PART 3: How I Met Your Beacon – Brute Ratel

mdsec.co.uk/2022/08/part-3-how-i-met-your-beacon-brute-ratel/

3 August 2022

ActiveBreach

Introduction

In part one, we introduced generic approaches to performing threat hunting of C2 frameworks and then followed it up with practical examples against Cobalt Strike in part two.

In part three of this series, we will analyse Brute Ratel, a command and control framework developed by Dark Vortex. As the C2 is lesser known, we can see it describes itself as follows:

> **Brute Ratel** is the most advanced Red Team & Adversary Simulation Software in the current C2 Market. It can not only emulate different stages of an attacker killchain, but also provide a systematic timeline and graph for each of the attacks executed to help the Security Operations Team validate the attacks and improve the internal defensive mechanisms. Brute Ratel comes prebuilt with several opsOpec features which can ease a Red Team's task to focus more on the analytical part of an engagement instead of focusing or depending on Open source tools for post-exploitation. Brute Ratel is a post-exploitation C2 in the end and however **does not** provide exploit generation features like metasploit or vulnerability scanning features like Nessus, Acunetix or BurpSuite.

The framework has come under close scrutiny in the past few months, having been allegedly abused by APT29 and the ransomware group BlackCat in recent times. Having an understanding of how we can generically detect this emerging C2 in our infrastructure is therefore useful intelligence for defenders.

Originally, all analysis was performed on Brute Ratel v1.0.7; the latest at the time of original review. However, a cursory update (contained at the end of this article) was performed discussing findings pertinent to v1.1 which was released shortly after our initial x33fcon presentation. One thing that should be noted with Brute Ratel is that the badger has only limited malleability and primarily from the perspective of the c2 channels; with the exception of v1.1 which added malleability for the sleep obfuscation techniques. As such it makes it possible to create very specific detections for the tool.

Brute Ratel's Loader

Brute Ratel's badger comes in a number of forms, including exe, DLL and shellcode. When the badger is injected, its reflective loader will instantly load all dependencies required for the badger. As the badger bundles a large amount of post-exploitation features, this leads to a significant number of DLLs being loaded on initialisation:

Time of Day	Process Name	PID	Operation	Path	Result	Detail
22:19:27.3294254	notepad.exe	8548	Load Image	C:\Windows\System32\cryptsp.dll	SUCCESS	Image Base: 0x7ffd...
22:19:27.3405444	notepad.exe	8548	Load Image	C:\Windows\System32\cryptbase.dll	SUCCESS	Image Base: 0x7ffd...
22:19:27.3472687	notepad.exe	8548	Load Image	C:\Windows\System32\cryptdll.dll	SUCCESS	Image Base: 0x7ffd...
22:19:27.3632003	notepad.exe	8548	Load Image	C:\Windows\System32\crypt32.dll	SUCCESS	Image Base: 0x7ffd...
22:19:27.3652582	notepad.exe	8548	Load Image	C:\Windows\System32\psapi.dll	SUCCESS	Image Base: 0x7ffd...
22:19:27.4036815	notepad.exe	8548	Load Image	C:\Windows\WinSxS\amd64_microsoft.windows.gdi...	SUCCESS	Image Base: 0x7ffd...
22:19:27.4079732	notepad.exe	8548	Load Image	C:\Windows\System32\netapi32.dll	SUCCESS	Image Base: 0x7ffd...
22:19:27.4107073	notepad.exe	8548	Load Image	C:\Windows\System32\samcli.dll	SUCCESS	Image Base: 0x7ffd...
22:19:27.4133385	notepad.exe	8548	Load Image	C:\Windows\System32\logoncli.dll	SUCCESS	Image Base: 0x7ffd...
22:19:27.4184777	notepad.exe	8548	Load Image	C:\Windows\System32\netutils.dll	SUCCESS	Image Base: 0x7ffd...
22:19:27.4229742	notepad.exe	8548	Load Image	C:\Windows\System32\srvcli.dll	SUCCESS	Image Base: 0x7ffd...
22:19:27.4252674	notepad.exe	8548	Load Image	C:\Windows\System32\ole32.dll	SUCCESS	Image Base: 0x7ffd...
22:19:27.4323592	notepad.exe	8548	Load Image	C:\Windows\System32\IPHLPAPI.DLL	SUCCESS	Image Base: 0x7ffd...
22:19:27.4418068	notepad.exe	8548	Load Image	C:\Windows\System32\msasn1.dll	SUCCESS	Image Base: 0x7ffd...
22:19:27.4467424	notepad.exe	8548	Load Image	C:\Windows\System32\secur32.dll	SUCCESS	Image Base: 0x7ffd...
22:19:27.4532209	notepad.exe	8548	Load Image	C:\Windows\System32\sspicli.dll	SUCCESS	Image Base: 0x7ffd...
22:19:27.4618411	notepad.exe	8548	Load Image	C:\Windows\System32\wtsapi32.dll	SUCCESS	Image Base: 0x7ffd...
22:19:27.4650746	notepad.exe	8548	Load Image	C:\Windows\System32\dbghelp.dll	SUCCESS	Image Base: 0x7ffd...
22:19:27.4691740	notepad.exe	8548	Load Image	C:\Windows\System32\version.dll	SUCCESS	Image Base: 0x7ffd...
22:19:27.4718988	notepad.exe	8548	Load Image	C:\Windows\System32\dnsapi.dll	SUCCESS	Image Base: 0x7ffd...
22:19:27.4741019	notepad.exe	8548	Load Image	C:\Windows\System32\nsi.dll	SUCCESS	Image Base: 0x7ffd...
22:19:27.4785349	notepad.exe	8548	Load Image	C:\Windows\System32\credui.dll	SUCCESS	Image Base: 0x7ffd...
22:19:27.4840403	notepad.exe	8548	Load Image	C:\Windows\System32\wininet.dll	SUCCESS	Image Base: 0x7ffd...
22:19:27.4956147	notepad.exe	8548	Load Image	C:\Windows\System32\iertutil.dll	SUCCESS	Image Base: 0x7ffd...
22:19:27.5000154	notepad.exe	8548	Load Image	C:\Windows\System32\profapi.dll	SUCCESS	Image Base: 0x7ffd...
22:19:27.5139276	notepad.exe	8548	Load Image	C:\Windows\System32\OnDemandConnRouteHelpe...	SUCCESS	Image Base: 0x7ffd...
22:19:27.5166682	notepad.exe	8548	Load Image	C:\Windows\System32\winhttp.dll	SUCCESS	Image Base: 0x7ffd...
22:19:27.5201859	notepad.exe	8548	Load Image	C:\Windows\System32\mswsock.dll	SUCCESS	Image Base: 0x7ffd...
22:19:27.5254157	notepad.exe	8548	Load Image	C:\Windows\System32\winnsi.dll	SUCCESS	Image Base: 0x7ffd...
22:19:27.7512557	notepad.exe	8548	Load Image	C:\Windows\System32\bcrypt.dll	SUCCESS	Image Base: 0x7ffd...
22:13:37.0402195	notepad.exe	8548	Load Image	C:\Windows\System32\notepad.exe	SUCCESS	Image Base: 0x7ffd...
22:13:37.0518518	notepad.exe	8548	Load Image	C:\Windows\System32\ntdll.dll	SUCCESS	Image Base: 0x7ffd...
22:13:37.0548401	notepad.exe	8548	Load Image	C:\Windows\System32\kernel32.dll	SUCCESS	Image Base: 0x7ffd...
22:13:37.0561712	notepad.exe	8548	Load Image	C:\Windows\System32\KernelBase.dll	SUCCESS	Image Base: 0x7ffd...
22:13:37.0746007	notepad.exe	8548	Load Image	C:\Windows\System32\gdi32.dll	SUCCESS	Image Base: 0x7ffd...

As we can see, the DLLs highlighted are all the DLLs that are loaded when the badger is injected. This list includes the loading of winhttp.dll and wininet.dll, which are not necessarily nefarious but are traditional loads for an egress beacon. There are however a number of less

common DLLs loaded, such as dbghelp.dll, credui.dll samcli.dll and logoncli.dll amongst others.

This behaviour allows us to create a signature for the image loads and leads to a high signal indicator that can be hunted for through image load telemetry.

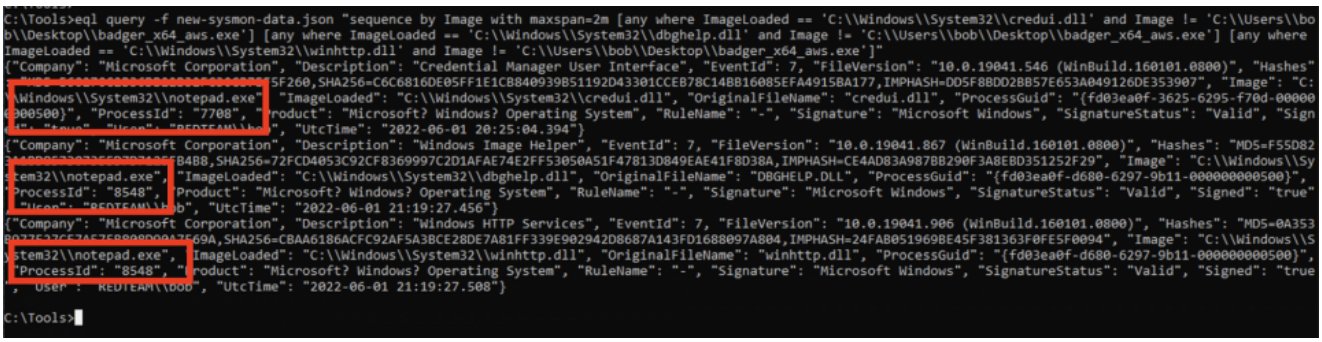
For example, using Elastic Query Language, we can search for the sequence of credui.dll, dbghelp.dll and winhttp.dll load events occurring in a process within 60 seconds of each other:

```
sequence by Image with maxspan=1m
        [any where ImageLoaded == 'C:\\Windows\\System32\\credui.dll']
        [any where ImageLoaded == 'C:\\Windows\\System32\\dbghelp.dll']
        [any where ImageLoaded == 'C:\\Windows\\System32\\winhttp.dll']
```

Using the EQL tool, or Elastic's cloud, we can search our event data, such as the following which was extracted from sysmon logs. Note, we're explicitly excluding the badger executable itself so we can only identify the injected badgers:

```
eql query -f sysmon-data.json "sequence by Image with maxspan=2m [any where
ImageLoaded == 'C:\\Windows\\System32\\credui.dll' and Image !=
'C:\\Users\\bob\\Desktop\\badger_x64_aws.exe'] [any where ImageLoaded ==
'C:\\Windows\\System32\\dbghelp.dll' and Image !=
'C:\\Users\\bob\\Desktop\\badger_x64_aws.exe'] [any where ImageLoaded ==
'C:\\Windows\\System32\\winhttp.dll' and Image !=
'C:\\Users\\bob\\Desktop\\badger_x64_aws.exe']"
```

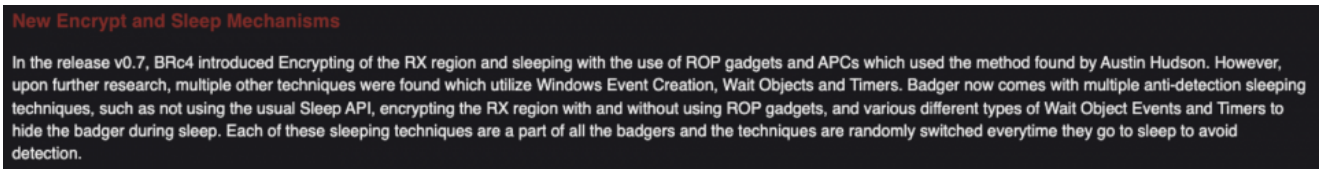
This leads to the following which shows the detection of the badger being injected in to notepad.exe:



This query is particularly powerful as it allows us to retrospectively hunt for indicators of Brute Ratel badgers in the network, without directly running code on the endpoints.

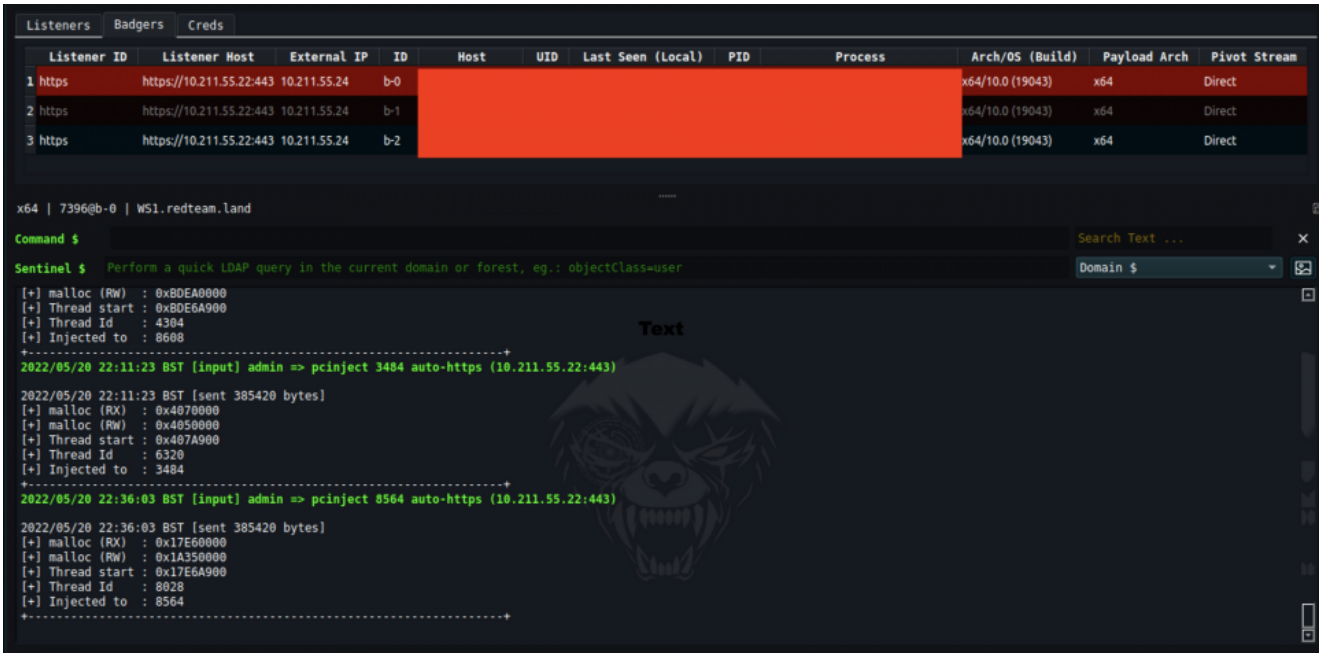
Brute Ratel In Memory

As most beacons remain memory resident, it is important to understand the footprint that is left behind in order to hunt for them. Reviewing the Brute Ratel documentation for the 1.0 release, it details its own implementation of obfuscate and sleep:

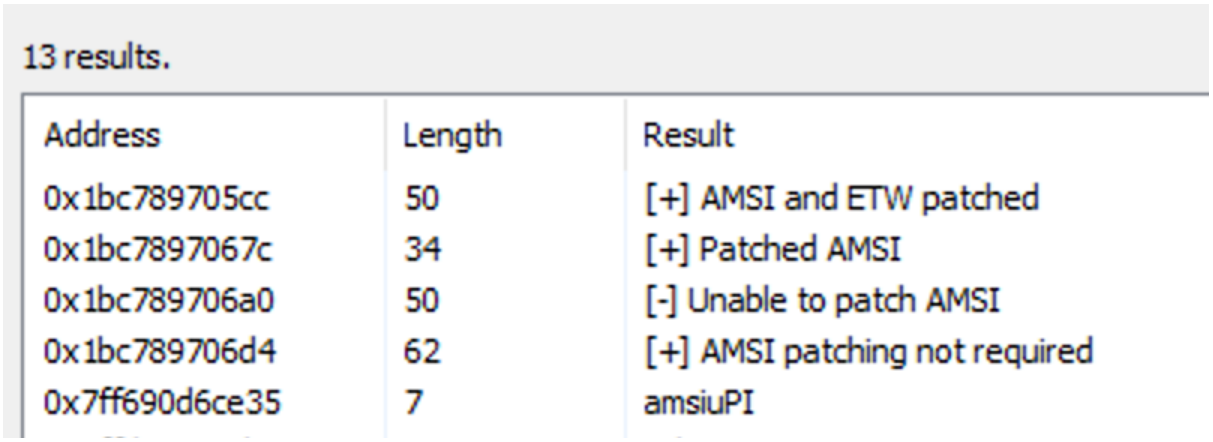


According to the release post, BRc4 uses a mixture of “Asynchronous Procedure Calls, Windows Event Creation, Wait Objects and Timers”. However, analysis of the badger was only able to find evidence of APC based execution; more on this later.

