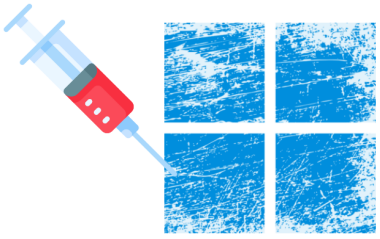
Process injection in 2023, evading leading EDRs


vanmieghem.io/process-injection-evading-edr-in-2023

vivami April 18, 2023

- Home
- Blog
- Projects



Tuesday. April 18, 2023 - 11 mins
Nowadays when I speak with my red team friends and touch upon the topic of process injection, the response is usually “Yes… but no…”. The risks of detection outweigh the need for having an implant “parasiting” in a host process. Typical process injection techniques stand out too much and more often than not is the injection linked to malicious activity. Occasionally, I like to pick-up this “AV evasion” hobby, and achieving process injection with arguably the most signatured malicious shellcode against today’s best endpoint protection, seemed like a fun exercise to me.

So in this blog post, we’ll walk through what combination of evasive techniques can be used to achieve process injection with zero detections or alerts.

My last year’s blog post is still relevant, and the techniques outlined there are used in this evasive loader as well. If you haven’t read that one (A blueprint for evading industry leading endpoint protection in 2022), I recommend reading that first. On top of those techniques, this post will cover the techniques in a similar fashion: no source code, but a blueprint of what readily available code can be glued together to achieve our objective.

1. A custom version of `GetProcAddress()`

Many evasive techniques rely on the use of the `WINAPI` function `GetProcAddress()` to obtain the virtual memory address of functions. For example, a typical way to obtain the memory address of an `Nt*` function to execute direct system calls (as first demonstrated in Combining Direct System Calls and sRDI to bypass AV/EDR) is:

```
GetProcAddress(GetModuleHandle(L"ntdll.dll"), L"NtWriteProcessMemory");
```

Everything in the above line is a signature for detection. The combination of strings `"NtWriteProcessMemory"` and `"ntdll.dll"` are suspicious, especially as arguments of `GetModuleHandle` and `GetProcAddress`. Both are used to resolve the memory location of a function, aiming to bypass the use of the exported (and potentially) API functions. Nowadays, `GetProcAddress` is a closely monitored function.

To evade both of these detection techniques, we can use our own implementation of `GetProcAddress()` that uses the process' `PEB` structure to obtain the memory address of (exported) functions in the executable. The `PEB` contains lots of information about the process, loaded modules (DLLs), functions etc., that we can read out ourselves to obtain their memory locations. Fetching the memory location of `ntdll.dll`, walking its `PEB` down to `AddressOfNames` gives us a list of `WINAPI`-function memory addresses which we can call directly. There are various implementations out there, most boil down to the same principle.

The strings we will obfuscate using previously described methods. You can also consider look-ups using hashes of API calls.

2. System calls using hardware breakpoints

This relatively new evasion technique to bypass hooks I first spotted in *@EthicalChaos*'s post In-Process Patchless AMSI Bypass. His post outlines how EDR hooks in `ntdll.dll` can be bypassed using hardware breakpoints and Vectored Exception Handlers (VEH), which avoid in-memory patching of `ntdll.dll` (indicator of malicious activity). The technique is fairly straight forward:

1. Register a VEH to handle the exception triggered by the breakpoint. VEHs are handled in the thread that raises the exception and the VEH has access to the corresponding thread context (including all registers).
2. Set a breakpoint on the memory address that you want to intercept execution for, i.e. `WINAPI NtWriteProcessMemory`. Setting the `DR7` register causes the OS to call the registered VEH.

```
GetThreadContext(myThread, &ctx);   /* get thread context */
ctx.Dr0 = (UINT64)&bp_addr;         /* address you want to break on */
ctx.Dr7 |= (1 << 0);                /* set first bit in DR7 */
ctx.Dr7 &= ~(1 << 16);              /* clear 16 an 17th bit */
ctx.Dr7 &= ~(1 << 17);
SetThreadContext(myThread, &ctx)    /* set the thread context, putting the
breakpoint in place */
```

3. From the VEH it's then possible to take over the control flow, and bypass the EDR hook.

@Dec0ne created HWSyscalls in which the above steps are implemented. In the VEH, it uses HalosGate to resolve the syscall number (SSN) when it detects the `WINAPI` is hooked (e.g. next instruction after the address in a `JMP` instruction). As a nice addition, HWSyscalls will point `RIP` (instruction pointer) to a `syscall; ret` instruction in `ntdll.dll`, making the return address (`RAX`) point back to `ntdll.dll` memory instead of directly from our loader's executable memory (indication of direct system calls).

