HAL
archives-ouvertes.fr

LockerGoga quickly reversed

Guillaume Bonfante, Corentin Jannier, Jean-Yves Marion, Fabrice Sabatier

To cite this version:

Guillaume Bonfante, Corentin Jannier, Jean-Yves Marion, Fabrice Sabatier. LockerGoga quickly reversed. MALCON 2019 14th International Conference on Malicious and Unwanted Software, Oct 2019, Nantucket, United States. hal-03178806

HAL Id: hal-03178806
https://hal.inria.fr/hal-03178806

Submitted on 24 Mar 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

LockerGoga quickly reversed

Guillaume Bonfante
LORIA-Université de Lorraine
CYBER-DETECT

Corentin Jannier
CYBER-DETECT

Jean-Yves Marion
LORIA-Université de Lorraine

Fabrice Sabatier
LORIA-CNRS

Abstract

Our objective is to illustrate the uses of the software GORILLE *that we developped at the High Security Lab*[1] *and more recently at* CYBER-DETECT. *The recent attacks of LockerGoga against Altran in France and Norsk Hydro in Norway illustrate the necessity to have advanced anti-malware defences.* GORILLE*'s basis are morphological analysis. As such, the main features of* GORILLE *are the following. It is robust with respect to heavy code obfuscations. It applies on dynamic data that can be forged within a virtual environment. Its detection engine is based on behaviour recognition. This contribution is an extended version of our Blog's post*[2].

Before we talk about reverse engineering, let us present the subject of our interest. `LockerGoga` is a malware that targeted two major companies at the begining of 2019. The first one is Altran in France [**?**] while the second one is Norsk Hydro [**?**]. The "success" of these two attacks show the need of new detection techniques. GORILLE is such a tool. It is now developed by CYBER-DETECT following research in morphological analysis at LORIA [**?**, **?**]. The attack in France happened in January and the one in Norway in March. Actually, those two attacks should have been stopped. Indeed, the GORILLE engine does the job. It detects `LockerGoga` and its variants as we will show it.

In a nutshell, GORILLE identifies malicious threats embedded in Linux, MacOS and Windows binary files. For this sake, GORILLE keeps a collection of malicious behaviours. Each binary file submitted to GORILLE is then scanned and as soon as a set of malicious inter-link behaviours is detected, GORILLE raises an alert. There is no magic behind, just several years of hard work at Loria's Computer Science Lab. For a full presentation of morphological analysis, we refer the reader to our previous contributions, see for instance [**?**, **?**, **?**]. But presently, there is no need to open the engine, looking at morphological analysis as a black box is sufficient.

In this contribution, we do not solve any specific scientific issue. We want to show that our earlier research works—which were attacking some hard scientific points—can/should be reconsidered as a whole. All the ingredients participate to the recipe, dynamic analysis, anti-anti-debugging/virtualization and finally morphological analysis. All these little steps contributed to the tool GORILLE that may serve at many levels within defenses: from detection to retro-engineering. We see this outcome as a strong stimulation to solve some apparently very focused issues. Put all together, they serve greater purposes.

1 A first step: the detection

Since GORILLE search process is based on a collection of malicious behaviours, the first question which comes in mind is whether or not GORILLE is able to detect LockerGoga. The database used for the experiments ("malware-static_24" in the figure below) contains $N = 32,812,355$ malicious behaviours. Among all of them, GORILLE identifies 60 malicious behaviours in the submitted sample of LockerGoga.

Signature based detection techniques need regular updates of their database. Our slogan is that morphological analysis is quite robust to malware versioning or malware repackaging. For that sake, we actually—but, let's confess it, we did it also for fun—used for the experiments our old malware database dating from 2013. And six years later, it is still up to date!

At first side, it could seem that 60 is not that much compared to the $m = 3573$ behaviours of `Hmir.tpz` and the $n = 9879$ behaviours of `LockerGoga`. But, nevertheless, it is significant. We address the question in two parts.

First, finding one behaviour is already meaningful. With the parameters in use, the order of magnitude of the set of

[1] `https://lhs.loria.fr/`
[2] See `http://www.cyber-detect.com/fr-blog.html`.

