**1. Utilizing Debugging Tricks for Unpacking Malware**

Refer to the technique outlined in "M11-1-Debugging Tricks for Unpacking Malware". Note that Ass4-1 is packed using UPX. You'll need to use a hardware breakpoint to identify the original malware payload's address before it was packed. Unzip Ass4-1 and rename the extracted file to include a .exe extension prior to debugging.

* **Q1-1**: Where did you set the hardware breakpoint, and why? Include a snapshot of your breakpoint. (1.5 points)

A white rectangular object with black border

Description automatically generated

^Location of the hardware breakpoint. The location of the breakpoint was set there due to how it represents the start of the program at PUSHAD. This is after following the dump within the ESP register, allowing me to put the break at the entry point of the packed sample itself for analysis -after stepping into the PUSHAD instruction-. This is also basically due to how the entry point of the packed binary starts at the PUSHAD instruction.

* **Q1-2**: At which instruction does the debugger halt due to the hardware breakpoint? Provide an explanation and a snapshot. (1.5 points)

A screenshot of a computer

Description automatically generated

^Location on where the debugger halted after running (after the hardware breakpoint was set). This has finished at this location right after the POPAD instruction due to how it finished decompressing the payload and will be shortly jumping into the OEP.

* **Q1-3**: What is the jump address? Include a snapshot. (1.5 points)

A screenshot of a computer

Description automatically generated

^Location of the JMP Address after the POPAD instruction.

* **Q1-4**: What is the first instruction following the jump execution? (1.5 points)

A screenshot of a computer

Description automatically generated

^First instruction following the JMP Execution.

This was found through the “step into” function which allows for further exploration into the call/function.

**2. Identifying the Injection Entry Point**

Using the approach from "M12-1-Debugging Code Injection", determine the injection entry point for Ass4-2. Remember to add a .exe extension to Ass4-2 before beginning the debugging process.

* **Q2-1**: On which instructions did you place breakpoints, and why? Provide a snapshot of your list of breakpoints. (1.5 points)

A screenshot of a computer

Description automatically generated

^List of instructions on which the breakpoints were setup.

This is due to the fact that in relation to Code Injection, a new process is generally made where memory is allocated for the process to be written into -with malicious code- which would then create a remote thread for the program/process to be ran in suspended mode to hide malicious intent.

* **Q2-2**: What is the target process ID, and how did you determine it? Include a snapshot. (1.5 points)

A screenshot of a computer

Description automatically generated^The target process ID.

This is determined by running the program after setting up the breakpoint within the Open Process API block.

* **Q2-3**: What is the address of the allocated memory block? Provide a snapshot. (1.5 points)

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

^The address of the allocated memory block is at 0. This is due to the handler, EAX, from the Open Process API which would have been sent to the VirtualAlloc API block being 0 <- address for the allocation of the memory block.

* **Q2-4**: What are the first 4 bytes of the injected buffer? Include a snapshot. (1.5 points)

A screenshot of a computer

Description automatically generated

^Viewing the WriteProcessMemory API block.

A screenshot of a computer

Description automatically generated

^Following the EAX registry buffer as revealed from the slides when looking into the code injection.

A screenshot of a computer screen

Description automatically generated

^First 4 bytes of the injected buffer This is due to the fact that the bytes originate from the WriteProcessMemory API block which is responsible for injecting the code itself.