

# Tackling Gootkit's Traps

By Authors & Contributors

Archived: 2026-04-05 16:19:01 UTC

Gootkit is an advanced banking trojan first discovered in mid-2014. Known for using various techniques to evade detection, the malware also has its own unique twists: it's partially written in JavaScript and it incorporates the node.js runtime environment.

Gootkit employs several checks in order to identify a virtual environment and halt its propagation sequence once that happens. In this post, we'll demonstrate a way to tackle these anti-research methods in order to have the sample executing in a virtual environment.

## First Glance

Running the sample in a virtual machine results in the creation of several threads, but no modifications to the system are made. Without interruptions, Gootkit's process would continue to run forever.

Running the sample in a debugger allows us to interrupt the process and inspect its execution flow. Whenever the debugger is paused, Gootkit's process is sleeping.

According to the return address, the call comes from the malware (not from a system library) and the parameter is -1, meaning: sleep forever.

How can we make the malware go to a more interesting execution flow in the code?

## Diving Deeper into the Assembly Code

Looking at the assembly of the function that calls the sleep API, there are three possible execution paths: #1 and #2 are paths that we could force the malware to take by patching the conditional jumps that lead to them; #3 is the path it takes before reaching the sleep API.

There are 3 possible paths of execution. We'll use a shortcut in order to decide quickly which one is the most interesting one to follow. That shortcut involves checking cross-references(xrefs) from each function, which will indicate where most of the activity would take place.

Execution path #1, contains a call to sub\_4047EC, which had no xrefs.

Execution path #2, contains a call to sub\_409450, which had just a few xrefs.

Execution path #3 creates a thread that will execute sub\_40C8C3, which had many xrefs.

So, execution path #3, the newly created thread, seems like the one to take because it has a massive function graph, indicating that all the activity will most likely happen there.

After placing a breakpoint at the beginning of the thread function(0x40C8C3), we start stepping through. One of the subroutines gets stuck in an endless loop, which means Gootkit is sleeping again!

Recap #1: There are two threads, one of which is sleeping forever, but it is not the interesting one. The second one is in a sleep loop, and our goal is to force it to take a different branch in order to execute the interesting functionality.

The only way to break the loop is to make the string comparison(StrStrIW) fail. The comparison parameters:

- Hard-coded string “Xeon”
- Data from  
`HKEY_LOCAL_MACHINE\HARDWARE\DESCRIPTION\System\CentralProcessor\0\ProcessorNameString`

If we changed this value in the registry, we could successfully overcome this hurdle. But, would that be the end? Would it run? Sadly, after making this change, Gootkit goes back to sleep in another endless loop, this time in a different subroutine.

Recap #2: Again, there are two threads, one that is sleeping forever (but not interesting); the other is running in a sleep loop.

After inspecting the condition that leads us to the sleep API call, it becomes clear that this branch is taken when the “GetProcAddress” fails. But, why would this simple API fail?

At second glance, it is evident that the first parameter of the “GetProcAddress” API, the HModule, is always equal to null because of the XOR instruction highlighted in red. The conclusion is that this API is meant to fail! This whole function is a trap, and our goal is to avoid entering it altogether.

## Following the Traps

We’ll rename the function to “trap” and then check all the function references to see how many traps are there.

There are 17 traps to overcome and they are all in the same function, so we assume this is the function making all the environment checks. We’ll have to check each condition that leads to a trap and make sure to change the environment (or patch the binary) in a way that would bypass the trap.

Figure 8 shows a code snippet from the sub\_409AE2 function.

Note that each “while” loop is performing string decryption on the sequences of bytes shown in the variables above the loop. When following the execution in a debugger, the strings are decrypted, and some meaningful indicators of VM checks are visible. (See appendix for decryption function details.)

In this code snippet, three checks are evident:

- MAC address check
- Checking the presence of “dbghelp.dll” — debugger indicator

- Checking the presence of “sbiedll.dll” — sandboxie indicator

By following the traps and patching the system accordingly, the environment is prepared for Gootkit to run in.

The rest of the checks include:

- Compare user name to "CurrentUser"/"Sandbox"
- Compare computer name to "SANDBOX"/"7SILVIA"
- *HKEY\_LOCAL\_MACHINE\HARDWARE\DESCRIPTION\System\SystemBiosVersion* compare with AMI, VirtualBox, BOCHS, INTEL 640000, 55274-640-2673064-23950, and other serials

After patching a virtual machine and running the sample, it’s clear that it is no longer stuck in an endless loop and that the sample continues its propagation in the system.

## The Road Not Taken

Patching the virtual environment by changing the MAC address and several registry values is one way to get the sample to run. It is the recommended way to go when running Gootkit samples in an automated analysis system. However, this post would not be complete without presenting the other solution: patching the binary itself.

Every time there’s a conditional jump to the trap function, one could change it to a simple jump opcode, thus skipping the trap function. This would direct the flow from the jump to the next code block while skipping the trap function.

However, this patch must be applied 17 times, and that’s a tedious task. Instead, since all the calls to the trap function come from one function, its preferable to patch and bypass the call to that function.

## Conclusion

When analyzing malware, especially intricate, multi-threaded ones like Gootkit, it is important to choose the interesting thread to follow and to keep in mind that even functions that look like they contain seemingly “ordinary” code may be used as a deception.

MD5: 47f8d6d3e4410218c4c64701cb7d7d3d

## Appendix

The decryption function is an XOR loop with a hard-coded key. The encrypted string and the key are hard-coded sequences of bytes.

In Python:

An example of the string decryption loop and its parameters:

```
encrypted_string = "\x3A\x03\x05\x15\x55\x0E\x25\x4F\x08\x1C\x5D\x62"
```

key = "\x49\x61\x6C\x70\x31\x62"

---

Source: <https://www.f5.com/labs/articles/threat-intelligence/tackling-gootkit-s-traps>