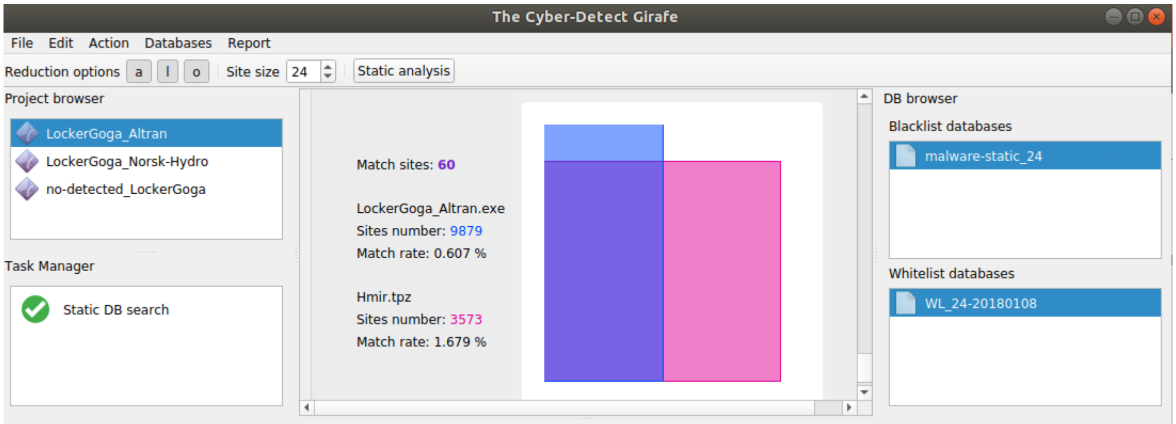


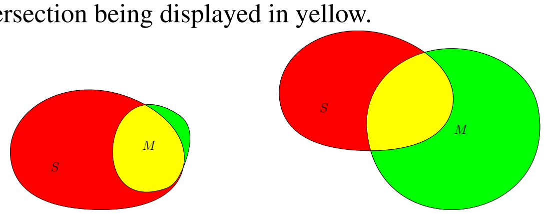
Figure 1. `LockerGoga`**'s proximity to** `Hmir.tpz`

all behaviours (they are stored as labeled graphs of size 24) is

$$|U| \sim 2^{24} \times 24^{48} \sim 2^{244}.$$

Thus, given $N \sim 2^{25}$, the probability of a false positive evaluates to $m \times N/|U| \sim 2^{-244+24+13} = 2^{-207}$. In other words, there is (almost) no chances we pick up a malware behaviour randomly. Nevertheless, one should be careful: the probability of a behaviour does not follow a uniform random law. For instance, there are quite frequent behaviours that are coming from third party libraries written by companies such as Microsoft or open source libraries. Collectively, third party libraries also denote behaviours. They occur in malware but they are not specific to malware. Thus, our tool stores them in a "white list" database (see "WL_24-20180108" in the figure below), such behaviours are then removed from maware databases. The observed false positive ratio is compatible with the theoretical one. Conclusion, we can state that the observed sample is definitely a malware. The second question is: can it be related to some particular one?

One may naively think that the number of common behaviours is the right measure. It is not. The issue can be explained as follows. In the following graph, we suppose we have a "green" malware M and a "red" sample S, the intersection being displayed in yellow.



Even if the intersection on the left is smaller, we can say that in this case, the sample S "extends" M which is not the case for the right drawing. So, it is important to take into account the size of the matched malware. Our proximity measure is based on a probabilistic argument which we briefly justify.

Let us suppose we have a malware, its behaviours should be in our database. Let us choose randomly n such behaviours among N. How many of these are common to the m behaviours of `Hmir.tpz`? Actually, the value follows an hypergeometric law[3]. Thus, according to it, the expected number of common behaviours is given by the formula:

$$E = \frac{n \times m}{N} \simeq 1.07.$$

Now, 60 takes a different flavor. But we can go further, the standard deviation is given by the formula

$$\sigma = \sqrt{\frac{n \times m \times (N-m) \times (N-n)}{N^2 \times (N-1)}}$$

which in the present case amounts approximatively to 1.03. As a conclusion, 60 is far beyond the expected value.

Actually, in our experiment, we could see that there is an other related guy to `LockerGoga`. It is called `Sheldor.db`. It has 47 common behaviours with `LockerGoga` but all these 47 behaviours are common to `Hmir.tpz`. Thus, there are no needs to make enquiries in that way.

To conclude, as we see, GORILLE detects 60 malicious behaviours in the yet undetected sample of LockerGoga. The technological advance of GORILLE allows to stop variants of (some) unknown threats.

