Playing with GuLoader Anti-VM techniques

Playing with GuLoader Anti-VM techniques

05.Aug.2020

Carlos Rubio and supported by the Blueliv Labs team

Threat Intelligence

GuLoader

GuLoader is one of the most widely used loaders to distribute malware throughout 2020. Among the malware families distributed by GuLoader, we can find FormBook, AgentTesla and other commodity malware. A recent research performed by Check Point suggests that GuLoader code is almost identical to a loader named as CloudEye and sold “legitimately” as a protection mechanism for binaries. At **Blueliv** we have been keeping track of it and have observed that some of the methods it uses to detect virtual machine execution and hook detection, although not new, are still quite effective, as we have seen that in several online sandboxes this loader does not run correctly. Our own sandbox was detected by GuLoader, but it is correctly handled now, executing this malware family without problems. In this blogpost, we will explain the techniques that this loader uses to thwart the analysts and their virtual machines/sandboxes, offering an application to check if a sandbox is detected by this technique or not.

Anti-Analysis & Anti-VM techniques

GuLoader uses the following techniques to make analysis tasks more difficult and to detect if it is running in a virtual machine:

- Using **ZwQueryVirtualMemory** to locate pages containing vm-related strings.
- Enumeration of windows (**EnumWindows**)
- Hooking **ntdll_DbgBreakPoint** and **ntdll_DbgUiRemoteBreakin**
- Checking breakpoints
- Hiding the thread (**NtSetInformationThread** with 0x11)
- Checking the existence of `C:\Program Files\Qemu-ga\qemu-ga.exe` and `C:\Program Files\qga\qga.exe`
- `RDTSC` Virtual machine detection
- Hook detection

Due to the fact that many of these techniques have been documented, we will focus on the following ones throughout this article:

- Hook detection
- RDTSC Virtual machine detection

Sandbox hook detection

GuLoader uses the following code fragment to detect if it's running in a sandbox with hooks installed.

```
_ANTI_HOOK

fnop

pop ebx

cld

cmp dword ptr [ebx], 0

jnz short _jmp_copied_NtAllocateVirtualMemory

clc

xor ecx, ecx

_copy:

clc

push dword ptr [eax+ecx]

pop dword ptr [ebx+ecx]

add ecx, 4

cmp ecx, 18h

jnz short _copy

cld

_jmp_copied_NtAllocateVirtualMemory:

fnop

jmp ebx

_ANTI_HOOK
```

With the push `dword ptr [eax+ecx]` and `pop dword ptr [ebx+ecx]` the NtAllocateVirtualMemory function is copied till `retn 0x18`:
ntdll_NtAllocateVirtualMemory

```
arg_0= byte ptr 4

mov eax, 15h

xor ecx, ecx
```

```
lea edx, [esp+arg_0]
call large dword ptr fs:0C0h
add esp, 4
retn 18h
```

Then it jumps to the code where it has copied the function, using `jmp ebx` and executes it. If it is running in an environment without hooks, the function will be executed correctly. But in the case that there is a hook in the function and this hook makes a relative jump, for example with opcode E9 (such as those performed by the cuckoo monitor), when the function that has been copied to another position is executed, it will jump to an unknown position and an exception will occur, causing an abrupt termination of the program.

RDTSC Virtual machine detection

Even though this technique is widely known, the interesting thing is the way that it has been implemented. GuLoader uses the following algorithm to detect if it is inside a virtual machine:

```
_VM_DETECT proc near
cld
_VM_DETECT_START:
fnop
xor edi, edi
nop
mov ecx, 186A0h
cld
nop
_VM_DETECT_CONTINUE:
push ecx
call _RDTSC_OPS
pop ecx
cmp edx, 32h
jl short _VM_DETECT_CONTINUE
add edi, edx
dec ecx
cmp ecx, 0
jnz short _VM_DETECT_CONTINUE
cmp edi, 0
jl short _VM_DETECT_START
cmp edi, 68E7780h
jge short _VM_DETECT_START
mov eax, edi
retn
_VM_DETECT endp
```

