

Analyzing a Brute Ratel Badger

blog.spookysec.net/analyzing-brc4-badgers/



09 Jul 2022

Now a days Brute Ratel (sometimes called the “Angry Monkey C2”) seems to be a hot topic within the information security community. There’s been lots of drama surrounding the author (ParanoidNinja), rumors of the C2 being backdoored, and even some blog_posts from well known and respected individuals within the security community indicating that the C2 framework is potentially being used by APT29 (aka the Russian State Sponsored groups).

So, with all these controversies, where do we go from here? Well, validating the claim that the C2 Framework is backdoored can be quite difficult to prove as that would involve me spending several thousand dollars to acquire the framework itself... So, that’s not exactly feasible. I can however get the next best thing. A Brute Ratel Beacon, or Agent (or as they like to call it, a “Badger”).

Acquiring a Badger for Analysis

How can we do this exactly? Fortunately, I have a VirusTotal Enterprise license! This means we can pull down (download) a publicly tagged “Brute Ratel” sample from the community. To do so, we’re going to use a search for something like `Comment:"Brute Ratel"` and see if we get any hits...

comment "Brute Ratel"

FILES 6 / 6 90 days

	Detections	Size	Sort by	Export	Tools	Help
			First seen	Last seen	Submitters	
<input type="checkbox"/> 31ACF37018A894FBCF644EC5029C3E19C947641A209CE3CE56071C1F5760869 No meaningful names peexe assembly overlay signed @4bits native	32 / 60	284.92 KB	2022-06-26 14:14:28	2022-06-26 14:14:28	1	
<input type="checkbox"/> F58AE9193802E98AF17E689E9F0BE3E9319C5027726D08802E3E82038014D46 Decret.iso img contains-pe	27 / 58	2.44 MB	2022-06-15 15:16:20	2022-06-15 15:16:20	1	
<input type="checkbox"/> 3AD53495851BAFC48CAF6D2227A434CA2E0B6EF9AB3BD40ABFE4EA8F318D37BBE badger_x64.exe peexe overlay runtime-modules checks-network-adapters direct-cpu-clock-access @4bits	40 / 69	250.79 KB	2022-05-20 20:45:39	2022-05-20 20:45:39	1	
<input type="checkbox"/> E42876E917541086F6719F88EE448955368758C708F78ED7308FE9A7808E669 /Volumes/17_05_2822/version.d11 peexe @4bits assembly	39 / 68	253.50 KB	2022-05-19 13:06:58	2022-05-19 13:06:58	1	
<input type="checkbox"/> 1FCT78E1854D5ACE8F1DE8CC395976881C7A83C7926C83172A300AA672698042C Roshan_CV.iso isoimage contains-pe	26 / 58	4.62 MB	2022-05-19 13:06:36	2022-05-19 13:06:36	1	
<input type="checkbox"/> 85D103C14EC2FEF86E7D8B7998BC450F47449348D66256A4E8B82D79E35236E DnsDr1ve.Update	18 / 57	270.57 KB	2022-05-19 10:20:36	2022-05-19 10:20:36	1	

VirusTotal

[Contact Us](#)

[Get Support](#)

[How It Works](#)

[ToS | Privacy Policy](#)

[Blog | Releases](#)

Community

[Join Community](#)

[Vote and Comment](#)

[Contributors](#)

[Top Users](#)

[Community Buzz](#)

Tools

[API Scripts](#)

[YARA](#)

[Desktop Apps](#)

[Browser Extensions](#)

[Mobile App](#)

Premium Services

[Intelligence](#)

[Hunting](#)

[Graph](#)

[API v3 | v2](#)

Documentation

[Intelligence](#)

[Hunting](#)

[Graph](#)