HWSyscalls is an easy to integrate module. We'll use that in our loader.

3. Threadless injection

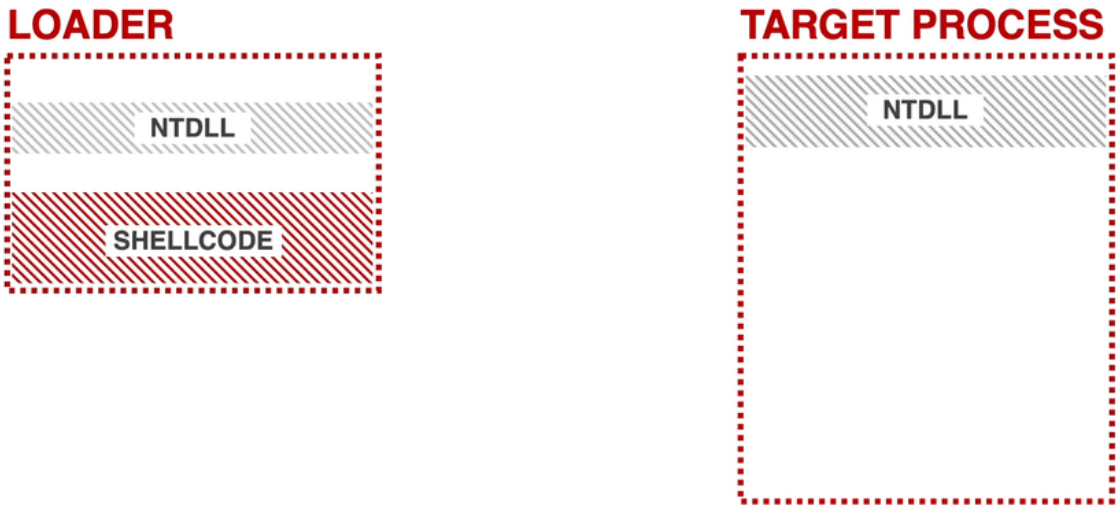
The next new technique, which is really the star of this loader, is Threadless Process Injection by *@EthicalChaos*. This technique only requires `VirtualAlloc`, `WriteProcessMemory` (and `VirtualProtect`) and avoids the use of `NtCreateThread` (hence "threadless", I assume). The absence of the last call breaks the typical process injection detection combination. It goes as follows:

1. Find a memory location (a "memory hole", or "code cave") in the remote process that is large enough to hold our shellcode and a small trampoline to.
2. Write the shellcode plus stub to the code cave. The stub will function as a trampoline.
3. Add a `JMP` instruction right after a commonly used `ntdll` function (e.g. `NtOpen`).
4. Wait for a legitimate thread to call `NtOpen`, follow the `JMP` instruction and execute our shellcode.
5. The trampoline redirect control flow back to the legitimate `NtOpen` instructions to continue the process execution and avoid a crash.

More details are available on the ThreadlessInject repo.

4. Evading common malicious patterns

This is really just a repetition of the same technique previously explained. I still believe one of the key detection techniques is a `VirtualAlloc` and `WriteProcessMemory` (or Nt equivalents) call for ~300KB of memory (the side of common implant's shellcode). Chunking those memory operations evade that detection, which DripLoader introduced 2 years ago. Let's also use this technique in our loader.



High level representation of the loader execution flow.

5. Sleep evasion

For a majority of the time the implant will be sleeping, waiting for the next C2 check-in. Once we have a successful execution of the implant's shellcode, hiding its presence in memory while sleeping is key for EDR evasion. There have been a few new implementations for sleep evasion, but not many write-ups, so let's expand a bit on this topic.

