Malware Project Analysis Report - kittyware.exe

Autumn 2024 CSS 579 A: Malware And Attack Reverse Engineering

Date: 10 December 2024

By

Nagendra Kaushik Godlaveti

Meenal Shah

Yash Mahesh Malpatak

Executive Summary

The malware, **kittyware.exe**, is a sophisticated malicious executable that exhibits multiple functionalities including anti-disassembly, anti-debugging, encryption/decryption, and system manipulation. It primarily aims to achieve persistence and modify system behavior by downloading and setting an image as the desktop wallpaper while also displaying a ransom message. Static and dynamic analysis revealed no signs of code packing or obfuscation in its PE sections.

- Persistence Mechanism: Creates a registry entry ('kittywarev') under "SOFTWARE\Microsoft\Windows\CurrentVersion\Run" to ensure it auto-starts with system boot.
- Payload Delivery: Downloads an image file ('straycatj.jpg') from a remote URL.
- System Modification: Alters desktop wallpaper settings via the registry path "Control Panel\DesktopV\WallPaper".
- Ransom Behavior: Displays a pop-up message demanding an exorbitant ransom, emphasizing the malware's intended extortion purpose.
- Anti-Analysis Techniques: Employs anti-disassembly and anti-debugging mechanisms to hinder reverse engineering.
- Encryption and Decryption: Utilizes cryptographic functions to secure its payload and associated data, with keys and encrypted strings decrypted during runtime.

Key Findings

Static Analysis

Malware Metadata

Basic Details

MD5	fc4f2b52671efa4e049513a4f68a0012
SHA-1	3f488998b208e7c02721203b81457d3cfc739723
SHA-256	956042e478a2230954e9ec8de51127bd86bc2152a1dcbddb996b25ccda875a19
File Type	Win32 EXE

Sections

Name	Virtual Size	Raw Size	Entropy
.text	23920	24064	5.69
.data	848	1024	4.24
.rdata	3444	3584	4.73
.tls	16	512	0

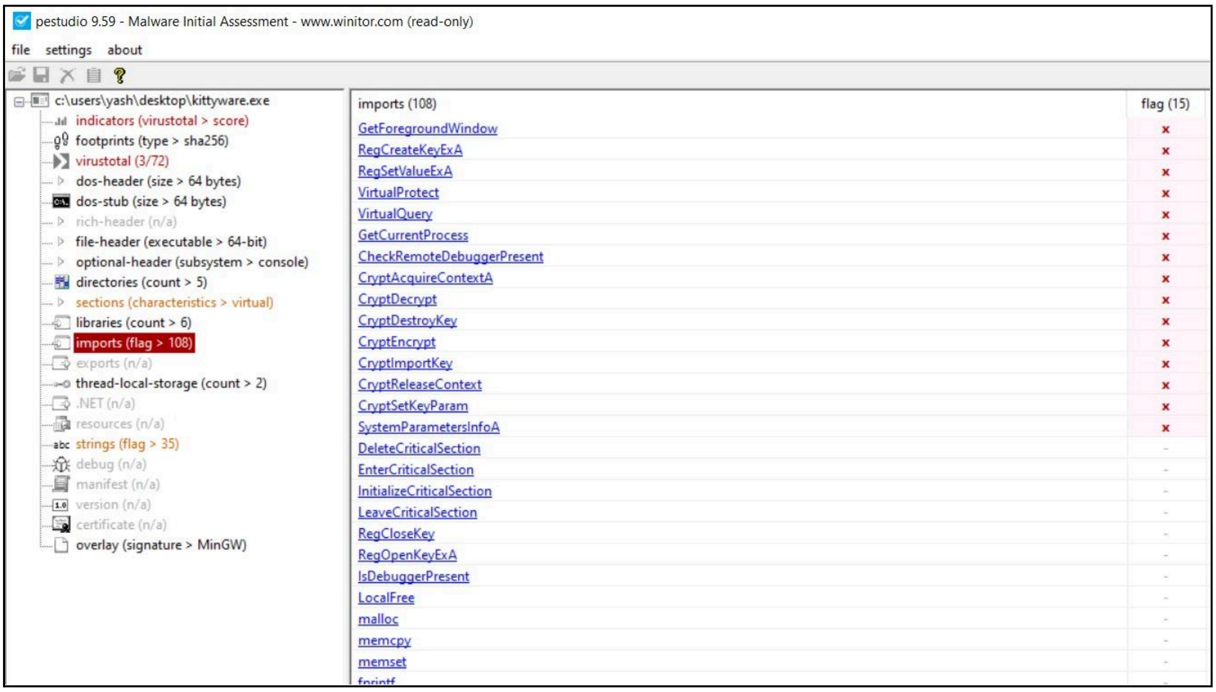
It is noteworthy that the presence of a '.tls' section in the file suggests that the entry point of the program has been modified from the standard entry point of a Win32 EXE file.

Packing Analysis

No unpacking mechanisms were detected in the databases of known packers (PEID, UPX, DIE), nor were they identified during the static or dynamic analysis performed in IDA. Additionally, if any packing was present, it would be unpacked during dynamic analysis while debugging.