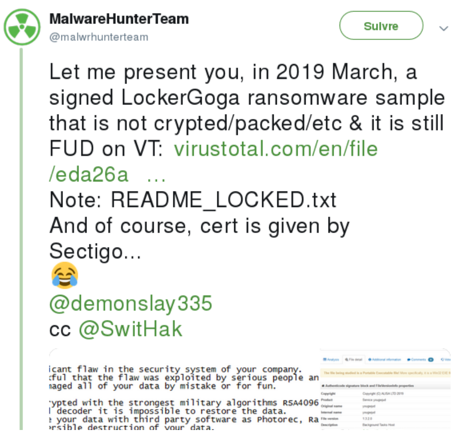
The alert being launched, it is time to start to do some retro-engineering.

2 LockerGoga wears different dresses

The sample named `LockerGoga_Altran` [?] corresponds to the malware that attacked Altran in Jan-

[3] As a matter of fact, with $m \ll N$, we could use in practice a simple binomial law. But, rigorously, it is hypergeometric.

uary 25th, 2019. On March 8th, 2019 that is two months later, **MalwareHunterTeam** discovered in [?] that a variant of `LockerGoga`, that we name here `no-detected_LockerGoga` was left undetected by all anti-virus products in Virus Total [?]. No shame with that, don't forget that malware detection is an heavily complicated problem, actually shown to be undecidable by Cohen [?] or by Adleman [?].

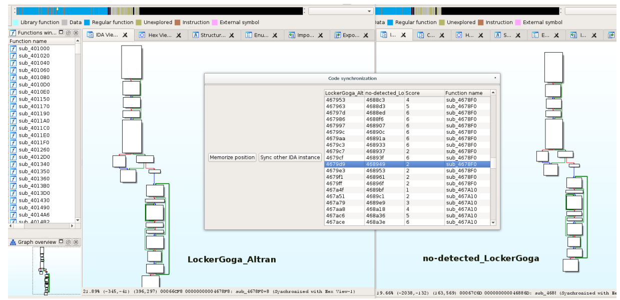


Actually, we can play with GORILLE a little bit more. Indeed, GORILLE is able to learn the specific malicious functionalities of `LockerGoga` by itself. First, we built the "LockerGoga" specific database which contains the 9879 behaviours (also mentioned as sites) in LockerGoga, not only bad ones. Indeed, LockerGoga incorporates, as usual in any software, see figure ??.

That said, we can compare all behaviours of LockerGoga_Altran, which were involved in Altran incident and the no-detected_LockerGoga of MalwareHunter. There are 4270 common behaviours, roughly the half, that are common between bot samples. See figure ??.

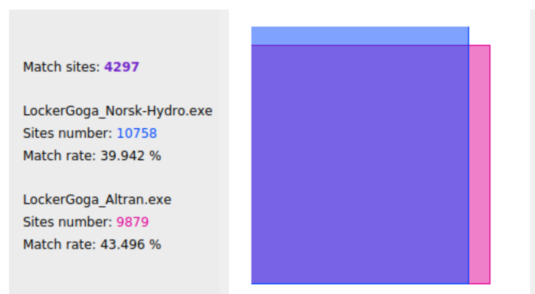
Then, using our tool `binsim` from the GORILLE suite, we can synchronize both codes, that is to find the precise correspondence between functions of LockerGoga_Altran and no-detected_LockerGoga. Actually, in that case, the correspondence was "too" easy. Once we know the correspondence, using our synchronization script, we can even see it within the IDA software[4].

[4] `https://www.hex-rays.com/products/ida/`



Some other tools may be used for code synchronisation. For instance, let us mention `bindiff` [?]. Compared to `binsim`, as their name suggest, `bindiff` will search for differences where `binsim` will perform—possibly wrong—connections. For malware, the "signal" being rather "noisy", we think our approach looks to be more productive.

And then came the Norsk Hydro's attack. We wanted to recognize the malware. Again, we find a clue in our main database. After comparison with the LockerGoga database, no doubt that both are very close.



3 Some LockerGoga's technicalities

Let us add few words on the retro-engineering of LockerGoga. We learnt from [?] that `LockerGoga` is using CryptoPP. Let's go. First, we learn CryptoPP and then, we use the function matching engine of GORILLE to simplify the IDA view. Functions within LockerGoga are automatically labeled with CryptoPP library's names.