In my last post, I used a half-baked memory obfuscation solution (it didn't encrypt the heap). It also uses a hook on the `Sleep()` function which leaves indicators in memory of `ntdll.dll`.

Most modern sleep evasion implementations are based on the FOLIAGE technique by Austin Hudson. One of them, Ekko by 5pider is probably the most widely used implementation nowadays.

Ekko (like FOLIAGE, but uses queued timers instead of queued APCs) uses Thread Pools to delegate the sleep obfuscation work to a worker thread. The worker thread handles the sleep obfuscation of the main thread (where beacon resides), and alerts the main thread when the implant execution should continue. It does so using the following steps:

1. Create a new `Event` and a `TimerQueue` to queue the obfuscation operations on.

```
hEvent      = CreateEventW( 0, 0, 0, 0 );
hTimerQueue = CreateTimerQueue();
```

2. Create a snapshot of the current (main) thread using `RtlCaptureContext` and save it in `&CtxThread` (the `WaitForSingleObject` call just waits for `RtlCaptureContext` to finish saving the snapshot).

```
 if ( CreateTimerQueueTimer( &hNewTimer, hTimerQueue, RtlCaptureContext,
&CtxThread, 0, 0, WT_EXECUTEINTIMERTHREAD ) ) {
        WaitForSingleObject( hEvent, 0x32 );
```

3. Then Ekko defines 6 different context structures that each hold an obfuscation operation to perform:

```
 memcpy( &RopProtRW, &CtxThread, sizeof( CONTEXT ) ); // 1. Set memory
protection to RW
 memcpy( &RopMemEnc, &CtxThread, sizeof( CONTEXT ) ); // 2. Encrypt memory
image, multi-byte RC4 without needing memory allocations
 memcpy( &RopDelay,  &CtxThread, sizeof( CONTEXT ) ); // 3. Delay (sleep) for
specified amount of time, using WaitForSingleObject on something that does not
become alertable
 memcpy( &RopMemDec, &CtxThread, sizeof( CONTEXT ) ); // 4. Decrypt the memory
image
 memcpy( &RopProtRX, &CtxThread, sizeof( CONTEXT ) ); // 5. Set memory
protection to RX
 memcpy( &RopSetEvt, &CtxThread, sizeof( CONTEXT ) ); // 6. Call SetEvent to
alert our main thread that the worker thread is finished.
```

4. Queue all the above calls into the thread pool for the worker thread to execute and alert the main thread when finished:

