Walkthrough

!!! Spoilers Ahead !!!

This walkthrough contains **spoilers and detailed instructions**.

If you want detailed instructions, please proceed to the next page.

Analysis of the Dropper: `procmon64` Step 1

1. **Right-click** the dropper and examine its **properties/description**.
2. Rename the dropper as `w.exe`
3. Start `procmon64` and add a filter where the **Process Name equals `w.exe`**.
4. **Execute** the dropper
5. Filter the `procmon64` log to identify **where the files are dropped and how they are executed**.

Analysis of the Dropped Files

Analyze the `msedgewebview4.exe` file

1. **Open in `pe-bear`**:
   - Examine the **imports**, **strings**, and **sections**. Does anything look suspicious?
2. **Open in `IDAFree`**:
   - **Understand the general layout**:
     - Find the function that **reads and loads the file**.
     - Find the function that **loops over the encrypted file content** and determine **where the encryption key is stored**.
   - **Understand how the encryption is implemented**:
     - Analyze the **XOR loop** (which suggests a **Pseudo-Random Number Generator (PRNG)**).
     - Find the **encryption key** (which serves as the **seed for the PRNG**).

Analyze the `SearchHost.bin` file

1. **Decrypt the bytecode** using `decrypt_bytecode.py` with the **correct key (seed)**.
2. **Check if it matches a well-known architecture** using `find_shellcode_arch.py`:
   - **What heuristic** does `find_shellcode_arch.py` implement?
3. **Open the shellcode in `ghidra`** (specifying the **correct architecture**):
   - Analyze the **API resolution function**:
     - Use `hash_x65599_exports.py` to **identify which APIs are imported** (e.g., `kernelbase!VirtualProtect`, `kernel32!CreateThread`).
     - Determine **how the API is used**.
   - Analyze the **main loop function**.

Toward the Second Stage: `procmon64` Step 2

1. **Continue monitoring with `procmon64`**:
   - Identify the **APIs that are invoked** by analyzing the **stack trace**.
   - Look for `GetComputerName` and `GetUserNameW` in the stack trace.
2. **Create a user**:

   ```
   net user <username> <pass> /add
   ```
3. **Add the user to the local administrators group** (optional):

   ```
   net localgroup administrators <username> /add
   ```
4. **copy the dropped files** into `c:\windows\temp` and **execute the dropper under the new user's credentials**:

   ```
   cd c:\windows\temp
   Start-Process powershell.exe -ArgumentList "-Command & { Start-Process msedgewebview4.e
   ```
5. **Attach to the process with `windbg`** (**activate all user processes view** in windbg - requires elevation):
   - Place a **breakpoint** on `kernelbase!VirtualProtect`.
   - Wait for the breakpoint to **hit**. Look at the stack trace with `kb`. **Identify the parameters**; we expect **`r8` to equal `0x40` (Read-Write-Execute - RWX)**.
   - **Execute `VirtualProtect`** and open `SystemInformer`. Find the process and look for the **RWX allocated memory regions**. **Dump the content** and **trim the dump** properly using a hex editor.
   - Greetings! You've found **Stage 2**!

Analysis of the stage-2

Stage-2 Malware Analysis

This section details the analysis of `stage-2.dll`, focusing on static and initial dynamic examination to understand its loading mechanism and primary function.

Static Analysis and Initial Triage

1. **File Triage with PE-bear:**
   - Load `stage-2.dll` into a PE viewer like **PE-bear**.
   - Examine the **PE Headers** and sections.
   - Analyze **Import Address Table (IAT)** and **Export Address Table (EAT)** to identify its dependencies and entry points.
   - Review embedded **Strings** for immediate indicators (e.g., file paths, domain names, API calls, custom error messages).

2. **Disassembly and Initial Code Review (IDA-Free):**
   - Open `stage-2.dll` in a disassembler (e.g., **IDA-Free**). Initial inspection suggests it is a **COM DLL** based on its exported functions.
   - **Review Standard COM Exports:**
     - Examine `DllRegisterServer` and `DllUnregisterServer` to confirm their conventional structure or look for malicious deviations.
   - **Identify Loader Invocation:**
     - Note that the **stage-1 loader** specifically invokes **Ordinal 1**, which corresponds to the export `DllGetClassObject`.

Reflective Loading Mechanism Discovery

3. Analysis of `DllGetClassObject` (Ordinal 1)

- **Unconventional implementation:** Inspect the control flow of `DllGetClassObject`. It does **not** follow the standard COM pattern.
- **Key indicators of reflective loading:** Note the presence of magic constants such as `0x6A4ABC5B`, `0x3CFA685D`, and `0x534C0AB8`.
- **Verification:** A quick search (Google or an LLM) shows these constants are commonly associated with reflective loader implementations (e.g., Metasploit-style reflective DLL injection). This strongly suggests a reflective loader is present.

