“Hey ESET, Wait for the Leak”: Dissecting the “OctoberSeventh” Wiper targeting ESET customers in Israel – Emanuele De Lucia

By edelucia

Published: 2024-10-19 · Archived: 2026-04-05 18:25:31 UTC

On October 2024, attackers targeted Israeli organizations by exploiting a trusted source: ESET’s local partner, **Comsecure**. Apparently they compromised **Comsecure**’s infrastructure and used it to send phishing emails disguised as official communications from ESET.

These emails contained a malicious download link purported to be a legitimate tool but actually housed wiper malware (I internally named **OctoberSeventh**) designed to disrupt data from victim systems.

What makes this attack particularly interesting is the exploitation of an established partner of a globally trusted cybersecurity firm, leveraging the inherent trust that customers place in such relationships. By embedding wiper malware in files that appeared coming from ESET, the attackers succeeded in distributing a destructive payload under the guise of a routine security update.

eset Digital Security Progress. Protected.

Government-backed attackers may be trying to compromise your device!

Dear Customer,

Your device has been identified among a list of devices currently being targeted by a state-backed threat actor. Information attained by ESET's Threat Intelligence Division has identified a geopolitically motivated threat group as having attempted to target your machine within 14 days of this email.

As part of ESET's Advanced Threat Defense program (ESET-ATD), ESET is providing you access to the ESET Unleashed program, *designed to counter advanced targeted threats*, for you to install on up to 5 devices of yours.

Your personal download link is:

https://backend.store.eset.co.il/pub/2eb524d79ce77d5857abe1fe4399a58d/ESETUnleashed_081024.zip

We request that you do not share this link with any one else.

Regards,

ESET ATD Team

Phishing email sample

The motivation behind this attack appears to be geopolitically driven, given its exclusive focus on Israeli entities during a time of heightened regional tension. In my opinion it illustrates the evolving sophistication of cyber threat actors who not only rely on advanced malware but also on sophisticated social engineering and supply chain infiltration techniques.

INSIGHTS

I tried to have a quick look to the code; the malware employs several self-defense mechanisms designed to thwart analysis attempts. Their core purpose is to identify if the malware is being executed within a debugger and, if so, to trigger a controlled crash, effectively terminating the analysis session and hiding its malicious behavior.

It first probes the CPU's capabilities, using **IsProcessorFeaturePresent** to determine if the **__fastfail** instruction is supported. This instruction, a low-level mechanism for triggering fast fail exceptions, provides a rapid method for terminating a program. If the CPU supports **__fastfail**, the malware immediately utilizes it, causing an abrupt halt to its execution and hindering any debugger's ability to follow the code's flow.

If the CPU does not support **__fastfail**, the malware resorts to a more direct approach, explicitly checking for a debugger's presence using the ***IsDebuggerPresent*** function. This Windows API function returns a clear indication of whether a debugger is attached to the malware's process. Should a debugger be detected, the malware proceeds to construct a custom exception.

This custom exception is crafted to mimic a legitimate program crash while obscuring the true cause of the failure. The malware first initializes a large buffer, **v18**, filling it with **null bytes**. It then writes specific values to this buffer, mimicking the structure of a **CONTEXT** record, which holds processor register data and thread context information typically captured during exceptions. The malware populates this buffer with values from various registers, including general-purpose registers (EAX, EBX, EDI, ESI, etc.), segment registers (CS, DS, ES, etc.), the stack pointer (ESP), the instruction pointer (EIP), and the flags register (EFLAGS).

To trigger the exception, the malware calls ***RaiseException***, using the exception code ***0xE06D7363***. This code, unlikely to be handled by any default exception handlers, ensures that the exception will be caught by the malware's top-level exception filter. This custom filter, designed to prevent interference from debugging tools, is likely programmed to terminate the malware's process upon encountering this specific exception, effectively halting further analysis within the debugger.

Following the **sub_4025E0** initializes a security cookie (*__security_cookie*) using a combination of system time, process/thread IDs, and performance counter, may be used for integrity checks later on.

The *cookie*, crafted from a combination of system time retrieved using ***GetSystemTimeAsFileTime***, process and thread IDs acquired via ***GetCurrentProcessId*** and ***GetCurrentThreadId***, and performance counter readings obtained using ***QueryPerformanceCounter***, acts as a fingerprint for the malware's code. By incorporating these system values, the security cookie becomes highly unique to each infection.

It begins by dynamically resolving the base address of the "KERNEL32.DLL" module. This is achieved by traversing the **Process Environment Block** (PEB), a system structure containing information about loaded modules, and locating the entry for "KERNEL32.DLL."

This approach avoids using the standard ***GetModuleHandle*** function, which could be easily hooked by security products, making the malware's actions more difficult to trace.

Further decrypted strings reveal resolution calls to **WinHttpOpen**, **WinHttpConnect**, **WinHttpOpenRequest**, **WinHttpSendRequest**, **WinHttpReceiveResponse** and **WinHttpCloseHandle**.

These are all standard **WinHTTP API** functions for making web requests. It then constructs an HTTP GET request to ***www[.]oref[.]org[.]il/alerts/RemainderConfig_eng.json*** using hardcoded user agent string ***Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/109.0.0.0***.

It then downloads the content of the JSON file and stores it in memory. If the download is successful it opens a file named "EFsoIAEBEwpwcDoBGAYBORFPXlECQ11eVA==" (decrypted to "***C:\Users\Public\conf.conf***") in write mode "w", copies itself under ***Users/Public*** and prepares the paths for file overwriting.

As seen, the malware utilizes a series of obfuscated strings, which are actually API function names encrypted using a simple **XOR** cipher with the hardcoded key "***Saturday, October 07, 2023, 6:29:00 AM.***" Decrypting the string "Hw4VET4NAwtNUjYi" reveals the **LoadLibraryA** function as well. See "Appendix" to get a script usefull to decrypt them.