1. The value `0x186A0` is stored in `ECX`. This value indicates the number of times `EDI` will be incremented with the result of the `_RDTSC_OPS` function as long as the result of the operation is greater than `0x 32`. 2. After that it will perform a call to the `_RDTSC_OPS` function, explained later in detail. For now, it is only necessary to know that this function will return a value higher than 0. 3. Then it checks if the value is higher than 0x32. If that's the case it adds the result in the `EDI` register and decreases the `ECX` value. Otherwise it returns to

`_VM_DETECT_CONTINUE` to call `_RDTSC_OPS` again. As a note, if the value returned by `_RDTSC_OPS` at this point is greater than 0 but less than 0x32 continuously, the program will remain in an infinite loop. 4. This will be done until `ECX` is 0, so the result of the function will be added to the `EDI` value with the *`add edi, edx`* operation `0x186A0` times. 5. Finally, it will check if the result of those increments is greater or equal to `0x68E7780` . The result must be lower to pass the virtual machine check. If not, the execution will return to `_VM_DETECT_START` . It is important to highlight that in a virtual machine without any modification or hook over `RDTSC` this is the point where the program will remain running in an infinite loop. Basically, the malware developer estimated that the addition of values returned by the execution of the function `_RDTSC_OPS` `0x186A0` times within a virtualized environment will result in a value above `0x68E7780` due to the overhead generated by the `_RDTSC_OPS` function. If the value of `RDTSC` has been artificially lowered below `0x32` to attempt to bypass similar techniques, the analysis will be stuck in this Anti-VM check forever.

_RDTSC_OPS function

GuLoader uses the following algorithm to obtain the execution times between two `RDTSC` calls, after a `CPUID EAX=1` as shown below:

```
_RDTSC_OPS

lfence

rdtsc

lfence

shl edx, 20h

or edx, eax

mov esi, edx

pusha

mov eax, 1

cpuid

bt ecx, 1Fh

jb short $+2

popa

lfence

rdtsc

lfence

shl edx, 20h

or edx, eax

sub edx, esi

cmp edx, 0

jle short _RDTSC_OPS

retn

_RDTSC_OPS
```

The algorithm performs the following operations:

1. First obtains the elapsed time in `EAX` (low-part) and `EDX` (high-part) through `RDTSC` .

2. Performs an `OR` operation between the high and the low part, and saves the result in `ESI` .

3. It makes the call to `CPUID` with `EAX=1` and then, thanks to the `bt ecx, 1Fh` instruction, it checks if it is running in a virtual machine. However, the result of the operation is not relevant, because the call to `CPUID` is made with the intention of generating a VM-Exit causing that the hypervisor passes the execution to the Virtual Machine Manager. This allows to differentiate if it is running in a virtual machine, because the call takes more time than in a physical machine.