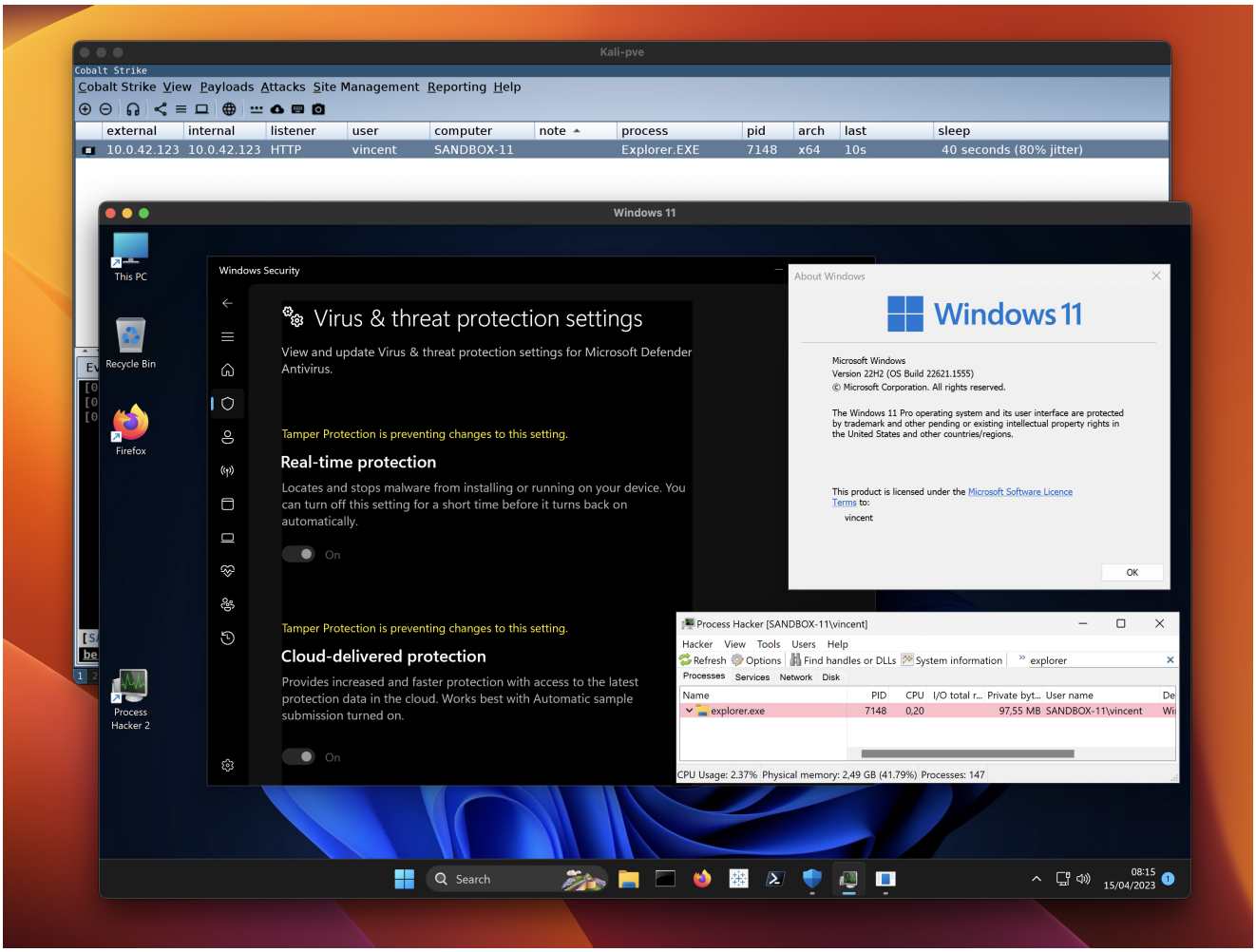
```
 CreateTimerQueueTimer( &hNewTimer, hTimerQueue, NtContinue, &RopProtRW, 100, 0,
WT_EXECUTEINTIMERTHREAD );
 CreateTimerQueueTimer( &hNewTimer, hTimerQueue, NtContinue, &RopMemEnc, 200, 0,
WT_EXECUTEINTIMERTHREAD );
 CreateTimerQueueTimer( &hNewTimer, hTimerQueue, NtContinue, &RopDelay,  300, 0,
WT_EXECUTEINTIMERTHREAD );
 CreateTimerQueueTimer( &hNewTimer, hTimerQueue, NtContinue, &RopMemDec, 400, 0,
WT_EXECUTEINTIMERTHREAD );
 CreateTimerQueueTimer( &hNewTimer, hTimerQueue, NtContinue, &RopProtRX, 500, 0,
WT_EXECUTEINTIMERTHREAD );
 CreateTimerQueueTimer( &hNewTimer, hTimerQueue, NtContinue, &RopSetEvt, 600, 0,
WT_EXECUTEINTIMERTHREAD );
```

This technique does not require hooks or other sketchy `RWX` give-aways for memory scanners.

Ekko is implemented in Cobalt Strike's 4.7+ sleepmask kit, I recommend enabling that. Additionally, you can consider adding patchless evasion of ETW and AMSI.

For this injection PoC we will use Kyle Avery's Cobalt Strike reflective loader AceLdr that implements the above for us. In addition, it also spoofs the return address while we sleep by pointing to a random other thread context using `NtSetContextThread` and "namazso's x64

return address spoofer” (his DEF CON 30 talk is highly recommended).



Process injection on Microsoft Defender for Endpoint with 0 detections (not screenshotted, you have to trust me).
So, that’s it; process injection in 2023, bypassing detection of (at least one) leading EDR solution. All in a pure C `.exe`, no fancy languages, runtimes, obscure file extensions or anything. Just “double-click and go”.

OK, but what about other EDRs? In my last blog post I showed the loader bypass CrowdStrike Falcon, for which I got into trouble. I don’t have access to other (good) EDR solutions. If you do and are happy to publish the results, please reach out.

Related Posts