The other library that is used by `LockerGoga` is `boost`. But which version of `boost`? To determine it, we learned `LockerGoga`'s behaviours and we searched for matching with different versions of `boost` compiled with different versions of Microsoft Visual C++. The output of GORILLE is:

```
BOOST 1.68 / msvc12
"boost_filesystem-vc120-mt-x32-1_68.dll": 19 matching sites / 1290 sites
4127 nodes = 425 small nodes + 1390 white nodes + 29 matched nodes + 2283 specific nodes
0.00\% 19 / 1290 ou 11148, 1.47\% 0.17\% : LockerGoga_Norsk-Hydro.exe
-----------------------------
BOOST 1.69 / msvc14.0
"boost_filesystem-vc140-mt-x32-1_69.dll": 19 matching sites / 1406 sites
4821 nodes = 667 small nodes + 1337 white nodes + 29 matched nodes + 2788 specific nodes
0.00\% 19 / 1406 ou 11148, 1.35\% 0.17\% : LockerGoga_Norsk-Hydro.exe
----------------------------
BOOST 1.69 / msvc14.1
"boost_filesystem-vc141-mt-x32-1_69.dll": 180 matching sites / 1620 sites
3801 nodes = 364 small nodes + 111 white nodes + 647 matched nodes + 2679 specific nodes
```