PeStudio Analysis

A few imports marked as suspicious by PEStudio were identified. These are presented in the screenshot below:



These following API functions mentioned below give a starting point of interest/overview while analysing the malware in a Disassembler:

- GetForegroundWindow
- RegCreateKeyExA, RegSetValueExA
- VirtualAlloc, VirtualProtect
- GetCurrentProcess
- CheckRemoteDebuggerPresent
- CryptAcquireContextA, CryptEncrypt, CryptDecrypt
- LoadLibraryA and GetProcAddress
- DeleteCriticalSection, InitializeCriticalSection
- malloc, free, fwrite, fclose
- VirtualFree, HeapFree:
- SetUnhandledExceptionFilter
- CreateFileA
- WriteFile, ReadFile

Analysis using IDA

The malware ‘kittyware.exe’ is a 64-bit Windows console application in the PE32+ format, designed for execution on modern Windows systems with x86-64 (64-bit) architecture. This malicious file is analysed using a 64-bit IDA Pro tool.

Strings

Significant amount of information can be gathered from the Strings section of IDA:

- Strings like ‘Successfully got crypt context.’, ‘Successfully got our key.’ and ‘Your files have been snatched!!! You owe us one million billion dollars!!!’ along with use of functions such as ‘CryptEnquireContextA’, ‘CryptDecrypt’, ‘CryptEncrypt’, ‘CryptSetKeyParam’ suggests that there are encryption and decryption techniques used by the malware.
- Strings such as ‘Failed to decrypt target URL.’ and the API functions ‘InternetOpenA’, ‘InternetOpenUrlA’, ‘InternetCloseHandle’, ‘InternetReadFile2’, ‘HttpQueryInfoA’ suggests that the payload may be downloaded from the Internet.
- Also, the strings ‘Successfully downloaded background!’ and ‘DEBUG:’, ‘Wrote image data to file.\n’ suggests that the target file in question may be an image of some kind.
- Strings such as ‘Failed to decrypt target autorun registry.’ along with several registry related functions such as ‘RegCreateKeyExA’, ‘RegOpenKeyExA’ and ‘RegCloseKey’, ‘RegSetValueExA’ suggests that a file is added to the autorun registry to gain persistence.
- Strings like ‘Successfully updated wallpaper.’ suggests that the malware changes the background of the desktop at runtime.
- Strings such as ‘ERROR when performing anti-disassembly.’ and ‘ERROR when performing anti-debugging.’ along with functions such as ‘IsDebuggerPresent’ and ‘CheckRemoteDebuggerPresent’ suggests that anti-disassembly and anti-debugging techniques are used by the malware.
- Finally, the ‘MessageBoxA’ function call suggests that a popup occurs on running the file.

Analysis of TLS Callbacks

On analyzing the exported TLS Callback functions, there was nothing of importance to be noted. The code is performing initialization of mingw libraries.

Name	Address	Ordinal
TlsCallback_0	00000001400046F0	
TlsCallback_1	00000001400046C0	
mainCRTStartup	00000001400013F0	[main entry]

Screenshot of exported functions

Anti Disassembly Techniques

- The malware used a common disassembly tactic by implementing a conditional jump to make an unconditional jump during execution, which can confuse the disassembler to disassemble the wrong code.

```
.text:000000014000301E 048 00
.text:0000000140003025 048 C7 45 F8 00 00 00 mov     [rbp+10h+var_18], 0
.text:0000000140003025 048 00
.text:000000014000302C 048 8B 55 FC          mov     edx, [rbp+10h+var_14]
.text:000000014000302F 048 89 D0             mov     eax, edx
.text:0000000140003031 048 31 C0             xor     eax, eax         ; Logical Exclusive OR
.text:0000000140003033 048 85 C0             test    eax, eax         ; Logical Compare
.text:0000000140003035 048 74 07             jz      short skip_dead_code ; Jump if Zero (ZF=1)

.text:0000000140003037 048 B8 EF BE AD DE    mov     eax, 0DEADBEEFh
.text:000000014000303C 048 90                nop                      ; No Operation
.text:000000014000303D 048 90                nop                      ; No Operation

.text:000000014000303E
.text:000000014000303E                       skip_dead_code:
.text:000000014000303E 048 48 8D 05 02 00 00 lea     rax, continue_execution ; Load Effective Address
.text:000000014000303E 048 00
.text:0000000140003045 048 FF E0             jmp     rax              ; Indirect Near Jump
```

As this code is detected by the disassembler already, there is no need to fix it.

