Dissecting GootLoader With Node.js

By Riley Porter, Mark Lim

Published: 2024-07-03 · Archived: 2026-04-05 23:14:27 UTC

Executive Summary

This article shows how to circumvent anti-analysis techniques from GootLoader malware while using Node.js debugging in Visual Studio Code. This evasion technique used by GootLoader JavaScript files can present a formidable challenge for sandboxes attempting to analyze the malware.

Sandboxes with limited computing resources can struggle to analyze a large volume of binaries. Malware often takes advantage of this to evade analysis by delaying its malicious actions, which is commonly described as "sleeping."

GootLoader is a backdoor and loader malware that its operators have actively distributed through fake forum posts. The infection process of GootLoader starts with a JavaScript file.

Palo Alto Networks customers are better protected from these threats through our Next-Generation Firewall with Cloud-Delivered Security Services including Advanced WildFire, as well as through Cortex XDR. If you think you might have been compromised or have an urgent matter, get in touch with the Unit 42 Incident Response team.

Related Unit 42 Topics	**GootLoader**, **Evasion**, **Memory Detection**

Background

Gootkit was first reported in 2014, and it underwent many changes over time. In 2020, at least one source identified a JavaScript-based type of malware named Gootkit Loader, which its operators distributed through fake forum posts. The group behind this campaign has kept the same distribution tactic and as of 2024 they continue using fake forum posts that are nearly identical in appearance.

Many security vendors shorten Gootkit Loader to GootLoader when referring to these JavaScript files. While the original Gootkit malware was a Windows executable, GootLoader is JavaScript-based malware, and it can deliver other types of malware, including ransomware.

Since January 2024, we have investigated several GootLoader samples. The infection chain is shown below in Figure 1.

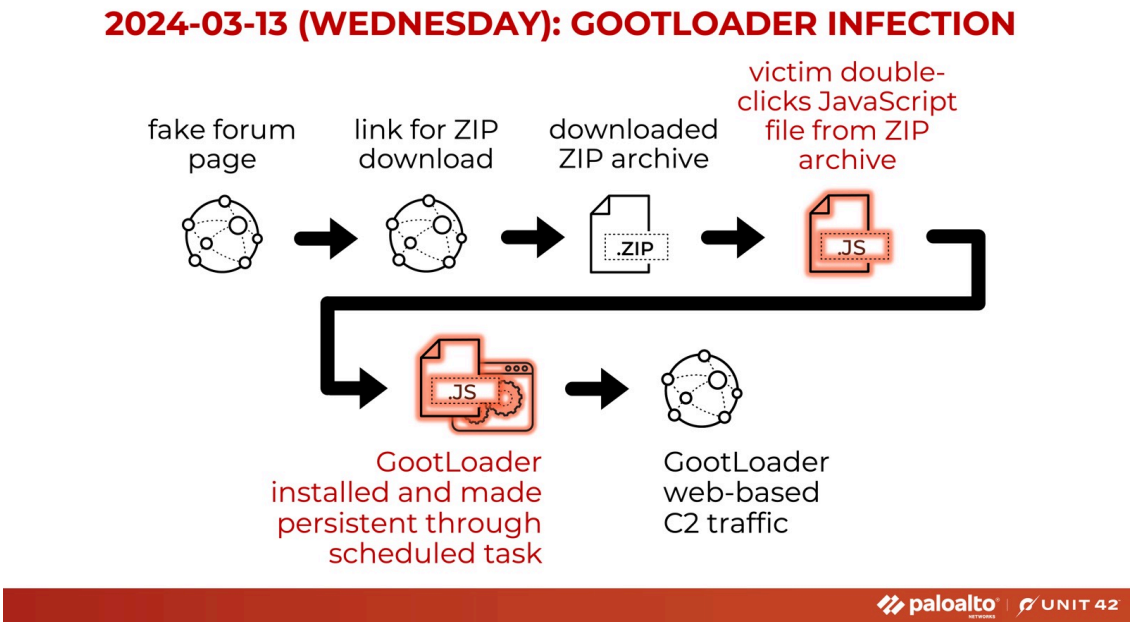

Figure 1. Flowchart for a GootLoader infection we saw in March 2024.

Sandboxing is a widely adopted method of identifying malicious binaries that involves analyzing the behavior of binaries within a controlled environment. Sandboxes encounter hurdles when analyzing a large volume of binaries with limited computing resources.

Malware often exploits these challenges by intentionally delaying malicious actions within the sandbox to conceal its true intent. These delaying actions are commonly described as the malware sleeping.

Common Ways for JavaScript Malware to Sleep

The most common way for malware to sleep is to simply call the methods Wscript.sleep() or setTimeout(). However, many sandboxes easily detect these methods. In the following paragraphs we dissect one of the least-mentioned methods GootLoader uses to evade detection.