In order to analyse the badger in memory, we first inject it to a process using the pcinject command, then put the badger to sleep using the sleep command:

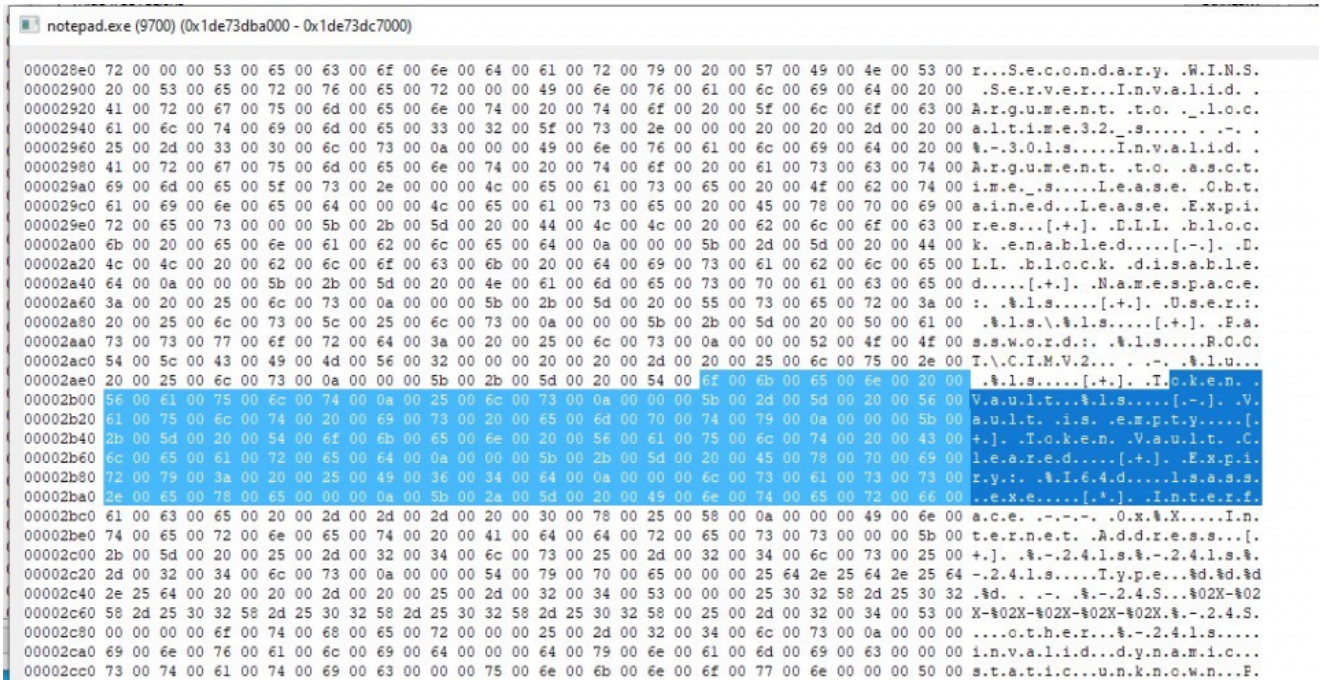


Once the badger is sleeping, we can recover the strings from the process using Process Hacker. Interestingly, while the badger is sleeping we can see strings such as the following:

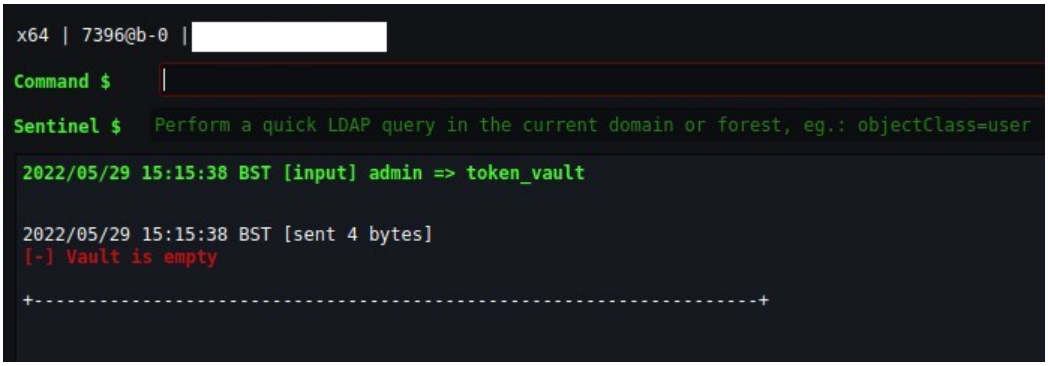


Initially this was quite surprising given the aforementioned purported sleep and obfuscate strategies described on the Brute Ratel blog.

Digging deeper, we can find that some interesting design decisions have been made where by many of the strings displayed in the operator's UI, are populated from the badger itself. For example, we can see the following in the memory of the badger while it is sleeping:

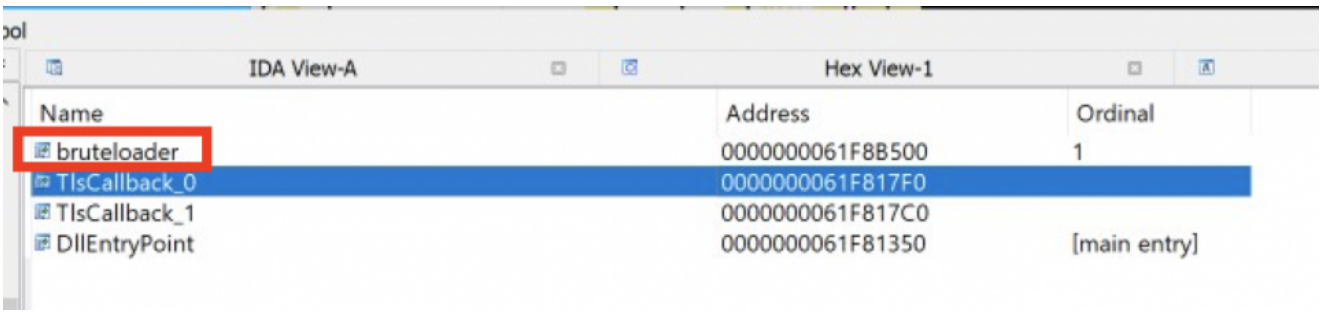


And these strings are then returned to the UI as we can see below:

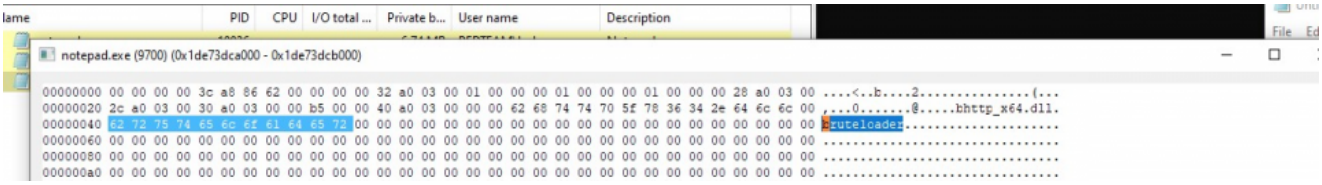


Digging deeper in to the badger, it was quickly apparent that only the .text section was being obfuscated on sleep, leaving the badger susceptible to all manner of signatures against strings and data.

To illustrate this, reversing the badger we can see the entry point for the loader as "bruteloader":



Searching for this string in memory while the badger is sleeping, we can quickly find it inside our notepad process:



These strings provide a good point on which to base a Yara rule for memory scanning on. For example, the following rule will search for either the bruteloader or bhttp_x64.dll strings in memory of a process:

```
rule brc4_badger_strings
{
meta:
    author = "@domchell"
    description = "Identifies strings used in Badger v1.0.x rDLL, even while
sleeping"
strings:
    $a = "bruteloader"
    $b = "bhttp_x64.dll"
condition:
    1 of them
}
```

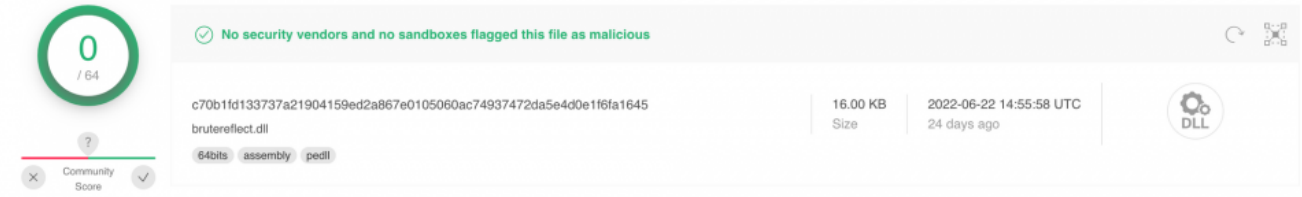
We can test these against our notepad process while the badger is sleeping to evidence its effectiveness:

```
C:\Tools\yara-v4.2.1-1934-win64>yara64.exe z:\brc4.yara 9700
brc4_badger_strings 9700

C:\Tools\yara-v4.2.1-1934-win64>yara64.exe --print-strings z:\brc4.yara 9700
brc4_badger_strings 9700
0x1de73dca040:$a: bruteloader
0x1de73dca032:$b: bhttp_x64.dll

C:\Tools\yara-v4.2.1-1934-win64>
```

It is unlikely the strings will exist in other processes, and using a simple one liner we can quickly find all the injected badgers on our test system:

```
C:\Tools\yara-v4.2.1-1934-win64>powershell -command "Get-Process | ForEach-Object {c:\yara64.exe z:\brc4.yara $_.ID}"
^C
C:\Tools\yara-v4.2.1-1934-win64>powershell -command "Get-Process | ForEach-Object {C:\Tools\yara-v4.2.1-1934-win64\yara64.exe z:\brc4.yara $_.ID}"
error scanning 4880: can not attach to process (try running as root)
brc4_badger_strings 7396
error scanning 5364: can not attach to process (try running as root)
error scanning 2420: can not attach to process (try running as root)
error scanning 4224: can not attach to process (try running as root)
error scanning 5168: can not attach to process (try running as root)
error scanning 460: can not attach to process (try running as root)
error scanning 560: can not attach to process (try running as root)
error scanning 2720: can not attach to process (try running as root)
error scanning 1004: can not attach to process (try running as root)
error scanning 892: can not attach to process (try running as root)
error scanning 900: can not attach to process (try running as root)
error scanning 8620: can not attach to process (try running as root)
error scanning 4244: can not attach to process (try running as root)
error scanning 0: could not open file
error scanning 716: can not attach to process (try running as root)
error scanning 2112: can not attach to process (try running as root)
error scanning 3112: can not attach to process (try running as root)
error scanning 8472: can not attach to process (try running as root)
brc4_badger_strings 8416
brc4_badger_strings 9700
brc4_badger_strings 10036
error scanning 8016: can not attach to process (try running as root)
brc4_badger_strings 7132
error scanning 3292: can not attach to process (try running as root)
error scanning 2452: can not attach to process (try running as root)
error scanning 108: can not attach to process (try running as root)
error scanning 4300: can not attach to process (try running as root)
error scanning 2312: can not attach to process (try running as root)
error scanning 696: can not attach to process (try running as root)
error scanning 1584: can not attach to process (try running as root)
error scanning 336: can not attach to process (try running as root)
error scanning 2800: can not attach to process (try running as root)
error scanning 376: can not attach to process (try running as root)
```

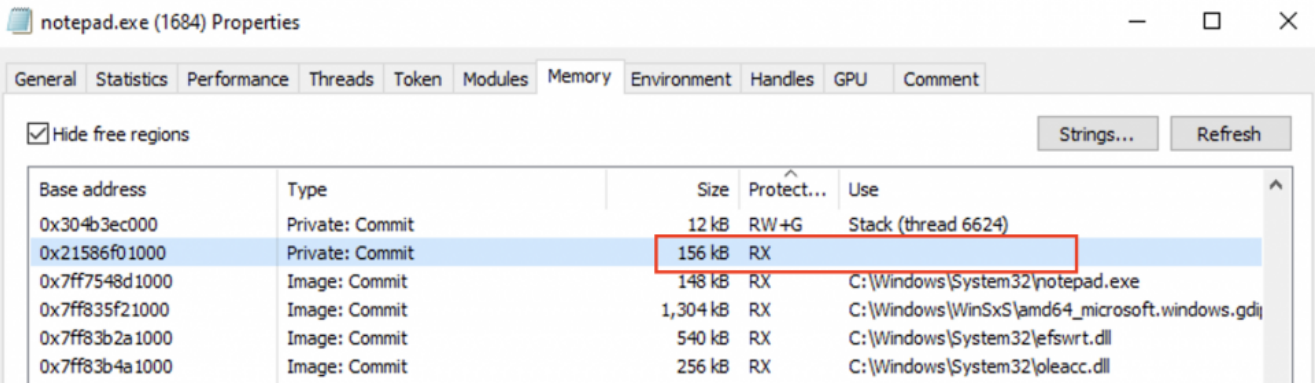
Plugging this Yara rule in to virus total, we can quickly find other samples, such as:



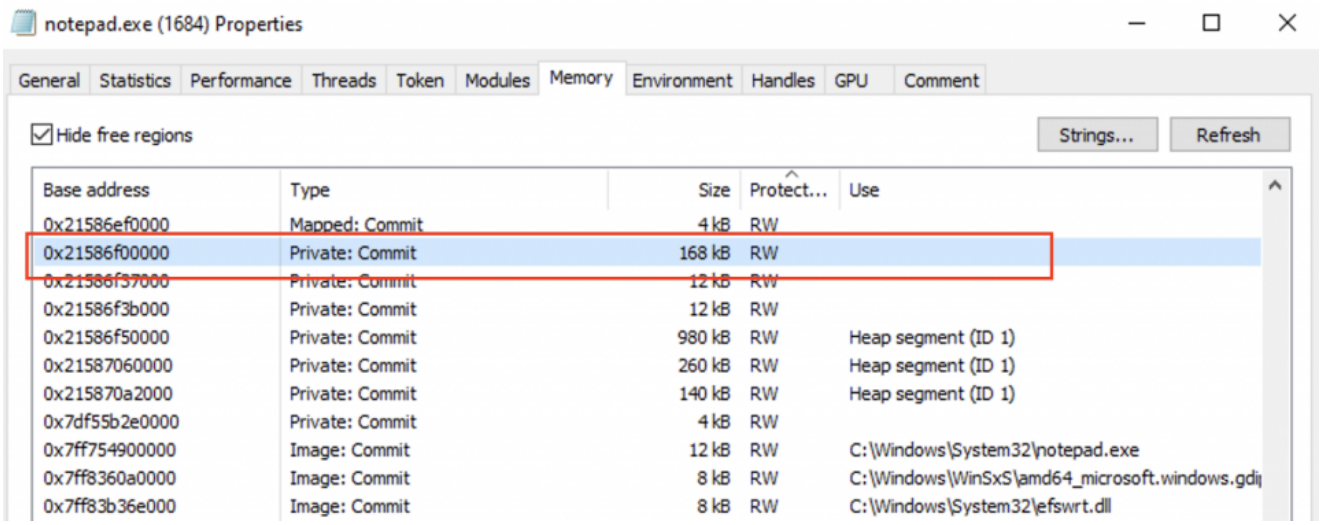
Page Permissions

Analysis of the Brute Ratel obfuscate and sleep strategy observed the badger to shuffle the page permissions for the badger during sleep in an attempt to evade prolonging executable permissions while the badger sleeps.