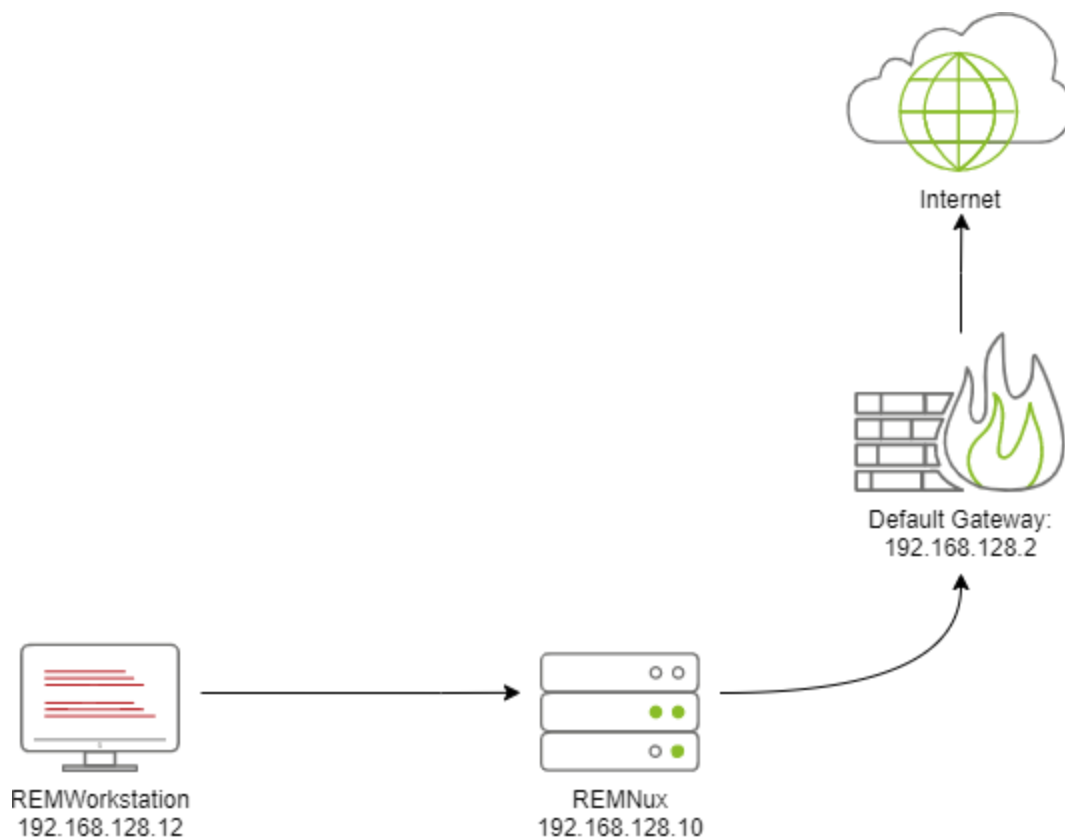
[API v3 | v2](#)

[Use Cases](#)

Suprise Surprise, we got six hits! Let's go with the most obvious one, badger_x64.exe (SHA256 Sum: 3ad53495851bafc48caf6d2227a434ca2e0bef9ab3bd40abfe4ea8f318d37bbe).

Lab Setup

For this lab, we will be using REMWorkstation + REMnux. Here's a diagram that breaks down the lab setup:



- REMWorkstation has the IP Address of 192.168.128.12
- REMNux has the IP Address of 192.168.128.10
- Default Gateway has the IP Address of 192.168.128.2
- REMNux **can** route to 192.168.128.2, but the route is not configured.
- If REMNux is configured to route to the Default Gateway, outbound traffic to the internet **is** allowed

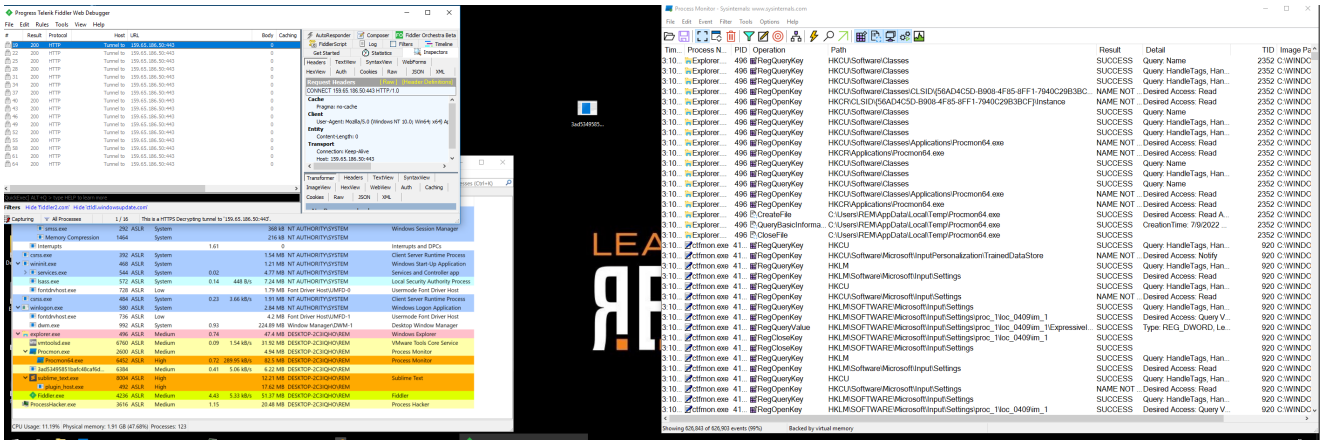
In addition:

- REMNux will have an iptables rule that will accept all and any traffic going into it.
- REMNux will be running FakeDNS and iNetSim
- REMNux will be running WireShark
- REMWorkstation will be running Fiddler

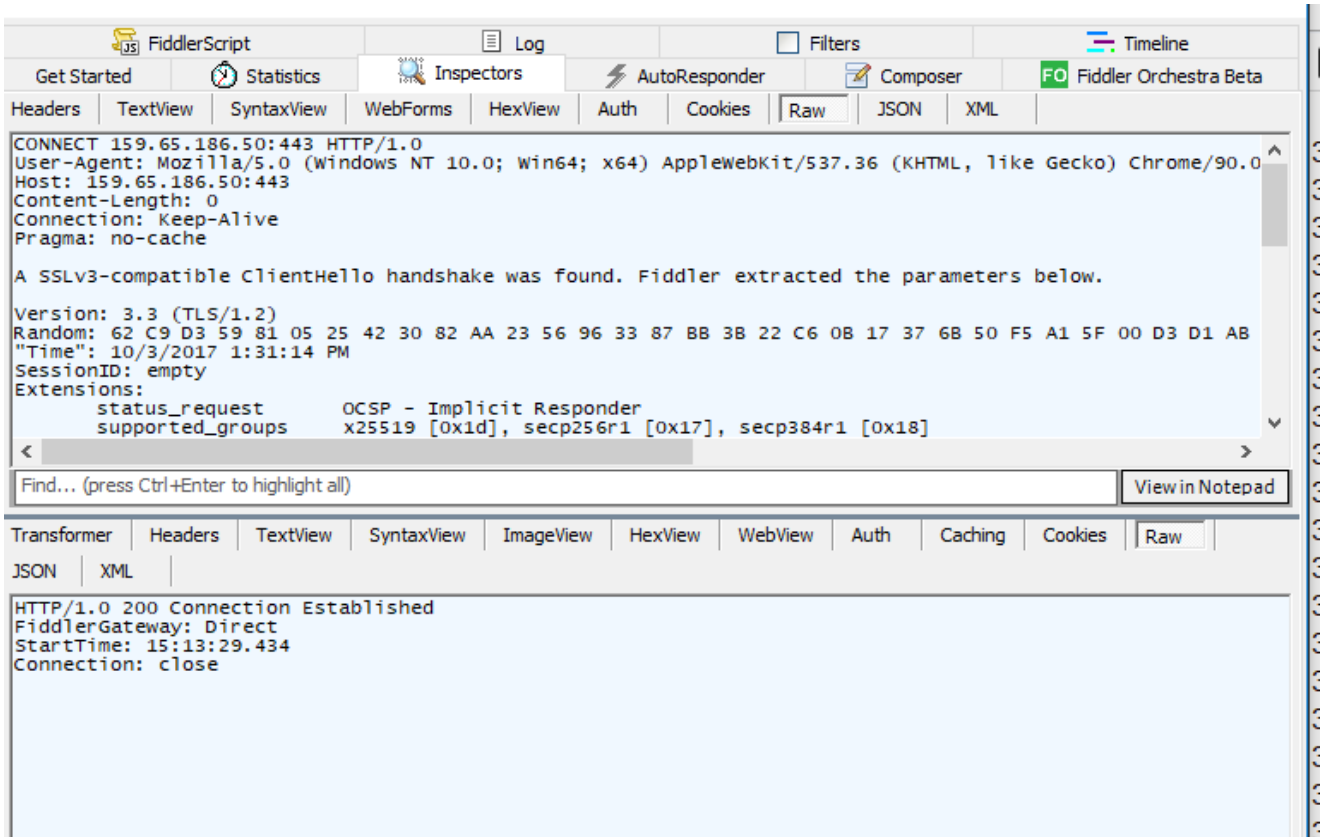
And thats our lab!

