# A blueprint for evading industry leading endpoint protection in 2022

vanmieghem.io/blueprint-for-evading-edr-in-2022/

vivami April 18, 2022

- Home
- Blog
- Projects

Monday. April 18, 2022 - 19 mins
About two years ago I quit being a full-time red team operator. However, it still is a field of expertise that stays very close to my heart. A few weeks ago, I was looking for a new side project and decided to pick up an old red teaming hobby of mine: bypassing/evading endpoint protection solutions.

In this post, I'd like to lay out a collection of techniques that together can be used to bypassed industry leading enterprise endpoint protection solutions. This is purely for educational purposes for (ethical) red teamers and alike, so I've decided not to publicly release the source code. The aim for this post is to be accessible to a wide audience in the security industry, but not to drill down to the nitty gritty details of every technique. Instead, I will refer to writeups of others that deep dive better than I can.

In adversary simulations, a key challenge in the "initial access" phase is bypassing the detection and response capabilities (EDR) on enterprise endpoints. Commercial command and control frameworks provide unmodifiable shellcode and binaries to the red team operator that are heavily signatured by the endpoint protection industry and in order to execute that implant, the signatures (both static and behavioural) of that shellcode need to be obfuscated.

In this post, I will cover the following techniques, with the ultimate goal of executing malicious shellcode, also known as a (shellcode) loader:

1. Shellcode encryption
2. Reducing entropy
3. Escaping the (local) AV sandbox
4. Import table obfuscation
5. Disabling Event Tracing for Windows (ETW)

6. Evading common malicious API call patterns
7. Direct system calls and evading “mark of the syscall”
8. Removing hooks in `ntdll.dll`
9. Spoofing the thread call stack
10. In-memory encryption of beacon
11. A custom reflective loader
12. OpSec configurations in your Malleable profile

## 1. Shellcode encryption

Let’s start with a basic but important topic, static shellcode obfuscation. In my loader, I leverage a XOR or RC4 encryption algorithm, because it is easy to implement and doesn’t leave a lot of external indicators of encryption activities performed by the loader. AES encryption to obfuscate static signatures of the shellcode leaves traces in the import address table of the binary, which increase suspicion. I’ve had Windows Defender specifically trigger on AES decryption functions (e.g. `CryptDecrypt`, `CryptHashData`, `CryptDeriveKey` etc.) in earlier versions of this loader.

```
ADVAPI32.dll
        140004000 Import Address Table
        140005930 Import Name Table
                0 time date stamp
                0 Index of first forwarder reference

                    C2 CryptAcquireContextW
                    C6 CryptDeriveKey
                    C7 CryptDestroyHash
                    D9 CryptHashData
                    C8 CryptDestroyKey
                    C4 CryptCreateHash
                    C5 CryptDecrypt
                    DC CryptReleaseContext
```

Output of dumpbin /imports, an easy giveaway of only AES decryption functions being used in the binary.

## 2. Reducing entropy

Many AV/EDR solutions consider binary entropy in their assessment of an unknown binary. Since we’re encrypting the shellcode, the entropy of our binary is rather high, which is a clear indicator of obfuscated parts of code in the binary.

There are several ways of reducing the entropy of our binary, two simple ones that work are:

1. Adding low entropy resources to the binary, such as (low entropy) images.
2. Adding strings, such as the English dictionary or some of `"strings C:\Program Files\Google\Chrome\Application\100.0.4896.88\chrome.dll"` output.

A more elegant solution would be to design and implement an algorithm that would obfuscate (encode/encrypt) the shellcode into English words (low entropy). That would kill two birds with one stone.

## 3. Escaping the (local) AV sandbox

Many EDR solutions will run the binary in a local sandbox for a few seconds to inspect its behaviour. To avoid compromising on the end user experience, they cannot afford to inspect the binary for longer than a few seconds (I've seen Avast taking up to 30 seconds in the past, but that was an exception). We can abuse this limitation by delaying the execution of our shellcode. Simply calculating a large prime number is my personal favourite. You can go a bit further and deterministically calculate a prime number and use that number as (a part of) the key to your encrypted shellcode.