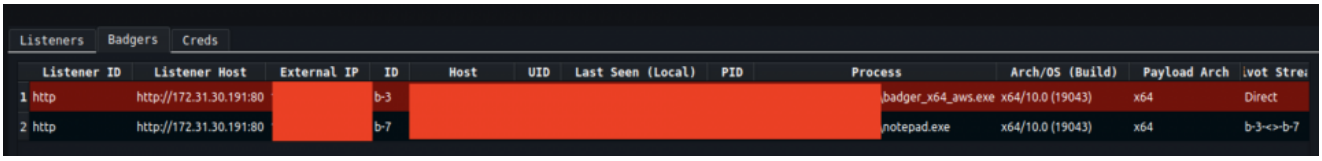
Below, we can see the badger operating on a sleep 0, the page permissions for the badger are PAGE_EXECUTE_READ on an unmapped page; this is necessary in order to perform tasking:



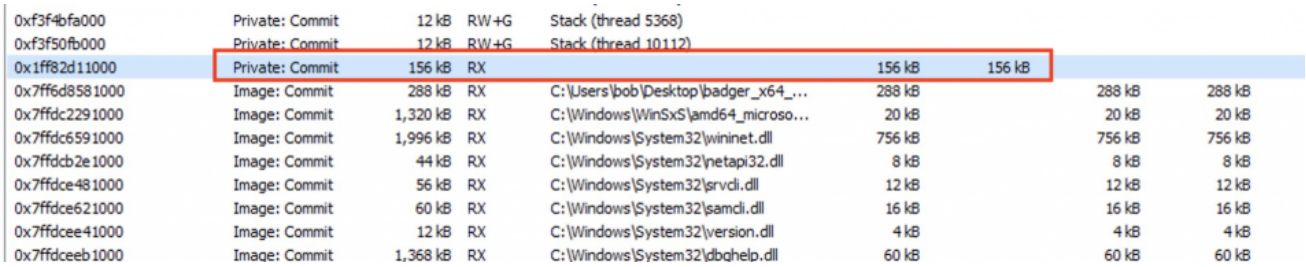
Putting the badger to sleep, we can see that the obfuscate and sleep strategy obfuscates the .text section and resets the page permissions for the badger to to PAGE_READWRITE:



Interestingly, we however note that this behaviour is not replicated while a SMB pivot is being performed, that is when two badgers are linked. Here we can see our two badgers linked and both on a 60 second sleep:



Analysis of the page permissions while two badgers are linked reveals that both remain PAGE_EXECUTE_READ, irrespective of the sleep time:

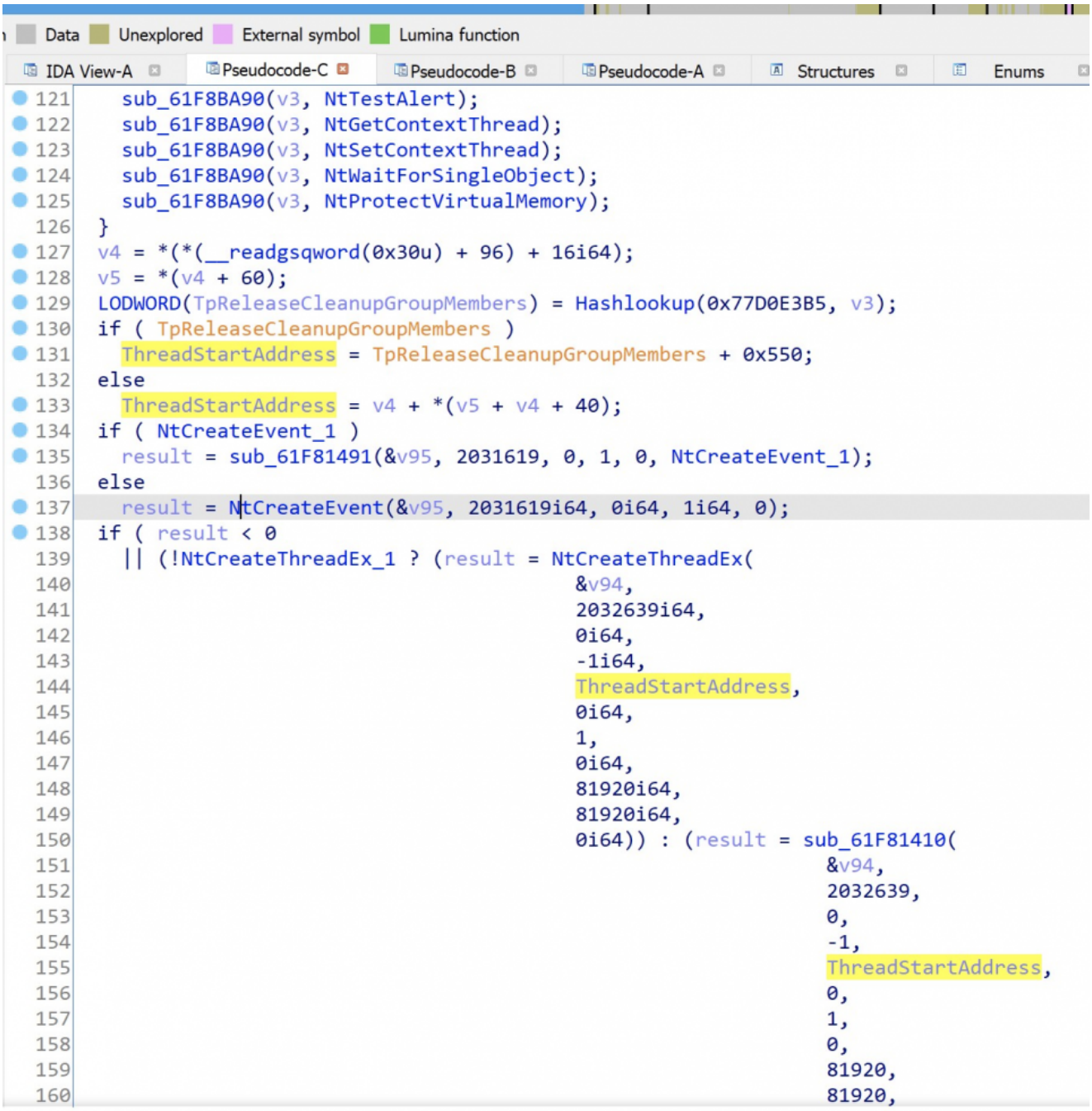


The conclusion is that the obfuscate and sleep strategy is only applicable to the .text section, and while no peer-to-peer pivot is present.

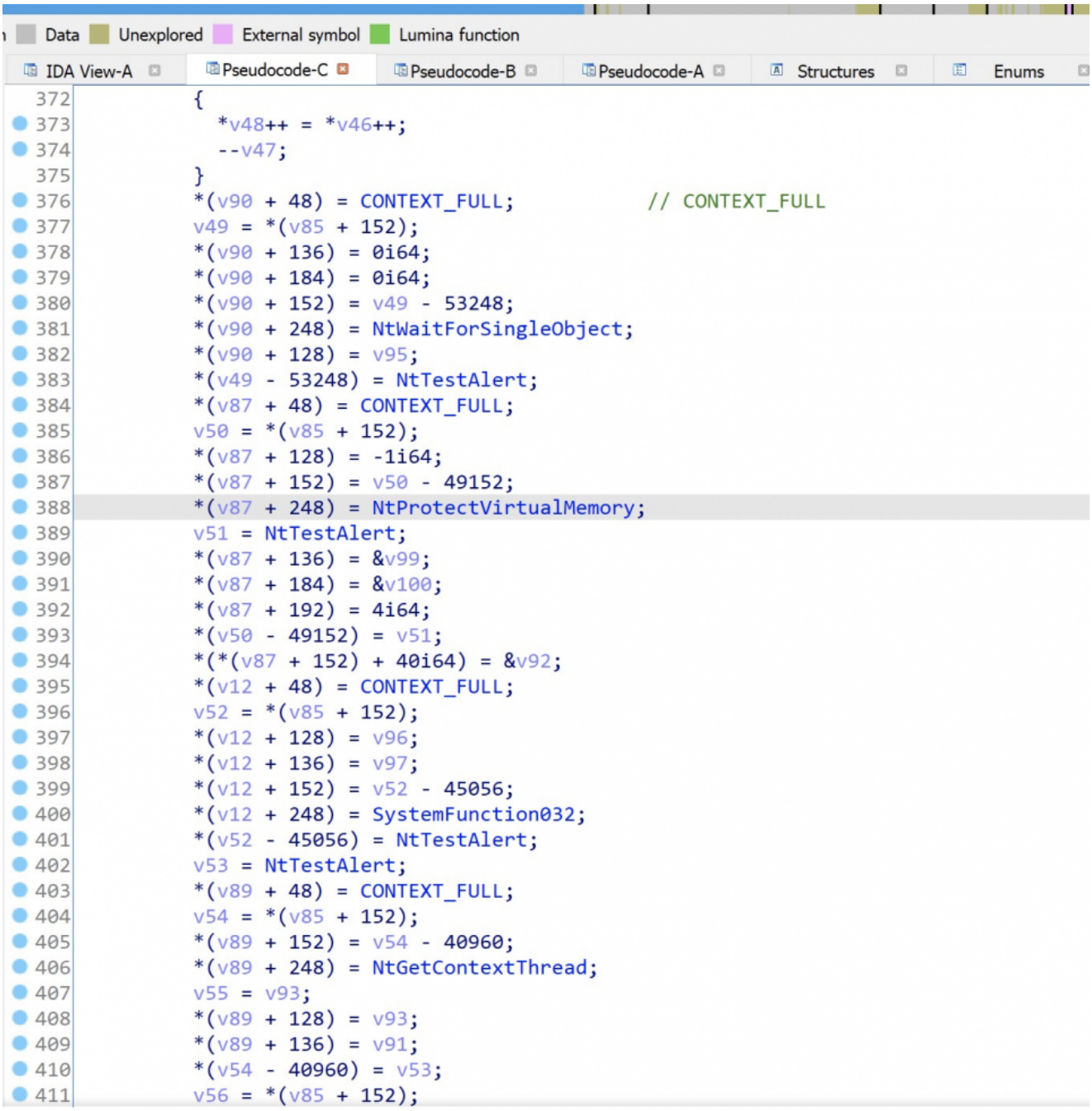
Curious to how the obfuscate and sleep functionality worked, we began to reverse engineer it. Walking through the sleep routine in windbg, we can get an initial flavour of what's happening; the badger is using WaitForSingleObjectEx to delay execution during a series of asynchronous procedure calls (APC), and leveraging an indirect syscall to execute NtTestAlert and force an alert on the thread:

```
00007ffa`b8dc16fc c3              ret
0:006> k
 # Child-SP          RetAddr               Call Site
00 00000000`02bc7fd8 00007ffa`bb0b04ff     KERNELBASE!WaitForSingleObjectEx+0x12c
01 00000000`02bc7fe0 00000000`00000000     ntdll!NtTerminateJobObject+0x1f
0:006> p
ntdll!NtTestAlert:
00007ffa`bb0b0500 4c8bd1          mov     r10,rcx
0:006> u ntdll!NtTerminateJobObject+0x1f
ntdll!NtTerminateJobObject+0x1f:
00007ffa`bb0b04ff 004c8bd1        add     byte ptr [rbx+rcx*4-2Fh],cl
00007ffa`bb0b0503 b8c0010000      mov     eax,1C0h
00007ffa`bb0b0508 f604250803fe7f01 test    byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],1
00007ffa`bb0b0510 7503            jne     ntdll!NtTestAlert+0x15 (00007ffa`bb0b0515)
00007ffa`bb0b0512 0f05            syscall
00007ffa`bb0b0514 c3              ret
00007ffa`bb0b0515 cd2e            int     2Eh
00007ffa`bb0b0517 c3              ret
```

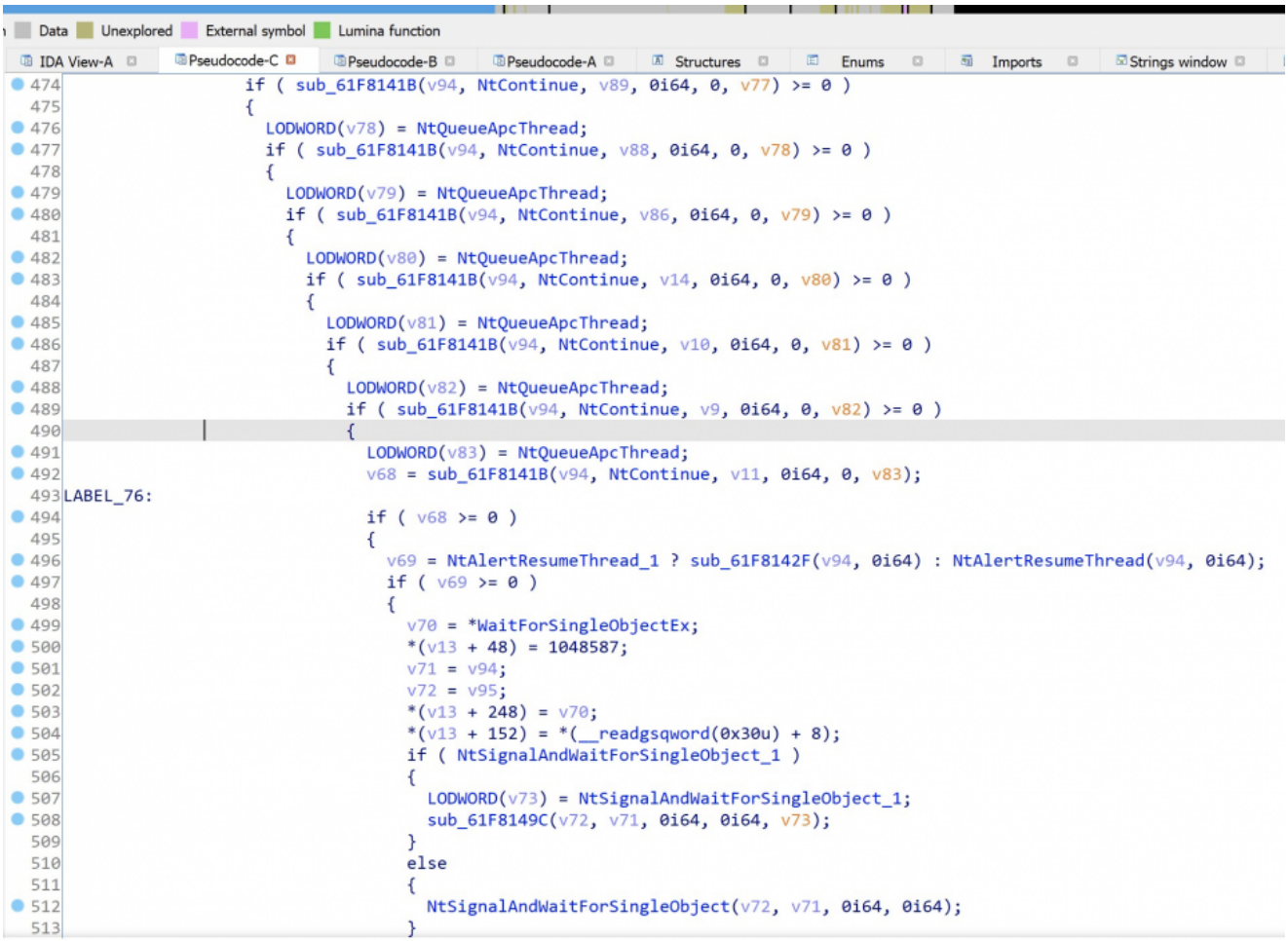
Diving in to IDA, we can get a better feel for what is happening. First it creates a new thread with the start address spoofed to a fixed location of TpReleaseCleanupGroupMembers+550:



A series of context structures are then created for a number of function calls, to NtWaitForSingleObject, NtProtectVirtualMemory, , SystemFunction032, NtGetContextThread and SetThreadContext:



Next, a number of APCs are queued against the NtContinue, with the intention of using it to proxy calls to the aforementioned context structures; this technique acts as a rudimentary form of ROP:



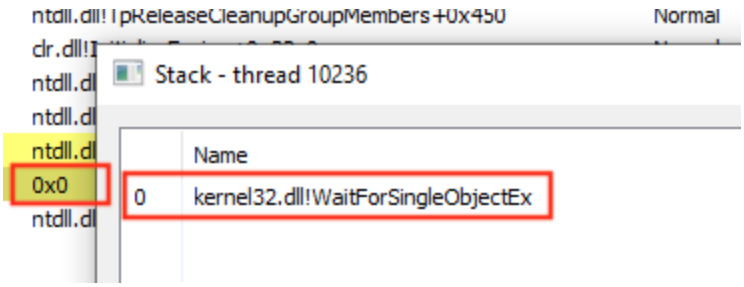
Having reverse engineered the sleeping technique, we soon realised that it it was very similar to @ilove2pwn_'s Foliage project, with the exception of the hardcoded thread start address.

