Cobalt Strike Defense Evasion

Attack Surface Reduction Rules

- **MS Office related**
  - Block all Office applications from creating child processes
    - LOLBAS, not all child processes are blocked by this rule (e.g. msbuild.exe is not, you can use the VBA macro to pass XML/C# code to msbuild.exe to get code execution).
    - COM Bypass: Instantiate suitable COM objects directly from VBA
  - Block Win32 API calls from Office macros
    - Use GadgetToJScript or DotNetToJScript instead of P/Invoke'ing directly from VBA
  - Block Office applications from injecting code into other processes
    - Replace CreateRemoteThread API in your shellcode runner, e.g. with QueueUserAPC
- **Block process creations originating from PSExec and WMI commands**
  - PsExec
    - Manually drop your own service binary to disk and manually create a service to run it
    - 'beacon> jump psexec(64)' also still works
  - WMI
    - Execute VBScript payloads via an event subscription, e.g. by configuring it with SharpWMI. To run .Net assembly with this method, generate VBS with Gadget2JScript.
- **Block credential stealing from the Windows Local Security Authority Subsystem (lsass.exe)**
  - Not all processes blocked. E.g. use WerFault.exe as spawnto to execute the MiniDumpWriteDump assembly

Network Inspection

- Limit number of egress listeners, use peer-to-peer/pivot listeners whenever possible.
- Avoid setting Beacon sleep time too low
- Well-chosen domains and IP addresses for redirectors (age, owner, classification, reputation etc.)
- Use correctly signed TLS certificates on the redirectors
- Choose host process according to Beacon's network protocol (e.g. msedge & HTTPS)
- Using certain categories (like "banking") for the C2 domain sometimes helps to reduce the likelihood of HTTPS inspection.
- Ensure that our payload is proxy-aware
- Set user-agent to a browser type that is permitted by the organization
- Consider the traces our C2 session will leave in the proxy logs. For example, instead of simply generating custom TCP traffic on ports 80 or 443, our session should conform to HTTP protocol standards.
- **Tunneling techniques**
  - Bypass domain, proxy, and IDS filters in one shot with Domain Fronting.
  - Bypass proxy, IPS, and firewall filters by tunneling the C2 traffic through allowed protocols, e.g. DNS tunneling.
  - Cobalt Strike External C2: Use external components to act as a communication layer between the Team Server and a Beacon payload. This can be directly over a protocol such as SSH, FTP... But also be indirectly via service, such as Office365, Slack, Google Drive and others. For example see https://lots-project.com/.

