

Ransomware Analysis and Defense

WannaCry and the Win32 environment

Justin Jones

August 2, 2017

Contents

1	Introduction	1
2	Analysis	1
2.1	WannaCry/WCry	1
2.1.1	Background	1
2.1.2	General File Data	2
2.1.3	Decompilation	5
2.1.4	Dynamic Analysis	7
2.1.5	Conclusion	9
3	The Defense Software	9
3.1	An Overview	9
4	Appendix A	10

1 Introduction

A type of malware known as *ransomware* has recently become very prevalent in the cyber security world, taking over user systems and demanding compensation for the safe return of all functionality and data. While initially not incredibly sophisticated, this type of software has evolved from simple scripts that change file extensions and make empty threats to full-blown attacks affecting hundreds of thousands of systems worldwide that implement sophisticated NSA-developed exploits as their propagation vector. In this paper I will explore a specific piece of malware known as *WannaCry* that recently made headlines around the world, performing a full static and dynamic analysis. Additionally, I will endeavor to write a useable piece of software to stop malware from launching within a Win32 environment.

2 Analysis

2.1 WannaCry/WCry

2.1.1 Background

WannaCry (referring to the general family consisting of all named variations of WannaCrypt, WCry, WanaCrypt, WanaCrypt0r, etc) came into prevalence during a massive attack starting

on May 12, 2017. This software utilizes an exploit called EternalBlue¹, a known vulnerability in the Server Message Block (SMB) protocol used by Microsoft Windows which was previously patched in a critical update outlined in KB4013389². As this vulnerability has been explored and detailed very thoroughly already, focus will be shifted to WanaCry's implementation and software aspects while avoiding the inner workings of the exploit.

The working sample of WanaCry has been obtained from theZoo³, with SHA256 hash:

`ed01ebfbc9eb5bbea545af4d01bf5f1071661840480439c6e5babe8e080e41aa`

and is positively identified by VirusTotal as a member of the WanaCry family⁴. The software is being tested in a Microsoft-supplied 32-bit Windows 7 appliance as a VMWare Player virtual host. The appliance is given host-only network access and all outgoing traffic is recorded with Wireshark during analysis. All relevant analysis files are supplied in the GitHub repository accompanying the paper, including the Wireshark pcap, IDA Pro idb, registry snapshots and a compressed file containing the extracted payload.

2.1.2 General File Data

Utilizing PEiD⁵, it can be seen that the program was packed using Microsoft Visual Studio C++ 6.0 for Win32 and we should have no trouble unpacking it ourselves.

Utilizing Dependency Walker⁶, we see that the program uses *ADVAPI32.DLL* which is the source of many cryptographic security functions implemented in Windows. In fact, this is where the SMB exploit *EternalBlue* is found, however had I not known ahead of time the program uses it I wouldn't gather that information from this examination until I go through the decompilation and execution. As it is, most if not all ransomware will want to utilize this dll if it intends to seriously encrypt any files. The inclusion of this library is the first indication that this software may be ransomware had I not already known.