- Another disassembly technique is implemented in the malware, making it more difficult for a malware analyst to determine the actual jump location. This is achieved by first storing the memory location of the jump in a registry and then calling the registry, rather than directly calling the memory location.

```
.text:000000014000303E
.text:000000014000303E                       skip_dead_code:
.text:000000014000303E 048 48 8D 05 02 00 00 lea     rax, continue_execution ; Load Effective Address
.text:000000014000303E 048 00
.text:0000000140003045 048 FF E0             jmp     rax              ; Indirect Near Jump
```

- The value for eax register is set by var_18, which is previously hard coded as 0. This makes the jump at 140003051 location always true.

```
.text:0000000140003047
.text:0000000140003047           continue_execution:
.text:0000000140003047 048 8B 45 F8  mov     eax, [rbp+10h+var_18]
.text:000000014000304A 048 85 C0     test    eax, eax         ; Logical Compare
.text:000000014000304C 048 0F 95 C0  setnz   al               ; Set Byte if Not Zero (ZF=0)
.text:000000014000304F 048 84 C0     test    al, al           ; Logical Compare
.text:0000000140003051 048 74 4F     jz      short loc_1400030A2 ; Jump if Zero (ZF=1)
```

Anti Debugging Techniques

- This malware uses an anti-debugging technique which is understood by the 'AntiDebugV' function call.

```
.text:000000014000369F
.text:000000014000369F                 loc_14000369F:
.text:000000014000369F E8 52 FA FF FF  call    _Z9AntiDebugv    ; Call Procedure
.text:00000001400036A4 84 C0           test    al, al           ; Logical Compare
.text:00000001400036A6 0F 94 C0        setz    al               ; Set Byte if Zero (ZF=1)
.text:00000001400036A9 84 C0           test    al, al           ; Logical Compare
.text:00000001400036AB 74 35           jz      short loc_1400036E2 ; Jump if Zero (ZF=1)
```

- The anti debugging process can further be confirmed by the use of common anti-debugging API calls such as 'IsDebuggerPresent & 'CheckRemoteDebuggerPresent', which checks for presence of a debugger.

```
.text:000000014000319F
.text:000000014000319F                          loc_14000319F:
.text:000000014000319F C7 45 FC 00 00 00 mov     [rbp+pbDebuggerPresent], 0
.text:000000014000319F 00
.text:00000001400031A6 48 8B 05 07 93 00 mov     rax, cs:__imp_GetCurrentProcess
.text:00000001400031A6 00
.text:00000001400031AD FF D0             call    rax ; __imp_GetCurrentProcess ; Indirect Call Near Procedure
.text:00000001400031AF 48 89 C1          mov     rcx, rax        ; hProcess
.text:00000001400031B2 48 8D 45 FC       lea     rax, [rbp+pbDebuggerPresent] ; Load Effective Address
.text:00000001400031B6 48 89 C2          mov     rdx, rax        ; pbDebuggerPresent
.text:00000001400031B9 48 8B 05 CC 92 00 mov     rax, cs:__imp_CheckRemoteDebuggerPresent
.text:00000001400031B9 00
.text:00000001400031C0 FF D0             call    rax ; __imp_CheckRemoteDebuggerPresent ; Indirect Call Near Procedure
.text:00000001400031C2 8B 45 FC          mov     eax, [rbp+pbDebuggerPresent]
.text:00000001400031C5 85 C0             test    eax, eax        ; Logical Compare
.text:00000001400031C7 74 47             jz      short loc_140003210 ; Jump if Zero (ZF=1)
```

Screenshot of CheckRemoteDebuggerPresent function call

```
.text:00000001400030F6
.text:00000001400030F6
.text:00000001400030F6                          ; Attributes: bp-based frame
.text:00000001400030F6
.text:00000001400030F6                          ; __int64 AntiDebug(void)
.text:00000001400030F6                          public _Z9AntiDebugv
.text:00000001400030F6                          _Z9AntiDebugv proc near
.text:00000001400030F6
.text:00000001400030F6                          pbDebuggerPresent= dword ptr -4
.text:00000001400030F6
.text:00000001400030F6 55                       push    rbp
.text:00000001400030F7 48 89 E5                 mov     rbp, rsp
.text:00000001400030FA 48 83 EC 30              sub     rsp, 30h        ; Integer Subtraction
.text:00000001400030FE 48 8B 05 D7 93 00 mov    rax, cs:__imp_IsDebuggerPresent
.text:00000001400030FE 00
.text:0000000140003105 FF D0                    call    rax ; __imp_IsDebuggerPresent ; Indirect Call Near Procedure
.text:0000000140003107 85 C0                    test    eax, eax        ; Logical Compare
.text:0000000140003109 0F 95 C0                 setnz   al              ; Set Byte if Not Zero (ZF=0)
.text:000000014000310C 84 C0                    test    al, al          ; Logical Compare
.text:000000014000310E 74 47                    jz      short loc_140003157 ; Jump if Zero (ZF=1)
```

Screenshot of IsDebuggerPresent function call

- Evading these anti-debugging functions for dynamic analysis was a simple patch over ‘call rax’ with ‘xor eax, eax’ assembly instructions.