## 4. Import table obfuscation

You want to avoid suspicious Windows API (WINAPI) from ending up in our IAT (import address table). This table consists of an overview of all the Windows APIs that your binary imports from other system libraries. A list of suspicious (oftentimes therefore inspected by EDR solutions) APIs can be found here. Typically, these are `VirtualAlloc`, `VirtualProtect`, `WriteProcessMemory`, `CreateRemoteThread`, `SetThreadContext` etc. Running `dumpbin /exports <binary.exe>` will list all the imports. For the most part, we'll use Direct System calls to bypass both EDR hooks (refer to section 7) of suspicious WINAPI calls, but for less suspicious API calls this method works just fine.

We add the function signature of the WINAPI call, get the address of the WINAPI in `ntdll.dll` and then create a function pointer to that address:

```
typedef BOOL (WINAPI * pVirtualProtect)(LPVOID lpAddress, SIZE_T dwSize, DWORD
flNewProtect, PDWORD lpflOldProtect);
pVirtualProtect fnVirtualProtect;

unsigned char sVirtualProtect[] = {
'V','i','r','t','u','a','l','P','r','o','t','e','c','t', 0x0 };
unsigned char sKernel32[] = { 'k','e','r','n','e','l','3','2','.','d','l','l', 0x0 };

fnVirtualProtect = (pVirtualProtect) GetProcAddress(GetModuleHandle((LPCSTR)
sKernel32), (LPCSTR)sVirtualProtect);
// call VirtualProtect
fnVirtualProtect(address, dwSize, PAGE_READWRITE, &oldProt);
```

Obfuscating strings using a character array cuts the string up in smaller pieces making them more difficult to extract from a binary.

The call will still be to an `ntdll.dll` WINAPI, and will not bypass any hooks in WINAPIs in `ntdll.dll`, but is purely to remove suspicious functions from the IAT.

## 5. Disabling Event Tracing for Windows (ETW)

Many EDR solutions leverage Event Tracing for Windows (ETW) extensively, in particular Microsoft Defender for Endpoint (formerly known as Microsoft ATP). ETW allows for extensive instrumentation and tracing of a process' functionality and WINAPI calls. ETW has components in the kernel, mainly to register callbacks for system calls and other kernel operations, but also consists of a userland component that is part of `ntdll.dll` (ETW deep dive and attack vectors). Since `ntdll.dll` is a DLL loaded into the process of our binary, we have full control over this DLL and therefore the ETW functionality. There are quite a few different bypasses for ETW in userspace, but the most common one is patching the function `EtwEventWrite` which is called to write/log ETW events. We fetch its address in `ntdll.dll`, and replace its first instructions with instructions to return 0 ( `SUCCESS` ).

```
void disableETW(void) {
	// return 0
	unsigned char patch[] = { 0x48, 0x33, 0xc0, 0xc3};	// xor rax, rax; ret

	ULONG oldprotect = 0;
	size_t size = sizeof(patch);

	HANDLE hCurrentProc = GetCurrentProcess();

	unsigned char sEtwEventWrite[] = {
'E','t','w','E','v','e','n','t','W','r','i','t','e', 0x0 };

	void *pEventWrite = GetProcAddress(GetModuleHandle((LPCSTR) sNtdll), (LPCSTR)
sEtwEventWrite);

	NtProtectVirtualMemory(hCurrentProc, &pEventWrite, (PSIZE_T) &size,
PAGE_READWRITE, &oldprotect);

	memcpy(pEventWrite, patch, size / sizeof(patch[0]));

	NtProtectVirtualMemory(hCurrentProc, &pEventWrite, (PSIZE_T) &size,
oldprotect, &oldprotect);
	FlushInstructionCache(hCurrentProc, pEventWrite, size);

}
```

I've found the above method to still work on the two tested EDRs, but this is a noisy ETW patch.