4. Makes a call to `RDTSC` and get again the time that has been spent in `EAX` (low-part) and `EDX` (high-part).

5. Performs an `OR` operation between the high and low parts, and subtracts the previous result stored in `ESI`.

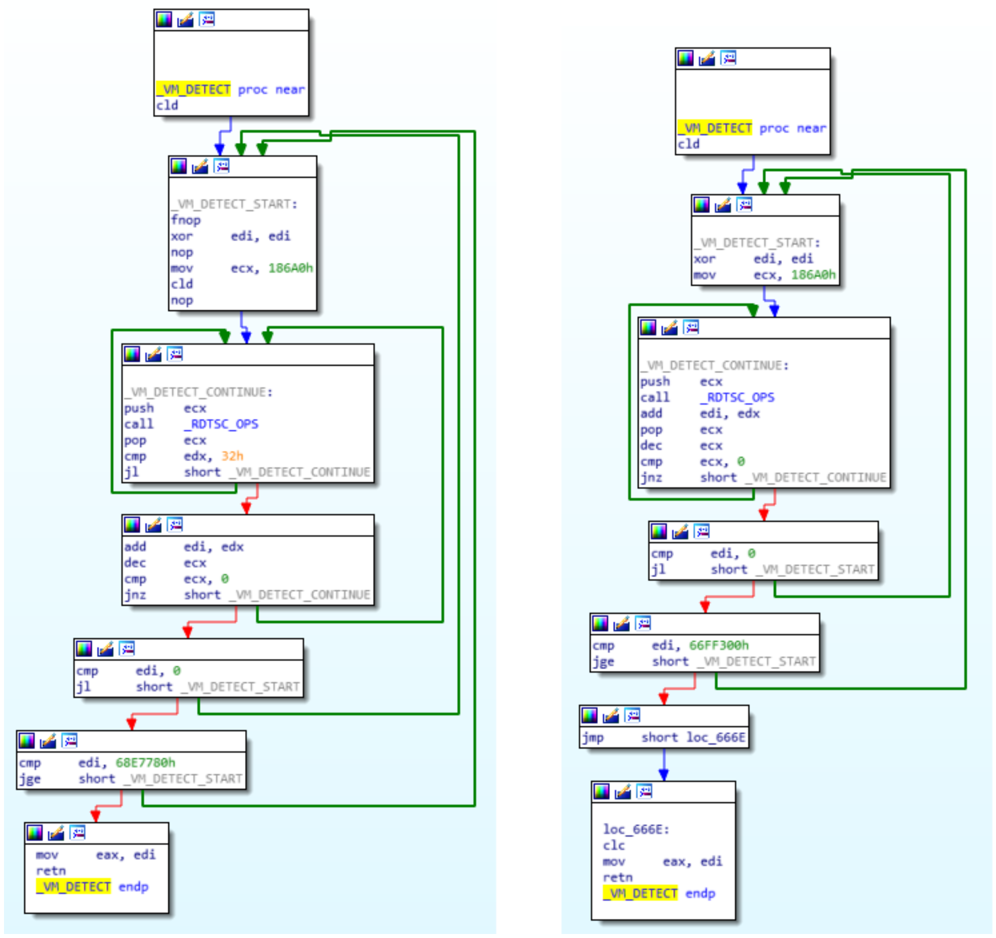
6. If the result is greater than 0, as would be expected in normal execution, the function will return the result in `EDX`, otherwise it will return to the start of the function. It is important to highlight that it is possible that in some sandboxes the sample will be locked in an infinite loop at this point., depending on the way the sandbox deals with the detection problem by `RDTSC`.

Implementation changes in some GuLoader versions

We also found some differences between different GuLoader samples, particularly in the functions responsible for performing the virtual machine check.

Sample	SHA256
GuLoader Blueliv Article	25018a8ff2a535ed05ebe8a1d2158a79dbeb53fc0be67d4e788bc936cb551b6d
GuLoader Checkpoint Shellcode	295cb5b21bafcafa8d770d5ce325893340037c8efe545691b8289aff82315539
CloudEyE Checkpoint Shellcode	3ca3f172d222d5f52be734079658e2a141d92e15c8edd4ea7515a72cf03a28a3

Comparing the sample that we used to write the article with the different shellcodes that can be found in the following Check Point article, it is possible to see how the `_VM_DETECT` function has gone through some modifications.



```
_VM_DETECT proc near
cld

_VM_DETECT_START:
fnop
xor     edi, edi
nop
mov     ecx, 186A0h
cld
nop

_VM_DETECT_CONTINUE:
push    ecx
call    _RDTSC_OPS
pop     ecx
cmp     edx, 32h
jl      short _VM_DETECT_CONTINUE

add     edi, edx
dec     ecx
cmp     ecx, 0
jnz     short _VM_DETECT_CONTINUE

cmp     edi, 0
jl      short _VM_DETECT_START

cmp     edi, 68E7780h
jge     short _VM_DETECT_START

mov     eax, edi
retn
_VM_DETECT endp
```

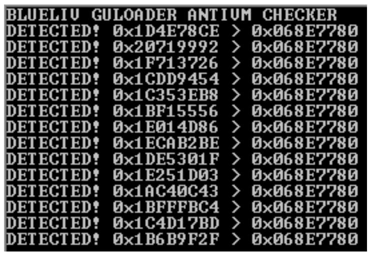
```
_VM_DETECT proc near
cld

_VM_DETECT_START:
xor     edi, edi
mov     ecx, 186A0h

_VM_DETECT_CONTINUE:
push    ecx
call    _RDTSC_OPS
add     edi, edx
pop     ecx
dec     ecx
cmp     ecx, 0
jnz     short _VM_DETECT_CONTINUE

cmp     edi, 0
jl      short _VM_DETECT_START

cmp     edi, 66FF300h
jge     short _VM_DETECT_START

jmp     short loc_666E

loc_666E:
clc
mov     eax, edi
retn
_VM_DETECT endp
```