Initialization of Encryption/Decryption Functions Analysis

- Initialization of encryption/decryption for further usage, later in the program is done by the function ‘Z14InitEncryptionv’.
- The key is found in the function ‘_Z6GetKeyv’, which is used throughout the file for encryption/decryption of various string elements. The key is stored in a variable called ‘keyIV’ with the value ‘732065766F6C2069h’ or ‘s evol i’ in String.
- The CryptDecrypt and CryptEncrypt functions of ‘wincrypt.h’ library are used to decrypt and encrypt the various string elements in the file. The list of variables decrypted are as follows:
  - TargetFile
  - TargetRegistryWp
  - TargetRegKeyWp
  - TargetRegistryRun
  - TargetRegKeyRun

GetModuleFileName

- After decrypting the various different variables as mentioned above, the malware calls GetModuleFileNameA, using which it gets the location where the malware is stored.

```
.text:000000014000396E
.text:000000014000396E loc_14000396E:
.text:000000014000396E lea     rax, [rbp+270h+var_2C0]
.text:0000000140003972 mov     rcx, rax
.text:0000000140003975 call    _ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEEC1Ev ; std::string::basic_string(void)
.text:000000014000397A lea     rax, [rbp+270h+Filename]
.text:000000014000397E mov     r8d, 104h       ; nSize
.text:0000000140003984 mov     rdx, rax        ; lpFilename
.text:0000000140003987 mov     ecx, 0          ; hModule
.text:000000014000398C mov     rax, cs:__imp_GetModuleFileNameA
.text:0000000140003993 ;   try {
.text:0000000140003993 call    rax ; __imp_GetModuleFileNameA
.text:0000000140003995 test    eax, eax
.text:0000000140003997 setnz   al
.text:000000014000399A test    al, al
.text:000000014000399C jz      loc_140003A56
```

Screenshot of GetModuleFileName function call

GetFunctionAddresses

- After the location of the malware is obtained, the function GetFunctionAddresses performs three tasks: decrypting encrypted function and library names for calling Internet functions using the WinINet library, retrieving the address of the exported function from the specified DLL, and using the decrypted networking functions to download an image from a specified URL.
- Decrypting encrypted WinInet functions: The following variables are decrypted by this function:
  - winInetLibary
  - eInternetOpenA
  - eInternetOpenUrlA
  - eInternetCloseHandle
  - eInternetReadFile
  - eHttpQueryInfo
  -
- The encryption of these functions suggest that the malware does not disclose the networking functions that are used, or to hide contact with the malware author's server.

```
.text:00000001400026FE 038 mov     [rbp+var_6], 64h ; 'd'
.text:0000000140002702 038 lea     rax, [rbp+var_6] ; Load Effective Address
.text:0000000140002706 038 mov     r8, rax
.text:0000000140002709 038 lea     rax, urlSize     ; Load Effective Address
.text:0000000140002710 038 mov     rdx, rax
.text:0000000140002713 038 lea     rax, url         ; Load Effective Address
.text:000000014000271A 038 mov     rcx, rax
.text:000000014000271D 038 call    _Z16EncryptOrDecryptILy96EEbRAT__hRmRKc ; Call Procedure
.text:0000000140002722 038 xor     eax, 1           ; Logical Exclusive OR
.text:0000000140002725 038 test    al, al           ; Logical Compare
.text:0000000140002727 038 jz      short loc_14000277B ; Jump if Zero (ZF=1)
```

- GetProcAddress:
  - This function is used to retrieve the address of the exported function from the specified DLL. In this case, the DLL used is 'Kernel32.dll'.

```
.text:000000014000228D
.text:000000014000228D loc_14000228D:
.text:000000014000228D lea     rax, [rbp+1E0h+var_220]
.text:0000000140002291 mov     rcx, rax
.text:0000000140002294 call    _ZNKSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEE5c_strEv ; std::string::c_str(void)
.text:0000000140002299 mov     rdx, rax        ; lpProcName
.text:000000014000229C mov     rax, cs:WinInet
.text:00000001400022A3 mov     rcx, rax        ; hModule
.text:00000001400022A6 mov     rax, cs:__imp_GetProcAddress
.text:00000001400022AD call    rax ; __imp_GetProcAddress
.text:00000001400022AF mov     cs:InternetOpenA, rax
.text:00000001400022B6 lea     rax, [rbp+1E0h+var_240]
.text:00000001400022BA lea     r8, aFailedToGetAdd ; "Failed to get address of function "
.text:00000001400022C1 lea     rdx, _Z3errB5cxx11 ; err
.text:00000001400022C8 mov     rcx, rax
.text:00000001400022CB call    _ZStplIcSt11char_traitsIcESaIcEENSt7__cxx1112basic_stringIT_T0_T1_EERKS8_PKS5_ ; std::operator+<char>(std::string const&,char const*)
.text:00000001400022CB ;   } // starts at 140002209
.text:00000001400022D0 mov     rax, cs:InternetOpenA
.text:00000001400022D7 test    rax, rax
.text:00000001400022DA jnz     short loc_140002330
```

Screenshot of GetProcAddress function call

- Downloading image using functions:
    - Using the above decrypted functions, SystemParametersInfoA and GetProcAddress, the malware downloads the image from the specified URL and saves the file.

