WastedLocker's techniques point to a familiar heritage

news.sophos.com/en-us/2020/08/04/wastedlocker-techniques-point-to-a-familiar-heritage/

August 4, 2020

It's a lot easier to change a ransomware's appearance (or obfuscate its code) than to change its underlying goals or behavior. After all, ransomware must necessarily reveal its intent when it strikes. But there are behavioral traits that ransomware routinely exhibits that security software can use to decide whether the program is malicious. Some traits – such as the successive encryption of documents – are hard for attackers to change.

The author of the WastedLocker ransomware cleverly constructed a sequence of maneuvers meant to confuse and evade behavior based anti-ransomware solutions. We'll discuss some of those tricks below, but it's also worth mentioning that a code analysis we've performed on WastedLocker shows something else we didn't expect: some of the specific techniques WastedLocker ransomware employs to obfuscate its code and perform certain tasks mirror the subroutines we've seen previously used by another ransomware, Bitpaymer, and in the Dridex trojan – too closely to have been a coincidence, in our opinion.

Evasion takes center stage

Ransomware creators are acutely aware that network or endpoint security controls pose a fatal threat to any operation, so they've developed a fixation on building complex logic for detecting and subverting those controls. Modern ransomware spends an inordinate amount of time attempting to thwart security controls.

Most of the advancements we observe in ransomware development can be categorized as survival skills, so the malware remains undetected just long enough to encrypt the target's files. Survival demands that static and dynamic endpoint protection struggle to make a

determination about a file based on the appearance of its code, and that behavioral detection tools are thwarted in their efforts to determine the root cause of the malicious behavior.

Many malware families employ code obfuscation techniques, like runtime packers, as a way to thwart analysis, but a few have taken this a step further. Bitpaymer, for example, uses a unique method that calls Windows API functions using a hash of the function call, rather than the call itself. WastedLocker appears to have adopted this technique and that adds an additional layer of obfuscation by doing the entire thing in memory, where it's harder for a behavioral detection to catch it.

Over the years, ransomware file system behaviors have, largely, remained consistent. (This year's Ryuk and REvil attacks exhibit the same file system behaviors as CryptoLocker from 2013, for example.) Ransomware defenses based on behavior monitoring are typically tailored to detect this universal telltale activity. Now that things have changed a bit, the tactics we use to detect this behavior will have to change as well.

Memory tricks may thwart behavior monitoring

Before diving into the tricks, you need to know that ransomware defenses based on behavior monitoring typically implement a minifilter driver. Minifilter drivers are kernel drivers that attach to the file system stack. Minifilters filter I/O operations in order to keep an eye on everything that happens to files. For example, the well-known Process Monitor utility from Sysinternals uses a minifilter driver to create a real-time log of file system activity. Most anti-ransomware solutions use a similar approach to keep an eye on what happens to files.

WastedLocker uses a trick to make it harder for behavior based anti-ransomware solutions to keep track of what is going on: using memory-mapped I/O to encrypt a file. Although it is unnecessary for ransomware to access documents as a memory-mapped file (MMF), the method is more common nowadays, as Maze and Clop also employ the same tactic.

This technique allows the ransomware to transparently encrypt cached documents in memory, without causing additional disk I/O. For behavior monitoring, this may be a problem. Tools used to monitor disk writes may not notice that ransomware is accessing a cached document, because the data is served from memory instead of from disk.

But the kicker here is that WastedLocker is closing the file once it has mapped a file in memory. You'd think this would result in an error, but the trick actually works because the Windows Cache Manager also opens a handle to the file once a file is mapped into memory.

Cache Manager's lazy writer

The Cache Manager is a kernel component that sits between the file system and the Memory Manager. If a process accesses the mapped memory, the memory manager will issue a page fault and the Cache Manager will read the necessary data from disk into memory. The more

memory is accessed, the more pages will be read from the disk into memory via so-called paging I/O.

The Memory Manager also keeps any eye on memory that has been modified, so-called dirty pages. If a process encrypts the mapped memory, the memory manager knows which pages need to be written back to disk. This writing is done by the Cache Manager's Lazy Writer.

The Cache Manager implements a write-back cache with lazy write. This means that dirty pages are allowed to accumulate for a short time and are then flushed to disk all at once, reducing the overall number of disk I/O operations. This also means that the writing is not done in the context of the process but in the context of the system (PID 4). It's this aspect that can be troublesome for anti-ransomware solutions, as it becomes harder for an anti-ransomware tool to determine which process wrote to the file.