Despite extensive debugging and reverse engineering of the badger, we unable to reveal any evidence of the "Windows Event Creation, Wait Objects and Timers" techniques referenced in the v1.0 blog post; indeed the APIs required for these techniques did not appear to be imported via the badger's hashed imports.

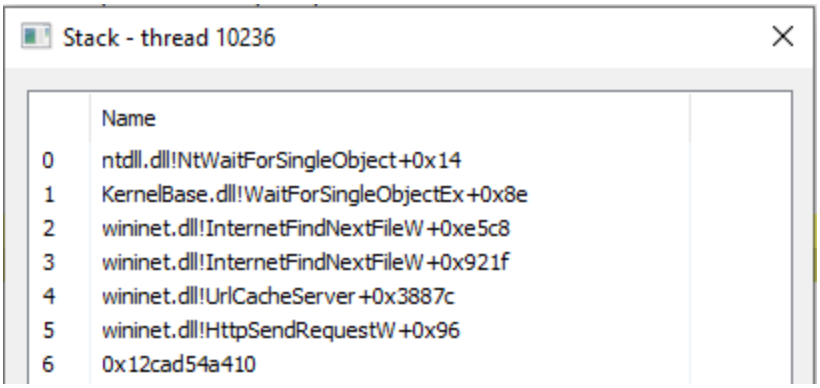
Brute Ratels Threads

To analyse how Brute Ratel threads look in memory, we injected the badger in to a fresh copy of notepad. Immediately, we can see there are some suspicious indicators in the threads used by the sleeping badger.

Firstly, we note that there is a suspicious looking thread with a 0x0 start address, and a single frame calling WaitForSingleObjectEx in the call stack:



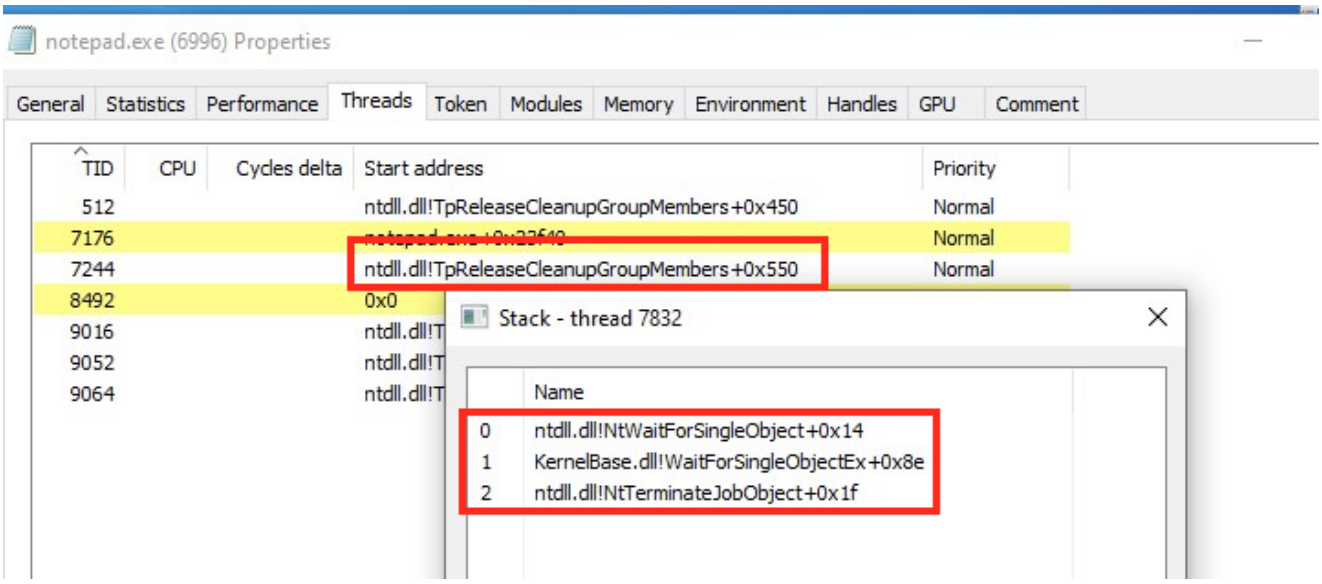
We can speculate that this thread is used for the HTTP comms based on analysis of the thread call stack while the badger is now sleeping:



Based on the information we gained from reverse engineering the obfuscate and sleep strategy, we noted that new threads were created with a hardcoded spoofed start address of ntdll!TpReleaseCleanupGroupMembers+0x550:

```
LODWORD(TpReleaseCleanupGroupMembers) = Hashlookup(0x77D0E3B5, v3);
if ( TpReleaseCleanupGroupMembers )
  ThreadStartAddress = TpReleaseCleanupGroupMembers + 0x550;
else
  ThreadStartAddress = v4 + *(v5 + v4 + 40);
if ( NtCreateEvent_1 )
  result = sub_61F81491(&v95, 2031619, 0, 1, 0, NtCreateEvent_1);
else
  result = NtCreateEvent(&v95, 2031619i64, 0i64, 1i64, 0);
if ( result < 0
  || (!NtCreateThreadEx_1 ? (result = NtCreateThreadEx(
                                        &v94,
                                        2032639i64,
                                        0i64,
                                        -1i64,
                                        ThreadStartAddress,
```

We were unable to find any instances of this occurring as a start address naturally, and as such leads to a trivial indicator for hunting Brute Ratel threads. In practice this looks as follows within our injected notepad process:



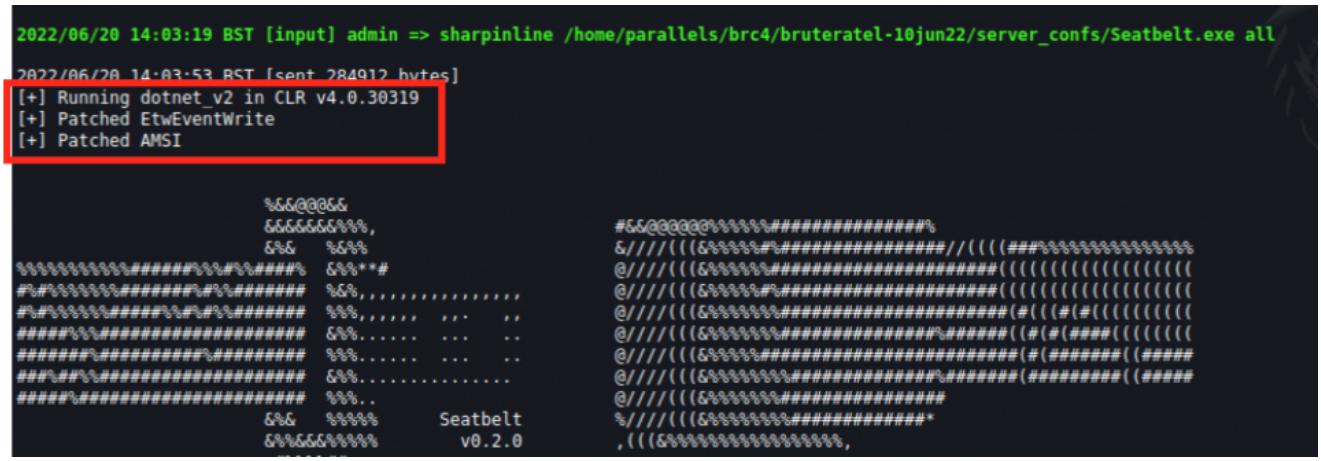
The call stack for the thread is also slightly irregular as it not only contains calls to delay execution, but also the first frame points to ntdll.dll!NtTerminateJobObject+0x1f. A deeper look at why NtNerminateJobObject is used highlights that this is simply a ROP gadget for NtTestAlert and is used to execute pending APCs on the thread:

```
00007ffa`b8dc16fc c3              ret
0:006> k
 # Child-SP          RetAddr               Call Site
00 00000000`02bc7fd8 00007ffa`bb0b04ff     KERNELBASE!WaitForSingleObjectEx+0x12c
01 00000000`02bc7fe0 00000000`00000000     ntdll!NtTerminateJobObject+0x1f
0:006> p
ntdll!NtTestAlert:
00007ffa`bb0b0500 4c8bd1          mov     r10,rcx
0:006> u ntdll!NtTerminateJobObject+0x1f
ntdll!NtTerminateJobObject+0x1f:
00007ffa`bb0b04ff 004c8bd1        add     byte ptr [rbx+rcx*4-2Fh],cl
00007ffa`bb0b0503 b8c0010000      mov     eax,1C0h
00007ffa`bb0b0508 f604250803fe7f01 test    byte ptr [SharedUserData+0x308 (00000000`7ffe0308)],1
00007ffa`bb0b0510 7503            jne     ntdll!NtTestAlert+0x15 (00007ffa`bb0b0515)
00007ffa`bb0b0512 0f05            syscall
00007ffa`bb0b0514 c3              ret
00007ffa`bb0b0515 cd2e            int     2Eh
00007ffa`bb0b0517 c3              ret
```

Memory Hooks

In our first post in this series, we detailed two potential approaches for detecting in-memory beacons based on memory hooks; by looking for signatures of known patches (e.g. ret to ntdll.dll!EtwEventWrite) and by detecting copy on write operations.

Applying these concepts to Brute Ratel, we note that the badger does not apply any memory hooks until its post-exploitation functionality is used by the operator. An example of this, would be the sharpinline command, which runs a .NET assembly in the current process:

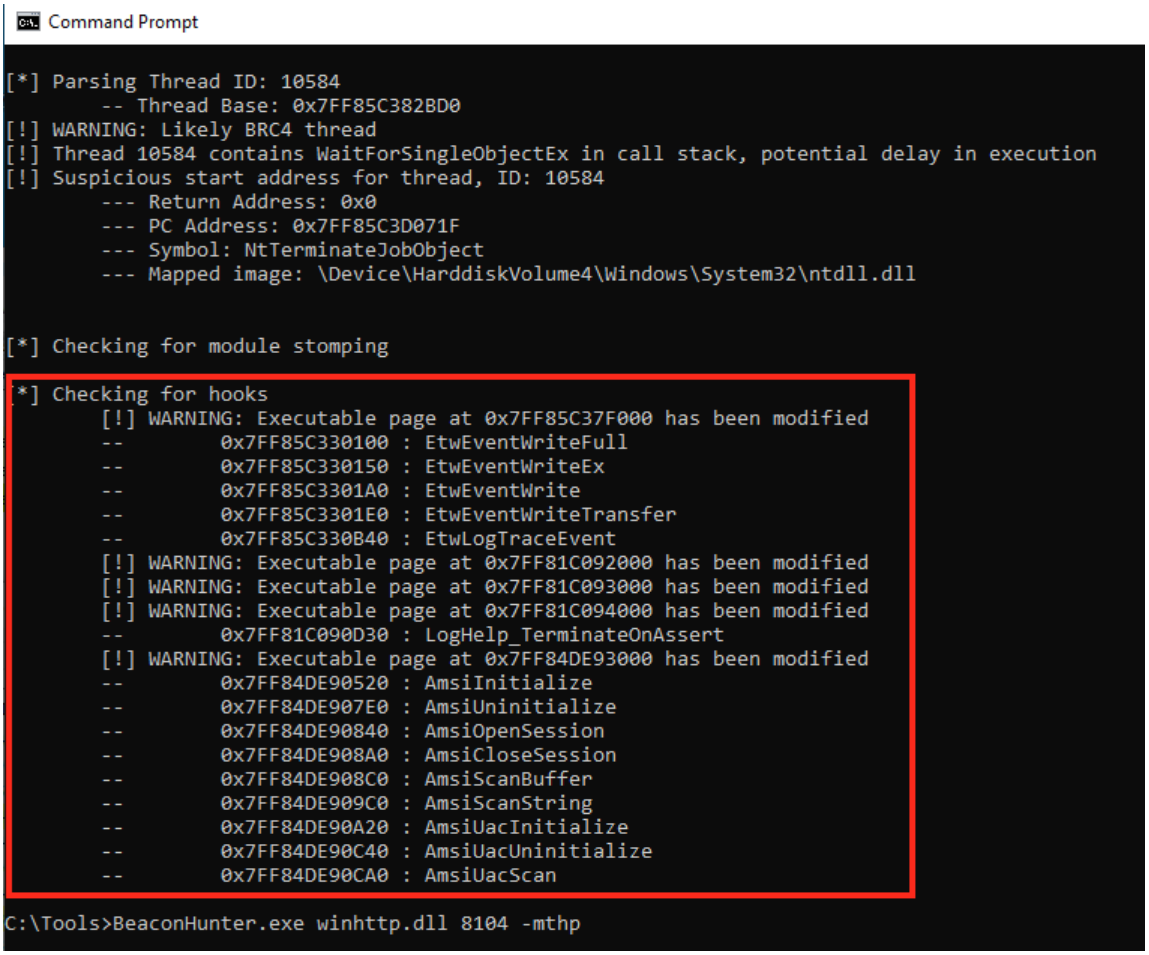


Once the assembly has completed and the beacon gone back to sleep, we can get a better understanding of whats going on by attaching a debugger and disassembling the values of ntdll.dll!EtwEventWrite and amsi.dll!AmsiScanBuffer:

```
0:016> uf ntdll!EtWEventWrite
ntdll!EtwEventWrite:
00007ffa`bb05f1a0 c3              ret
0:016> uf amsi!AmsiScanBuffer
DBGHELP: downstreamstore*https://msdl.microsoft.com/down
amsi!AmsiScanBuffer:
00007ffa`a78635e0 b857000780      mov     eax,80070057h
00007ffa`a78635e5 c3              ret
0:016>
```

As shown above, these are simple and persistent patches to disable .NET ETW data and inhibit AMSI. As the patches are persistent, we can detect them by either of the aforementioned techniques, since not only will we receive a high signal detection due to the first instruction of EtwEventWrite being a ret, but also an indicator that the pages where EtwEventWrite resides have been modified due to the clearing of the shared bit.