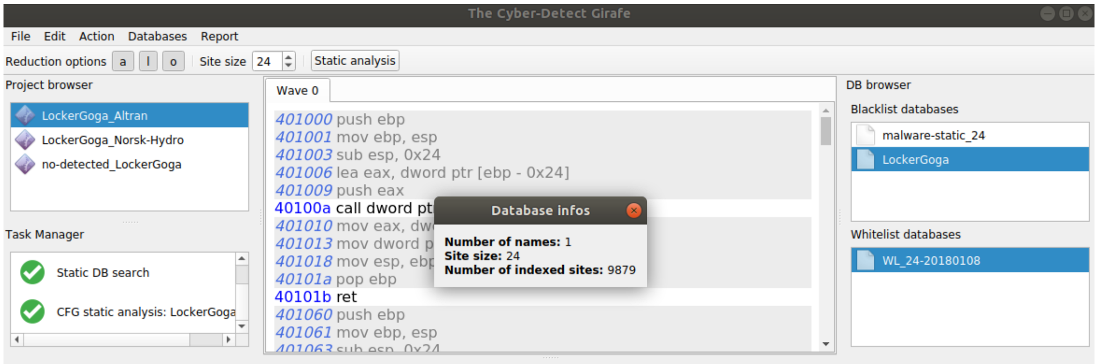


Figure 2. Learning `LockerGoga`

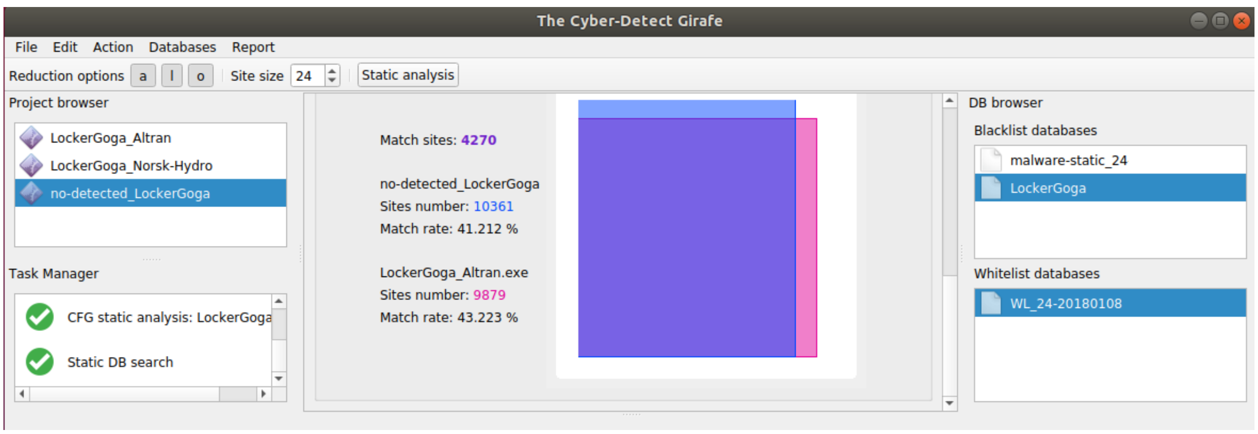


Figure 3. Altran's `LockerGoga` **versus MalwareHunters** `LockerGoga`

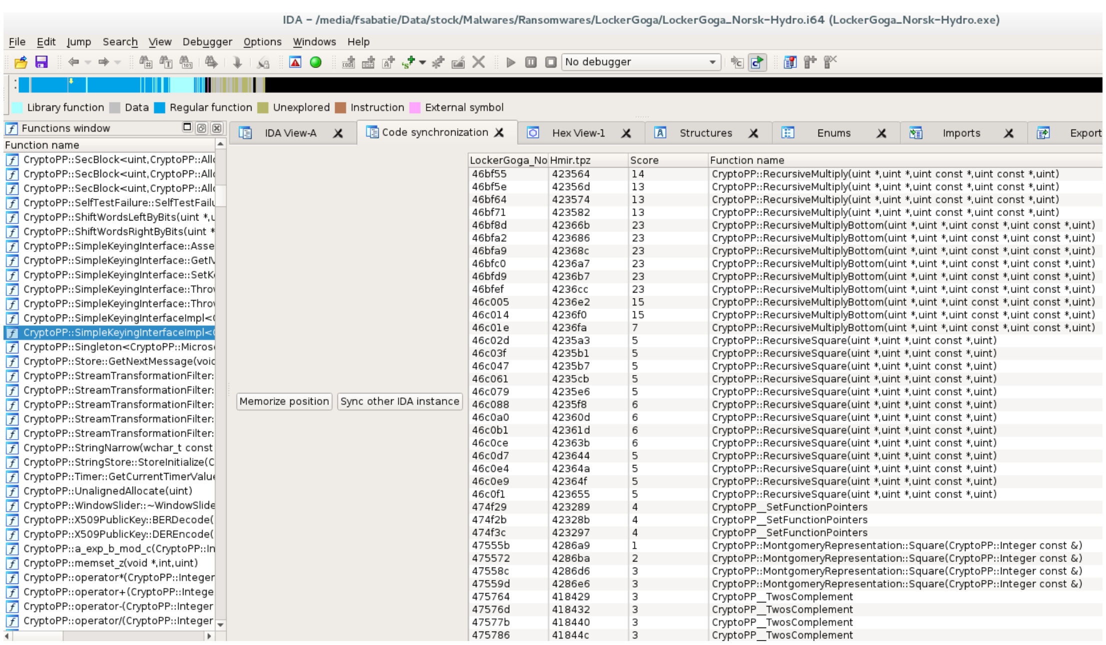
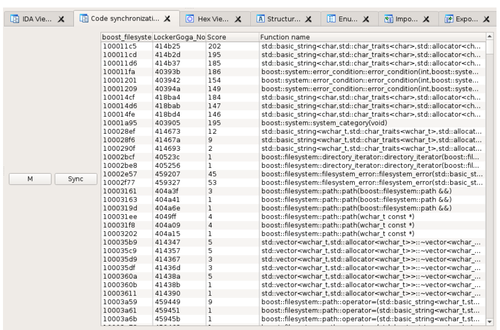


Figure 4. `LockerGoga` **reversed within IDA**

```
0.18\% 180 / 1620 ou 11148, 11.11\% 1.61\% : LockerGoga_Norsk-Hydro.exe
---------------------------
...
```

The best match we find corresponds to `boost.1_69` compiled with Visual C++ 14.1. This gives us a good indication that the malware was built quite recently. With respect to the compiler, the malware is posterior to March 2017 according to Microsoft documentation. With respect to boost version, it is posterior to December 12th, 2018. So, yes, `LockerGoga` was really fresh meat.