Complications

The WastedLocker ransomware closes its file handle right after it has mapped the file into memory as can be seen in the following screenshot:

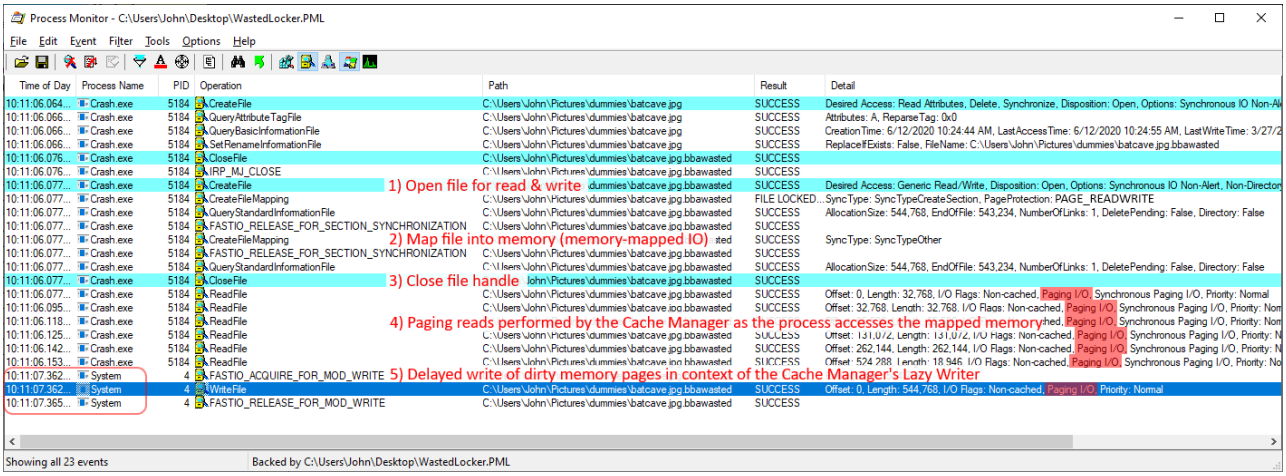
```
DWORD __userpurge OpenFileForEncryption@<eax>(DWORD *a1@<esi>, const WCHAR *lpFileName)
{
  DWORD v2; // edi@1
  HANDLE hFile; // eax@1
  DWORD fileSize; // eax@3
  bool v5; // zf@3
  LPVOID v6; // eax@6
  HANDLE hFileMappingObject; // [esp+4h] [ebp-8h]@2
  DWORD dwNumberOfBytesToMap; // [esp+8h] [ebp-4h]@1
  HANDLE hFile2; // [esp+14h] [ebp+8h]@1

  v2 = 0;
  dwNumberOfBytesToMap = 0x4000000;          1) Open file for read & write
  hFile = CreateFileW(lpFileName, 0xC0000000, 0, 0, 3u, FILE_ATTRIBUTE_NORMAL, 0);// GENERIC_READ | GENERIC_WRITE
  hFile2 = hFile;
  if ( hFile == INVALID_HANDLE_VALUE )
    return GetLastError();
  hFileMappingObject = CreateFileMappingW(hFile, 0, PAGE_READWRITE, 0, 0, 0);
  if ( hFileMappingObject )
  {
    memset(a1, 0, 0x28u);
    fileSize = GetFileSize(hFile2, a1 + 7);
    v5 = a1[7] == 0;
    a1[6] = fileSize;
    if ( v5 && fileSize < 0x4000000 )
      dwNumberOfBytesToMap = fileSize;       2) Map file into memory (memory-mapped IO)
    v6 = MapViewOfFile(hFileMappingObject, 6u, 0, 0, dwNumberOfBytesToMap);// FILE_MAP_WRITE | FILE_MAP_READ
    if ( v6 )
    {
      a1[8] = dwNumberOfBytesToMap;
      a1[1] = hFileMappingObject;
      a1[2] = v6;
      a1[9] = 6;
    }
    else
    {
      v2 = GetLastError();
      CloseHandle(hFileMappingObject);
    }
  }
  else
  {
    v2 = GetLastError();
  }
  CloseHandle(hFile2);                       3) Close file handle
  return v2;
}
0000635E OpenFileForEncryption:7
```

Closing the file handle right after the file has been mapped into memory is allowed as the documentation states:

> *Mapped views of a file mapping object maintain internal references to the object, and a file mapping object does not close until all references to it are released. Therefore, to fully close a file mapping object, an application must unmap all mapped views of the file mapping object by calling UnmapViewOfFile and close the file mapping object handle by calling CloseHandle.* ***These functions can be called in any order****.*

Anti-ransomware solutions that correlate activity based on CreateFile and CloseFile operations will miss all the disk I/O performed by the Cache Manager in response to mapped memory operations. This can be observed by the following screenshot:



Ultimately, the Cache Manager will release its internal handle to the memory-mapped file. This may happen after a few minutes, but we have observed that the Cache Manager closes the handle only after several hours.