¹<http://bit.ly/2spdT15>

²<https://support.microsoft.com/en-us/help/4013389/title>

³<http://thezoo.morirt.com/>

⁴<http://bit.ly/2s93pCl>

⁵<https://www.aldeid.com/wiki/PEiD>

⁶<http://www.dependencywalker.com/>

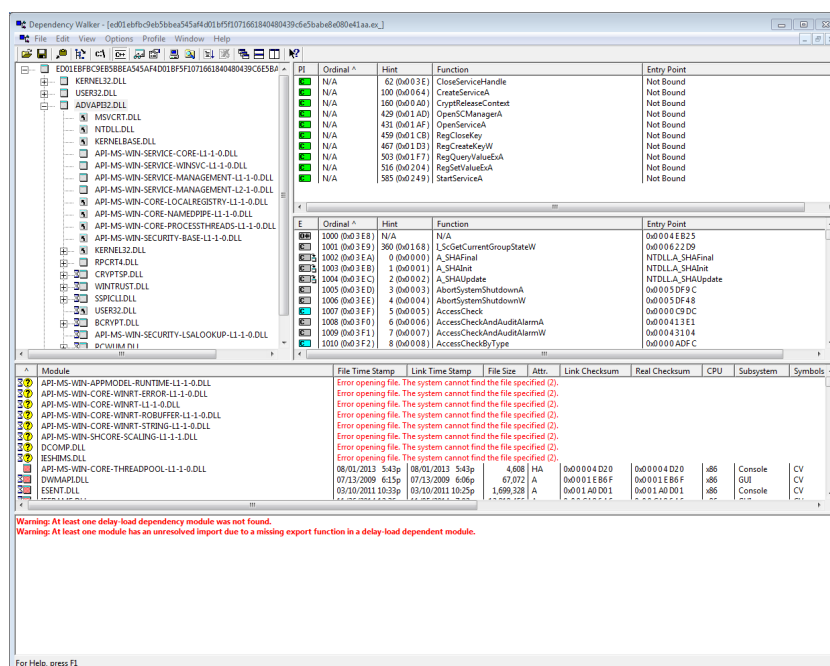


Figure 1: Dependency Walker overview of WannaCry

Looking into the functionality imported from this library, we see *CRYPT32.DLL*, and the imports immediately indicate this program is performing a large amount of cryptographic functions:

CryptDecrypt	CryptDeriveKey	CryptDestroyHash	CryptDestroyKey	CryptDuplicateHash
CryptDuplicateKey	CryptEncrypt	CryptEnumProviderTypesA	CryptEnumProviderTypesW	CryptEnumProvidersA
CryptEnumProvidersW	CryptExportKey	CryptGenKey	CryptGenRandom	CryptGetDefaultProviderA
CryptGetDefaultProviderW	CryptGetHashParam	CryptCreateHash	CryptGetProvParam	CryptGetUserKey
CryptHashData	CryptHashSessionKey	CryptImportKey	CryptReleaseContext	CryptSetHashParam
CryptSetKeyParam	CryptSetProvParam	CryptSetProviderA	CryptSetProviderExA	CryptSetProviderExW
CryptSetProviderW	CryptSignHashA	CryptSignHashW	CryptVerifySignatureA	CryptVerifySignatureW
CryptContextAddRef	CryptAcquireContextW	CryptGetKeyParam	CryptAcquireContextA	

We can also see that it uses the kernel library and the user library, specifically utilizing *GDI32.DLL* which gives access to local registry functions *RegCloseKey*, *RegEnumValueW*, and *RegOpenKeyExW* and is used for displaying graphics and creating GUIs. We also see *MSVCRT.DLL*, again indicating it was packed with Microsoft Visual Studio.

Utilizing PEvent⁷ to examine the PE header, we see a compile time of 11/20/2010. As this is a very recent exploit, this is not likely the true compile time for obvious reasons. More likely, the time was either faked or the system clock of the compiling machine was not accurately set. The subsystem indicated is *IMAGE_SUBSYSTEM_WINDOWS_GUI* which offers further evidence that the program utilizes a graphical interface. The virtual size and raw data size are about the same, indicating a more sophisticated packer was likely not used and the program was generated by a compiler.