## 6. Evading common malicious API call patterns

Most behavioural detection is ultimately based on detecting malicious patterns. One of these patters is the order of specific WINAPI calls in a short timeframe. The suspicious WINAPI calls briefly mentioned in section 4 are typically used to execute shellcode and therefore heavily monitored. However, these calls are also used for benign activity (the `VirtualAlloc`, `WriteProcess`, `CreateThread` pattern in combination with a memory

allocation and write of ~250KB of shellcode) and so the challenge for EDR solutions is to distinguish benign from malicious calls. Filip Olszak wrote a great blog post leveraging delays and smaller chunks of allocating and writing memory to blend in with benign WINAPI call behaviour. In short, his method adjusts the following behaviour of a typical shellcode loader:

1. Instead of allocating one large chuck of memory and directly write the ~250KB implant shellcode into that memory, allocate small contiguous chunks of e.g. <64KB memory and mark them as `NO_ACCESS`. Then write the shellcode in a similar chunk size to the allocated memory pages.
2. Introduce delays between every of the above mentioned operations. This will increase the time required to execute the shellcode, but will also make the consecutive execution pattern stand out much less.

One catch with this technique is to make sure you find a memory location that can fit your entire shellcode in consecutive memory pages. Filip's DripLoader implements this concept.

The loader I've built does not inject the shellcode into another process but instead starts the shellcode in a thread in its own process space using `NtCreateThread`. An unknown process (our binary will de facto have low prevalence) into other processes (typically a Windows native ones) is suspicious activity that stands out (recommended read "Fork&Run – you're history"). It is much easier to blend into the noise of benign thread executions and memory operations within a process when we run the shellcode within a thread in the loader's process space. The downside however is that any crashing post-exploitation modules will also crash the process of the loader and therefore the implant. Persistence techniques as well as running stable and reliable BOFs can help to overcome this downside.

## 7. Direct system calls and evading "mark of the syscall"

The loader leverages direct system calls for bypassing any hooks put in `ntdll.dll` by the EDRs. I want to avoid going into too much detail on how direct syscalls work, since it's not the purpose of this post and a lot of great posts have been written about it (e.g. Outflank).

In short, a direct syscall is a WINAPI call directly to the kernel system call equivalent. Instead of calling the `ntdll.dll` `VirtualAlloc` we call its kernel equivalent `NtAlocateVirtualMemory` defined in the Windows kernel. This is great because we're bypassing any EDR hooks used to monitor calls to (in this example) `VirtualAlloc` defined in `ntdll.dll`.

In order to call a system call directly, we fetch the syscall ID of the system call we want to call from `ntdll.dll`, use the function signature to push the correct order and types of function arguments to the stack, and call the `syscall <id>` instruction. There are several tools that arrange all this for us, SysWhispers2 and SysWhisper3 are two great examples. From an evasion perspective, there are two issues with calling direct system calls:

1. Your binary ends up with having the `syscall` instruction, which is easy to statically detect (a.k.a “mark of the syscall”, more in “SysWhispers is dead, long live SysWhispers!”).
2. Unlike benign use of a system call that is called through its `ntdll.dll` equivalent, the return address of the system call does not point to `ntdll.dll`. Instead, it points to our code from where we called the syscall, which resides in memory regions outside of `ntdll.dll`. This is an indicator of a system call that is not called through `ntdll.dll`, which is suspicious.

To overcome these issues we can do the following:

1. Implement an egg hunter mechanism. Replace the `syscall` instruction with the `egg` (some random unique identifiable pattern) and at runtime, search for this `egg` in memory and replace it with the `syscall` instruction using the `ReadProcessMemory` and `WriteProcessMemory` WINAPI calls. Thereafter, we can use direct system calls normally. This technique has been implemented by klezVirus.
2. Instead of calling the `syscall` instruction from our own code, we search for the `syscall` instruction in `ntdll.dll` and jump to that memory address once we’ve prepared the stack to call the system call. This will result in an return address in RIP that points to `ntdll.dll` memory regions.

Both techniques are part of SysWhisper3.