Gain Persistance

- The malware gains persistence by making changes to the registry key. The malware checks if the registry key is existing or not. In case, the registry key is not present, the malware creates a key and sets the value as required.

```
.text:00000001400033E8
.text:00000001400033E8 loc_1400033E8:
.text:00000001400033E8 mov     rax, [rbp+10h+arg_0]
.text:00000001400033EC mov     rcx, rax
.text:00000001400033EF call    _ZNKSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEE5c_strEv ; std::string::c_str(void)
.text:00000001400033F4 mov     rdx, rax        ; lpSubKey
.text:00000001400033F7 mov     [rsp+70h+lpdwDisposition], 0 ; lpdwDisposition
.text:0000000140003400 lea     rax, [rbp+10h+hKey]
.text:0000000140003404 mov     [rsp+70h+var_38], rax ; phkResult
.text:0000000140003409 mov     [rsp+70h+lpSecurityAttributes], 0 ; lpSecurityAttributes
.text:0000000140003412 mov     [rsp+70h+samDesired], 20006h ; samDesired
.text:000000014000341A mov     dword ptr [rsp+70h+phkResult], 0 ; dwOptions
.text:0000000140003422 mov     r9d, 0          ; lpClass
.text:0000000140003428 mov     r8d, 0          ; Reserved
.text:000000014000342E mov     rcx, 0FFFFFFFF80000001h ; hKey
.text:0000000140003435 mov     rax, cs:__imp_RegCreateKeyExA
.text:000000014000343C call    rax ; __imp_RegCreateKeyExA
.text:000000014000343E test    eax, eax
.text:0000000140003440 setnz   al
.text:0000000140003443 test    al, al
.text:0000000140003445 jz      short loc_140003456
```

Screenshot of Registry key functions used

```
.text:0000000140003371
.text:0000000140003371 loc_140003371:
.text:0000000140003371 mov     rax, [rbp+10h+arg_0]
.text:0000000140003375 mov     rcx, rax
.text:0000000140003378 call    _ZNKSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEE5c_strEv ; std::string::c_str(void)
.text:000000014000337D mov     rdx, rax        ; lpSubKey
.text:0000000140003380 lea     rax, [rbp+10h+hKey]
.text:0000000140003384 mov     [rsp+70h+phkResult], rax ; phkResult
.text:0000000140003389 mov     r9d, 20006h     ; samDesired
.text:000000014000338F mov     r8d, 0          ; ulOptions
.text:0000000140003395 mov     rcx, 0FFFFFFFF80000001h ; hKey
.text:000000014000339C mov     rax, cs:__imp_RegOpenKeyExA
.text:00000001400033A3 call    rax ; __imp_RegOpenKeyExA
.text:00000001400033A5 test    eax, eax
.text:00000001400033A7 setnz   al
.text:00000001400033AA test    al, al
.text:00000001400033AC jz      loc_1400034A4
```

Screenshot of Registry key functions used

```
.text:00000001400034EC
.text:00000001400034EC loc_1400034EC:
.text:00000001400034EC mov     rax, [rbp+10h+arg_10]
.text:00000001400034F0 mov     rcx, rax
.text:00000001400034F3 call    _ZNKSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEE4sizeEv ; std::string::size(void)
.text:00000001400034F8 lea     esi, [rax+1]
.text:00000001400034FB mov     rax, [rbp+10h+arg_10]
.text:00000001400034FF mov     rcx, rax
.text:0000000140003502 call    _ZNKSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEE5c_strEv ; std::string::c_str(void)
.text:0000000140003507 mov     rbx, rax
.text:000000014000350A mov     rax, [rbp+10h+arg_8]
.text:000000014000350E mov     rcx, rax
.text:0000000140003511 call    _ZNKSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEE5c_strEv ; std::string::c_str(void)
.text:0000000140003516 mov     rdx, rax        ; lpValueName
.text:0000000140003519 mov     rax, [rbp+10h+hKey]
.text:000000014000351D mov     [rsp+70h+samDesired], esi ; cbData
.text:0000000140003521 mov     [rsp+70h+phkResult], rbx ; lpData
.text:0000000140003526 mov     r9d, 1          ; dwType
.text:000000014000352C mov     r8d, 0          ; Reserved
.text:0000000140003532 mov     rcx, rax        ; hKey
.text:0000000140003535 mov     rax, cs:__imp_RegSetValueExA
.text:000000014000353C call    rax ; __imp_RegSetValueExA
.text:000000014000353E test    eax, eax
.text:0000000140003540 setnz   al
.text:0000000140003543 test    al, al
.text:0000000140003545 jz      short loc_140003553
```