Dynamic Analysis - Malware Detonation

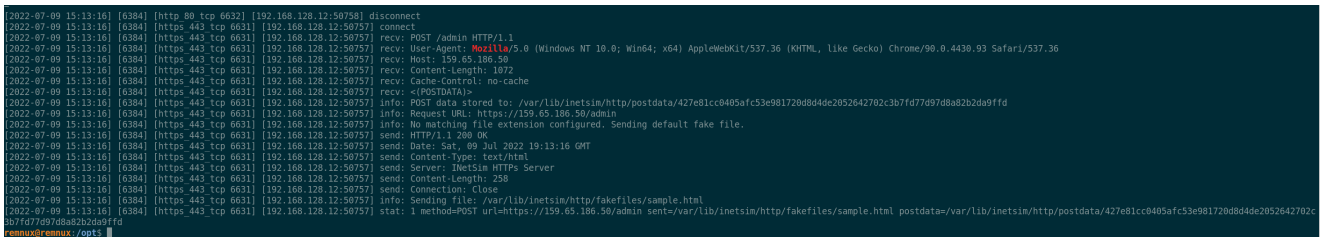
Now that we have our sample acquired, and you're familiar with my lab setup, let's double click some EXEs!



So, right off the bat, we can see some beacons to 156.65.186.50 over HTTPS. Looking at these requests in Fiddler, we can see that the sample is using the 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** with no extra headers.



This is surprisingly bare. Let's pivot over to iNetSim and see whats going on over there.



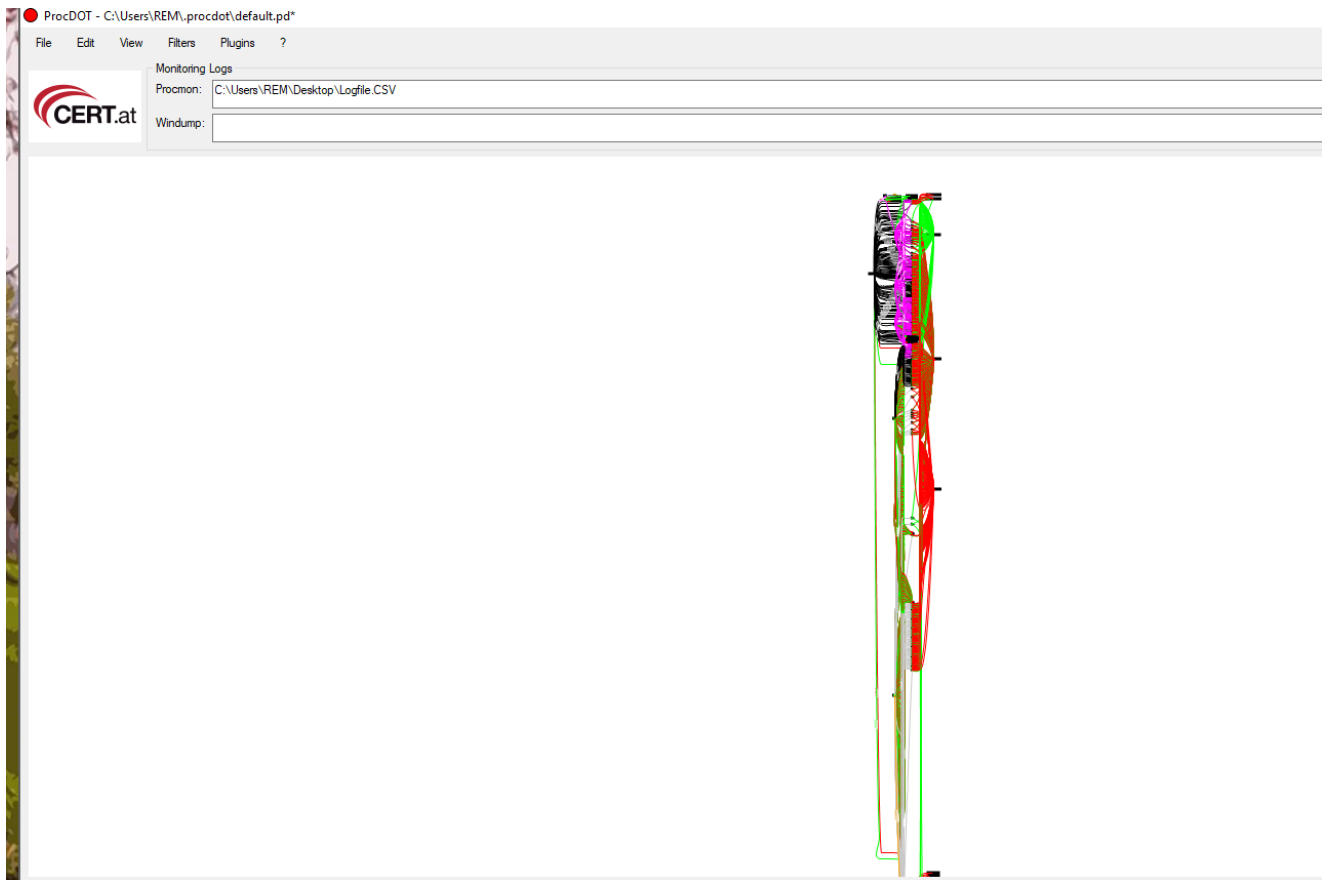
On that side, we can see a little bit more. The file that the “Badger” requested is `/admin` , and there is also some POST data that we missed!