AntiVirus

- **On-Disk Detections**
  - Use encrypted payload (AES, XOR, etc.) in the shellcode runner.
  - Customize the Artifact Kit bypass template(s) you use before building payloads.
  - Put only a loader on disk, load shellcode into memory, e.g. from your webserver.
  - Use ordinals instead of the Win32API's name to define the DllImport attribute in your shellcode runner C# code. This can help circumventing some AV signatures.
  - Use D/Invoke as a replacement for P/Invoke in your shellcode runner C# code. This can avoid the identification of suspicious P/Invoke usage if AV inspects a compiled .NET assembly.
  - **VBA specific**
    - VBA Stomping (only cached pre-compiled P-code remains in document). Only works it victim's version of MS Office is the same. Create manually or automate with EvilClippy.
    - Obfuscate and encrypt PowerShell code (for e.g. reflective DLL load) inside the VBA macro.
    - Remote Template Injection: Send document without malicious code to victim, which then downloads and loads a template holding a malicious macro.
- **In-Memory Detections**
  - Customize Ressource Kit bypass templates
  - Apply amsi_disable in Malleable C2 and/or use own bypasses (attack assembly instructions, vital structures...).
  - Bypass AMSI in JScript to e.g. protect .Net2JS generated shellcode runners.
    - Attack AmsiEnable registry key
    - Prevent AMSI.DLL from loading
  - Bypass "Event Tracing for Windows" to avoid detection of .NET assemblies in memory.
    - If executing an exe on disk from a shell: Just set 'COMPlus_ETWEnabled' environment variable to '0'.
    - If executing from Beacon: Use a BOF like 'InlineExecute-Assembly' or a custom loader based on the User Defined Reflective Loader (UDRL) to execute assemblies.
  - Activate Sleep Mask, customize and recompile to avoid in-memory detection of it's deobfuscation routine.
- **Attack AV itself**
  - Attack AV configuration by removing signature updates, disable certain sub-functions, set exclusions, ...
  - Change the \Device\BootDevice symbolic link in the Windows Object Manager so that when Defender's WdFilter driver is unloaded and loaded again by its Tamper Protection feature, another file is mapped in memory in place of the original WdFilter.sys, rendering it effectively useless. As shown by unDefender (https://github.com/APTortellini/unDefender)
- **Behavioural Detections**
  - **Initial Access**
    - The CreateRemoteThread Win32 API is well known for process injection and AVs often block it by default. Use alternative Win32 APIs like QueueUserAPC.
    - Use lower-level native APIs as an alternative to the Win32 APIs VirtualAllocEx, WriteProcessMemory and VirtualProtectEx. One possibility is using the Nt*Section set of API's to create a new section with the shellcode within our current process and then map it into the target process with NtMapViewOfSection.
    - If you want to inject into svchost.exe (because it makes network activity look legitimate) as an unprivileged user, use process hollowing: Start in suspended state, inject, resume with ResumeThread Win32 API.
  - **Lateral Movement**
    - Change the lateral movement default spawnto (not the post-exec one) with 'ak-settings spawnto_x64 ...' or, in older CS versions set in Artifact Kit sources and recompile, import.
    - Processes created through jump psexec always look suspicious. Immedeately migrate to legit process and remove the old one.
    - Aggressor can be used to register new (possibly more OSEC safe) techniques under 'jump' and 'remote-exec'. E.g. DCOM.
    - Use BOFs for code execution inside the Beacon host process to avoid the fork & run pattern of powershell, powerpick and execute-assembly. Be aware of possible stability issues.
  - **Post-Ex**
    - Set post-ex spawnto according to current situation.
    - PPID Spoofing: Set PPID according to current situation. Set in C2 profile, on Beacon cmdline or in your own code using the PROC_THREAD_ATTRIBUTE_PARENT_PROCESS attribute of CreateProcess.
    - Use Command Line Spoofing to hide command line arguments from the Logs: Create a process with "fake" arguments (to be logged) in a suspended state. Then change arguments (to be executed) and resume the process.
    - Create only unsuspicious image load events (e.g. when using 'execute-assembly' use a 'spawnto' binary that naturally loads clr.dll).
    - Set custom pipe names in Malleable C2 profile (mostly used to retrieve output from some fork & run commands and in the SMB Beacon itself).
    - Bypass "Event Tracing for Windows" to avoid detection of .NET assemblies in memory.
      - If executing an exe on disk from a shell: Just set 'COMPlus_ETWEnabled' environment variable to '0'.
      - If executing from Beacon: Use a BOF like 'InlineExecute-Assembly' or a custom loader based on the User Defined Reflective Loader (UDRL) to execute assemblies.
    - Avoid Beacon's region in memory being set to RWX ('set userwx "false" in C2 profile)
    - Cleanup memory region that contains Beacon's no longer needed reflective loader ('set cleanup "true" in C2 profile)
    - Replace suspicious strings in Beacon in Malleable C2 profile. E.g. 'strrep "beacon.x64.dll" "data4.dll"'
    - Activate Sleep Mask, customize and recompile to avoid in-memory detection of the deobfuscation routine.
    - Enable Thread Stack Spoofing when recompiling Artifact Kit.

Application Whitelisting

- **Microsoft AppLocker**
  - Exploit weak custom rules introduced by the admins.
  - Bypass with NTFS Alternate Data Streams
  - Writable subfolders of allowed paths, like C:\Windows\Tasks.
  - Living Off The Land Binaries, Scripts and Libraries that are not explicitly blocked.
  - Bypass PowerShell Constrained Language Mode with PS custom runspaces. Manually with custom code or by using 'powerpick' in CS.
  - **DLL-based bypasses**
    - If DLL-enforcement is not enabled: Execute from DLLs with rundll32 or use DLL hijacking/sideloading.
    - Bypass DLL-enforcement with reflective injection of untrusted DLLs, e.g. using Invoke-ReflectivePEInjection.ps1.
  - **Bypass AppLocker Scripting Rules**
    - Execute JScript code with mshta instead of wscript.exe.
    - Execute JScript code through XSL Transformation (XSLT).
  - Bypass with third party execution if available (Python, Perl, Java, ...)
  - Got SYSTEM? By default, AppLocker is not even applied to Administrators
- **Windows Defender Application Control (WDAC)**
  - Living Off The Land Binaries, Scripts and Libraries that are not on the blocklist.
  - Abuse custom wildcard FilePaths that are writable (if Runtime FilePath Rule Protection is disabled)
  - Abuse user modifiable binaries (if Runtime FilePath Rule Protection is disabled)
  - Abuse trusted applications that have script or code execution capabilities and/or support plugins (e.g. Visual Studio Code, Sublime Text, Notepad++ ...)
  - **Trusted Signers (Organisation implements it's own code signing mechanisms).**
    - If you can gain access to an exported private key (.pfx) (e.g. a backup copy) of the CA used for code signing and the associated password, use it to sign any binary and bypass WDAC.
    - Make the ADCS CA sign your own certificate signing request, e.g. when Web Enrollment role is installed and you have access to a code signing template. Cobalt Strike can use that cert to sign payload binaries on the fly. Or use your signing keys signed by the CA to manually sign binaries with signtool.exe.
  - Exploiting vulnerabilities in allowed applications (DLL & COM hijacks, overflows, double-freeing, deserialization...) to execute your own code.