## 8. Removing hooks in `ntdll.dll`

Another nice technique to evade EDR hooks in `ntdll.dll` is to overwrite the loaded `ntdll.dll` that is loaded by default (and hooked by the EDR) with a fresh copy from `ntdll.dll`. `ntdll.dll` is the first DLL that gets loaded by any Windows process. EDR solutions make sure their DLL is loaded shortly after, which puts all the hooks in place in the loaded `ntdll.dll` before our own code will execute. If our code loads a fresh copy of `ntdll.dll` in memory afterwards, those EDR hooks will be overwritten. RefleXXion is a C++ library that implements the research done for this technique by MDSec. RelfeXXion uses direct system calls `NtOpenSection` and `NtMapViewOfSection` to get a handle to a clean `ntdll.dll` in `\KnownDlls\ntdll.dll` (registry path with previously loaded DLLs). It then overwrites the `.TEXT` section of the loaded `ntdll.dll`, which flushes out the EDR hooks.

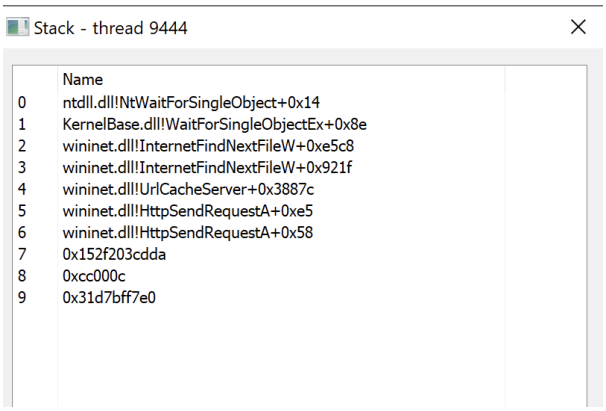
I recommend to use adjust the RefleXXion library to use the same trick as described above in section 7.

## 9. Spoofing the thread call stack

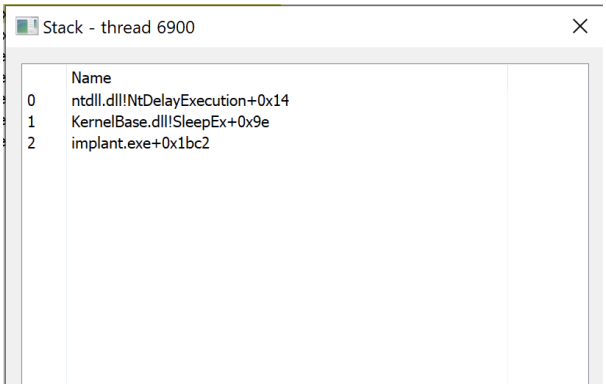
The next two sections cover two techniques that provide evasions against detecting our shellcode in memory. Due to the beaconing behaviour of an implant, for a majority of the time the implant is sleeping, waiting for incoming tasks from its operator. During this time the implant is vulnerable for memory scanning techniques from the EDR. The first of the two evasions described in this post is spoofing the thread call stack.

When the implant is sleeping, its thread return address is pointing to our shellcode residing in memory. By examining the return addresses of threads in a suspicious process, our implant shellcode can be easily identified. In order to avoid this, want to break this connection between the return address and shellcode. We can do so by hooking the `Sleep()` function. When that hook is called (by the implant/beacon shellcode), we overwrite the return address with `0x0` and call the original `Sleep()` function. When `Sleep()` returns, we put the original return address back in place so the thread returns to the correct address to continue execution. Mariusz Banach has implemented this technique in his ThreadStackSpoofer project. This repo provides much more detail on the technique and also outlines some caveats.

We can observe the result of spoofing the thread call stack in the two screenshots below, where the non-spoofed call stack points to non-backed memory locations and a spoofed thread call stack points to our hooked Sleep ( `MySleep` ) function and "cuts off" the rest of the call stack.




Default beacon thread call stack.




Spoofed beacon thread call stack.

