Overlay RAT Malware Uses AutoIt Scripting to Bypass Antivirus Detection

securityintelligence.com/overlay-rat-malware-uses-autoit-scripting-to-bypass-antivirus-detection/

November 8, 2017

Overlay RAT Malware Uses AutoIt Scripting to Bypass Antivirus Detection

Malware November 8, 2017

By Gadi Ostrovsky co-authored by Limor Kessem 6 min read

IBM X-Force Research follows the cybercrime arena across the globe to map the trends that shape online crime in each region. Brazil is a thriving region for financial malware, where malicious developers create various malware types to target local users with identity theft and online banking fraud.

In the past year, we have observed the rise of malware, such as Client Maximus and similar codes, that uses remote access with overlay screens for bank fraud operations in Brazil. Recently, we detected a remote access Trojan (RAT) malware that uses the same overall technique, but with an added twist to its antivirus evasion method.

Read the white paper: Shifting the balance of power with cognitive fraud prevention

A Delphi-Based RAT Pulling AutoIt Tricks

Malware developers that target Brazilian banks are often concerned with evading antivirus (AV) software. To evade detection, they commonly attempt to disable the running AV or find another workaround to avoid it.

X-Force Research recently observed an overlay RAT malware using the AutoIt framework to bypass AV detection in attacks against Brazilian bank targets. The AutoIt framework is an open source tool. It’s a BASIC-like scripting language designed to automate functions in the Windows user interface as well as general scripting tasks. AutoIt runs on all versions of Windows.

Within this context, the malware's developer uses AutoIt to prevent static AV detection from recognizing the malware's hash signature. To accomplish that, the malware's developer compiled the malicious code with an AutoIt script and runs it as a valid AutoIt framework process where the malicious payload is loaded into an AutoIt process memory address space.

Figure 1: Valid AutoIt process running malicious AutoIt code, viewed in ProcessExplorer

Figure 2: A hex view of the compiled AutoIt code

After its decompilation, the script executes a certain flow of events:

1. The script loads a dynamic link library (DLL) from disk and decrypts it with a hardcoded key implemented by the DESDecryptFile function.
2. It then loads the DLL into memory using the dllFromMemory function.
3. Finally, the script calls the DLL's entry point via a DLLCALL function

Figure 3: The script's execution flow

The Decryption Process

The decryption of the DLL runs first. The encryption selected by the malware's developer is based on the Advanced Encryption Standard algorithm in Cipher Block Chaining mode (AES-CBC). It was implemented in AutoIt for use by this RAT malware.

Figure 4: The decryption function implements the AES-CBC algorithm (Source: IBM X-Force)

Loading the DLL Into Memory

The DLL loading process happens in a few stages. The malware will use each stage to allocate memory space, protect it and then relocate it.

1. Loading the Library

To begin, the malware loads the DLL in an operation that it executes by calling the LoadLibraryEx application program interface (API) with DON'T_RESOLVE_DLL_REFERENCES, indicating that the Windows kernel should not load DLL dependencies.

During this process, the operating system loads the DLL into memory and adjusts code sections without resolving imports. This simplifies the operation of copying sections later.

2. Memory Allocation

To allocate memory space, the malware calls the VirtualAlloc API with a PAGE_READWRITE flag to enable read-only or read/write access to the committed region of pages. Additionally, memory allocation flags MEM_RESERVE and MEM_COMMIT are required to reserve virtual memory pages.

3. Copy Sections

Once memory has been reserved, the file's contents can be copied to the system.

4. Protect Memory

At this point, the code section tasked with running the code must be allowed to execute it. To do this, the malware's developer calls the VirtualProtect API with PAGE_EXECUTE_READWRITE rights. This API changes the protection on a region of committed pages in the virtual address space of the calling process, which means it can control whether an application is allowed to access the memory. In this case, the malware aims to hide its malicious code.

5. Base Relocation

Since all memory addresses in the code/data sections of a library are stored in locations relative to the base address defined by *ImageBase* in the *OptionalHeader*, a conflict arises if the library can't be imported to a given memory address. This means the references must all be adjusted or relocated. To modify the list accordingly, this malware uses the *FixReloc* function, followed by some defining parameters.

Figure 5: Using the FixReloc function to relocate addresses in memory

6. Resolving the Imports Table

The malware's file further contains a list of functions it will need to import for each DLL. Because function addresses are not static, new location values are needed for each address. These can be deduced by loading all the DLLs that were not yet loaded into memory and mapping them into the process address space.

To fix the import address table, the malware uses the FixImports function, as shows in the image below.

Figure 6: Using the FixImports function to calibrate the import address table

7. Notifying the Library

To finalize loading the malware’s DLL into the virtual address space, the developer calls DllMain as an entry point.

8. Freeing Up the Library and Wiping Traces

At the end of this routine, the malware releases the DLL that was loaded to memory during the first stage, calling the FreeLibrary API. This frees the loaded DLL module and, if necessary, decrements its reference count, a step taken by the malware’s developer to wipe traces of the malicious code.

To hide its internal strings, the malware uses a primitive asymmetric algorithm and a hardcoded key to decrypt it, which makes it easier for researchers to break once that key is located.

Figure 7: Implementation of the decryption algorithm as viewed in the IDA-Pro tool

The RAT code itself is written in Delphi, which is a programming language that has become synonymous with malicious codes written in Brazil. It is packed with an Ultimate Packer for Executables (UPX) obfuscator.