Using BeaconHunter, we can rapidly detect these hooks based on resolving the exports on the modified pages, providing a strong indicator that malicious tampering has taken place:



Brute Ratel C2 Server

Moving away from the endpoint, as hunters we also have an interest in detecting the command-and-control infrastructure as this may assist in providing us with sufficient intelligence to detect beaconing based on network telemetry.

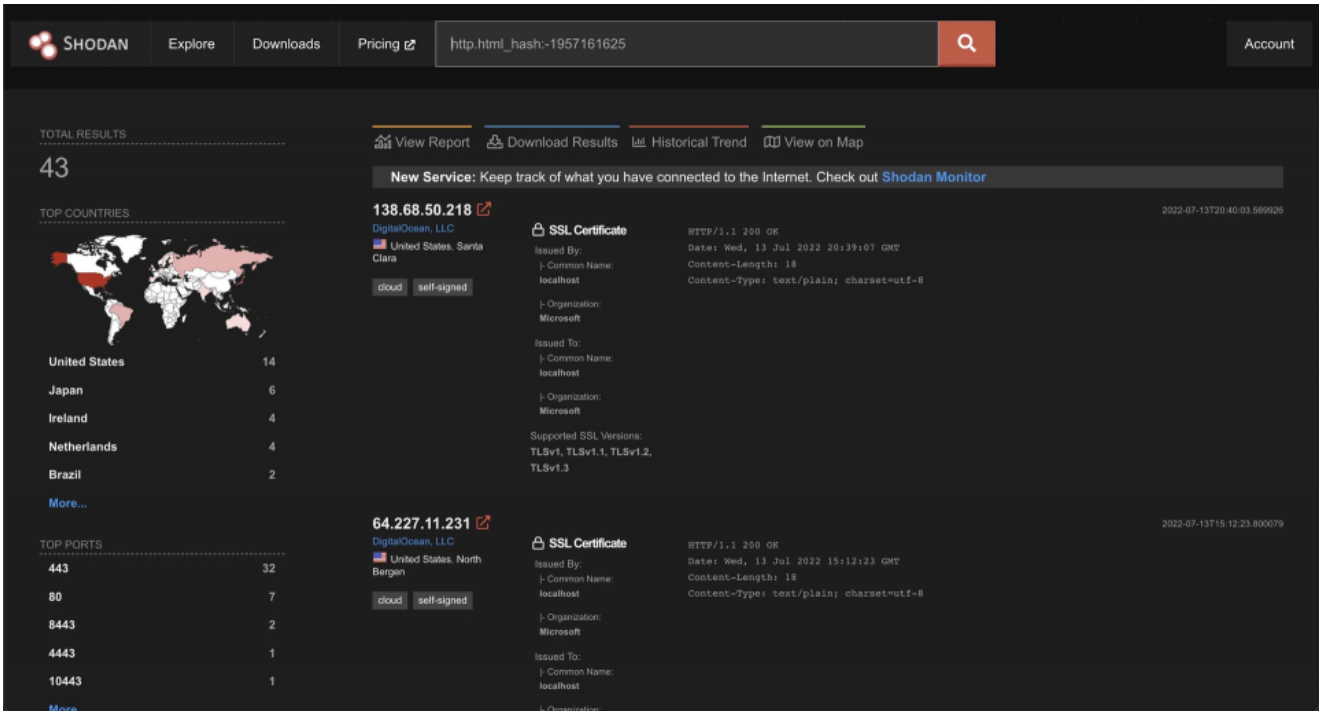
The C2 server for Brute Ratel is developed in golang, and by default only allows the operator to modify the default landing page for the C2. To fingerprint the C2 server, we discovered it was possible to generate an unhandled exception when sending a POST request containing base64 to any URI. For example, consider the following base64 POST data compared with the the plaintext:

It is likely this occurs as the expected input for the base64 decoded POST data should conform to the C2 traffic format. A simple Nuclei rule might help us in scanning for this kind of infrastructure:

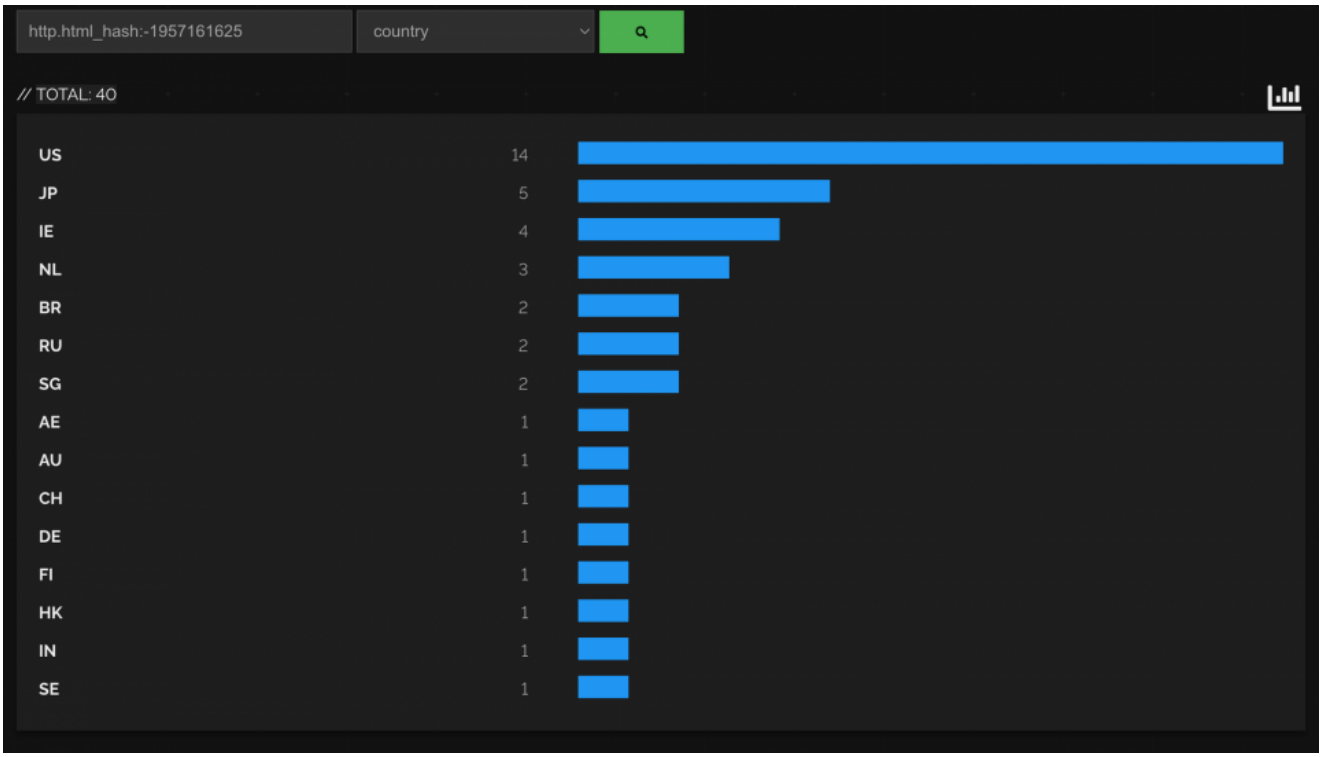
```
id: brc4-ts

info:
  name: Brute Ratel C2 Server Fingerprint
  author: Dominic Chell
  severity: info
  description: description
  reference:
    - https://
  tags: tags
requests:
  - raw:
      - |-
        POST / HTTP/1.1
        Host: {{Hostname}}
        Content-Length: 8

        Zm9vYmFy
```

Outside of direct interaction with the C2, it is also possible to detect C2 infrastructure where the operator has not manually redefined the default landing page based on a hash of the HTML (http.html_hash=-1957161625).

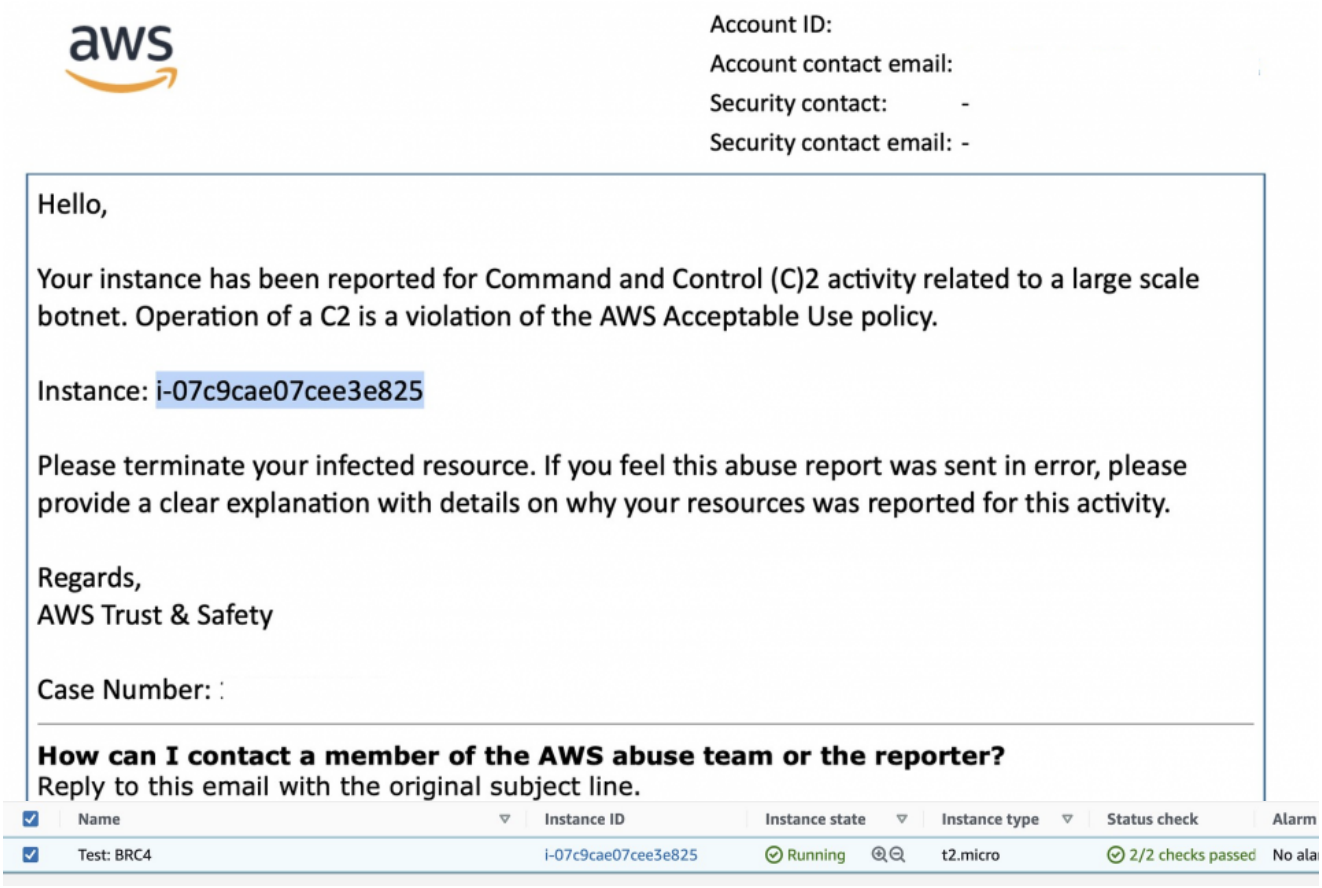
Using a simple Shodan query, we can quickly find live infrastructure exposed to the Internet:



Although only around 40 team servers were identified, we can get a better picture of where these are located based on the geographical spread:



It is quite likely some of these techniques are already known, as based on reports against our test infrastructure, defenders are actively hunting these C2 servers:



Brute Ratel Configurations

Analysis of the Badger revealed that Brute Ratel maintains an encrypted configuration structure in memory which includes details on the C2 endpoints. Being able to extract this from either artifacts or from running processes can prove helpful for defenders. Our analysis revealed that this configuration is held in a base64 and RC4 encrypted blob using a fixed key of “bYXJm/3#M?:XyMBF” in the artifacts for the badger. While the configuration is stored plaintext in memory for the sleeping badger.