Stepping Into the Code

In this section we leverage Node.js debugging in Visual Studio Code to analyze the following GootLoader file on a Windows host:

- SHA256 hash: c853d91501111a873a027bd3b9b4dab9dd940e89fcfec51efbb6f0db0ba6687b
- File size: 860,920 bytes
- File name: what cards are legal in goat format 35435.js
- First submitted to VirusTotal: Jan. 9, 2024

In our debugging endeavor for GootLoader files, we use a Windows host with Node.js JavaScript runtime and Visual Studio Code installed. In this environment, we can step through the code using Node.js debugging in the Visual Studio Code editor.

This environment offers an effective approach to comprehend the malware's flow control and execution logic. Typically, Windows Script Host (wscript.exe) runs standalone JavaScript files in a Windows environment. However, by employing Node.js and Visual Studio Code, we can step through the JavaScript file's execution, set breakpoints in the code and use the immediate window to evaluate expressions. While this approach offers significant advantages, certain JavaScript functions might not be supported by Node.js.

As an obfuscation technique, the authors of GootLoader have interwoven lines of GootLoader code among legitimate JavaScript library code. Throughout our debugging process, we observed the code execution that appeared to be seemingly stuck within the confines of a particular loop. Below, Figure 2 shows a snippet of code from one of these loops.

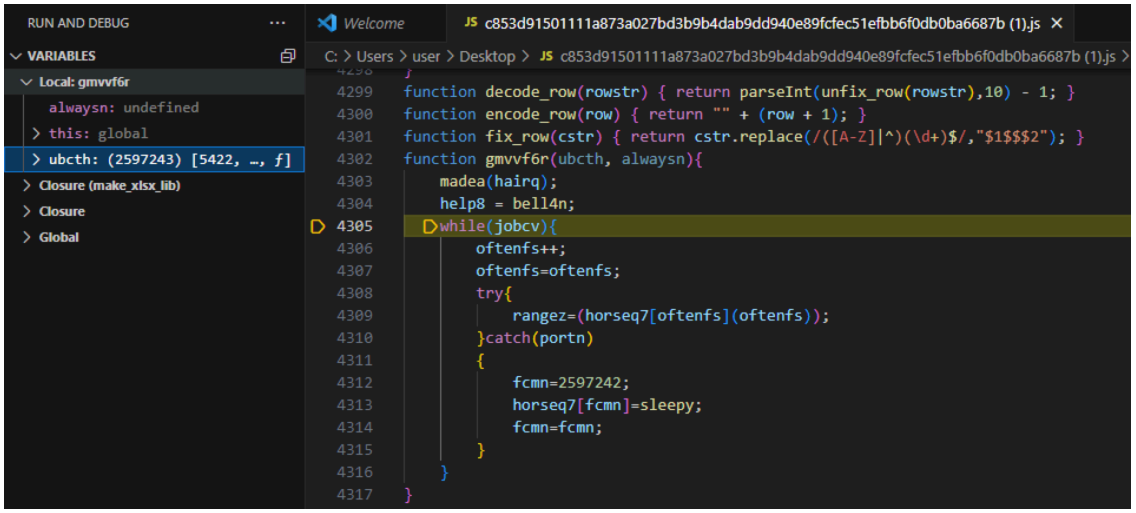

Figure 2. Code execution from a GootLoader sample that appeared to be stuck in a loop when analyzing the file using Node.js debugging in Visual Studio Code.

To gain a better understanding of these loops, let's delve into the surrounding code from the loop in Figure 2. Below, Figure 3 shows an isolated rendition of the original code that we will focus on.

```
oftenfs = 8242;
function gmvvf6r(ubcth, alwaysn){
    madea(hairq);
    help8 = bell4n;
    while(jobcv){
        oftenfs++;
        oftenfs=oftenfs;
        try{
            rangez=(horseq7[oftenfs](oftenfs));
        }catch(portn)
        {
            fcmn=2597242;
            horseq7[fcmn]=sleepy;
            fcmn=fcmn;
        }
    }
}
```

Figure 3. Code loop from Figure 2.

In Figure 3, the while function within the code causes an infinite loop, because the variable jobcv is consistently assigned the value 1. Additionally, the variable oftenfs acts as a counter, which has been initialized with the value 8242.

The pivotal line within this loop is rangez=(horseq7[oftenfs](oftenfs));. The successful execution of this line relies on the function array horsqe7 pointing to an actual function. The loop persists until the counter oftenfs reaches the value 2597242, at which the function array horsqe7 references the sleepy function.

This made the code appear to be stuck in a loop, because within our analysis environment, it took over 10 minutes for the counter oftenfs to attain the value 2597242.

