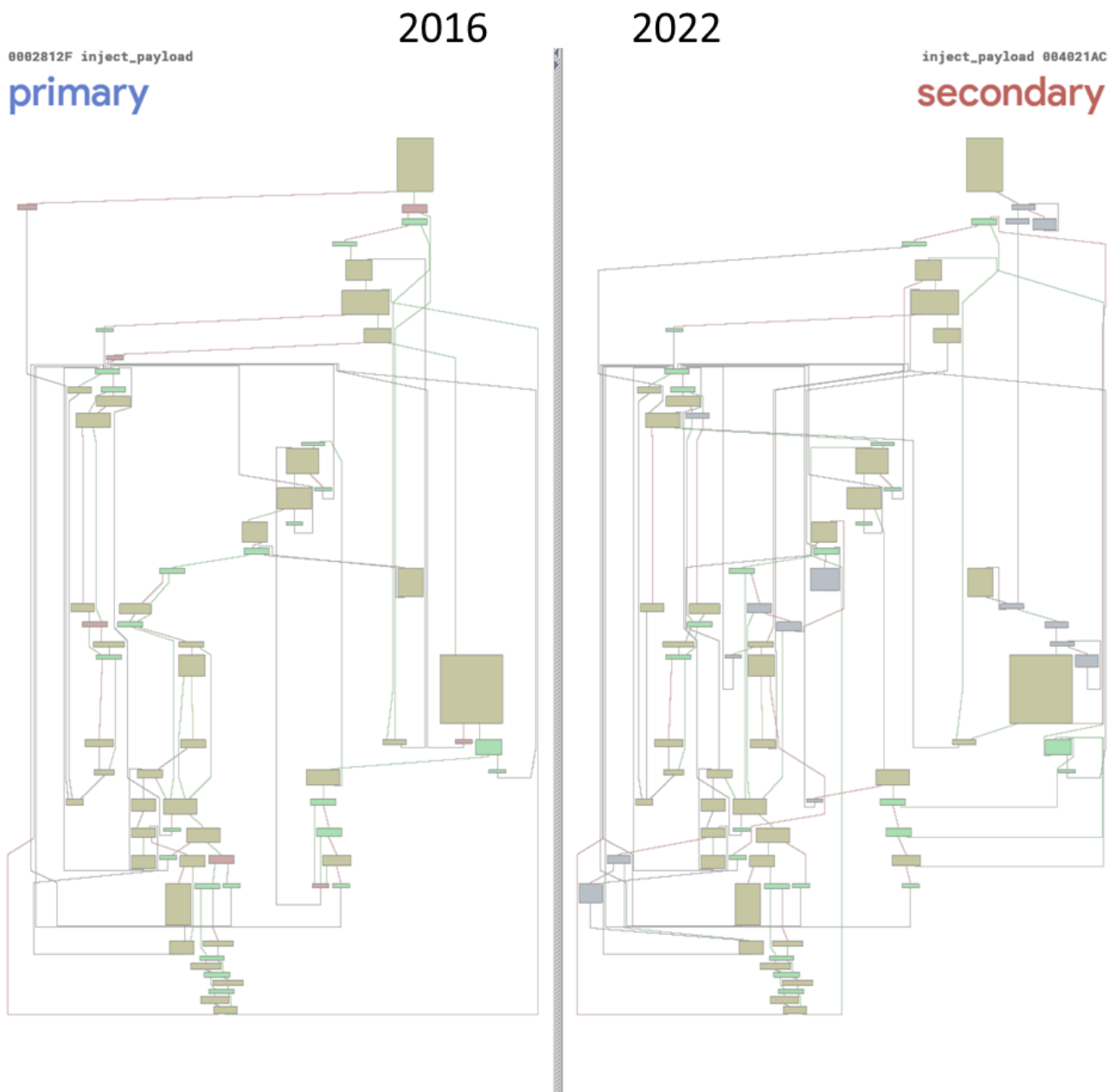


Figure 13 – Control flow graph – on the main injection function. Diffing 2016-07_Cerber: 24aa45280c7821e0c9e404f6ce846f1ce00b9823 VS 2022-12_Remcos: a1f73365b88872de170e69ed2150c6df7adcdc9c

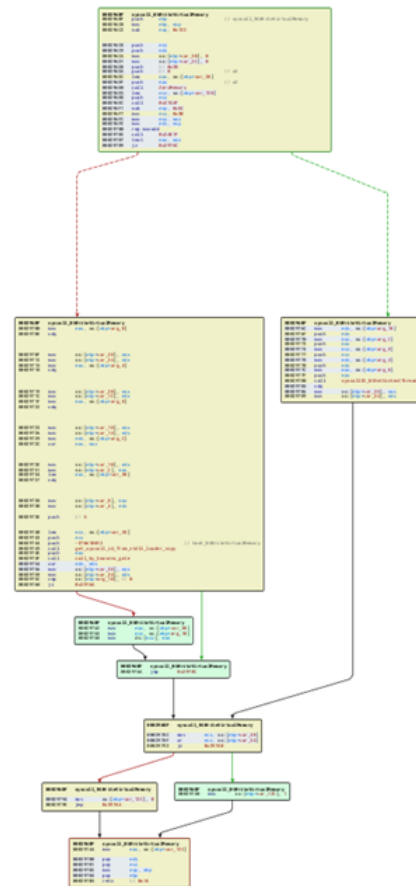


Figure 14 – Diffing kernel direct call of NtWriteVirtualMemory 2022-12_Remcos: a1f73365b88872de170e69ed2150c6df7adcdc9c VS 2016-07_Cerber: 24aa45280c7821e0c9e404f6ce846f1ce00b9823