We developed the following config extractor that can be used against both on-disk artifacts for BRC4 v1.0.x or injected sleeping badgers with Brute Ratel 1.0.x and 1.1.x:

```
#define _CRT_SECURE_NO_WARNINGS

#include <stdio.h>
#include <stdlib.h>
#include <Windows.h>
#include <string>
#include <vector>

#pragma comment(lib, "Crypt32.lib")

std::string HexDump(void* pBuffer, DWORD cbBuffer)
{
        PBYTE pbBuffer = (PBYTE)pBuffer;
        std::string strHex;

#define FORMAT_APPEND_1(a)      { char szTmp[256]; sprintf(szTmp, a); strHex +=
szTmp; }
#define FORMAT_APPEND_2(a,b)    { char szTmp[256]; sprintf(szTmp, a, b); strHex +=
szTmp; }

        for (DWORD i = 0; i < cbBuffer;)
        {
                FORMAT_APPEND_2("0x8x  ", i);

                DWORD n = ((cbBuffer - i) < 16) ? (cbBuffer - i) : 16;

                for (DWORD j = 0; j < n; j++)
                {
                        FORMAT_APPEND_2("%02X ", pbBuffer[i + j]);
                }

                for (DWORD j = 0; j < (16 - n); j++)
                {
                        FORMAT_APPEND_1("   ");
                }

                FORMAT_APPEND_1(" ");

                for (DWORD j = 0; j < n; j++)
                {
                        FORMAT_APPEND_2("%c", (pbBuffer[i + j] < 0x20 || pbBuffer[i +
j] > 0x7f) ? '.' : pbBuffer[i + j]);
                }

                FORMAT_APPEND_1("\n");

                i += n;
        }

        return strHex;
}
```

```
BOOL ReadAllBytes(std::string strFile, PBYTE* ppbBuffer, UINT* puiBufferLength)
{
	BOOL bSuccess = FALSE;
	PBYTE pbBuffer = NULL;

	*ppbBuffer = NULL;
	*puiBufferLength = 0;

	FILE* fp = fopen(strFile.c_str(), "rb");
	if (fp)
	{
		fseek(fp, 0, SEEK_END);
		long lFile = ftell(fp);
		fseek(fp, 0, SEEK_SET);

		if (!(pbBuffer = (PBYTE)malloc(lFile)))
			goto Cleanup;

		if (fread(pbBuffer, 1, lFile, fp) != lFile)
			goto Cleanup;

		*ppbBuffer = pbBuffer;
		*puiBufferLength = (UINT)lFile;

		pbBuffer = NULL;
		bSuccess = TRUE;
	}

Cleanup:
	if (fp) fclose(fp);
	if (pbBuffer) free(pbBuffer);
	return bSuccess;
}

void Brc4DecodeString(BYTE* pszKey, BYTE* pszInput, BYTE* pszOutput, int cchInput)
{
	BYTE szCharmap[0x100];

	for (UINT i = 0; i < sizeof(szCharmap); i++)
	{
		szCharmap[i] = (char)i;
	}

	UINT cchKey = strlen((char*)pszKey);

	BYTE l = 0;

	for (UINT i = 0; i < sizeof(szCharmap); i++)
	{
		BYTE x = szCharmap[i];
		BYTE k = pszKey[i % cchKey];
		BYTE y = x + k + l;
```

```
                l = y;
                szCharmap[i] = szCharmap[y];
                szCharmap[y] = x;
        }

        l = 0;

        for (UINT i = 0; i < cchInput; i++)
        {
                BYTE x = szCharmap[i + 1];
                BYTE y = x + l;
                l = y;
                BYTE z = szCharmap[y];
                szCharmap[i + 1] = z;
                szCharmap[y] = x;
                x = x + szCharmap[i + 1];
                x = szCharmap[x];
                x = x ^ pszInput[i];
                pszOutput[i] = x;
        }
}

BOOL MatchPattern(PBYTE pbInput, PBYTE pbSearch, DWORD cbSearch, BYTE byteMask)
{
        BOOL bMatch = TRUE;

        for (DWORD j = 0; j < cbSearch; j++)
        {
                if (pbSearch[j] != byteMask && pbInput[j] != pbSearch[j])
                {
                        bMatch = FALSE;
                        break;
                }
        }

        return bMatch;
}

PBYTE FindPattern(PBYTE pbInput, UINT cbInput, PBYTE pbSearch, DWORD cbSearch, BYTE
byteMask, UINT* pcSkipMatches)
{
        if (cbInput > cbSearch)
        {
                for (UINT i = 0; i < cbInput - cbSearch; i++)
                {
                        BOOL bMatch = MatchPattern(pbInput + i, pbSearch, cbSearch,
byteMask);

                        if (bMatch)
                        {
                                if (!*pcSkipMatches)
                                {
```

```
                                        return &pbInput[i];
                                }

                                (*pcSkipMatches)--;
                        }
                }
        }

        return NULL;
}

BOOL LocateBrc4Config(PBYTE pbInput, UINT cbInput, PBYTE* ppbConfig)
{
#define XOR_RAX_RAX                     0x48, 0x31, 0xC0,
#define PUSH_RAX                        0x50,
#define MOV_EAX_IMM32           0xB8, 0xab, 0xab, 0xab, 0xab,
#define MOV_RAX_IMM64           0x48, 0xB8, 0xab, 0xab, 0xab, 0xab, 0xab, 0xab, 0xab,
0xab,
#define PUSH_IMM32                      0x68, 0xab, 0xab, 0xab, 0xab,
#define MOV_EAX_0                       0xB8, 0x00, 0x00, 0x00, 0x00,

        BYTE Pattern1[] =
        {
                XOR_RAX_RAX
                PUSH_RAX
                MOV_EAX_IMM32
                PUSH_RAX
                MOV_RAX_IMM64
                PUSH_RAX
                MOV_RAX_IMM64
                PUSH_RAX
                MOV_RAX_IMM64
                PUSH_RAX
                MOV_RAX_IMM64
                PUSH_RAX
                MOV_RAX_IMM64
                PUSH_RAX
                MOV_RAX_IMM64
        },
        Pattern2[] =
        {
                XOR_RAX_RAX
                PUSH_RAX
                MOV_RAX_IMM64
                PUSH_RAX
                MOV_RAX_IMM64
                PUSH_RAX
                MOV_RAX_IMM64
                PUSH_RAX
                MOV_RAX_IMM64
                PUSH_RAX
                MOV_RAX_IMM64
```

```
                PUSH_RAX
                MOV_RAX_IMM64
                PUSH_RAX
                MOV_RAX_IMM64
        };

        UINT cSkipMatches = 0;

        if (cbInput < 100)
        {
                return FALSE;
        }

        PBYTE pbConfigStart = FindPattern(pbInput, cbInput, Pattern1,
sizeof(Pattern1), 0xab, &cSkipMatches);

        if (!pbConfigStart)
        {
                cSkipMatches = 0;

                pbConfigStart = FindPattern(pbInput, cbInput, Pattern2,
sizeof(Pattern2), 0xab, &cSkipMatches);

                if (!pbConfigStart)
                {
                        return FALSE;
                }
        }

        BYTE Pattern3[] = {
                PUSH_IMM32
                MOV_EAX_0
                PUSH_RAX
                MOV_EAX_0
                PUSH_RAX
                MOV_EAX_0
                PUSH_RAX
        };

        cSkipMatches = 0;

        PBYTE pbConfigEnd = FindPattern(pbConfigStart, cbInput - (pbConfigStart -
pbInput), Pattern3, sizeof(Pattern3), 0xab, &cSkipMatches);

        if (!pbConfigEnd)
        {
                return FALSE;
        }

        *ppbConfig = (PBYTE)malloc(pbConfigEnd - pbConfigStart);

        if (!*ppbConfig)
```

```
        {
                return FALSE;
        }

        memset(*ppbConfig, 0, pbConfigEnd - pbConfigStart);

        pbConfigStart += 4; // skip: XOR_RAX_RAX / PUSH_RAX

        BYTE Pattern4[] = {
                MOV_EAX_IMM32
                PUSH_RAX
        },
        Pattern5[] = {
                MOV_RAX_IMM64
                PUSH_RAX
        };

        for (UINT uiIndex = 0, i = 0; i < pbConfigEnd - pbConfigStart;)
        {
                if (MatchPattern(pbConfigStart + i, Pattern4, sizeof(Pattern4),
0xab))
                {
                        (*ppbConfig)[uiIndex++] = pbConfigStart[i + 4];
                        (*ppbConfig)[uiIndex++] = pbConfigStart[i + 3];
                        (*ppbConfig)[uiIndex++] = pbConfigStart[i + 2];
                        (*ppbConfig)[uiIndex++] = pbConfigStart[i + 1];

                        i += sizeof(Pattern4);
                }
                else if (MatchPattern(pbConfigStart + i, Pattern5, sizeof(Pattern5),
0xab))
                {
                        (*ppbConfig)[uiIndex++] = pbConfigStart[i + 9];
                        (*ppbConfig)[uiIndex++] = pbConfigStart[i + 8];
                        (*ppbConfig)[uiIndex++] = pbConfigStart[i + 7];
                        (*ppbConfig)[uiIndex++] = pbConfigStart[i + 6];
                        (*ppbConfig)[uiIndex++] = pbConfigStart[i + 5];
                        (*ppbConfig)[uiIndex++] = pbConfigStart[i + 4];
                        (*ppbConfig)[uiIndex++] = pbConfigStart[i + 3];
                        (*ppbConfig)[uiIndex++] = pbConfigStart[i + 2];

                        i += sizeof(Pattern5);
                }
                else if (MatchPattern(pbConfigStart + i, Pattern3, sizeof(Pattern3),
0xab))
                {
                        break;
                }
                else
                {
                        return FALSE;
                }
```

```
        }

        std::string config = (char*)*ppbConfig;
        std::reverse(config.begin(), config.end());

        strcpy((char*)*ppbConfig, config.c_str());

        return TRUE;
}

BOOL FromBase64(char* pszString, PBYTE* ppbBinary, UINT* pcbBinary)
{
        DWORD cbBinary = 0;

        if (FAILED(CryptStringToBinaryA(pszString, 0, CRYPT_STRING_BASE64, NULL,
&cbBinary, NULL, NULL)))
        {
                return FALSE;
        }

        *ppbBinary = (PBYTE)malloc(cbBinary + 1);

        if (!*ppbBinary)
        {
                return FALSE;
        }

        if (FAILED(CryptStringToBinaryA(pszString, 0, CRYPT_STRING_BASE64,
*ppbBinary, &cbBinary, NULL, NULL)))
        {
                return FALSE;
        }

        *pcbBinary = cbBinary;

        return TRUE;
}

BOOL ScanProcessForBadgerConfig(HANDLE hProcess, std::string& badgerId,
std::vector<std::wstring>& configStrings)
{
        SIZE_T nBytesRead;
        PBYTE lpMemoryRegion = NULL, pbBadgerStateStruct = NULL;

        printf("[+] Searching process memory for badger state ...\n");

        while (1)
        {
                MEMORY_BASIC_INFORMATION mbi = { 0 };

                if (!VirtualQueryEx(hProcess, lpMemoryRegion, &mbi, sizeof(mbi)))
                {
```

```
                        break;
                }

                if ((mbi.State & MEM_COMMIT) && !(mbi.Protect & PAGE_GUARD) &&
                        ((mbi.Protect & PAGE_READONLY) || (mbi.Protect &
PAGE_READWRITE) || (mbi.Protect & PAGE_EXECUTE_READWRITE)))
                {
                        //printf("[+] Searching process memory at 0x%p (size
0x%x)\n", lpMemoryRegion, mbi.RegionSize);

                        PBYTE pbLocalMemoryCopy = (PBYTE)malloc(mbi.RegionSize);

                        if (!ReadProcessMemory(hProcess, lpMemoryRegion,
pbLocalMemoryCopy, mbi.RegionSize, &nBytesRead))
                        {
                                //printf("[!] Unable to read memory at 0x%p\n",
lpMemoryRegion);
                        }
                        else
                        {
                                for (UINT i = 0; i < mbi.RegionSize - 128 &&
!pbBadgerStateStruct; i++)
                                {
                                        if (memcmp(pbLocalMemoryCopy + i, "b-", 2) ==
0)
                                        {
                                                char* pszEndPtr = NULL;
                                                int badgerId =
strtoul((char*)pbLocalMemoryCopy + i + 2, &pszEndPtr, 10);

                                                if (pszEndPtr !=
(char*)pbLocalMemoryCopy + i + 2 && pszEndPtr && *pszEndPtr == '\\' &&
strnlen(pszEndPtr, 100) > 16)
                                                {
                                                        pbBadgerStateStruct =
lpMemoryRegion + i;
                                                        break;
                                                }
                                        }
                                }
                        }

                        free(pbLocalMemoryCopy);
                        pbLocalMemoryCopy = NULL;
                }

                lpMemoryRegion += mbi.RegionSize;
        }

        if (!pbBadgerStateStruct)
        {
                printf("[!] Failed to find badger state\n");
```

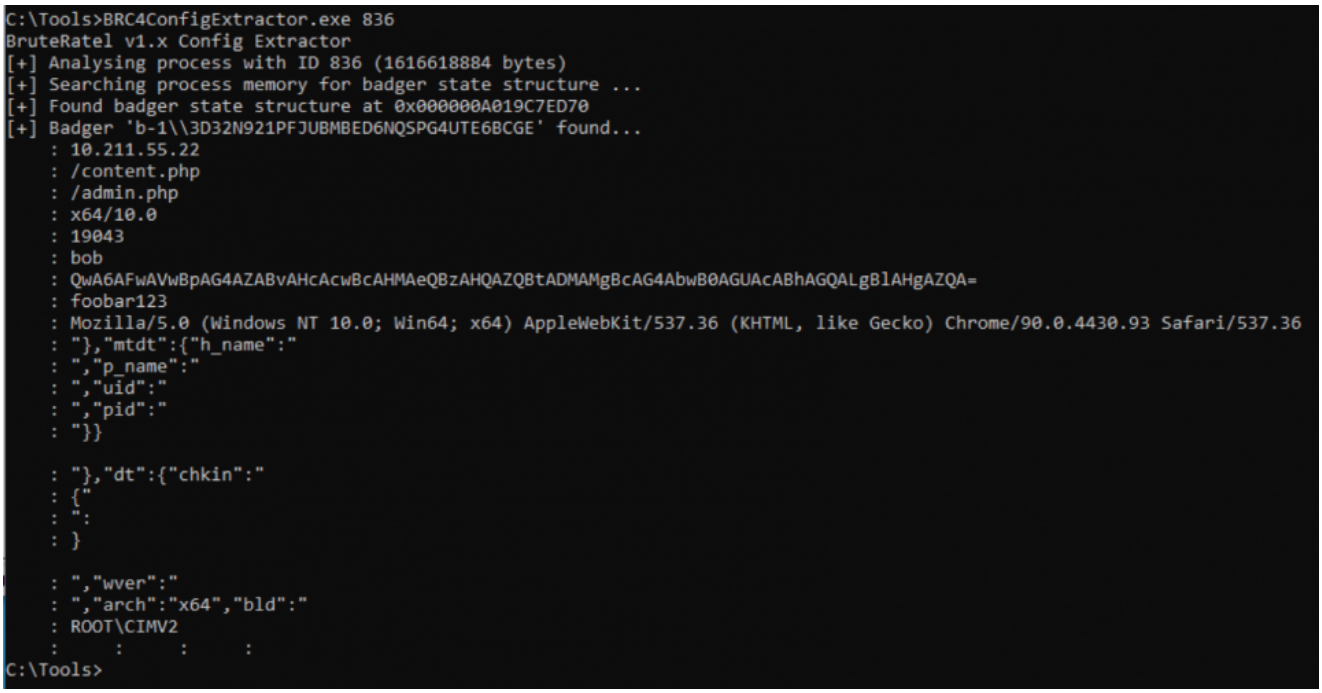
```
                return FALSE;
        }

        printf("[+] Found badger state at 0x%p\n", pbBadgerStateStruct);

        BYTE BadgerState[0x1000];

        memset(BadgerState, 0, sizeof(BadgerState));

        if (!ReadProcessMemory(hProcess, pbBadgerStateStruct, BadgerState, 0x1000,
&nBytesRead))
        {
                if (GetLastError() != ERROR_PARTIAL_COPY)
                {
                        printf("[!] Unable to read badger state at 0x%p\n",
pbBadgerStateStruct);
                        return FALSE;
                }
        }

        badgerId = (char*)BadgerState;

        BYTE ConfigString[1024];

        memset(ConfigString, 0, sizeof(ConfigString));

        for (UINT i = 0x100 + (0x10 - ((DWORD64)pbBadgerStateStruct & 0xf)); i <
sizeof(BadgerState); i += sizeof(DWORD64))
        {
                DWORD64 pMem = *(DWORD64*)(BadgerState + i);

                if (pMem)
                {
                        ConfigString[0] = 0;

                        if (!ReadProcessMemory(hProcess, (LPVOID)pMem, ConfigString,
1024, &nBytesRead) || nBytesRead != 1024)
                        {
                                continue;
                        }

                        BOOL bIsValid = ConfigString[0] != 0;
                        std::wstring badgerString;

#define MIN_STRING_LENGTH       5

                        if (bIsValid)
                        {
                                char* pszConfigString = (char*)ConfigString;

                                for (UINT j = 0; j < nBytesRead && pszConfigString[j]
!= 0; j++)
```

```
                                {
                                        if (!isprint(pszConfigString[j]) && !
(pszConfigString[j] == '\t' || pszConfigString[j] == '\r' || pszConfigString[j] ==
'\n'))
                                        {
                                                break;
                                        }

                                        badgerString.push_back(pszConfigString[j]);
                                }

                                bIsValid = badgerString.size() >= MIN_STRING_LENGTH;
                        }

                        if (!bIsValid)
                        {
                                badgerString.clear();
                                bIsValid = TRUE;

                                WCHAR* pwszConfigString = (WCHAR*)ConfigString;

                                for (UINT j = 0; j < nBytesRead / sizeof(WCHAR) &&
pwszConfigString[j] != 0; j++)
                                {
                                        if (!iswprint(pwszConfigString[j]) && !
(pwszConfigString[j] == '\t' || pwszConfigString[j] == '\r' || pwszConfigString[j] ==
'\n'))
                                        {
                                                break;
                                        }

                                        badgerString.push_back(pwszConfigString[j]);
                                }

                                bIsValid = badgerString.size() >= MIN_STRING_LENGTH;
                        }

                        if (bIsValid)
                        {
                                configStrings.push_back(badgerString);
                        }
                }
        }

        return TRUE;
}

int main(int argc, char *argv[])
{
        PBYTE key = (PBYTE)"bYXJm/3#M?:XyMBF";

        printf("BruteRatel v1.x Config Extractor\n");
```

```
        if (argc < 2)
        {
                printf(
                        "Usage: Brc4ConfigExtractor.exe <file> [key]\n"
                        "    <file|pid> - file to scan for config, or running process
ID\n"
                        "    [key]  - key if not default\n"
                );

                return 1;
        }

        if (argc > 2)
        {
                key = (PBYTE)argv[2];
        }

        if (atoi(argv[1]) == 0)
        {
                PBYTE pbBadger = NULL;
                UINT cbBadger = 0;

                if (!ReadAllBytes(argv[1], &pbBadger, &cbBadger))
                {
                        printf("[!] Input file '%s' not found\n", argv[1]);
                        return 1;
                }

                printf("[+] Analysing file '%s' (%u bytes)\n", argv[1], cbBadger);

                PBYTE pbConfigText = NULL;

                if (!LocateBrc4Config(pbBadger, cbBadger, &pbConfigText))
                {
                        printf("[!] Failed to locate BRC4 config\n");
                        return 1;
                }

                printf("[+] Located BRC4 config: %s\n", pbConfigText);

                PBYTE pbBinaryConfig = NULL;
                UINT cbBinaryConfig = 0;

                if (!FromBase64((char*)pbConfigText, &pbBinaryConfig,
&cbBinaryConfig))
                {
                        printf("[!] Failed to decode BRC4 config from base64\n");
                        return 1;
                }

                Brc4DecodeString(key, pbBinaryConfig, pbBinaryConfig,
```

```
cbBinaryConfig);

                printf("[+] Decoded config: %.*s\n", cbBinaryConfig, pbBinaryConfig);
        }
        else
        {
                DWORD dwPid = atoi(argv[1]);

                printf("[+] Analysing process with ID %u\n", dwPid);

                HANDLE hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, dwPid);

                if (!hProcess)
                {
                        printf("[!] Failed to open process\n");
                        return 1;
                }

                std::string badgerId;
                std::vector<std::wstring> configStrings;

                if (!ScanProcessForBadgerConfig(hProcess, badgerId, configStrings))
                {
                        printf("[!] Failed to locate badger configuration in
memory\n");
                        return 1;
                }

                printf("[+] Badger '%s' found...\n", badgerId.c_str());

                for (auto configString : configStrings)
                {
                        printf("    : %S\n", configString.c_str());
                }

                CloseHandle(hProcess);
        }

        return 0;
}
```

