Accessing PEB and TEB in Assembly and C++

Use Case in Malware Development

By Reda Ouzidane

Introduction

The **TEB** (Thread Environment Block) and **PEB** (Process Environment Block) are Windows internal structures that store metadata about threads and processes. Although undocumented in the official Windows API, they are vital in reverse engineering, malware development, anti-debugging, and evasion techniques.

In this document, we explore how to access these structures using Assembly and C++, and discuss their relevance in the context of malware development.

Accessing the TEB in Assembly

Assembly Code (getTEB.asm)

```
.code
getTEB proc
    mov rax, gs:[0x30]     ; Get pointer to TEB (Thread Environment
        Block)
    ret
getTEB endp
end
```

Listing 1: Retrieve TEB Pointer

Explanation

- ‘gs:[0x30]‘ holds the address of the TEB in 64-bit Windows. - This function returns the TEB pointer via ‘RAX‘.

C++ Integration for TEB

```
#include <windows.h>
#include <stdio.h>

extern "C" PVOID getTEB(void);

int main() {
    PVOID pTEB = getTEB();
    printf("[*] TEB Address: 0x%p\n", pTEB);
    return 0;
}
```

Listing 2: C++ Code Using getTEB()

Use Case

- Debugging thread-local data - Retrieving LastErrorValue - Implementing custom error handlers

Custom Error with TEB

Assembly: CustomError.asm

```
.code
CustomError proc
    xor eax, eax
    call getTEB
    mov eax, dword ptr[rax + 0x68] ; Offset for LastErrorValue
    ret
CustomError endp
end
```

Listing 3: Fetch Last Error via TEB

Explanation

- Retrieves the ‘LastErrorValue‘ stored in the TEB at offset ‘0x68‘.

Accessing the PEB in Assembly

Assembly Code (getPEB.asm)

```
.code
getPEB proc
    mov rax, gs:[0x30]       ; Get TEB
    mov rax, [rax + 0x60]    ; Get PEB from TEB
    ret
getPEB endp
end
```

Listing 4: Retrieve PEB Pointer

Explanation

- TEB is at ‘gs:[0x30]‘ - PEB is at offset ‘0x60‘ from TEB in 64-bit processes

C++ Integration for PEB

```
#include <windows.h>
#include <stdio.h>

extern "C" PVOID getPEB(void);

int main() {
    PVOID pPEB = getPEB();
    printf("[*] PEB Address: 0x%p\n", pPEB);
    return 0;
}
```

Listing 5: C++ Code Using getPEB()

Optional: Minimal PEB Structure

If the default headers don’t define the PEB, you can manually define a minimal one:

```
typedef struct _PEB {
    BYTE Reserved1[2];
    BYTE BeingDebugged;
    BYTE Reserved2[1];
    PVOID Reserved3[2];
    PVOID ImageBaseAddress;
} PEB, *PPEB;
```

Listing 6: Minimal PEB Definition

Why Is This Useful in Malware Development?

- **Anti-Debugging:** Accessing the ‘BeingDebugged‘ flag from PEB helps detect if a debugger is attached.
- **Evasion:** Custom error handling without using WinAPI avoids triggering antivirus hooks.
- **Stealth:** Bypassing Windows API reduces visibility to monitoring tools and EDRs.
- **Manual Mapping:** Accessing the ImageBaseAddress from the PEB is helpful when parsing PE headers in custom loaders or shellcode.

Conclusion

Accessing the TEB and PEB manually gives developers low-level control and insight into thread/process execution. While primarily used in advanced debugging and reverse engineering, it is also critical in stealthy malware design. Understanding these internals is a fundamental step in mastering Windows internals, red teaming, and defensive security.