Exposing an AV-Disabling Driver Just in Time for Lunch

securityintelligence.com/exposing-av-disabling-drivers-just-in-time-for-lunch/

January 4, 2017

Home / Advanced Threats

Exposing an AV-Disabling Driver Just in Time for Lunch

Advanced Threats January 4, 2017
By Lior Keshet 8 min read
The IBM X-Force Security Research team detected a malicious AV-disabling driver while investigating new remote overlay malware attacking banks in Brazil. The AV-disabling driver is part of a financial malware designed to empty infected victims' bank accounts. What a way to start my morning.

Decoding an AV-Disabling Driver

It's 9:00 a.m. and I've yet to have my coffee. Where do I start?

When trying to understand what an unknown piece of software does, there are two main approaches: the dynamic approach, whereby we execute the code using tools such as sysinternals and a debugger to see what it does; and the static approach, whereby we examine the code using a disassembler or decompiler.

In this case, the malware is part of actual attacks, and time is of the essence. My goal is to understand exactly what this malicious driver does as quickly as possible and, ultimately, reduce mean time to detection and response. Since this particular driver doesn't contain a large amount of code, I decide to start with the static approach.

Disassembly

By now, two good things have already happened. First, I have my coffee in hand. Second, I have acquired an open instance of Interactive Disassembler (IDA) Pro to analyze a piece of code I've never seen before.

The first thing I notice is the scarcity of functions. With only a dozen to examine, maybe I'll have time for lunch today.

At this point, it's a good idea to look at the list of imported functions and strings for clues regarding the software's functionality. The strings don't look promising in this case:

The imports, however, are a different story:

Here I found the malware's string creation and conversion (ASCII to Unicode) functions and, more interestingly, registry-writing functions. Malware authors use various methods to hide the malware's list of imports, so we must account for the additional functions that may not appear in the screenshot above.

Since the code is so short, I decided to start at the beginning, the DriverEntry.

The first function call is compiler-generated and related to the security cookie. This isn't interesting, so I skip forward:

We can see in the figure above that this function calls sub_4011EA several times, each time with a different parameter between 1 and 5.

Function Overview

Let's take a moment to inspect the call graph, starting at sub_4011EA:

We see that sub_401160 uses the registry-writing functions. Let's check what exactly is being written. While we're at it, let's rename that function to something more meaningful: Write_To_Registry.

Since assembly code tends to be longer and harder to grasp, I'll use IDA's Hex-Rays plugin, which translates the assembly to a C syntax code. This will significantly decrease the amount of code that is shown. Keep in mind, however, that the generated code isn't always accurate.

As shown above, Hex-Rays did a near-perfect job translating the code. Now we can easily understand what's going on, with a little help from Microsoft's Developer Network (MSDN) documentation. It seems the AV-disabling driver tries to access a specific registry key, which it receives as a parameter. If that key exists, the AV-disabling driver tries to write it into a value, which IDA identifies as ValueName. The data to be written, as we can see in the previous image, is "4."

But why only a near-perfect job? Note that the parameter the function receives is shown as an object type of HANDLE, but it is cast to a Unicode string at the very first line of code. To an experienced researcher, this doesn't make any sense. From a quick look at the ObjectAttributes documentation from MSDN, we find that ObjectName indeed points to a string. On the other hand, a HANDLE is a numerical value and casting it into a string, once again, doesn't make sense.

So what's going on here? It looks like IDA misdetected the types.

Advancing Backwards

At this point, there are two pivotal questions:

1. Where does the ObjectName parameter come from?
2. Where does the ValueName come from?

To answer the first question, we'll take a step back and look at the function that calls the function we just examined, which I renamed Write_To_Registry:

The parameter we're interested in from the above screenshot, UnicodeString, originated from sub_4010EA. This is also the function that determines our ValueName from the previous figure. Now we can answer the above questions by examining a single function. This is great news because I'm getting hungry.

The function sub_4010EA gets two parameters: a1 and a2. a2 is a pointer to a string, and based on the calls to this function that we've seen previously, we can tell it's actually an output parameter.

The first part of the function is a loop that runs 0x1a7 times and fills a buffer with values taken from a different buffer after having each byte XORed by 0x8. This is simple decryption. Byte_403357 is used as a flag that signals whether the decryption already happened, verifying the XOR operation doesn't happen more than once.

A Binary Blob

I extracted the binary buffer and decrypted it. This is the result:

The code seems to contain two parts. The first, in blue, looks like binary data. The second, in red, starts in the last line of the first part and looks like encrypted data.