Running the extractor tool on either an artifact or a running process (even while sleeping), will extract the Brute Ratel configuration state for the process or artifact:



Updated v1.1 Analysis

Shortly after our talk on this subject at x33fcon, Brute Ratel announced a new version of the software. As such, it seemed appropriate to analyse this to ensure defenders have accurate advice given the recent uptake in Brute Ratel by threat actors.

Analysis of Obfuscate and Sleep Techniques

One of the things that struck us about the v1.1 release, was the declaration that the author had discovered new sleep and obfuscate techniques. As stated in this YouTube video "**Brute Ratel C4 v/s Nighthawk and Open Source Sleep Obfuscation Techniques**", the author says "I didn't even knew (SIC) about this technique until Austin released the blog post on this. However, Brute Ratel does not use either of these two techniques that we have seen over here." in reference to the APC technique used in Foliage and the Timer based technique as used in MDSec's Nighthawk and as reverse engineered here and a proof of concept implementation released here. Noting that this video appeared a short time after the Ekko release.

Reverse engineering of the obfuscate in sleep techniques used within Brute Ratel v1.1 reveal that three sleeping strategies are now available. The first, as we have previously documented is an extremely similar implementation to @ilove2pwn_'s Foliage, if not an exact copy.

The second implementation, reverse engineering revealed to be an almost identical implementation of @c5pider's Ekko code (and originally discovered by Peter Winter-Smith and used in MDSec's Nighthawk). For example, consider the following taken from Ekko:

github.com/Cracked5pider/Ekko/blob/main/Src/Ekko.c#L97

```
81
82          // VirtualProtect( ImageBase, ImageSize, PAGE_EXECUTE_READWRITE, &OldProtect );
83          RopProtRX.Rsp  -= 8;
84          RopProtRX.Rip   = VirtualProtect;
85          RopProtRX.Rcx   = ImageBase;
86          RopProtRX.Rdx   = ImageSize;
87          RopProtRX.R8    = PAGE_EXECUTE_READWRITE;
88          RopProtRX.R9    = &OldProtect;
89
90          // SetEvent( hEvent );
91          RopSetEvt.Rsp  -= 8;
92          RopSetEvt.Rip   = SetEvent;
93          RopSetEvt.Rcx   = hEvent;
94
95          puts( "[INFO] Queue timers" );
96
97          CreateTimerQueueTimer( &hNewTimer, hTimerQueue, NtContinue, &RopProtRW, 100, 0, WT_EXECUTEINTIMERTHREAD );
98          CreateTimerQueueTimer( &hNewTimer, hTimerQueue, NtContinue, &RopMemEnc, 200, 0, WT_EXECUTEINTIMERTHREAD );
99          CreateTimerQueueTimer( &hNewTimer, hTimerQueue, NtContinue, &RopDelay,  300, 0, WT_EXECUTEINTIMERTHREAD );
100         CreateTimerQueueTimer( &hNewTimer, hTimerQueue, NtContinue, &RopMemDec, 400, 0, WT_EXECUTEINTIMERTHREAD );
101         CreateTimerQueueTimer( &hNewTimer, hTimerQueue, NtContinue, &RopProtRX, 500, 0, WT_EXECUTEINTIMERTHREAD );
102         CreateTimerQueueTimer( &hNewTimer, hTimerQueue, NtContinue, &RopSetEvt, 600, 0, WT_EXECUTEINTIMERTHREAD );
103
104         puts( "[INFO] Wait for hEvent" );
105
106         WaitForSingleObject( hEvent, INFINITE );
107
```

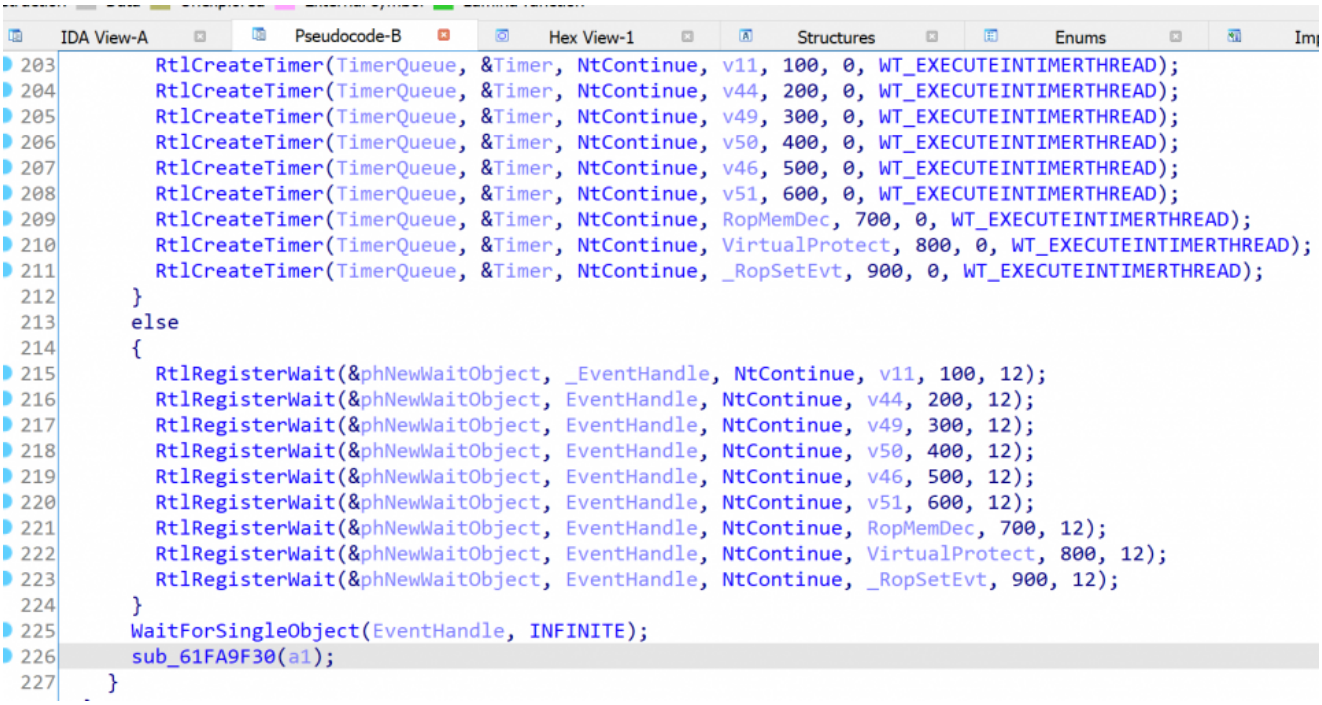
Compare this with the technique implemented inside Brute Ratel:

IDA View-A | Pseudocode-B | Hex View-1 | Structures | Enums | Imp

```
180     v26[17] = v17;
181     v27 = RopMemDec;
182     RopMemDec[31] = v25;
183     v27[19] -= 8i64;
184     v27[16] = v54;
185     v28 = *VirtualProtect_1;
186     v27[17] = v55;
187     v29 = VirtualProtect;
188     VirtualProtect[31] = v28;
189     v30 = NtSetEvent;
190     v29[19] -= 8i64;
191     v29[16] = ImageBase;
192     v29[17] = ImageSize;
193     v29[23] = PAGE_EXECUTE_READ;
194     v29[24] = &OldProtect;
195     RopSetEvt = _RopSetEvt;
196     _RopSetEvt[31] = v30;
197     _EventHandle = EventHandle;
198     RopSetEvt[19] -= 8i64;
199     RopSetEvt[16] = _EventHandle;
200     RopSetEvt[17] = 0i64;
201     if ( UseTimerQueue == 1 )
202     {
203       RtlCreateTimer(TimerQueue, &Timer, NtContinue, v11, 100, 0, WT_EXECUTEINTIMERTHREAD);
204       RtlCreateTimer(TimerQueue, &Timer, NtContinue, v44, 200, 0, WT_EXECUTEINTIMERTHREAD);
205       RtlCreateTimer(TimerQueue, &Timer, NtContinue, v49, 300, 0, WT_EXECUTEINTIMERTHREAD);
206       RtlCreateTimer(TimerQueue, &Timer, NtContinue, v50, 400, 0, WT_EXECUTEINTIMERTHREAD);
207       RtlCreateTimer(TimerQueue, &Timer, NtContinue, v46, 500, 0, WT_EXECUTEINTIMERTHREAD);
208       RtlCreateTimer(TimerQueue, &Timer, NtContinue, v51, 600, 0, WT_EXECUTEINTIMERTHREAD);
209       RtlCreateTimer(TimerQueue, &Timer, NtContinue, RopMemDec, 700, 0, WT_EXECUTEINTIMERTHREAD);
210       RtlCreateTimer(TimerQueue, &Timer, NtContinue, VirtualProtect, 800, 0, WT_EXECUTEINTIMERTHREAD);
211       RtlCreateTimer(TimerQueue, &Timer, NtContinue, _RopSetEvt, 900, 0, WT_EXECUTEINTIMERTHREAD);
```

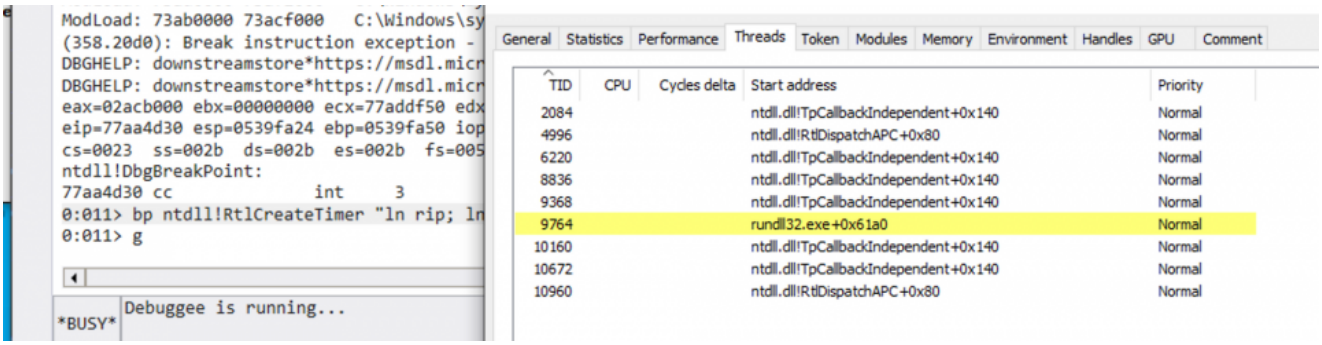
As you can see, the code is almost identical; indeed the few changes include replacing the WinApi calls for CreateTimerQueueTimer with the Rtl wrapper RtlCreateTimer, noting that the breakpoints for Rtl wrappers were avoided (likely intentionally) in the aforementioned video demonstration.

This brings us to the third technique used by Brute Ratel which is a variation of timers and is not publicly documented. We can see here that this technique uses a subtle variation on timers and instead proxies the timer through RtlRegisterWait:



While this technique is not publicly documented, it has been available in Nighthawk for some time, coincidentally with the same values used for many of the constants. Further coincidences arise with other undocumented/unpublished features arising in the Brute Ratel v1.1 release.

So far, we have only discussed the sleeping techniques available in the x64 implementation of Brute Ratel. Analysis of the x86 implementation shows that the obfuscate and sleep strategies are fixed to the aforementioned APC Foliage based implementation (noting the breakpoints never hit):

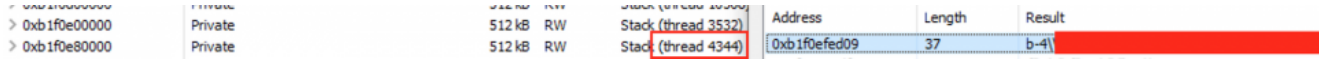


To date there are no public or open source x86 implementations of obfuscate and sleep strategies that use timers, limiting the available opportunities to easily integrate such code without custom development.