Now that we have identified the closest version of boost, we can recompile it to get the corresponding symbols in a `pdb` file (that is in Program Database format). This will help IDA for the disassembling process, but more importantly it will provide the exact name (together with their profile) of identified functions. We developed an IDA script in PYTHON that 1) reads the `pdb` file, 2) build a synchronisation file mapping addresses within `LockerGoga` to addresses within `boost`. This leads to the following tab in IDA:



Then, within IDA, identified functions are presented with their profile. For instance, an excerpt of `LockerGoga` within IDA:

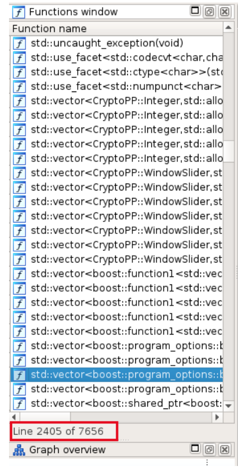
```
; Attributes: bp-based frame

; void __thiscall boost::exception_detail::clone_impl<boost::exception_detail::error_in
??1?$clone_impl@U?$error_info_injector@Vlogic_error@std@@@exception_detail@boost@@@exce

var_C= dword ptr -0Ch
var_4= dword ptr -4

this = ecx
push    ebp
mov     ebp, esp
push    0FFFFFFFFh
push    offset __ehhandler$??_G?$error_info_injector@Vtoo_many_positional_options_error
mov     eax, large fs:0
push    eax
push    esi
mov     eax, ___security_cookie
xor     eax, ebp
push    eax
lea     eax, [ebp+var_C]
mov     large fs:0, eax
mov     esi, this
```

Among the 7656 functions found by IDA within `LockerGoga` (see picture below) we identified without difficulties 178 functions of `boost`. Finally, if we add functions from `CryptoPP` and standard libraries, we identified correctly 2500 functions. All in all, that corresponds to one third of the whole program. Due to the precision (no false positive) of the method, this work is less tedious compared to techniques based on FLIRT signatures (for which a manual inspection is often needed).



Up to this point, everything was done statically. But, GORILLE can take benefit of our dynamic analysis framework. It is based on DYNAMORIO with special efforts to make it transparent to anti-virtualization techniques. And LockerGoga use some of them. There are suspicious `cpuid` instructions. But, our tool also observed a "SystemKernelDebuggerInformation" that is clearly a protection.

There is also call to "OutputDebugString", but that one serves for other purposes. Notice that these calls are made in a separate process that is launched by the original one. And yes, we follow sub-processes and threads. Here is a view

Second point, our tool did not see any self-modification tricks. Thus the static analysis of the file is sufficient. Again, that information saves so much time for the retro-engineer. Nevertheless, the execution of the malware is obfuscated. The main process launch some sub-processes that serve to hide/encrypt data. For instance we can read within the execution trace the following call:

```
0x00a1e492 call [0x7692103d]  WINAPI CreateProcessW(
_In_ [0x0044e00c] "C:\Windows\system32\cmd.exe"
_In_Out_ [0x0044e010] 0x004d7f18
_In_ [0x0044e014] 0x00000000
_In_ [0x0044e018] 0x00000000
_In_ [0x0044e01c] FALSE
_In_ [0x0044e020] 0x00000000
_In_ [0x0044e024] 0x00000000
_In_ [0x0044e028] NULL
_In_ [0x0044e02c] 0x0044e09c
_Out_ [0x0044e030] 0x0044e110
)
Return TRUE
```

So, `LockerGoga` first creates a process that launch a command stored at address `0x004d7f18` which contains the string

```
"C:\Windows\system32\cmd.exe \
```

```
/c move /y e:\Exec\LockerGoga_Norsk-Hydro.exe \
C:\Windows\TEMP\tgytutrc720.exe"
```

that is, it launch a copy of itself. We can observe it a little bit afterwards:

```
[0x004d7f18] "C:\Windows\system32\cmd.exe \
/c move /y e:\Exec\LockerGoga_Norsk-Hydro.exe \
C:\Windows\TEMP\tgytutrc720.exe"
[PROCESS_INFORMATION]
[0x0044e110] 0x000001b4
...
```

Using the same trick, it runs a new process:

```
[0x004d8ed0] "C:\Windows\TEMP\tgytutrc720.exe -m"
[PROCESS_INFORMATION]
...
```

where option `m` stands for master process. The flag serves to manage encryption.

And just for fun. Was the malware difficult to code? It could be. There is a mysterious call to the debugger.

```
 0x0118d5fb call [0x7694b2b7]  WINAPI \
 OutputDebugStringA(
_In_ [0x003bf930] "C:\\Program Files\\ \
Common Files\\System\\msadc\\msadcor.dll"
)
```

4 Conclusion

All right, GORILLE sees LockerGoga. Does it mean it will discover every malware? No, of course not. But, it clearly sees (some) malware that others don't see. We think that a panel of detecting engine using different technologies is much stronger than a simple anti-virus software and want to contribute to this aim.