LOLSnif – Tracking Another Ursnif-Based Targeted Campaign

By Deutsche Telekom AG

Published: 2020-05-14 · Archived: 2026-04-05 20:31:25 UTC

Tool leaks are very interesting occurrences in cyber security. On one hand, they may burn the leaked tool since it got either publicly known or it facilitates the analysis of the tool significantly. On the other hand, it transfers capabilities to other, possibly lower-skilled actors. In my new blog post I analyze a newer version of Ursnif, which has been turned into an exploration tool and downloader. Because: In case of malicious software, source code leaks are very remarkable events. But first a small classification to get started. A notable example from the cyber security history was the leak of the exploit code “Eternal Blue” (CVE-2017-0144) in 2017. This very powerful remote exploit empowered many lower-skilled actors to conduct intrusions with ease.

Such leaks have occurred every now and then and they will most likely continue to occur in the years to come. The leak of the banking Trojan Zeus definitely shaped the malware landscape during the 2010s. Its source code was leaked in 2010. What followed was a plethora of offspring that share the same code base with the original Zeus banking Trojan. For instance, the zeusmuseum lists 28 families including Citadel, Pandabanker, and more recently Zloader. Many families continued to operate as banking Trojans like their ancestor but some of them adopted another purpose like the downloader Zloader.

Besides the Zeus source code leak, there are many more interesting leaks, each of them deserves its own story. In this blog article, I focus on another malware family whose source code got leaked in 2014. This family is called Ursnif (also known as Gozi2/ISFB). Ursnif is a fully-fledged banking Trojan. There has been many great technical as well as historical work on this family in the recent years. For instance, Maciej Kotowicz wrote a very good write-up on this family in 2016 and the podcast “Malicious Life” produced two episodes about the history of Gozi, the ancestor of Ursnif. Today, the source code of Ursnif can be easily found on the Internet. Many actors took this solid code base and based their work on it.

In this blog post, I analyze a recent Ursnif variant, which was repurposed to be a reconnaissance tool and downloader. It is a modern variant utilizing techniques like LOLBins, Component Object Model (COM) interfaces to instrument the Internet Explorer and as a consequence bypass local proxies as well as Windows Management Instrumentation (WMI) to interact with the operating system in place of native Win32 APIs. Campaigns associated with this variant start with a cascade of several layers of obfuscation, which ensures that the detection rate is very low, even several days after a campaign has started. This variant plays an important role in these campaigns which appear to be targeted cybercrime attacks.

It was first publicly mentioned in August 2019, which is in line with the registration dates of some associated domains and it gained momentum in autumn and winter 2019/2020. Recently, a blog article scrutinized one of its campaigns in April 2020, pointing out its use of LOLBins. In a nutshell, LOLBins stands for “Living Off the Land Binaries”, which means to utilize system tools that are already on a target system in order to carry out nefarious purposes. For instance, there are dozens of Windows system tools that can be utilized to download files, dump

credentials, or execute other binaries. As a consequence, attacks do not need to bring their own tools and may hide their traces in other legitimate uses of these tools.

To the best of my knowledge, it is not yet tracked under its own name. Since the Ursnif variant that I will look into in this blog article makes use of LOLBins and there is already a reference to this ("Ursnif via LOLbins"), I will refer to it as LOLSnif in the following.

Digging Deeper into A Recent Campaign

At first, I will look into a recent LOLSnif campaign started on 2020-04-07 in order to get an idea of the actors TTPs. Later, I will zoom out and review recent activity of other campaigns of the associated actor.

Each campaign begins with a cascade of several layers of obfuscation. It all starts with a spam email that contains an encrypted ZIP archive. The mail mentions the password for decryption and encourages the recipient to open it.

The next layer is a 1.34 MB sized, heavily obfuscated JavaScript file called "my_presentation_v8s.js" (4d98790aa67fb14f6bedef97f5f27ea8a60277dda9e2dcb8f1c0142d9619ef52). Its first submission to VirusTotal was on 2020-04-07 and its initial detection rate was very low. This JavaScript file ultimately drops a PE executable with file name "UtBuefba.txt" as well as a text file to "AppData\Local\Temp". The text file contains what appears to be an eight character long ASCII key that the PE executable requires to unpack properly. This might be a way to thwart analysis if only the PE executable is shared.

The DLL is first launched via "regsvr32.exe -s [redacted]\AppData\Local\Temp\\UtBuefba.txt". The option "-s" stands for silent registration, which does not show a dialog box. Even though DLLs to be loaded with regsvr32 are required to comprise the exports "DllRegisterServer" and "DllUnregisterServer", this DLL does not export them. It has just a regular DLL entry point. However, this file unpacks an APlib-compressed Ursnif DLL. This DLL (e3d89b564e57e6f1abba05830d93fa83004ceda1dbc32b3e5fb97f25426fbda2) is not known on VirusTotal as of 2020-04-17.