Screenshot of Registry key functions used

Encryption Of Files

- After gaining persistence, the malware proceeds to encrypt a string masquerading as important system files, so that the user can no longer access their files.

```
text:00007FF782723E33 2F8 48 89 C1          mov     rcx, rax
text:00007FF782723E36 2F8 E8 85 07 00 00    call    _ZNKSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEE5c_strEv ; Call Procedure
text:00007FF782723E3B 2F8 48 89 C1          mov     rcx, rax        ; Str
text:00007FF782723E3E 2F8 E8 5D 19 00 00    call    strlen          ; Call Procedure
text:00007FF782723E43 2F8 89 45 A4          mov     [rbp+270h+var_2CC], eax
text:00007FF782723E46 2F8 C6 85 0F 02 00 00 mov     [rbp+270h+var_61], 65h ; 'e'
text:00007FF782723E46 2F8 65
text:00007FF782723E4D 2F8 48 8D 8D 0F 02 00 lea     rcx, [rbp+270h+var_61] ; Load Effective Address
text:00007FF782723E4D 2F8 00
text:00007FF782723E54 2F8 48 8D 55 A4       lea     rdx, [rbp+270h+var_2CC] ; unsigned int *
text:00007FF782723E58 2F8 48 8D 45 A8       lea     rax, [rbp+270h+var_2C8] ; Load Effective Address
text:00007FF782723E5C 2F8 49 89 C8          mov     r8, rcx         ; char *
text:00007FF782723E5F 2F8 48 89 C1          mov     rcx, rax        ; unsigned __int8 **
text:00007FF782723E62 2F8 E8 17 DA FF FF    call    _Z16EncryptOrDecryptRPhRmRKc ; Call Procedure
text:00007FF782723E67 2F8 83 F0 01          xor     eax, 1          ; Logical Exclusive OR
text:00007FF782723E6A 2F8 84 C0             test    al, al          ; Logical Compare
text:00007FF782723E6C 2F8 74 35             jz      short loc_7FF782723EA3 ; Jump if Zero (ZF=1)
```

Screenshot of the encryption call

Message Box

- Following encryption, a message box pops up containing the string ‘Your files have been snatched!!! You owe us one million billion dollars!!!’. ‘MessageBoxA’ function pops up the message box.
- The MessageBoxA function is called by giving the handle received by GetForegroundWindow, ensuring that the MessageBox is visible to the user immediately.
- This is to intimidate the user and ask for money to decrypt the user’s files.

```
.text:0000000140003EFE lea     rax, unk_140008689
.text:0000000140003F05 mov     [rbp+270h+lpCaption], rax
.text:0000000140003F0C lea     rax, aYourFilesHaveB ; "Your files have been snatched!!! You ow"...
.text:0000000140003F13 mov     [rbp+270h+lpText], rax
.text:0000000140003F1A mov     rcx, [rbp+270h+lpCaption]
.text:0000000140003F21 mov     rdx, [rbp+270h+lpText] ; lpText
.text:0000000140003F28 mov     rax, [rbp+270h+hWnd]
.text:0000000140003F2F mov     r9d, 0          ; uType
.text:0000000140003F35 mov     r8, rcx         ; lpCaption
.text:0000000140003F38 mov     rcx, rax        ; hWnd
.text:0000000140003F3B mov     rax, cs:__imp_MessageBoxA
.text:0000000140003F42 call    rax ; __imp_MessageBoxA
.text:0000000140003F44 jmp     short loc_140003F8E
```

Screenshot of MessageBoxA function call

Dynamic Analysis

Running via Debugger

The debugger is executed after patching the anti-debugging function calls found in static analysis.