Next, we stepped into the sleepy function. Inside the sleepy function, we observed a familiar function array name from Figure 3. This function array, horseq7, is assigned with a function named indicated6 as shown below in Figure 4.

```
3800  function sleepy(molecule9, cost6, kwuiem, tyvgoye03){
3801      were8 = decimalv+evend+homz+nabs+efgqvorm+dofhpam+dream1+
          atomc+wwamd+saidv+epyh+printy+ifyux+tireu+wall1+master8;
3802      horseq7[5210044] = indicate6;
3803      madea(khqrlh);
3804  }
```

Figure 4. Finding the horseq7 function array name inside the sleepy function.

After more delays, code execution will land inside the indicate6 function. This time the lclft4 function is assigned into the function array horseq7 as shown below in Figure 5.

```
3548  function indicate6(own8o, qobqfy, atnuskq){
3549      talll= faced8( bell4n(were8),map4);
3550      horseq7[6001779] = lclft4;
3551  }
```

Figure 5. Inside the indicate6 function.

Again with more delays, code execution will reach the course83 function shown below in Figure 6. The function course83 is where the actual malicious code begins execution.

```
4810  function course83(ncffnxn, pattern8, edjoaqn, lead4){
4811      talll[znqxcxu] (tallbell4l[pwnwi]) (horseq7);
4812      horseq7 = camet;
4813  }
```

Figure 6. Inside the course83 function.

Finally, debugging the course83 function unveils and deobfuscates JavaScript code that initiates GootLoader's malicious functions. Below, Figure 7 shows a section of the deobfuscated malicious GootLoader code.

```
_wscript = WScript;
wscript_shell = _wscript['CreateObject']("WScript.Shell");
scripting_filesystem_object = _wscript['CreateObject']("Scripting.FileSystemObject");
scheduler_service = _wscript['CreateObject']("Schedule.Service");
scheduler_service['connect']();
scheduler_folder = scheduler_service['GetFolder']("\\");
try{
    defensive_driving_task = scheduler_folder['GetTask'](Defensive_Driving);
}
catch(IhllTwx){
    defensive_driving_task = false;
}
if (defensive_driving_task == false) {
    BqdABYzF = scripting_filesystem_object['GetFolder'](wscript_shell['ExpandEnvironmentStrings']
    ('%APPDATA%'))['SubFolders']; WIyd = 396-(Math['floor'](396/BqdABYzF['Count'])*BqdABYzF['Count']);
    _counter = 0;
    malware_directory = false;
    for(_folder_enumerator = new Enumerator(BqdABYzF);
     !_folder_enumerator['atEnd']();
     _folder_enumerator['moveNext']()) {
        NJGHOUn = _folder_enumerator['item']();
        if (WIyd==_counter) malware_directory = NJGHOUn;
        _counter++;
```

Figure 7. Snippet of deobfuscated malicious GootLoader code.

The creators of GootLoader employed time-consuming while loops with arrays of functions to deliberately delay the execution of malicious code. This method effectively implements an evasion technique, inducing sleep periods to obfuscate the malicious nature of GootLoader.

Table 1 lists the counter values and their assigned functions in the order they were called from the GootLoader JavaScript code.

Counter Value	Function Name
2597242	sleepy
5210044	indicate6
6001779	lclft4
6690534	course83

Table 1. Counter values and their assigned functions from the GootLoader sample.

Conclusion

Leveraging our insights gained from analyzing the evasion technique used by GootLoader, we can enhance our ability to detect, analyze and develop effective countermeasures against malicious software. Through continuous collaboration and knowledge sharing, we can collectively stay ahead of cybercriminals to help safeguard our digital systems and networks.

Palo Alto Networks customers are better protected from GootLoader and similar threats through the following products:

- Next-Generation Firewall with Cloud-Delivered Security Services including Advanced WildFire detect the files mentioned within this report as malicious.

- The script described is prevented by Behavioral Threat Protection as part of Cortex XDR.

If you think you might have been compromised or have an urgent matter, get in touch with the Unit 42 Incident Response team or call:

- North America Toll-Free: 866.486.4842 (866.4.UNIT42)
- EMEA: +31.20.299.3130
- APAC: +65.6983.8730
- Japan: +81.50.1790.0200

Palo Alto Networks has shared these findings with our fellow Cyber Threat Alliance (CTA) members. CTA members use this intelligence to rapidly deploy protections to their customers and to systematically disrupt malicious cyber actors. Learn more about the Cyber Threat Alliance.

Indicators of Compromise

SHA256 Hashes of GootLoader JavaScript Files

- b939ec9447140804710f0ce2a7d33ec89f758ff8e7caab6ee38fe2446e3ac988
- c853d91501111a873a027bd3b9b4dab9dd940e89fcfec51efbb6f0db0ba6687b

---

Source: https://unit42.paloaltonetworks.com/javascript-malware-gootloader/