## 10. In-memory encryption of beacon

The other evasion for in-memory detection is to encrypt the implant's executable memory regions while sleeping. Using the same sleep hook as described in the section above, we can obtain the shellcode memory segment by examining the caller address (the beacon code that calls `Sleep()` and therefore our `MySleep()` hook). If the caller memory region is `MEM_PRIVATE` and `EXECUTABLE` and roughly the size of our shellcode, then the memory segment is encrypted with a XOR function and `Sleep()` is called. Then `Sleep()` returns, it decrypts the memory segment and returns to it.

Another technique is to register a Vectored Exception Handler (VEH) that handles `NO_ACCESS` violation exceptions, decrypts the memory segments and changes the permissions to `RX`. Then just before sleeping, mark the memory segments as `NO_ACCESS`, so that when `Sleep()` returns, it throws a memory access violation exception. Because we registered a VEH, the exception is handled within that thread context and can be resumed at the exact same location the exception was thrown. The VEH can simply decrypt and change the permissions back to RX and the implant can continue execution. This technique prevents a detectible `Sleep()` hook being in place when the implant is sleeping.

Mariusz Banach has also implemented this technique in ShellcodeFluctuation.

## 11. A custom reflective loader

The beacon shellcode that we execute in this loader ultimately is a DLL that needs to be executed in memory. Many C2 frameworks leverage Stephen Fewer's ReflectiveLoader. There are many well written explanations of how exactly a relfective DLL loader works, and Stephen Fewer's code is also well documented, but in short a Reflective Loader does the following:

1. Resolve addresses to necessary `kernel32.dll` WINAPIs required for loading the DLL (e.g. `VirtualAlloc`, `LoadLibraryA` etc.)
2. Write the DLL and its sections to memory
3. Build up the DLL import table, so the DLL can call `ntdll.dll` and `kernel32.dll` WINAPIs
4. Load any additional library's and resolve their respective imported function addresses
5. Call the DLL entrypoint

Cobalt Strike added support for a custom way for reflectively loading a DLL in memory that allows a red team operator to customize the way a beacon DLL gets loaded and add evasion techniques. Bobby Cooke and Santiago P built a stealthy loader (BokuLoader) using Cobalt Strike's UDRL which I've used in my loader. BokuLoader implements several evasion techniques:

- Limit calls to `GetProcAddress()` (commonly EDR hooked WINAPI call to resolve a function address, as we do in section 4)

- AMSI & ETW bypasses
- Use only direct system calls
- Use only `RW` or `RX`, and no `RWX` ( `EXECUTE_READWRITE` ) permissions
- Removes beacon DLL headers from memory

Make sure to uncomment the two defines to leverage direct system calls via HellsGate & HalosGate and bypass ETW and AMSI (not really necessary, as we've already disabled ETW and are not injecting the loader into another process).