⁷<http://wjradburn.com/software/>

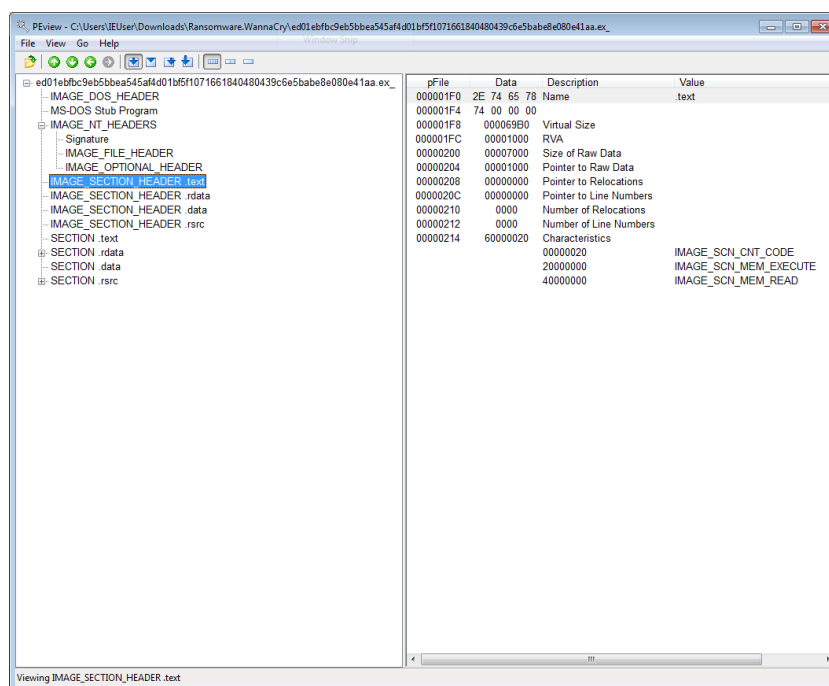


Figure 2: PE headers for WannaCry

Examination with Resource Hacker⁸ indicates that the software is trying to masquerade as diskpart.exe, and offers no useful information otherwise because the WannaCry files are encrypted in the "WNcry@2017" archive and are password protected, which we will see in the IDA Pro analysis.

⁸<http://angusj.com/resourcehacker/>

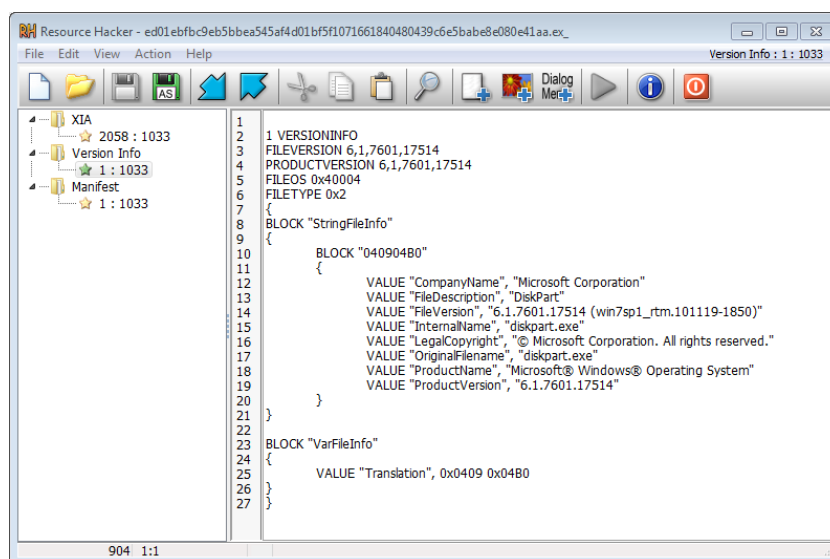


Figure 3: Resource Hacker & WannaCry

2.1.3 Decompile

Utilizing the IDA call flow graph, it can be seen that in addition to the basic startup functionality the program also calls `_WinMain@16`, from which the rest of the function calls will cascade from as this is where we unpack the archive. The software utilizes try/catch/finally blocks and extensive memory manipulation to accomplish its task, however in many cases I found that the creator(s) seemingly failed to turn on optimization in their compiler as shown in Fig 4.

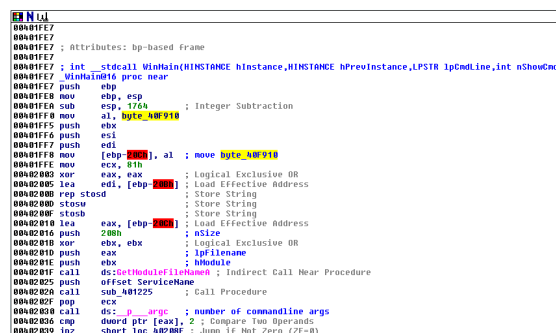


Figure 4: Poor Optimization

Additionally, after examining the try/catch/finally blocks, I believe that if any files were missing from the archive or otherwise corrupted upon extraction, the program would likely fail to fully take over the system and not execute. Before moving on through the code, a quick look at the Strings window confirms this is some variant of WannaCry. We can also see the filenames included in the archive, as the software has the names hard-coded for opening and manipulation, in Fig 5. In some cases, the functionality of these files isn't fully uncovered until the dynamic analysis.