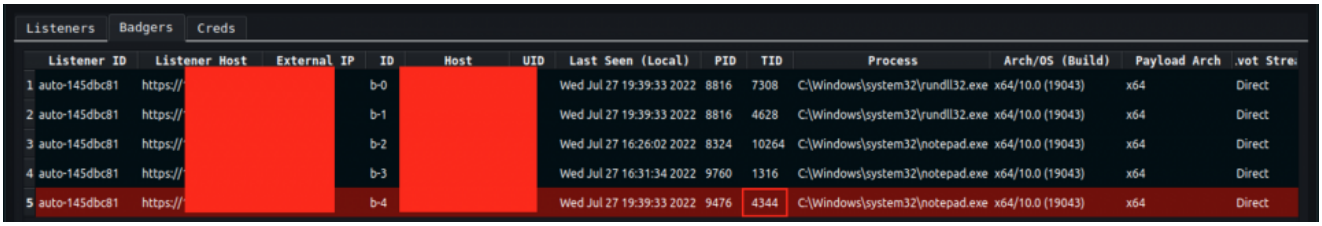
In Memory Detections

One of the updates in the v1.1 release implies that the .rdata section is now also obfuscated, in order to hide strings such as “[+] AMSI Patched” which were exposed in the memory of the sleeping badger. However, even cursory memory analysis shows there remains many exposed strings within the memory of the sleeping badger. As a result, this means there are many opportunities to pluck out Brute Ratel processes on an endpoint, even while the badger is sleeping. For example, consider the Brute Ratel C2 data which is stored in a JSON format, simply searching for one of its unique parameters in memory such as “chkin” will allow us to spot a badger:

Or simply searching for the badger identifier (e.g. b-) will find them scattered all over both the heap and the stack. As a bonus, this can act as simple mechanism to spot the thread that Brute Ratel is operating from, for example:



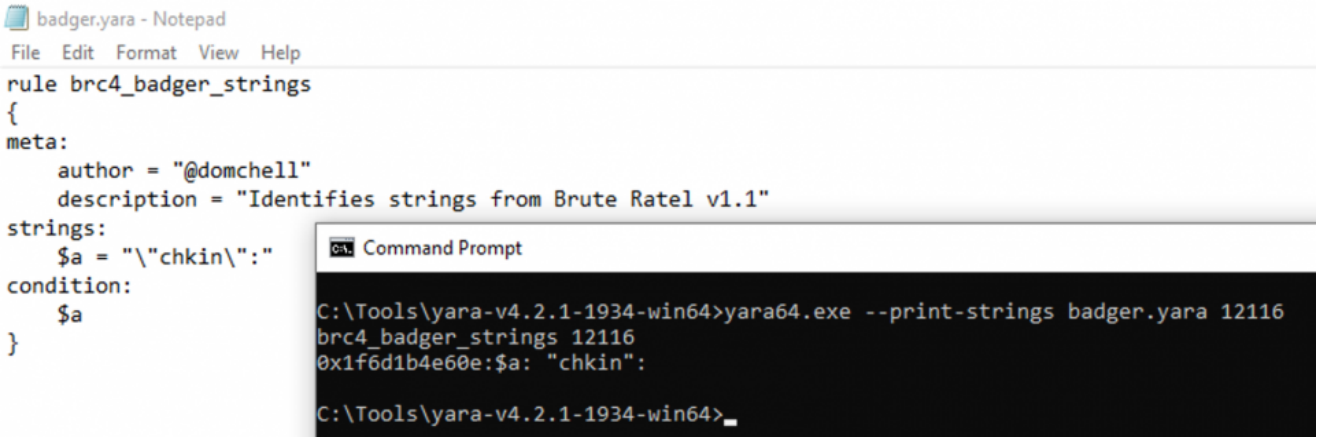
Here we can see the presence of the “b-4\” on the stack of thread 4344. We can confirm that is indeed the thread for Brute Ratel from the UI:



With this in mind, we’re able to build a simple but effective Yara rule to pluck sleeping Brute Ratel processes from memory:

```
rule brc4_badger_strings
{
meta:
    author = "@domchell"
    description = "Identifies strings from Brute Ratel v1.1"
strings:
    $a = "\"chkin\":"
condition:
    $a
}
```

Executing the Yara rule, we can spot the sleeping badger:



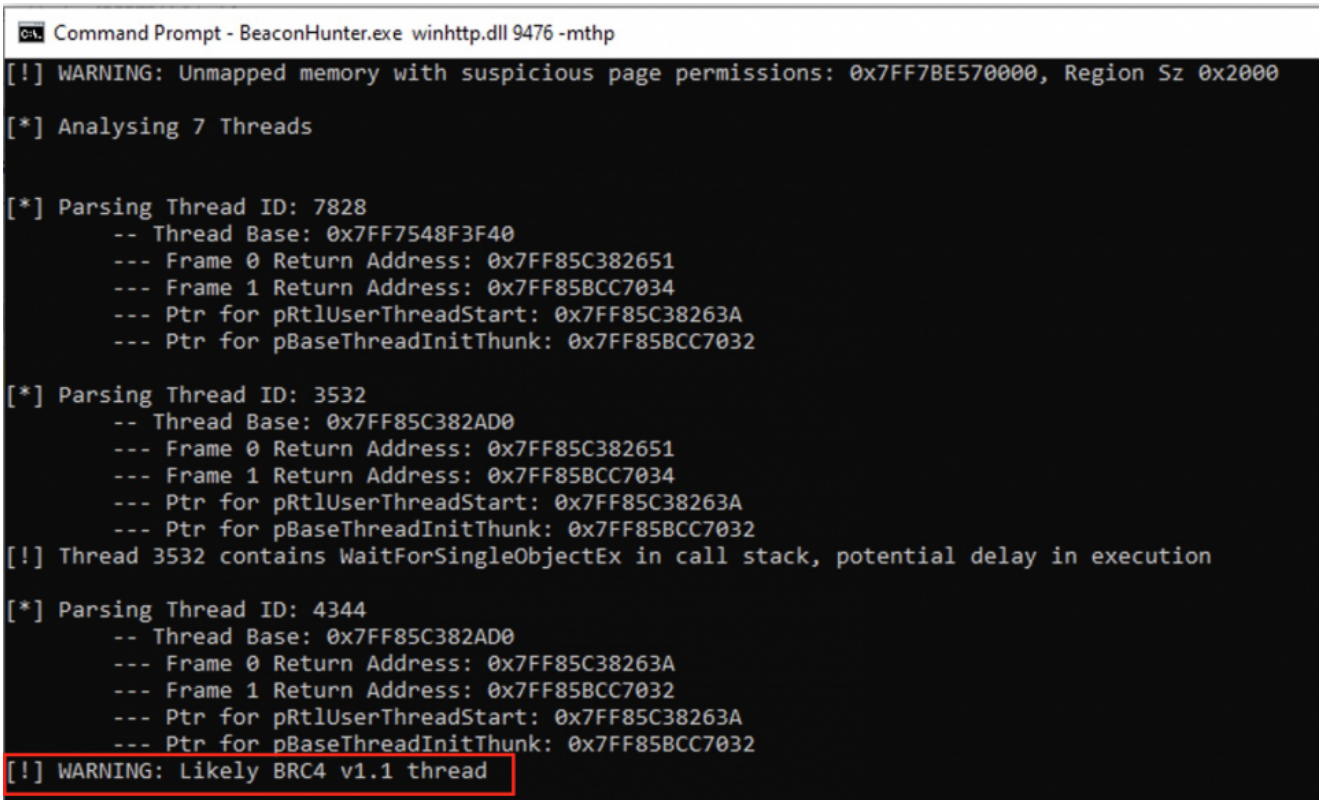
The detections documented in v1.0 for post-exploitation actions such as suspicious copy on write operations remain relevant and still offer an effective means of detection for BRC4 post-exploitation.

Thread Stack Spoofing

In the v1.0 release of Brute Ratel, as we noted the start address of the thread is hardcoded to ntdll!TpReleaseCleanupGroupMembers+0x550. Version 1.1 proclaims to offer “full thread stack masquerading”. Analysis of the stack spoofing for Brute Ratel reveals a simplistic implementation of rewriting the threads call stack. This process occurs just prior to the badger going to sleep, using the aforementioned timer technique. In an attempt to make the thread appear more legitimate, a new thread stack is created with hardcoded addresses for the first two frames. The addresses hardcoded are at offsets 0xa and 0x12 from RtlUserThreadStart and BaseThreadInitThunk respectively:

```
26      BRC4_memcpy(v51, v42, 1232i64);
27      v8 = v52;
28      v9 = NtWaitForWorkViaWorkerFactory;
29      v52->ContextFlags = CONTEXT_FULL;
30      v8->Rip = v9;
31      ret_addr = read_ret_addr();
32      v11 = RopProtRW;
33      v8->Rsp = ret_addr;
34      v12 = v52;
35      _Rsp = v52->Rsp;
36      v52->Rsp = _Rsp - 40960;
37      *(_Rsp - 40960) = RtlAcquireSRWLockExclusive + 288;
38      *(v12->Rsp + 8) = BaseThreadInitThunk + 18;
39      *(v12->Rsp + 56) = RtlUserThreadStart + 10;
40      v14 = *VirtualProtect_1;
41      v15 = SystemFunction032;
42      v11[19] -= 8i64;
43      v11[31] = v14;
44      v16 = v44;
45      v11[16] = ImageBase;
```

We were able to identify any other threads using these hardcoded start addresses, as such it becomes trivial to identify any Brute Ratel threads on a system. To detect these threads, we updated BeaconHunter accordingly to identify threads with the first two frames at RtlUserThreadStart+0xa and BaseThreadInitThunk+0x12:



Updated rDLL Extraction

Shortly after our analysis at x33fcon, Brute Ratel announced an update to the method in which the artifacts hide the reflective DLL. Analysis of these artifacts revealed that this is achieved using RC4 to encrypt the reflective DLL with a random key; the PE header is then stomped. The 8 byte RC4 key is appended to the encrypted reflective DLL, followed by 400 bytes of base64 configuration file.

We developed the following tool targeting Brute Ratel v1.1 to extract the reflective DLL from DLL and EXE artifacts:

```
//
// only works with BRC4 1.1 binaries.
//
#include <algorithm
#include <windows.h>
#include <cstdio>
#include <string>
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <iomanip>

typedef struct _RC4_CTX {
    BYTE       x, y;
    BYTE       s[256];
} RC4_CTX, *PRC4_CTX;

std::vector<BYTE>
ReadData(std::string path) {
    std::ifstream instream(path, std::ios::in | std::ios::binary);
    std::vector<BYTE> input((std::istreambuf_iterator<char>(instream)),
std::istreambuf_iterator<char>());
    return input;
}

bool
WriteData(std::string path, std::vector<BYTE> data) {
    std::ofstream outstream(path, std::ios::out | std::ios::binary);
    std::copy(data.begin(), data.end(), std::ostreambuf_iterator<char>(outstream));
    return outstream.good();
}

BYTE
start_sig[]={
#if defined(_WIN64)
    0x55, 0x50, 0x53, 0x51, 0x52, 0x56, 0x57, 0x41, 0x50, 0x41, 0x51, 0x41, 0x52,
0x41, 0x53, 0x41,
    0x54, 0x41, 0x55, 0x41, 0x56, 0x41, 0x57, 0x48, 0x89, 0xE5, 0x48, 0x83, 0xE4,
0xF0, 0x48, 0x31,
    0xC0, 0x50
#else
    0x60, 0x89, 0xE5, 0x83, 0xE4, 0xF8, 0x31, 0xC0, 0x50
#endif
};

BYTE
end_sig[]={
#if defined(_WIN64)
    0x41, 0x5F, 0x41, 0x5E, 0x41, 0x5D, 0x41, 0x5C, 0x41, 0x5B, 0x41, 0x5A, 0x41,
0x59, 0x41, 0x58,
    0x5F, 0x5E, 0x5A, 0x59, 0x5B, 0x58, 0x5D, 0xC3
```

```
#else
    0x83, 0xC4, 0x10, 0x61, 0xC3
#endif
};

void
RC4_set_key(
    PRC4_CTX c,
    PVOID    key,
    UINT     keylen)
{
    UINT i;
    UCHAR j;
    PUCHAR k=(PUCHAR)key;

    for (i=0; i<256; i++) {
        c->s[i] = (UCHAR)i;
    }

    c->x = 0; c->y = 0;

    for (i=0, j=0; i<256; i++) {
        j = (j + (c->s[i] + k[i % keylen]));
        UCHAR t = c->s[i];
        c->s[i] = c->s[j];
        c->s[j] = t;
    }
}

void
RC4_crypt(
    PRC4_CTX c,
    PUCHAR   buf,
    UINT     len)
{
    UCHAR x = c->x, y = c->y, j=0, t;

    for (UINT i=0; i<len; i++) {
        x = (x + 1);
        y = (y + c->s[x]);
        t = c->s[x];
        c->s[x] = c->s[y];
        c->s[y] = t;
        j = (c->s[x] + c->s[y]);
        buf[i] ^= c->s[j];
    }
    c->x = x;
    c->y = y;
}

std::vector<BYTE>
extract_encrypted_rdll(PBYTE ptr, DWORD maxlen) {
```

```
    std::vector<BYTE> outbuf;
    printf("Searching %ld bytes.\n", maxlen);

    for (DWORD i=0; i<maxlen;) {
        if (!memcmp(&ptr[i], end_sig, sizeof(end_sig))) {
            printf("Reached end of signature...\n");
            break;
        }
    #if defined(_WIN64)
        if ((ptr[i] & 0x40) == 0x40 && (ptr[i+1] & 0xB0) == 0xB0)
        {
            BYTE buf[8];

            buf[0] = ptr[i + 9];
            buf[1] = ptr[i + 8];
            buf[2] = ptr[i + 7];
            buf[3] = ptr[i + 6];
            buf[4] = ptr[i + 5];
            buf[5] = ptr[i + 4];
            buf[6] = ptr[i + 3];
            buf[7] = ptr[i + 2];

            outbuf.insert(outbuf.end(), buf, buf + sizeof(buf));
            i += (ptr[i + 10] == 0x41) ? 12 : 11;
        } else i++;
    #else
        if ((ptr[i] & 0xB0) == 0xB0 && (ptr[i+5] & 0x50) == 0x50) {
            BYTE buf[4];

            buf[0] = ptr[i + 4];
            buf[1] = ptr[i + 3];
            buf[2] = ptr[i + 2];
            buf[3] = ptr[i + 1];

            outbuf.insert(outbuf.end(), buf, buf + sizeof(buf));
            i += 6;
        } else i++;
    #endif
    }
        std::reverse(outbuf.begin(), outbuf.end());
    return outbuf;
}

int
main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("usage: decrypt_brc4 <DLL|EXE>\n");
        return 0;
    }

    std::vector<BYTE> inbuf, infile = ReadData(argv[1]);
    DWORD len=0, ptr=0;
```

```
if (infile.empty()) {
    printf("Nothing to read.\n");
    return 0;
}

do {
    auto dos = (PIMAGE_DOS_HEADER)infile.data();
    auto nt = (PIMAGE_NT_HEADERS)(infile.data() + dos->e_lfanew);
    auto s = IMAGE_FIRST_SECTION(nt);

    for (DWORD i=0; i<nt->FileHeader.NumberOfSections; i++) {
        char Name[IMAGE_SIZEOF_SHORT_NAME + 1] = {0};
        memcpy(Name, s[i].Name, IMAGE_SIZEOF_SHORT_NAME);

        if (std::string(Name) == ".data") {
            len = s[i].SizeOfRawData;
            ptr = s[i].PointerToRawData;
            break;
        }
    }

    if (!len) {
        printf("Unable to locate .data section.\n");
        break;
    }

    printf("Searching %ld bytes for loader...\n", len);

    for (DWORD idx=0; idx<len - sizeof(start_sig); idx++) {
        if(!memcmp(infile.data() + ptr + idx, start_sig, sizeof(start_sig))) {
            printf("Found signature : %08lX\n", ptr + idx);
            inbuf = extract_encrypted_rdll(infile.data() + ptr + idx, len - idx);
            break;
        }
    }

    if (inbuf.size()) {
        printf("size : %zd\n", inbuf.size());
        RC4_CTX c;
        BYTE key[8+1] = {0};
        memcpy((char*)key, inbuf.data() + inbuf.size() - 400 - 8, 8);

        //
        // Decrypt RDLL. The additional 400 bytes are base64 configuration.
        //
        RC4_set_key(&c, key, 8);
        RC4_crypt(&c, inbuf.data(), inbuf.size() - 400);

        //
        // Fix DOS header.
        //
```

```
            inbuf[0] = 'M';
            inbuf[1] = 'Z';
            WriteData(std::string(argv[1]) + ".dll", inbuf);
        }
    } while (FALSE);

    return 0;
}
```

Conclusion

In summary, we've highlighted a number of techniques to detect Brute Ratel both in its artifacts, in-memory, through threat hunting and across the network. As this framework grows in popularity with threat actors, it is important to understand the many ways in which it can be detected. As a side note, we have also illustrated how the framework takes close inspiration from the many available open source community tools; knowledge of these can assist in reverse engineering the framework and provide a better understanding of its capabilities (and by virtue its detection points).

This blog post was written Dominic Chell.

written by

MDSec Research

Ready to engage with MDSec?

Get in touch
Stay updated with the latest
news from MDSec.

Newsletter Signup Form