Before I continue with the analysis of the DLL, let me note that the techniques that the actor utilizes in their cascade of obfuscated layers ensure very low detection rates of their tools within the first days.

Dissecting LOLSnif

The DLL that I unpacked in the previous section is part of a recent fork of Ursnif, which I will refer to as LOLSnif in the following. LOLSnif comprises two main components: the loader and the worker component. The first drops the latter. Both are PE DLLs with slightly broken headers, where the DOS header magic number ("MZ") as well as the PE header magic number ("PE") are overwritten. While this may hinder some sandbox solutions to properly dump the payloads, this can be easily fixed. Notably both are x86 binaries. During my research I did not find any x64 binaries related to this malware.

The Loader

The unpacked DLL (e3d89b564e57e6f1abba05830d93fa83004ceda1dbc32b3e5fb97f25426fbda2) comprises two exports: the DLL entry point and DllRegisterserver that is required for DLL registration with regsvr32. It

resembles an Ursnif loader and uses the same configuration mechanism (JJ structure).

The following screenshots illustrates a JJ structure right beneath the section listing of the PE header. Each configuration section starts off with the magic number 0x4A4A ("JJ") and holds an offset to the data as well as a CRC32 tag to describe the data type (e.g. additional payload). A profound introduction into the configuration format of Ursnif can be found here.

JJ structure

The loader comprises two configuration sections. The first section is an "INSTALL_INI" that is required to install the payload of the second section. The second section contains an APlib-compressed DLL that is the actual worker. Before this second DLL is unpacked the loader checks whether the operating system has a Russian or Chinese localization. If so, then the loader silently stops its operation.

The Worker

LOLSnif's worker still resembles an original Ursnif worker. For instance, the general structure of the project is still preserved. This includes peculiarities like the configuration storage using JJ structures and the encrypted ".bss" section that holds strings and other data that deserves protection. Although there are some slight changes to the encryption algorithm. However, there are several interesting enhancements that I will detail later on.

Its configuration format is the same as its ancestor. The following screenshot shows a configuration blob that is highlighted with colors. Its first part compromises structures with offsets to strings and information on the data type. At the end of the blob (grey area) are these strings stored. For instance, the area highlighted in blue contains the command and control server addresses. This area is followed by a list of public DNS server. Notably this blob contains configuration parameters for a DGA (Domain Generation Algorithm), e.g. the domain where to download keywords for domain generation (constitution[.]org). However, LOLSnif does not comprise any functionality to generate domain names.

Configuration blob

As its ancestor Ursnif, LOLSnif comes with a encrypted ".bss" section that holds, for example, strings. This section is decrypted on startup. However, the decryption algorithm differs slightly from the original code base. The following screenshot depicts the decryption algorithm. The decryption key is still based on the sample's compilation date, which is hard-coded in the binary. However, there are some modifications to this key. Also, the original Ursnif utilized a rolling XOR algorithm, where LOLSnif's algorithm is based on add / sub instructions. The full list of decrypted strings can be found in Appendix B (pdf, 121.5 KB).

Decryption algorithm

Furthermore, some strings do not have references. Hence, they are dead and not needed. While this may be an issue with the disassembler, this is rather an issue with the code. The strings are defined in Ursnif's code as macros (see header file "Common/cschar.h"). Therefore, the authors of the malware must keep track of them. This may suggest that this functionality is not yet implemented, or the author forgot to clean up all unused functionality like in the DGA example.

The interaction with the registry is implemented in two ways: via native Win32 APIs (e.g. RegOpenKey, RegEnumKey, RegCloseKey) and via Windows Management Instrumentation (WMI). There are wrapper functions that take a Boolean argument in order to decide whether to use the Win32 API or WMI. For instance, the following screenshot shows a wrapper function for setting a string value of a registry key. The first argument (annotated as “use_wmi” in the disassembly) determines the use of native API or WMI.

Wrapper function

LOLSnif seems to be only built for x86. However, it clearly anticipates that it will run on x64 platforms as well. For instance, it utilizes the flag “__ProviderArchitecture” to request data from a 64 bit WMI provider as an 32 bit application.

Another interesting aspect is that LOLSnif makes extensive use of COM interfaces. It was reported that this malware instruments the Internet Explorer via IWebBrowser in order to contact its Command and Control servers (CC). This tactic allows it to bypass any proxy configuration in a cooperate network since Internet Explorer typically knows how to talk to the proxy. LOLSnif anticipates that there might be invalid certificate warnings since it searches in the DOM via IHTMLElement for the string “invalidcert”. If it encounters this string, then it confirms the warning to send the payload to the CC server. However, the current sample does not utilize https but rather plain http. Furthermore, the malware sets Internet Explorer as standard browser in order to avoid any popups during operation.