Comparison of the _VM_DETECT function in different samples

The function on the left corresponds to the sample that we have analyzed throughout the article and the function on the right corresponds to the shellcodes mentioned in Check Point's article. We can see that the function on the left adds a check after calling `_RDTSC_OPS` where it checks if the value is less than `0x32`. As mentioned previously, this new check may be due to the fact that there are sandboxes that after the call to the `_RDTSC_OPS` function return values in `EDX` below the normal. So GuLoader can avoid to be fully executed in those sandboxes where it will stay in an infinite loop at this point. We can also see that the value compared in `EDI` in the function on the left is `0x68E7780`, which is greater than `0x66FF300` (function on the right). It is possible that the variation of this value is due to the fact that there are non-virtualized machines whose value after these operations is above `0x66FF300` and in some update they have had to increase the value up to `0x68E7780`, although this is just an assumption.

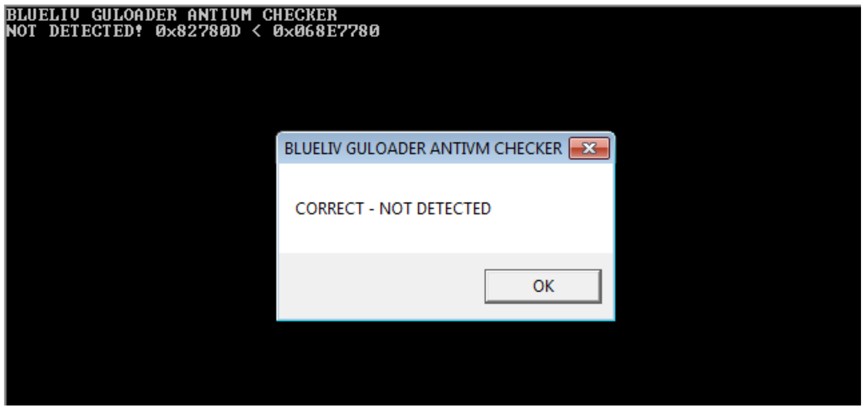
Blueliv Anti-VM detection tool

This technique was quite interesting to us because some of the existent sandboxes were not dealing with the issue. That's why we thought it was a good idea to create a tool to help to detect the problem. This tool permits to know if a sandbox is detected by the technique described above (the most restrictive version). You can find this tool in our repository. If anyone is interested in discussing these technique or taking a look at the source code in order to fix the problem in a sandbox, be free to reach to us. The tool has been created from the code of the GuLoader sample without modifications, to avoid modifying its behavior. If the sandbox is detected, it displays by console the `EDI` value after the `0x186A0` loop have been completed.



```
BLUELIV GULOADER ANTIVM CHECKER
DETECTED! 0x1D4E78CE > 0x068E7780
DETECTED! 0x20719992 > 0x068E7780
DETECTED! 0x1F713726 > 0x068E7780
DETECTED! 0x1CDD9454 > 0x068E7780
DETECTED! 0x1C353EB8 > 0x068E7780
DETECTED! 0x1BF15556 > 0x068E7780
DETECTED! 0x1E014D86 > 0x068E7780
DETECTED! 0x1ECAB2BE > 0x068E7780
DETECTED! 0x1DE5301F > 0x068E7780
DETECTED! 0x1E251D03 > 0x068E7780
DETECTED! 0x1AC40C43 > 0x068E7780
DETECTED! 0x1BFFFBC4 > 0x068E7780
DETECTED! 0x1C4D17BD > 0x068E7780
DETECTED! 0x1B6B9F2F > 0x068E7780
```

>However, if the sandbox is not detected, a message appears on the screen indicating that it does not detect the sandbox.



```
BLUELIV GULOADER ANTIVM CHECKER
NOT DETECTED! 0x82780D < 0x068E7780
```

BLUELIV GULOADER ANTIVM CHECKER

CORRECT - NOT DETECTED

If the tool does not display any value, there are two possibilities:

> The result of the operation between the `RDTSCs` is less than or equal to 0.
>
> The result returned by the `_RDTSC_OPS` function is less than 0x32.