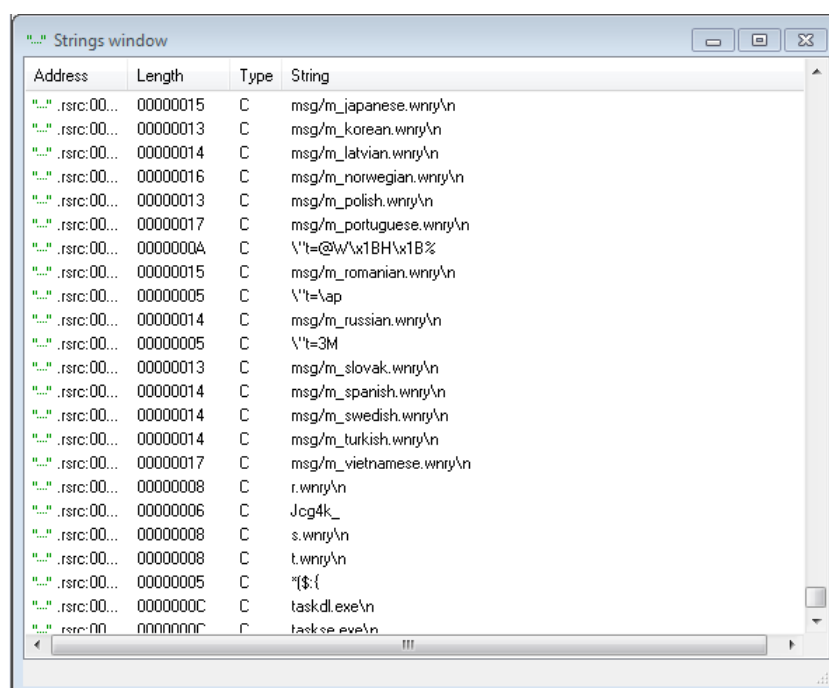


Figure 5: *.wnry filenames

Filenames & Descriptions

b.wnry	Background image
c.wnry	Configuration
s.wnry	Tor communication (endpoints from c.wnry)
u.wnry	User interface
t.wnry	Default keys
msg	A folder of language packs

Moving into the software workflow, the program enters the main method, `_WinMain@16`, and extracts a password protected archive named "WNcry@2o17" and creates the necessary services and procedures. The program retrieves host information, as well as the correct language file. It then loops through the extracted files in the current working directory, and elevates privileges. It then collects and terminates a number of processes that it has pushed into the system stack.⁹ This is seemingly so that the files won't be in use which allows the processes to encrypt them without trouble. The software then continues to loop through directories and encrypts files as it goes, appending them with the `.WCry` extension, seemingly for 'shock value' with `tasksche.exe`. It currently attacks 151 different file types, which can be found in the decompilation, but this count seems to vary among the different variants. The program deletes Shadow Volume Copies and disables backup restoration. It finally displays the 'lock' screen along with the rest of the UI contained in `u.wnry` and implemented as `@WanaDecryptor@.exe`. Once the payment is verified, the program then goes through and decrypts all of the encrypted files, and exits. The program utilizes a TOR client for communication with the attackers to verify payment, which is retrieved before the encryption begins.

⁹After searching online, I discovered these processes were specifically related to database and mail servers.

One of the notable early discoveries was that there was a built in 'kill-switch' that would stop the software from fully taking over systems and encrypting files, however it should be noted that the version of software I have does not include this kill-switch. It seems to be the only difference between the 'original' and the version that I am examining.

2.1.4 Dynamic Analysis

The file is being launched directly from `~/Downloads/Ransomware.WannaCry/`. Upon initial launch, the wallpaper is immediately changed, and you can see the dropper extracts the files from the embedded archive into the current directory.