This Ursnif variant makes use of LOLBins (as also pointed out in this blog). For instance, LOLBins are utilized to start the malware from the registry (mshta.exe + powershell.exe). Furthermore, LOLSnif is capable of download and execute further modules as well as payloads. For instance, a recent blog post mentions that the associated actor dropped a Cobalt Strike BEACON as well as a legitimate TeamViewer VNC client.

Zooming Out: Tracking LOLSnif’s /api1 Activity Throughout the Last Months

Based on the campaign that I observed in April 2020, I pivoted to track LOLSnif activity throughout the last months. There are several links we can leverage to find more samples and domains. Appendix A (pdf, 93.0 KB) lists all IoCs that I found during my investigation. First, the string decryption algorithm of LOLSnif differs from the original code base. Therefore, I hunted for more samples based on this algorithm. This yielded several samples from which I was able to extract the configuration. The following table shows the unique configurations that I was able to find.

Hash (SHA256 prefix)	Botnet ID	Server ID	Serpent Key	Compilation Timestamp
8d700ea	1000	730	W7fx3j0lFvOxT2kF	Thu Oct 31 14:55:25 2019 UTC

c206f90	2000	730	W7fx3j0lFvOxT2kF	Wed Dec 4 16:56:24 2019 UTC
7307f15	3000	730	W7fx3j0lFvOxT2kF	Wed Feb 19 11:18:20 2020 UTC
f48e634	2000	730	U7yKaYwFde7YtppY	Wed Feb 19 11:18:24 2020 UTC
8ffe59d	3000	730	W7fx3j0lFvOxT2kF	Wed Apr 1 17:32:33 2020 UTC

All samples share the same server ID (730) and the same RSA key (not listed here). There are three botnet IDs (1000, 2000, 3000) that follow an increasing pattern with each campaign. Their compilation timestamps seem to be legit. All but the last share the domain ”wensa[.]at” or a subdomain of it. As we later will see, the actor utilized this domain for roughly six months. The serpent CBC key is “W7fx3j0lFvOxT2kF” in all cases but one. This outlier belongs to the botnet ID 2000, which has the key “U7yKaYwFde7YtppY”. Interestingly, there are two samples that were compiled within four seconds on 2020-02-19, which suggests that there is some form of build automation.

Another way is to pivot on the CC infrastructure. First, all samples use “/api1” as part of their URLs, e.g. “been.dianer[.]at/api1”. Second, all domains but one uses the “.at” TLD. There is one TOR “.onion” domain “6buzj3jmnvrak4lh.onion”. If we utilize passive DNS information regarding the already known domains / IPs, then we find further interesting information. There are many domains that the associated actor utilized for several months. The domain “wensa[.]at” was probably utilized for the longest time, roughly six months.

The following timeline shows the unique configurations and some of the domains. The samples are described by their SHA256 prefix and their serpent CBC key prefix, e.g. 8d700ea_W7. The domains have a first seen date (FS) and some of them a last seen date (LS). The domains lamanak[.]at and kamalak[.]at that the actor probably registered at the very beginning of their operation are still from time to time associated with IPs of their infrastructure. Furthermore, I suppose that the aforementioned wensa[.]at domain was replaced by the pipen[.]at domain in March 2020.

Timeline

The associated actor does either not care about OpSec or is very sure of their capabilities since they are reusing domains and IPs several times and they utilize them sometimes for very long periods of time.

Conclusion

In this blog, I've scrutinized a recent Ursnif variant to which I refer as LOLSnif and its recent activity. The associated actor possibly conducts targeted cybercrime operations that are still ongoing. At the heart of these campaigns is a variant of the Ursnif Trojan that was repurposed as a downloader and reconnaissance tool to meet the actor's special needs.

The techniques utilized by this actor such as LOLBins, heavy obfuscation, as well as COM interfaces and the TTPs observed by others in later stages of an attack such as utilization of post exploitation frameworks (Cobalt Strike) and VNC clients (TeamViewer) suggest that this actor is involved in targeted attacks. The attacks are ongoing since month without much public attention. This maybe the reason why the associated actor reuses domains and IPs over and over again.

Nevertheless, this actor should not be overlooked and maybe more data in the future (e.g. from Incident Response Engagements) helps to shed some light on the ultimate goal of this actor.

Appendix A: IoCs (pdf, 93.0 KB)

Appendix B: Full Dump of Decrypted Strings (pdf, 121.5 KB)

More information[Cybersecurity: TA505's Box of Chocolate]
[Cybersecurity: Dissecting Emotet - part one]
[Cybersecurity: Dissecting Emotet - part two]