STRINGS DECRYPTION ROUTINE

The string decryption routine within this wiper utilizes a custom **XOR** cipher implementation, augmented by a character mapping technique. The function **sub_401160** serves as the core decryption engine, accepting the encrypted string and a hardcoded key string ("**Saturday, October 07, 2023, 6:29:00 AM**") as input. The initial step involves Base64 decoding. The function determines the encoded data's length by traversing the input string until it encounters a non-Base64 character, using the lookup table **byte_41DE18** to identify characters outside the **Base64** alphabet.

A memory buffer, sized to hold the decoded bytes, is allocated. Decoding proceeds in blocks of four encoded characters, which are mapped to their numeric values via the **byte_41DE18** table. These values are then bit-shifted and combined to generate three bytes of decoded data, which are written to the buffer. This process iterates until the entire input string is decoded. Following Base64 decoding, the function allocates a second buffer to store the decrypted string. The size of this buffer is calculated based on the decoded data's length and the key string's length.

The **XOR** decryption operates on each byte of the decoded data. For each byte, using the decoded data's index and the key's length, the index of the corresponding key character is determined. This ensures cyclic repetition of the key string to match the decoded data's length. The core XOR operation is then applied between the current decoded byte and the **ASCII** value of the corresponding key character. This generates a single decrypted byte, which is written sequentially to the output buffer, forming the decrypted string character by character.

The function relies on the **byte_41DE18** lookup table to optimize character-to-numeric value conversion, eliminating the need for repeated calls to the ord function. This table-driven approach streamlines the **XOR** operation by providing readily available numeric values for each **ASCII** character. Upon completing the decryption, **sub_401160** null-terminates the output buffer, resulting in a valid C-style string containing the decrypted data. This string is then returned to the calling function for use within the malware.

CORE WIPER LOGIC

The function **sub_4016C0** serves as the operational core of the wiper malware, orchestrating the files manipulation and system disruption activities. Upon entering this function, the malware first initiates a short delay using the **Sleep** function, pausing for *1000 milliseconds* and enumerating drives using ***GetLogicalDrives***.

Following the malware calls **sub_4015B0**, which appears to lay as the groundwork for the wiper's operations. The malware proceeds to resolve the addresses of essential Windows API functions from "KERNEL32.DLL" and "User32.dll," utilizing decrypted strings to mask its intentions. These functions include ***CreateFileW*** (decrypted from **dword_D593C8**), **SetFileAttributesW** (decrypted from **dword_D593D0**), **CloseHandle** (decrypted from **dword_D59504**), **PostMessageW** (decrypted from **dword_D593E0**), and **SendMessageW** (decrypted from **dword_D593E4**).

After this, the malware enters its main operational loop supporting *multi-threading*. This loop forms the heart of the wiper's attack strategy, driving a continuous cycle of file destruction and system manipulation. Within each iteration of the loop, the malware calls ***CreateFileW***, creating handles to the targeted files and directories. With file handles secured it calls the **sub_408050**.

This function, designed for data writing, overwrites the contents of the targeted files with large blocks of data, effectively rendering them unusable. Significantly, this initialization involves filling the buffer with *null bytes* (ASCII value 0) and random bytes could be used to disrupt the targeted files.

A crucial observation pointing towards overwriting lies in the size parameter passed to **sub_408050**. This parameter appears to be directly related to the target file size, suggesting the wiper might be writing a stream of bytes equal to the file's size, completely filling it and effectively obliterating the original data.

To further hide its actions, the malware employs *SetFileAttributesW* to potentially modify the attributes of the affected files. The wiper's manipulation appears to extend beyond simple file overwriting as it also utilizes *MoveFileWithProgressW* (decrypted from **dword_D59510**) to potentially prevent an easy recovery.

Before returning, the malware displays the message "***Hey ESET, wait for the leak.. Doing business with the occupiers puts you in scope!***" by using **MessageBoxA**.

ATTRIBUTION

Diffucult to say; My personal speculations are directed towards **TA402** (https://malpedia.caad.fkie.fraunhofer.de/actor/ta402) but I have no definitive evidence. Obviously, these cannot be considered conclusive in attributing this attack 🙂 . TA402 is believed to be aligned with Palestinian espionage interests, primarily targeting intelligence collection efforts.

IoC

SHA256: 2abff990d33d99a0732ddbb3a39831c2c292f36955381d45cd8d40a816d9b47a

APPENDIX

Python script to decrypt internal malware strings:

```
 import base64

def decrypt_string(encrypted_string, key_string):
  """
  Author:
    Emanuele De Lucia

  What it does:
    Decrypts the given string using a XOR cipher with the logic of OctoberSeventh wiper.

  Args:
    encrypted_string: The string to decrypt.

    key_string: The key to use for decryption.
  Returns:
    The decrypted string.
  """
  key_len = len(key_string)
  decrypted_bytes = bytearray()


  decoded_bytes = base64.b64decode(encrypted_string)


  for i, byte in enumerate(decoded_bytes):
    decrypted_bytes.append(byte ^ ord(key_string[i % key_len]))


  return decrypted_bytes.decode()
```

```
encrypted_string = "AxMbFhcXEkoeZiYRBxs="
key_string = "Saturday, October 07, 2023, 6:29:00 AM"
decrypted_string = decrypt_string(encrypted_string, key_string)

print(f"Decrypted string: {decrypted_string}")
```

Source: https://www.emanueledelucia.net/hey-eset-wait-for-the-leak-dissecting-the-octoberseventh-wiper-targeting-eset-customers-in-israel/