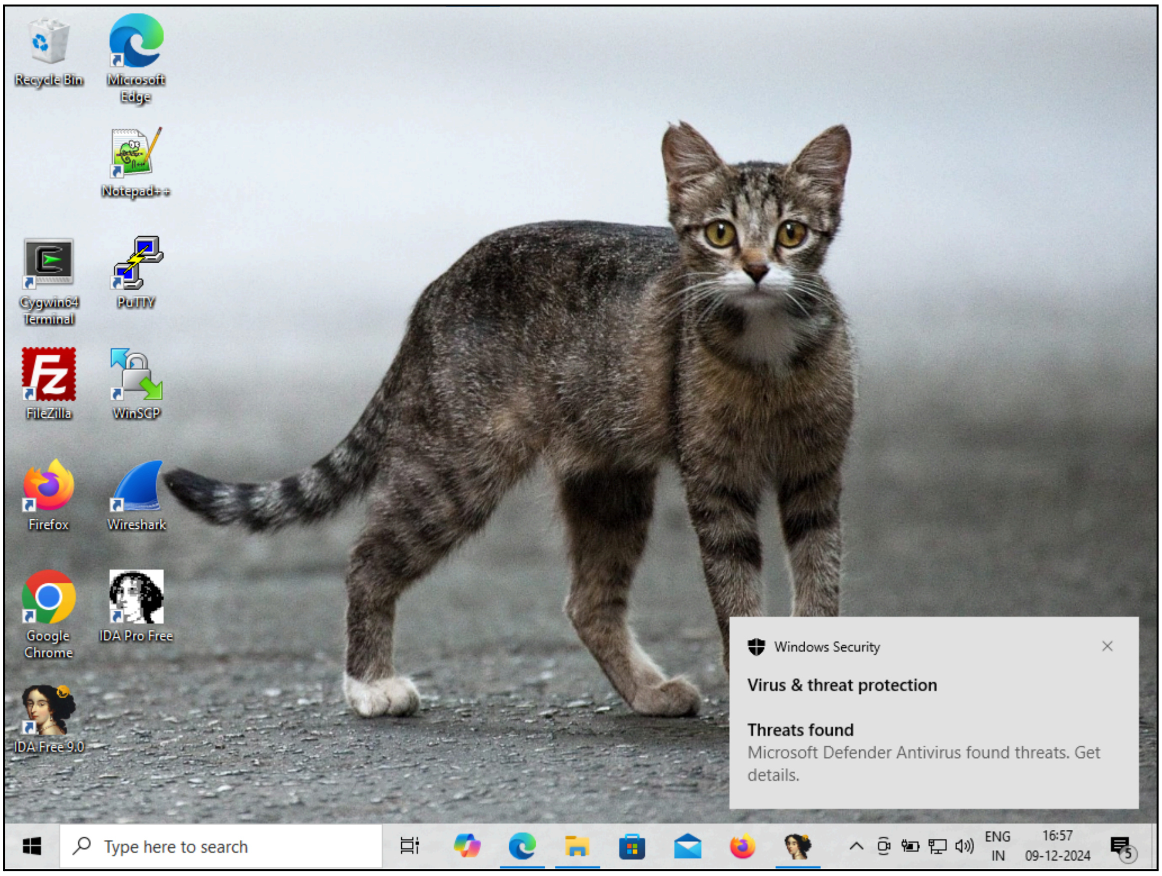
Encryption/Decryption Functions

Stepping through the program, the debugging process was able to decrypt all the encrypted strings, functions and library names obfuscated via encryption. The following are the values found while debugging:

- TargetFile = ‘straycatj.jpg’
- TargetRegistryWp = ‘Control Panel\\DesktopVЦ’
- TargetRegKeyWp = ‘WallPaper’
- TargetRegistryRun = ‘SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run’
- TargetRegKeyRun = ‘kittywarev’
- winInetLibary = ‘wininet.dll’
- eInternetOpenA = ‘InternetOpenA’
- eInternetOpenUrlA = ‘InternetOpenUrlA’
- eInternetCloseHandle = ‘InternetCloseHandle’
- eInternetReadFile = ‘InternetReadFile2’
- eHttpQueryInfo = ‘HttpQueryInfoA’

GetFunctionAddresses

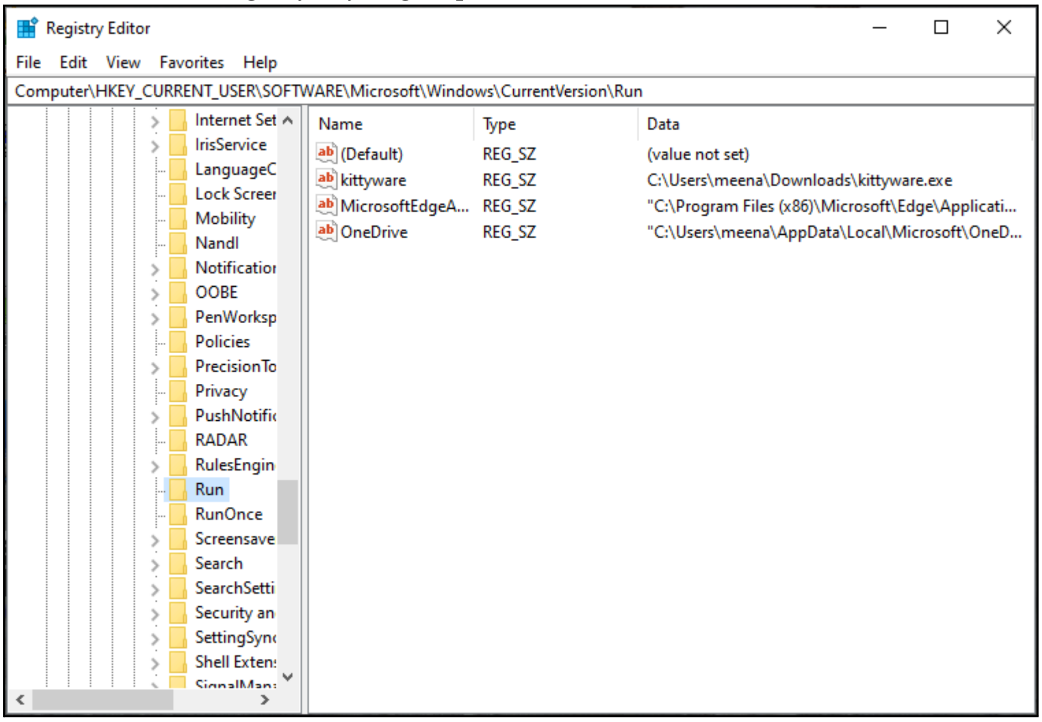
- Post decrypting the ‘WinInet’ functions, target information and malware location, the malware downloads an image file in the same directory as the malware called straycatj.jpg.
- The malware proceeds to update the desktop wallpaper with this file using ‘SystemParametersInfoA’.