TrickGate shellcode's construction elements

As mentioned above, the shellcode has been constantly updated, but the main functionalities exist on all the samples since 2016. An overview of the shellcode's building-blocks can be described as follows:

- API hash resolving.
- Load to memory and decrypt the payload.
- Injection using direct kernel calls.
 - Manually map a fresh copy of ntdll.
 - Dynamically retrieve the kernel syscall numbers.
 - Invoke the desired syscalls.
 - Inject and run the payload.

API hash resolving.

When we analyzed the TrickGate code, no constant strings can be found. Many times, TrickGate intentionally adds clean code and debug strings to throw off any analysis. To hide the needed strings and its intentions, TrickGate uses a common technique called API hashing, in which all the needed Windows APIs are hidden with a hash number. Until January 2021, TrickGate used to hash the shellcode string with CRC32. In the newer version, TrickGate started using a custom hash function.

The equivalent Python hashing functions used in the last 2 years:

```
def hash_str_ror1(str):
    h = 8998
    for c in str:
        h += ord(c) + (((h >> 1) & 0xffffffff) | ((h << 7) & 0xffffffff))
    return h & 0xffffffff

def hash_str21(str):
    h = 8998
    for c in str:
        h = ord(c) + (0x21 * h)
    return h & 0xffffffff
```

The following Kernel32 API names have been hashed in TrickGate samples:

API NAME	CRC32	hash_str_ror1	hash_str21
CloseHandle	0xB09315F4	0x7fe1f1fb	0xd6eb2188
CreateFileW	0xA1EFE929	0x7fe63623	0x8a111d91
CreateProcessW	0x5C856C47	0x7fe2736c	0xa2eae210
ExitProcess	0X251097CC	0x7f91a078	0x55e38b1f
GetCommandLineW	0xD9B20494	0x7fb6c905	0x2ffe2c64
GetFileSize	0xA7FB4165	0x7fbd727f	0x170c1ca1
GetModuleFileNameW	0XFC6B42F1	0xff7f721a	0xd1775dc4
GetThreadContext	0x649EB9C1	0x7fa1f993	0xc414ffe3
IsWow64Process	0x2E50340B	0xff06dc87	0x943cf948
ReadFile	0x95C03D0	0x7fe7f840	0x433a3842
ReadProcessMemory	0xF7C7AE42	0x7fa3ef6e	0x9f4b589a
SetThreadContext	0x5688CBD8	0xff31bf16	0x5692c66f

VirtualAlloc	0x9CE0D4A	0x7fb47add	0xa5f15738
VirtualFree	0xCD53F5DD	0x7f951704	0x50a26af

Figure 15 – API hashing.

Load to memory and decrypt the payload.

TrickGate always changes the way the payload is decrypted, so unpacking solutions that we observe now will not work on the next update. Most of the samples use a custom decryption method but on older samples we also saw known cyphers such as RC4 implementation or the use of Windows APIs for encryption.

Injection using direct kernel calls:

After decrypting the payload, the shellcode then injects it into a newly created process. After the process is created using the `create_suspended` flag, the injection is done by a set of direct calls to the kernel. For every one of these `ntdll` API calls:

- `NtCreateSection`
- `NtMapViewOfSection`
- `NtUnmapViewOfSection`
- `NtWriteVirtualMemory`
- `NtResumeThread`

The following actions are executed:

- Manually map a fresh copy of `ntdll` from the disk.
- Resolve the address of a given hash in the newly mapped `ntdll`.
- Dynamically extract the requested System Service Number (SSN).
- Direct kernel Invoke with the SSN.

For Windows 64-bit: Switch to 64-bit mode using “[Heaven’s Gate](#)” technique and `SYSCALL` SSN