EDR

- Avoid to use attacker tools on the entry point altogether and only start a proxy (e.g. an obfuscated Chisel client) to tunnel attacker tools into the AD.
- **Remmove the EDR's hooks**
  - "Unhook" both IAT and inline hooks by patching them to restore their original values. NOT RECOMMENDED! as the EDR can re-hook them and raise an alert.
  - Enumerate which APIs are hooked by the EDR and use comparable alternatives
  - Manual Mapping: Use D/Invoke's 'Map.MapModuleToMemory()' method to read and map a new instance of original ntdll.dll from disk into the process (additionally to the hooked ntdll.dll) and call APIs from there. Opsec: Free the mapped instance of ntdll from the process after use.
  - Process Mitigation Policy: If the EDR's DLL is not signed by Microsoft, we can prevent it from being loaded into processes via the PROC_THREAD_ATTRIBUTE_MITIGATION_POLICY. Implement in your own code or for post-ex commands in Cobalt Strike use the 'blockdlls' command.
- **Evade the EDR's hooks**
  - Direct syscalls: Avoid calls to the needed APIs over kernel32.dll and ntdll.dll entirely. Instead use syscalls to talk to the kernel directly. NOTE: As the SSN varies between Windows versions it hast to be obtained dynamically. NOTE: Calling syscall directly from your malware is pretty suspicious and bad OPSEC. See indirect syscalls below.
    - Syscalls with D/Invoke: Use D/Invoke's 'GetSyscallStub' method to read the original syscall stub from ntdll on disk and copy it into a new memory region within our calling process. Use it for a direct syscalls bypassing the API. Opsec: Free the memory region after use with 'NtFreeVirtualMemory' to avoid leaving stubs in memory.
    - Syscalls with SysWhispers2 and Artifact Kit: Use SysWhispers2 to generate header and assembly files that facilitate making syscalls from native code. Use those in the changes to Artifact Kit's sources that integrate the use of syscalls into it's generation of payloads instead of the default use of Win32 APIs.
  - Indirect syscalls: Replace the original syscall instruction with a jump instruction pointing to a memory address of NTDLL where it stores the syscall instruction. This avoids having syscall instructions in your payload and ensures the syscall execution is always originated from legitimate NTDLL, which is better OPSEC
  - Customize Cobalt Strike's reflective loader used in Beacon workflows with the 'User Defined Reflective Loader' (UDRL). Or use publicly available alternative loaders such as BokuLoader or Elusive95ce
- **Manipulate kernel callbacks with a malicious kernel driver:** Even the use of syscalls will not help to avoid that the creation of new processes and threads is logged via CreateProcessNotify and CreateThreadNotify. Malicious drivers can be used to disable the kernel callback.
  - Find a vulnerability in a driver that's already installed and exploit it.
  - Bring your own malicious driver that you can load yourself as local admin. In this case be aware of Driver Signature Enforcement (DSE). To bypass it:
    - With full physical & administrative access to a device you can enable "Test Signing" and reboot. This scenario is usually not applicable in Red Team Operations.
    - Install a signed Driver with known vulnerabilities which provides arbitrary memory read/write from userland (e.g. CVE-2018-19320 & CVE-2018-19321). Use it to disable driver signing and load your malicious driver. OPSEC: re-enable driver signing immediately.
- Disable Logging: Phant0m (https://github.com/hlldz/Phant0m) detects and kills the threads responsible for the Event Log service. Thus, while the Event Log service appears to be running in the system the system does not collect logs. OPSEC: Should only be used for short periods of time.