Another look at sub_4010EA reveals that the buffer we see here is passed to sub_401000, and so is the parameter a1. The result of sub_401000 will be returned to sub_4010EA within v3, which will then be passed to the output parameter a2, which, fortunately, is the value we're interested in.

At this point, I am tempted to change my approach. I can execute the driver within a virtual machine (VM), connect with a kernel debugger and examine the return values of sub_401000. I'm not sure which approach is quicker, so I decide to stick with the static approach, at least for now.

Sub_401000

The next function, sub_401000, is a little more complicated. First, let's look at how it's being called:

It receives a1 from before, the XORed buffer and a constant — in this case, 1.

The marked code above shows a search for the a2 parameter within a loop, which advances by seven with each iteration. At this point, I don't want to get into the parsing process of the binary data since it may consume too much time. So what's next?

Inside the first loop, v9 marks where in the buffer we found the parameter. At the end of this function, we see calls to two additional functions, the second of which receives data from the XORed buffer at an offset that depends on the value of v9.

An Educated Guess

I'm guessing the two function calls, sub_4010EA and sub_401000, are related to encryption or decryption.

Let's look at the first one:

The top, marked part is an initialization of an array of length 256 (0x100), when each member of the array gets a value equal to its index. In other words, a[i] = i. Given this reality and the fact that the decryption consists of two functions, I strongly suspect the calls are related to RC4 encryption.

Another look at the clock tells me it's 11:30. If my intuition is wrong, this research is going to take a while longer. Again, I consider switching to the dynamic approach, but I decide to give the static method one last shot. I'll assume the calls do, in fact, indicate RC4 encryption and re-examine sub_401000 to understand what exactly is being decrypted and with what key.

I changed the names of the variables in this function to fit the RC4 theory:

The key looks to be 8 bytes long. I follow the variable to where it's stored and find the part where the key is generated:

The code iterates the first 8 bytes of the XORed buff and derives the RC4 key by XORing each byte with 0x8. Note that the key length here is 8, which fits what we just saw.

We are left to determine what is being decrypted. Again, it will take a while to answer this question accurately. What we do know, or can at least guess, is that some parts of the XORed_buff are being decrypted using RC4 with a key that we already have.

One Last Shortcut

To save time, I decided not to check what exactly is being decrypted. Instead, I would simply try to decrypt everything. RC4 is a stream cipher, so the decryption of each character depends on the cipher's current state. To decrypt everything, I'll have to first decrypt the buffer starting at each character up to 0x1A7 options.

The code looks something like this:

The outer loop starts at the end of the key, which is located at the start of the XORed buffer, and iterates the rest of the buffer. The internal loop performs RC4 decryption.

As I expected, the result contains a lot of garbage data, but it also contains plenty of meaningful stuff:

There are, in fact, five such strings; each one corresponds to a different antivirus (AV) solution. The path within the string represents the location of the AV's driver service.

One Step Backward, Two Steps Forward

Let's go back and look at the first function, overwrite_AV_reg_service, which is called five times with parameters of 1–5:

The function gets an argument, decrypts the corresponding string from the encrypted buffer using the get_string function and writes data into the decrypted registry path using set_registry_value.

Piecing the Puzzle

Now that we know what the driver's different functions do, we can look at the function call graph once again:

The first function, main_func, is called five times with parameters of 1–5. For each parameter, get_string decrypts a different AV-related path. Then, set_registry_if_key_exists overwrites the registry path of that AV's driver and prevents it from being loaded into the system.

Disable AV, Reload Without Resistance

We also noticed that the malware using this driver causes the system to reboot after installing the driver. This causes the targeted AV software not to be loaded after the system restores, enabling the malware to execute without disturbance.

The driver performs this action because the user-mode code can't overwrite AV registry data; it employs self-protections to prevent exactly that. However, when executed by a driver, which can carry out more actions on a deeper privilege level, it is much harder to prevent such actions.

And that's it. We now know what the driver does, and I even have a few minutes to spare for lunch!

Final Disclaimers

When deciding between the dynamic and static approaches, I chose the latter method because it allowed me to understand how everything works internally. By dynamically obtaining the strings, I could learn what the driver does but not exactly how it does it.

I conducted most of my research without using Hex-Rays due to the plugin's inherent inaccuracies. I showed code mostly using Hex-Rays, however, for the sake of approachability.