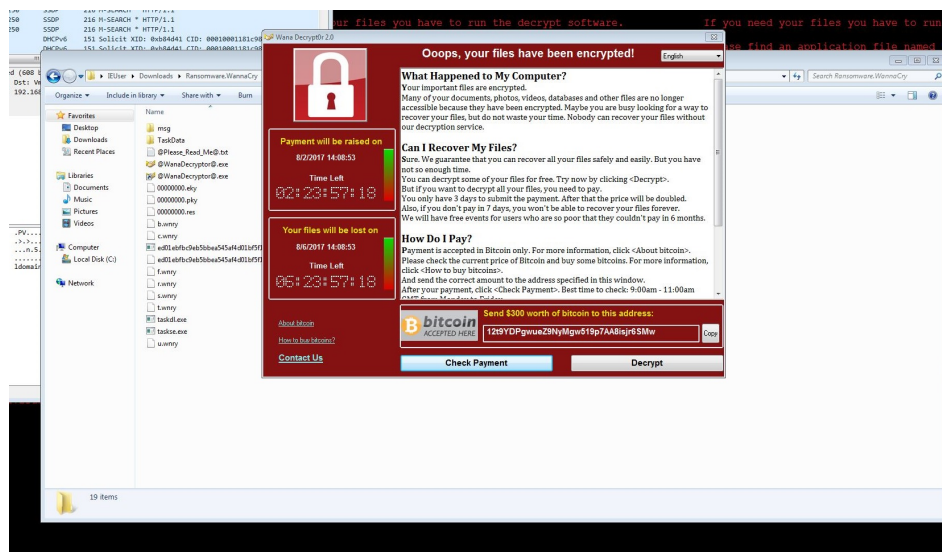


Figure 6: Initial Infection

Whenever the program is launched, a UAC prompt pops up requesting elevated privileges (Fig 7). If the victim denies the privileges, the software is still able to continue with encryption but cannot delete Shadow Volumes or backups.

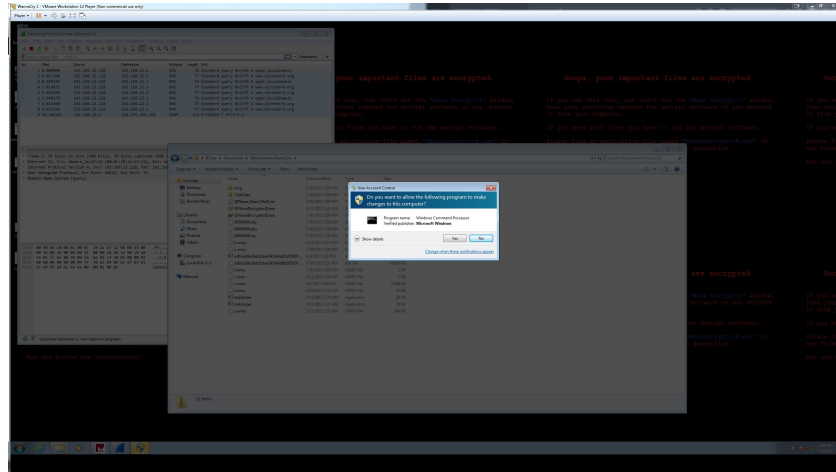


Figure 7: UAC Prompt on first run

The program creates additional files to accompany the files extracted with the archive. *f.wnry* stores a list of decrypted files, and *r.wnry* is a copy of @Please_Read_Me@.txt. *WanaDecryptor.exe* is the user GUI implementation, and the TOR client is retrieved into the *TaskData* folder. Additionally, RSA key files are created for the encryption/decryption processes. The actual encryption simply navigates down through directories encrypting files, always modifying the file headers, leaving a copy of @Please_Read_Me@.txt and a shortcut to @WannaDecryptor@.exe (Fig 8) in the directory.

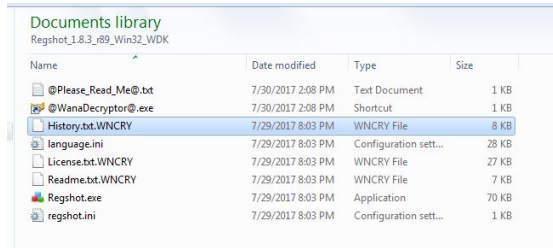


Figure 8: An example directory after infection

The software completely destroys the registry, in the case of this test machine deleting 324 keys. The program then adds 1547 new keys and modifies the remaining registry keys, entirely focused on wholly taking over the system. I was unable to find any saved record of these changes for the program, so it doesn't seem that the host is actually completely restorable upon payment, as the software claims.

The IP address for the victim in this test is *192.168.22.254*, and the VMware Virtual Adapter IP address is *192.168.22.1*. The initial Wireshark capture was started on the victim network adapter previous to infection, and showed no traffic with no network applications active. Once the infection is initialized, we see regular SSDP transmissions followed by DHCP Solicit XID requests attempting to discover network devices and obtain new host information. Continuing with the capture, we see TCP requests and TLSv1 packets to retrieve the key whenever we try to verify payment, indicative of the TOR communication lines.