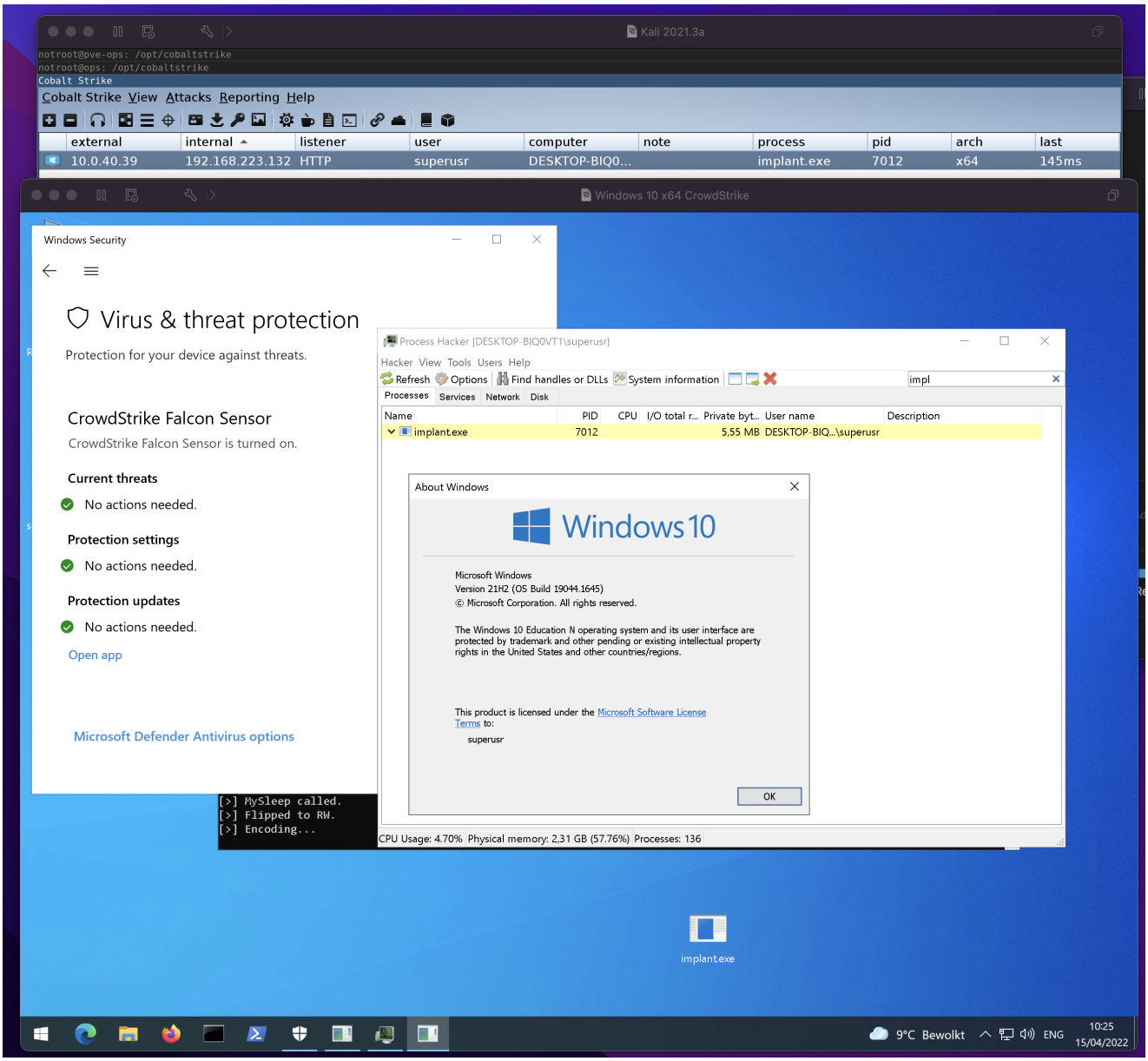
## 12. OpSec configurations in your Malleable profile

In your Malleable C2 profile, make sure the following options are configured, which limit the use of `RWX` marked memory (suspicious and easily detected) and clean up the shellcode after beacon has started.