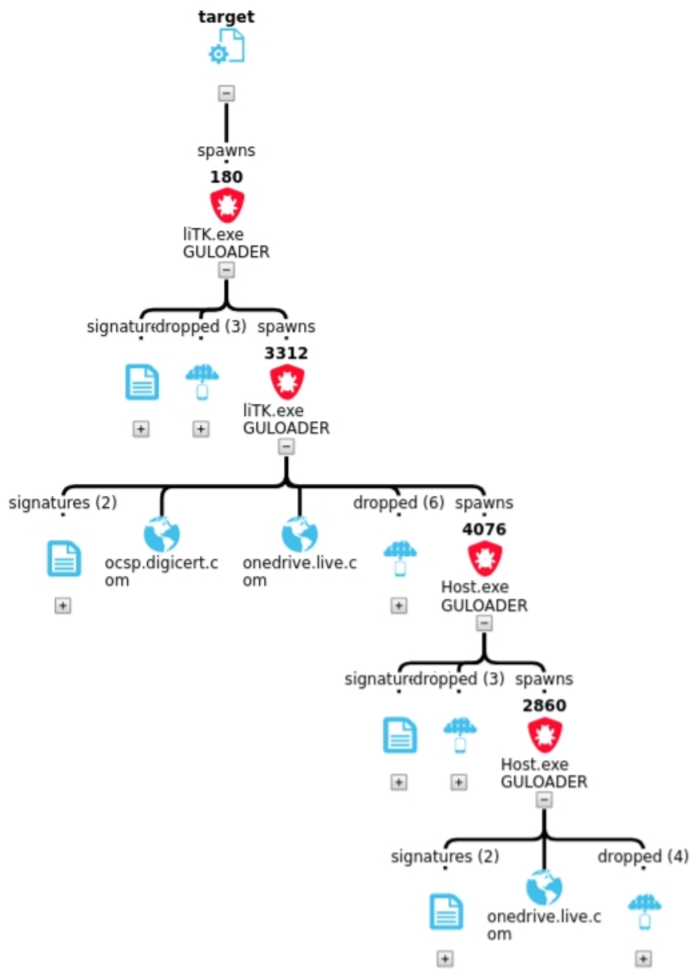
In both cases the execution would stay in an infinite loop. It is recommended to analyze the application step by step to find the problem. We have conducted some tests using the article's sample in several well-known public sandboxes and have found that the sample does not run correctly in most of them. In those that allow access to the API log it is possible to see that the last logged API calls are:

- **NtCreateFile** of `C:\Program Files\Qemu-ga\qemu-ga.exe`
- **NtCreateFile** of `C:\Program Files\qga\qga.exe`

This is because the `RDTSC` and anti-hook checks are performed after those calls, and may be avoiding the correct execution of the sample. We hope our tool can help to fix those problems.

Blueliv sandbox execution graph

We can see below the execution graph generated by the **Blueliv sandbox, using our kernel monitor,** during the execution of this sample (`25018a8ff2a535ed05ebe8a1d2158a79dbeb53fc0be67d4e788bc936cb551b6d`):



As explained in this article, during the execution of the sample it performs several anti-virtual machine and anti-hook checks in order to thwart the analysis, but our sandbox was able to bypass them and continue with the execution flow. When executed it runs twice (`liTK.exe`), then the GuLoader sample creates persistence and changes its name to ***host.exe*** and finally downloads from Microsoft OneDrive the final payload. **The final payload could not be retrieved if the sandbox would not correctly bypass the mentioned anti-vm techniques.**

Conclusions

Cybercriminals and malware authors are always evolving, modifying their tools and using techniques to evade virtual machines and difficult the analysis of the samples. GuLoader is a clear example of this, a really active loader which makes use of some modifications of well-known techniques like the RDTSC check to go further in the detection of sandboxes. The tool we providewill help to confirm if GuLoader is able to detect a sandbox and we hope that the affected online sandboxes can benefit from it and fix this problem. Keeping a sandbox in a good shape and with all the latest anti-vm techniques under control is crucial to be able to detonate samples and artifacts, and bypass packers and obfuscation layers. At **Blueliv** we know that and we investigate and fix all the problems that we detect so the number of extracted IOCs and our malware classification keeps having a good quality. This is especially important for our internal investigations, for our customers and for our **Threat Exchange Network**, where our sandbox can be used free of charge. ***This blog post was authored by Carlos Rubio and supported by the Blueliv Labs team.***

Visit our threat intelligence product page