Overlay RAT Malware: Still the Preferred Attack in Brazil

After running the malware in our labs, we recreated the attack flow and discovered that it is yet another remote overlay malware designed to target online banking users in Brazil. X-Force Research does not see many classic banking Trojans operating in Brazil. If any such Trojans are operating in the region, they are entirely Zeus-based with a local twist, such as Zeus Panda. It’s worth noting that Zeus Panda has not been active in Brazil since it was discovered in August 2016. Most cybercriminals attacking Brazilian banking customers stick with RATs. As long as those types of attacks continue to serve them, threat actors are unlikely to see a need for change.

Overlay RAT malware has a typical flow of events on user endpoints. After its deployment, it monitors the user’s browser activity by checking the browser window’s title for bank names. If a targeted tab is found, the malware launches two elements:

- A full-screen image or webpage to block the victim from the real bank’s page; and
- A RAT to take control of the victim’s endpoint and the banking session he or she may have already authenticated.

The malware’s operator remotely initiates a fraudulent transaction from the victim’s endpoint and may prompt the user to provide additional details by using the fake overlay screen.

This is only the latest of many similar codes that have long plagued Brazilian users. Like others of its kind, such as Malfies and Dybuk, one of this RAT's goals is to evade antivirus detection and go unnoticed until the user is under its operator's control.

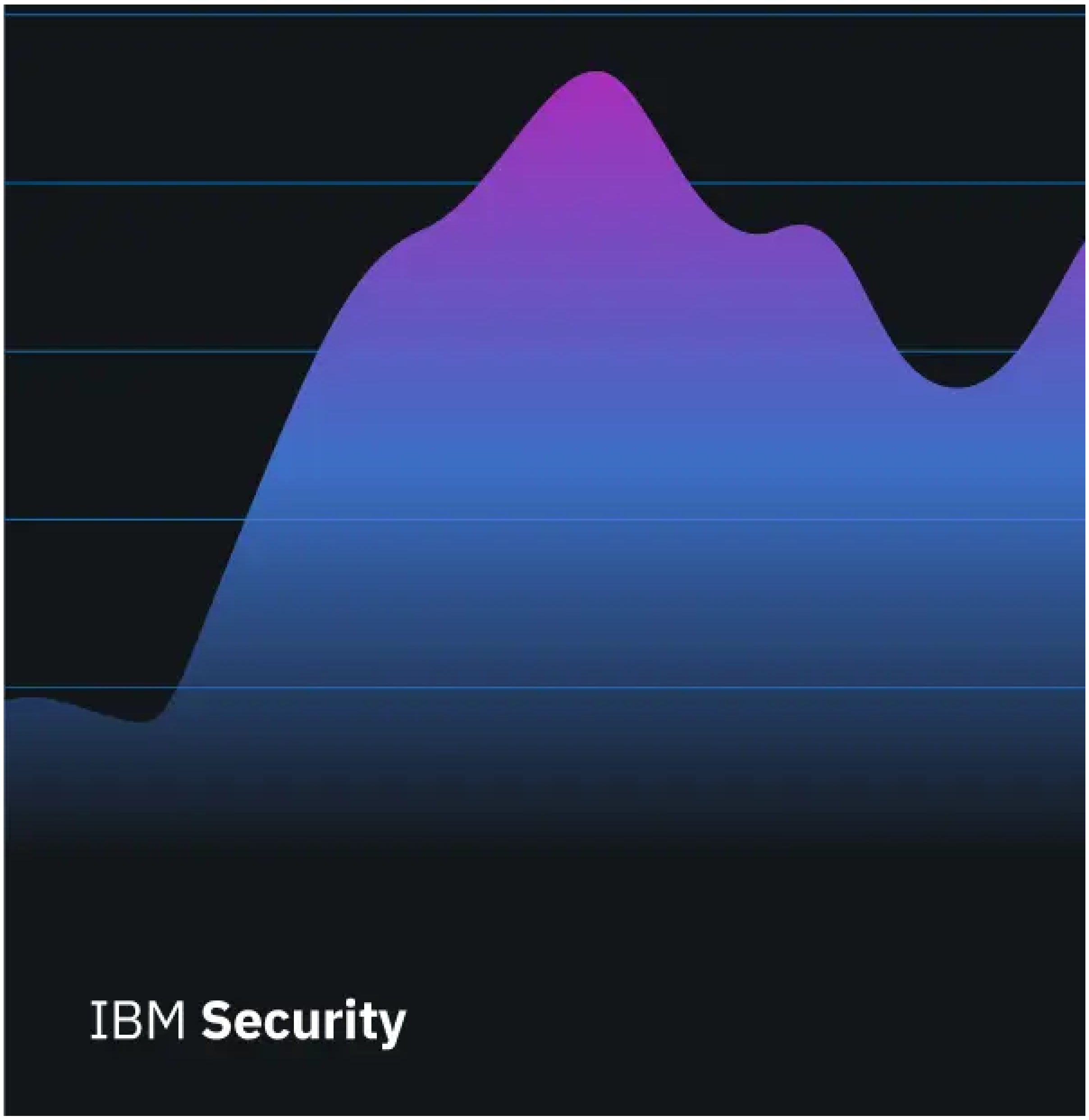
IBM X-Force analyzed the following samples for this research.

- MD5: EFB42A37102E377F030E98ADA65420EF (RAT)
- MD5: 8CF2318A99C3157E11645226F1626235 (AutoIt compiled script)
- MD5: B06E67F9767E5023892D9698703AD098 (AutoIt framework)

Read the white paper: Shifting the balance of power with cognitive fraud prevention

Gadi Ostrovsky
Malware Researcher, IBM

Gadi Ostrovsky is one of the top security researchers at IBM Security's Trusteer group. He joined the company in 2014, coming from a deep technical backgro...


IBM Security