Screenshot of changed desktop image

Gain Persistence

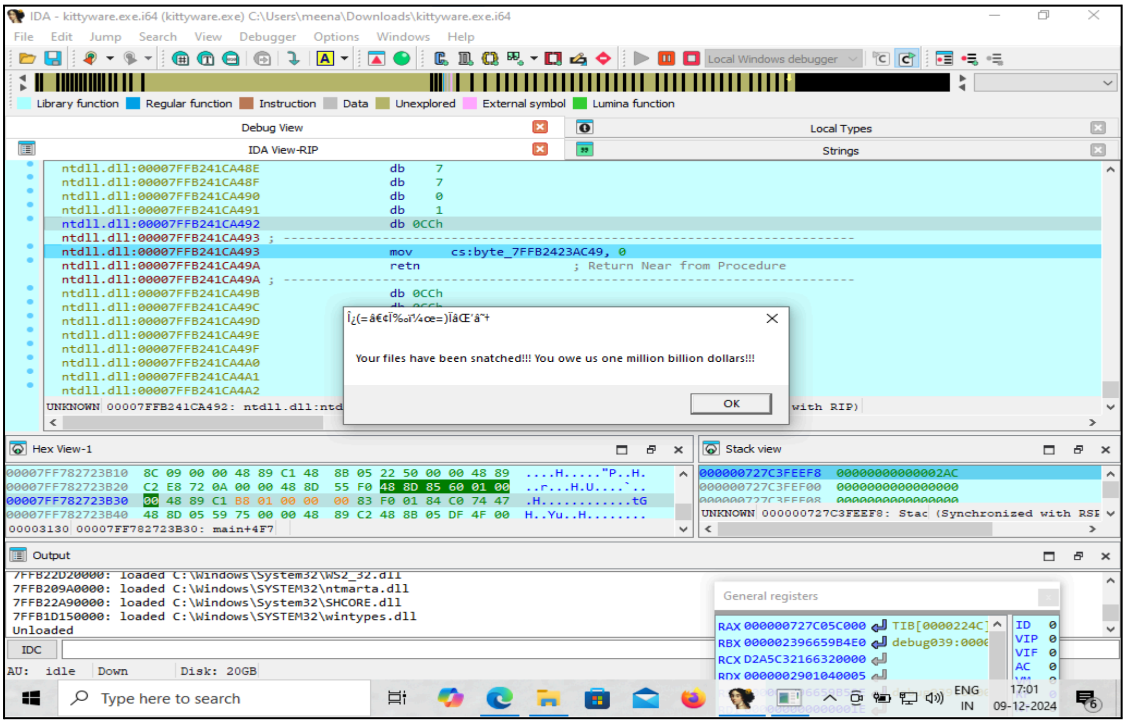
Following the change in desktop background, the malware modifies registry keys by adding the malware into the AutoRun registry key to gain persistence.



Screenshot of malware's entry in Registry Editor

Message Box

- Once the string is encrypted, the messagebox pops up on top of ‘GetForegroundWindow’ which denotes the string ‘Your files have been snatched!!! You owe us one million billion dollars!!!’ as shown in the below screenshot.
- This pop up is for intimidating the user to make them pay the amount asked, in return to either decrypt the files (string in this case) or handover the key for decrypting. Hence, this malware can also be termed as Ransomware.



Security Mitigations to evade the ‘kittyware.exe’ Malware

Category	Technical Mitigation	Description
System-Level Protections	Registry Monitoring and Hardening	Enable policies to monitor and restrict changes to critical registry keys (e.g., auto-run keys).
	Application Whitelisting	Use tools like Microsoft AppLocker or Bit9 to allow only approved executables to run.
	Memory Protections	Enable DEP (Data Execution Prevention) and ASLR (Address Space Layout Randomization) to prevent memory-based exploits.
	Anti-Debugging Evasion	Patch common anti-debugging API responses (e.g., IsDebuggerPresent, CheckRemoteDebuggerPresent).
	Disable TLS Callbacks	Modify the PE file structure to nullify malicious use of TLS callbacks during static analysis.

Incident Handling and Recovery	Patch Anti-Debugging Code	Modify assembly code to disable anti-debugging checks, e.g., replacing call rax with xor eax, eax.
Network-Level Protections	Firewall Rules	Configure firewalls to block outbound traffic to suspicious or untrusted domains and IP addresses.
	DNS Filtering	Use DNS filtering solutions to block connections to known malicious domains.
	Proxy for Internet Access	Route all external network traffic through a secure proxy to inspect and filter malicious activity.
	Sandbox Network Activity	Isolate suspicious executables in a virtual network environment for detailed inspection.
File and Payload Protections	File Integrity Monitoring	Deploy tools like Tripwire to monitor changes in critical system files and directories.
	Detect Malicious Behavior	Use tools like Sysmon to log suspicious file creation, registry modification, or process execution activities.
	Digital Signature Enforcement	Only allow execution of files signed with trusted digital certificates.
Advanced Detection Techniques	Behavioral Analysis	Use behavioral monitoring tools to detect abnormal process execution, memory allocation, or registry changes.
	Anti-Ransomware Tools	Deploy specialized tools that monitor for ransomware-like behaviors, such as rapid file encryption.
	Memory Analysis	Perform memory analysis to detect and dump malicious in-memory operations or payloads.

Conclusion

'Kittyware.exe' is a ransomware that combines various malicious techniques to achieve persistence, extort the victim, and manipulate system settings. It ensures auto-execution by creating a registry entry and delivers its payload by downloading an image file from a remote server. The malware modifies the desktop wallpaper and displays a ransom message, emphasizing its extortion intent. Anti-analysis features, such as anti-disassembly and anti-debugging, are used to complicate reverse engineering. Additionally, Kittyware employs encryption and decryption to safeguard its data and payload, decrypting them during runtime.