For Windows 32-bit: Call `SYSENTER` SSN

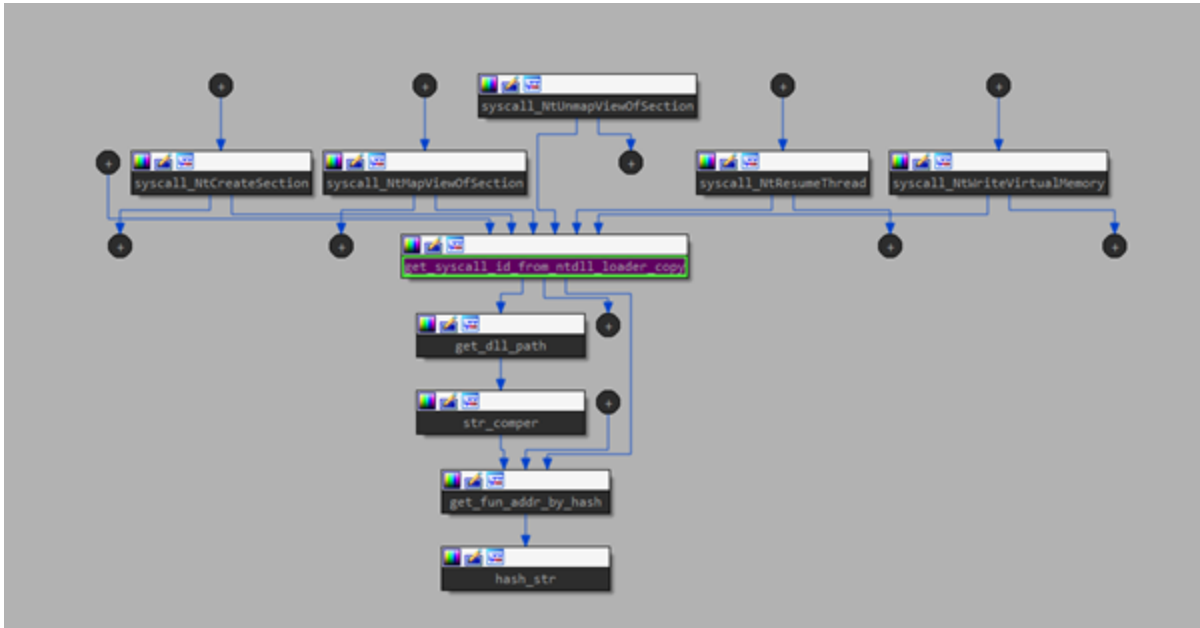


Figure 16 – Function call graph SYSCALL ID from Manually mapped DLL.

The way TrickGate invokes direct-syscalls is intriguing, as it uses a technique similar to Hell's Gate. Hell's Gate is a technique presented publicly in 2020 as a way to dynamically retrieve and execute direct syscall numbers. Here you can find samples dating to 2016 which manage to accomplish the equivalent action to retrieve and execute direct system calls without the need to maintain a System Service Descriptor Table (SSDT).

```

95  farproc = fun_addr_by_hash;
96  while ( *farproc != 0xB8 )           // mov eax DWORD
97  {
98      if ( *farproc == 0xE9 )         // jmp
99      {
100         farproc += *(_DWORD*)(farproc + 1) + 5;
101     }
102     else if ( *farproc == 0xEA )     // ljmp
103     {
104         farproc = *(unsigned __int8 **)(farproc + 1);
105     }
106     else                             // next instruction
107     {
108         ++farproc;
109     }
110 }
111 syscall_num = *(_DWORD *)++farproc;
112 if ( lpAddress )
113     VirtualFree((int)lpAddress, 0, MEM_RELEASE);
114 return syscall_num;
115 }

```

Figure 17 – SSN dynamically extracted 2016-07_Cerber:
24aa45280c7821e0c9e404f6ce846f1ce00b9823

The injection module has been the most consistent part over the years and has been observed in all TrickGate shellcodes since 2016.

Conclusion

We created strings correlating the most wanted malware in the last 6 years to a single Packer-as-a-Service named TrickGate, whose transformative abilities make it hard to identify and track. Understanding the packer's building blocks is of crucial importance to detect the threat, as blocking the packer will protect against the threat in an early stage, before the payload starts to run.

Packers often get less attention, as researchers tend to focus their attention on the actual malware, leaving the packer stub untouched. However, the identified packer can now be used as a focal point to detect new or unknown malware.

Analyzed samples.

03d9cbee9522c2c8a267b7e9599a9d245c35c7ac

043ae57e01ebd0a96fa30b92821b712504cfde03

1a455baf4ce680d74af964ea6f5253bbeeach3de

22f26496f2e8829af9f5cfcd79c47e03fe9a21bb

24aa45280c7821e0c9e404f6ce846f1ce00b9823

30e0181a018fa7dcbd2344dc32adcf77cf840ebe

3437ea9b7592a4a05077028d54ef8ad194b45d2f

3817bad277aa50016e08eed35e92d4a3b5247633

4380044a9517a08514459005836c5f92e4a33871

4f6fa448454b581d6c8e7aa6ed3ef72e66062bf8

666c5b23521c1491adeeee26716a1794b09080ec

75d999d431819311abf8bd048cd084acdc5f4e1

7f456f8b01fc8866aeed4678a14479b6eaa62fed

975629358fbba0344ef0dae4d22697ceb2a32b4

977800bd7be3c5c9f2c0dac7f4806e586d8f7b1a

9f20d00b4ec898a33e130720d4d29e94070e1575

a1f73365b88872de170e69ed2150c6df7adcdc9c

a661541c4cbeb1db859f6cec6c53979b5633c75e

afbe838c881e5b223351ff8fa05ddeb3678581ba

b2d58dfce71ce9c509fab1f00ce04c9526c60695

e6dccb4b1fc5ab116b6bc1321346b35dbf42f387

fa5c79321dd4cc2fea795d6ebe2e823abe33ca6f

[GO UP](#)

[BACK TO ALL POSTS](#)