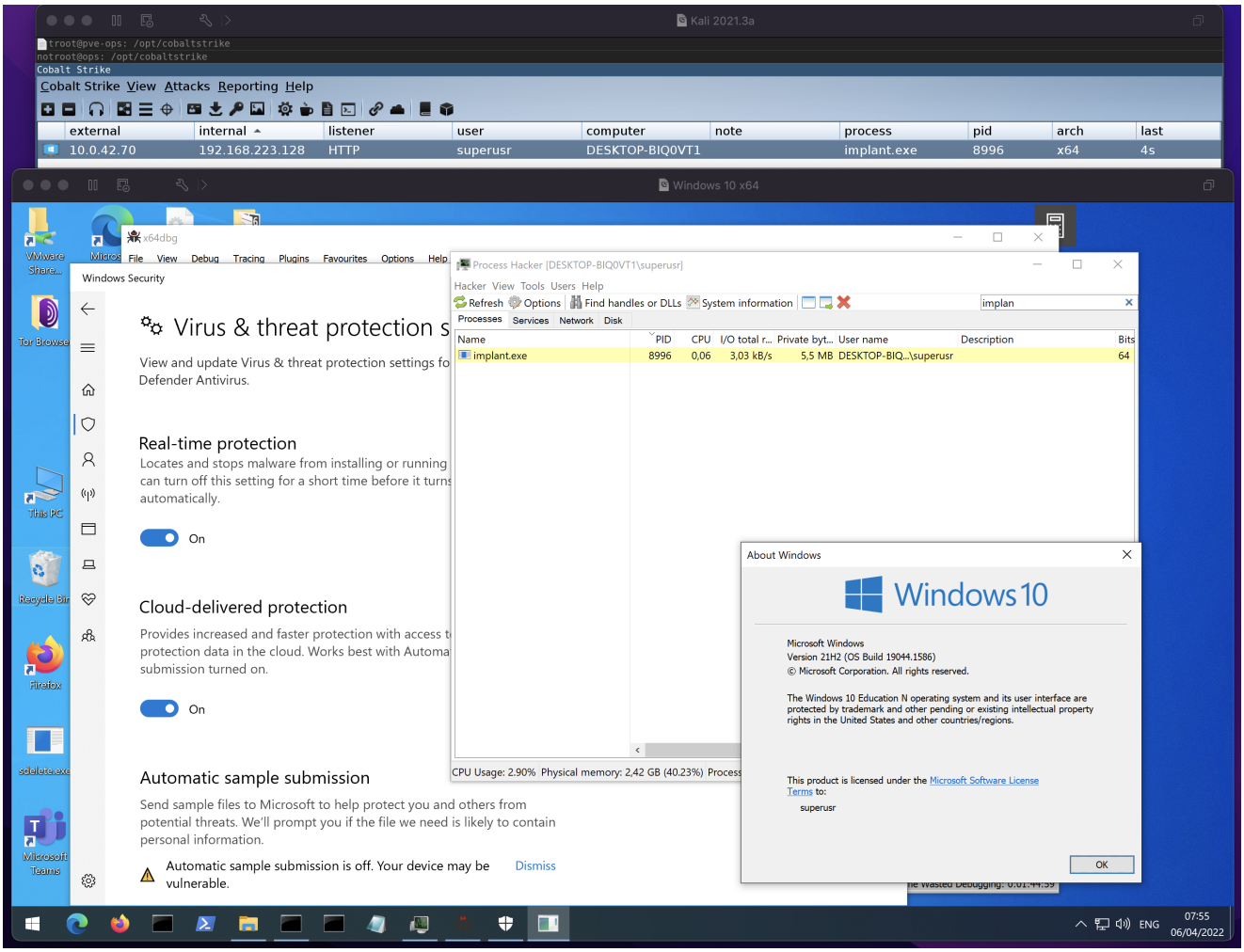
```
set startrwx        "false";
set userwx          "false";
set cleanup         "true";
set stomppe         "true";
set obfuscate       "true";
set sleep_mask      "true";
set smartinject     "true";
```

## Conclusions

Combining these techniques allow you to bypass (among others) Microsoft Defender for Endpoint and CrowdStrike Falcon with 0 detections (tested mid April 2022), which together with SentinelOne lead the endpoint protection industry.




CrowdStrike Falcon with 0 alerts.




Windows Defender (and also Microsoft Defender for Endpoint, not screenshotted) with 0 alerts.
Of course this is just one and the first step in fully compromising an endpoint, and this doesn't mean "game over" for the EDR solution. Depending on what post-exploitation activity/modules the red team operator choses next, it can still be "game over" for the implant. In general, either run BOFs, or tunnel post-ex tools through the implant's SOCKS proxy feature. Also consider putting the EDR hooks patches back in place in our `Sleep()` hook to avoid detection of unhooking, as well as removing the ETW/AMSI patches.

It's a cat and mouse game, and the cat is undoubtedly getting better.

## Related Posts

Towards generic .NET assembly obfuscation (Pt. 1)