2.1.5 Conclusion

The WannaCry ransomware is reasonably sophisticated, and takes advantage of the EternalBlue vulnerability as a reliable propagation vector as evidenced by the infection count of an estimated 230,000 machines. The best option to prevent passive infection is to apply Microsoft update KB4013389, and to otherwise follow best practices regarding downloaded files, email attachments, etc. I additionally tested a program called *Wanakiwi*¹⁰ that successfully retrieved a decryption key, however it only works if the system hasn't been restarted since infection and must be used as soon as possible to ensure the needed information is still in memory. The registry keys were not restored upon decryption, so once the needed information is retrieved it is best to perform a fresh install of the OS in any case and check all devices on the network, including network-attached storage.

3 The Defense Software

3.1 An Overview

The endeavor to write a defense program was based around a desire to become more familiar with the Windows NT environment and general antivirus development processes. The general operation of the software can be seen below in Alg. 1. A system callback, `SetCreateProcessNotifyRoutineEx`, is used to listen for process creation by using `PsSetCreateProcessNotifyRoutineEx`. This is implemented through a simple kernel-level driver that will launch a system *ProcessEvent* that can be caught by any processes that choose to listen. Specifically for this implementation, upon launch the process handle is passed into a queue of processes to be 'processed'. Whenever a new item enters this queue, the executable path is retrieved, and the file is hashed. If there is no known definition based on this hash, the hash is sent to VirusTotal utilizing the public C++ API, version 2.0. The result is then used to make a new entry into the database, and process creation either continues or is denied as deemed necessary by the results. I considered any process with more than 5 references to be unsafe, although realistically one could deny launch for even 1 VT reference.

The first hurdle with developing this software was creating a kernel driver. Having no previous experience myself, I read *Windows Internals, 6th ed.*¹¹ and also followed a tutorial on CodeProject.com¹². Finally, I also referenced a Microsoft-supplied example driver found on their Windows Example drivers GitHub repository.¹³ Because the kernel should not be made to wait for a response that is any longer than a few milliseconds (at most!), the execution is denied by default and the process will be relaunched if it has no known results from VirusTotal. Naturally in the process of working with a kernel driver, I learned much of the Windows NT API to handle events and perform actions around those events.

¹⁰<https://github.com/gentilkiwi/wanakiwi>

¹¹<https://docs.microsoft.com/en-us/sysinternals/learn/windows-internals>

¹²<http://bit.ly/2lxXIv5>

¹³<https://github.com/Microsoft/Windows-driver-samples>

Initialization:
Register PsSetCreateProcessNotifyRoutineEx callback;
Wait for process creation
Upon creation, run the following:
Data: Process Handle
Hash the Process Executable;
if *hash in database* **then**
 if {*db_entry*}.*allow* == *True* **then**
 allow execution;
 print allowance message;
 else
 deny execution;
 print denial message;
 end
else
 transmit hash to VirusTotal;
 {receive reply}
 if *no VT results* **then**
 allow execution;
 print allowance message;
 else
 deny execution;
 print denial message;
 end
 construct new database entry;
 set tuple launch permission;
 add to database;
end

Algorithm 1: A generalized algorithm

pagebreak
not working?

4 Appendix A

List of affected file extensions

.der .pfx .key .crt .csr .pem .odt .ott .sxw .stw .uot .max .ods .ots .sxc .stc .dif .slk .odp .otp .sxd .std .uop .odg .otg .sxm .mml .lay .lay6 .asc .sqlite3 .sqlitedb .sql .accdb .mdb .dbf .odb .frm .myd .myi .ibd .mdf .ldf .sln .suo .cpp .pas .asm .cmd .bat .vbs .dip .dch .sch .brd .jsp .php .asp .java .jar .class .wav .swf .fla .wmv .mpg .vob .mpeg .asf .avi .mov .mkv .flv .wma .mid .djvu .svg .psd .nef .tiff .tif .cgm .raw .gif .png .bmp .jpg .jpeg .vcd .iso .backup .zip .rar .tgz .tar .bak .tbk .paq .arc .aes .gpg .vmx .vmdk .vdi .sldm .sldx .sti .sxi .hwp .snt .onetoc2 .dwg .pdf .wks .rtf .csv .txt .vsdx .vsd .edb .eml .msg .ost .pst .potm .potx .ppam .ppsx .ppsm .pps .pot .pptm .pptx .ppt .xltm .xltx .xlc .xlm .xlt .xlw .xlsb .xlsm .xlsx .xls .dotx .dotm .dot .docm .docb .docx .doc

fix this for-
matting