4. Reflective loader execution flow

- The reflective loader maps the DLL into memory and transfers execution to its internal entry point.
- The reflective loader eventually calls `DllEntryPoint`, which is the **true main** for the stage-2 payload.

Payload Execution Flow

5. `DllEntryPoint` analysis

- **Thread creation:** `DllEntryPoint` creates a new thread to run a function we label `start_function`. The loader waits on that thread. This detaches payload execution from the loader and helps evade analysis.

6. `start_function` dissection

- **API resolution:** One of the first tasks of `start_function` is a PEB walk to dynamically resolve required APIs. This avoids static imports and complicates static analysis.

Key derivation and 3DES decryption chain

7. Important function: `sub_7FFC86652680`

- `sub_7FFC86652680` takes as input a `flink` (a DLL pointer) and a pointer to `&unk_7FFC86655370` (a constant).
- Nested calls reveal this repeated pattern:

```
LOBYTE(n8_2) = 101;
v24 = sub_7FFC86651500(v16, *a2, n8_2);
LOBYTE(n8_3) = 100;
v26 = sub_7FFC86651500(v24, a2[1], n8_3);
LOBYTE(n8_4) = 101;
v28 = sub_7FFC86651500(v26, a2[2], n8_4);
n8_10 += 8;
```

This structure is distinctive and suggests a multi-pass block cipher application.

8. Byte table used by `sub_7FFC86651500` performs mathematical transformations using the following byte array:

```
.byte_7FFC86655000: db 3Ah, 32h, 2Ah, 22h, 1Ah, 12h, 0Ah, 2, 3Ch, 34h, 2Ch
.byte_7FFC8665500B: db 24h, 1Ch, 14h, 0Ch, 4, 3Eh, 36h, 2Eh, 26h, 1Eh, 16h
.byte_7FFC86655016: db 0Eh, 6, 40h, 38h, 30h, 28h, 20h, 18h, 10h, 8, 39h, 31h
.byte_7FFC86655022: db 29h, 21h, 19h, 11h, 9, 1, 3Bh, 33h, 2Bh, 23h, 1Bh, 13h
```

Correlating the constants and call structure reveals DES-like S-box / permutation tables. The calling sequence implements 3DES (Triple DES) in ECB mode (no IV). Block size is 8 bytes (64 bits).

9. Key size and origin The key length is 24 bytes, matching the size of a global variable passed to:

```
sub_7FFC86652D10(byte_7FFC86655A80);
```

`sub_7FFC86652D10` generates a random value and performs operations on a specific DLL; therefore the key can be recovered:

- *statically* (by reversing the generation routine)
- *dynamically* (by using the debugger to read the generated bytes).

To speed up the process, this walkthrough uses the dynamic approach (debugger).

Debugger setup & dynamic key recovery (IDA)

10. **Debugger setup (IDA)**: In IDA's Debugger options, set the default local debugger. In Process options, set the target application to:

```
C:\Windows\System32\rundll32.exe
```

As parameters, pass the renamed sample DLL and a dummy exported ordinal, e.g.:

```
C:\Users\user\2stage\2stage.dll, #2
```

Place a breakpoint on the first instruction of the DLL main (press `F2` on that instruction).

11. **Run and advance to start_function**: Launch the debugger (green arrow) and wait the execution breaks at DLL main.
12. **Key discovery (dynamic approach)**: Open `sub_7FFC86652D10` and set the IP to its start (use Ctrl+N or "Set IP") Follow the execution till an instruction immediately after it writes the global key (e.g., `byte_7FFC86655A80`).

The key is derived from opcodes of a randomly chosen export (e.g., from `ntdll`). In our sample the extracted 24-byte key was:

```
0000DEADB8D18B4C017FDEAD082504F6C32EDEAD050F0375 (hex representation)
```

Read memory at the global key address and dump 24 bytes. Format them as hex: that is your 3DES key `(K1||K2||K3)`.

13. **Confirming the decryption chain**: Cross-reference the decryption routine (`sub_7FFC86652680` and callers). All `unk_*` values referenced by callers correspond to encrypted blobs in the stage-2 payload.

```
Use CyberChef or any 3DES tool:
Algorithm: Triple DES (3DES)
Mode: ECB (no IV)
Key: 24-byte key recovered (hex)
Input format: Hex (raw ciphertext), e.g. 281a5261479d3d42087515567ec409c3
```

Decrypt all encrypted blobs. The decrypted outputs reveal stage-2 strings and behavior. One buffer at `0x00007FFC86655450` should be converted to code in IDA (press C).