(For more information on the Windows Cache Manager, refer to Windows Internals, Part 2.)

Code evolution from an unexpected source

Interestingly, an analysis of the WastedLocker code gave rise to a hypothesis that it may be an evolutionary descendent of Bitpaymer. Analysts familiar with both found noteworthy similarities (possibly even rewritten code) that seem to be more than a coincidence.

Abuse of Alternate Data Streams (ADS)

Both Bitpaymer and WastedLocker abuse ADS in the same way: The malware finds a clean system file, copies itself to the clean file's ADS, and then executes itself as a service component of the clean file. This makes it appear that the clean file is the source of the ransomware behavior. They both accomplish this using the same technique: They reset the privileges of the targeted system file using icacls.exe in order to add the ADS component, and then copy the clean system file to the %APPDATA% folder.

Customized API resolving method

Bitpaymer uses custom API resolve functions to call Windows APIs using a hash value, rather than the API function's name. The same code was also used by Dridex malware, and was consistently seen in many earlier Bitpaymer variants. With WastedLocker, the author did a major upgrade to the codebase by removing these functions. Instead, it calls the Windows API directly in memory.

The change it has improved the efficiency of the malware execution without spending much time in computing the hash and calling the API dynamically. Since this custom API Resolve function gets called in every single API call, the similar-behaving function code looked totally different during analysis.

UAC bypass

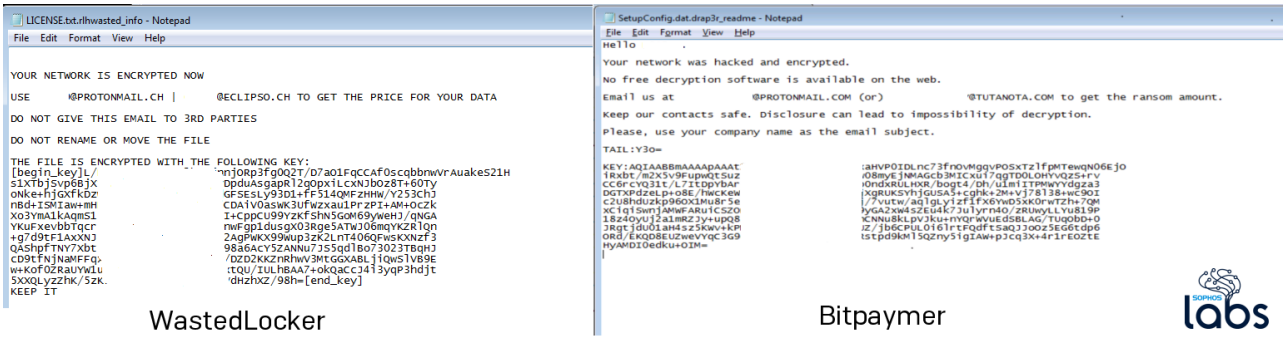
Both ransomware use a similar User Account Controls bypass technique to elevate the clean, hijacked process to run the ransomware code (using the ADS technique, above). Bitpaymer adds a .cmd file to the registry key ("HKCU\Software\Classes\mscfile\shell\open\command"), such that, when an elevated eventvwr.exe file is executed, it checks the registry key (by default) and that, in turn, executes the .cmd file that runs the ransomware binary. With WastedLocker, it uses winsat.exe and winmm.dll to run the ransomware binary (ADS component) by patching the winmm.dll.

Encryption methods

Bitpaymer has slowly over time improved the encryption method it uses. Initial variants of Bitpaymer use an RC4 key for encrypting the file content, and it further encrypts the RC4 key using a 1024-bit RSA public key. But later variants of Bitpaymer (as well as current versions of WastedLocker) made some improvements by using AES 256 bit CBC mode for encrypting the files, along with a 4096-bit RSA public key. Both these ransomware also encodes the key information with Base64, and stores the encoded key in the ransom note.

Ransom note composition

Both customize the ransom note for each of the victim by adding the organization name in the ransom note. WastedLocker also adds the organization name to the ransom note file name as a prefix.



Similar style of command line arguments

WastedLocker can perform certain operations when its main executable is launched using specific arguments, as did some earlier versions of BitPaymer. Both malware use numbers as arguments and the numbers they both use to indicate the operation the malware is supposed to perform are the same (eg., -1 indicates the main/initial execution, -2 issues a command to copy the malware and run it using ADS, and -3 indicates that it will begin the file encryption process.

While none of these alone, or even in combination, is enough to definitively say that, for instance, the same creator was responsible for both ransomware, the number of similarities is so striking as to raise questions about whether the malware author(s) of Bitpaymer and WastedLocker are connected in some collaborative way.

Reference IoC

Sample:
BCDAC1A2B67E2B47F8129814DCA3BCF7D55404757EB09F1C3103F57DA3153EC8