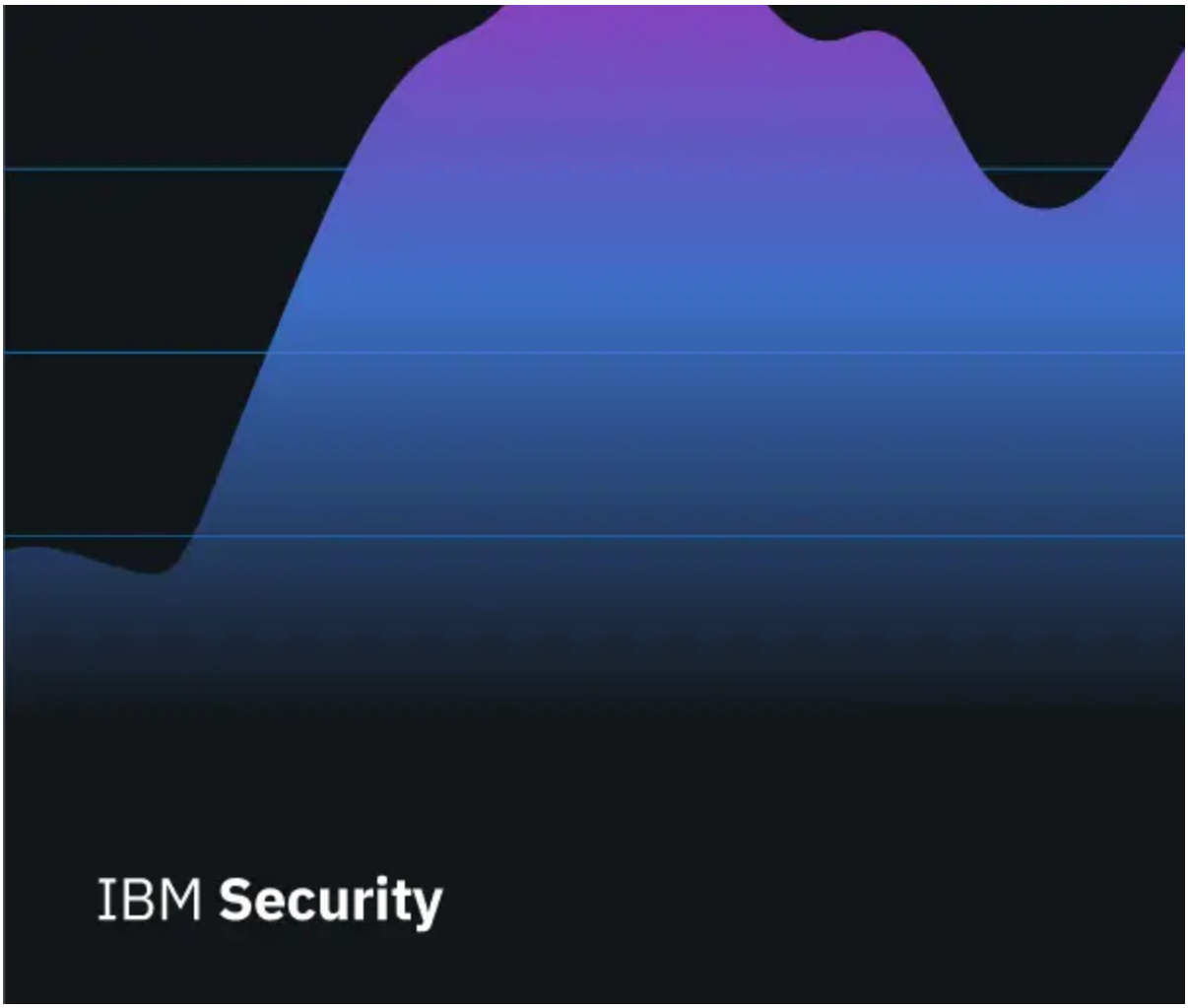
Finally, in truth, I am allowed to eat my lunch even if I do not complete a given task in the morning — and I don't always show up for work at 9:00 a.m. sharp.

Driver MD5: 48b872f91f1ff3f96594bf480ebf3dcc

Read the white paper: How to outsmart Fraudsters with Cognitive Fraud Detection

Lior Keshet
Malware Research Technical Lead, IBM Trusteer

Lior is a malware research technical lead at IBM Security's Trusteer's group. He has been a core member of the Trusteer cybercrime labs for the past four yea...


IBM Security