Let’s see if we can find that in Fiddler... Unfortunately, I could not find the request in Fiddler, I’ll have to revert and redetonate the sample in a bit...

Edit: Fiddler actually caused some issues w/ cutting the POST data off to inetsim :(.

Procmon/ProcDot Analysis

For now - Let’s move over to ProcMon and ProcDot and see what the badger is looking for.



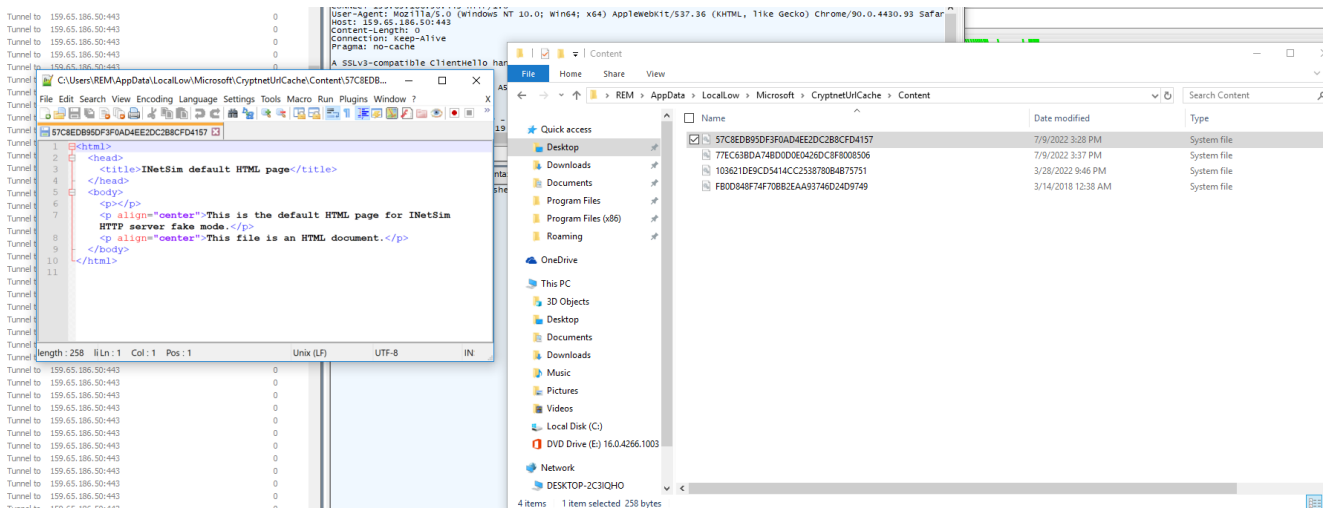
Starting out, this is an absolutely massive graph. Let’s start from the top and work our way down.

At the top:



It appears that the badger is first checking to see if there are any registry keys correlated to a proxy on the system. Since no proxies are in place, BRC4 likely found nothing.

On the far right, we can see a couple of cached web page responses saved to disk. If you'd like to read that data - all it contains is the iNetSim HTTP Response.

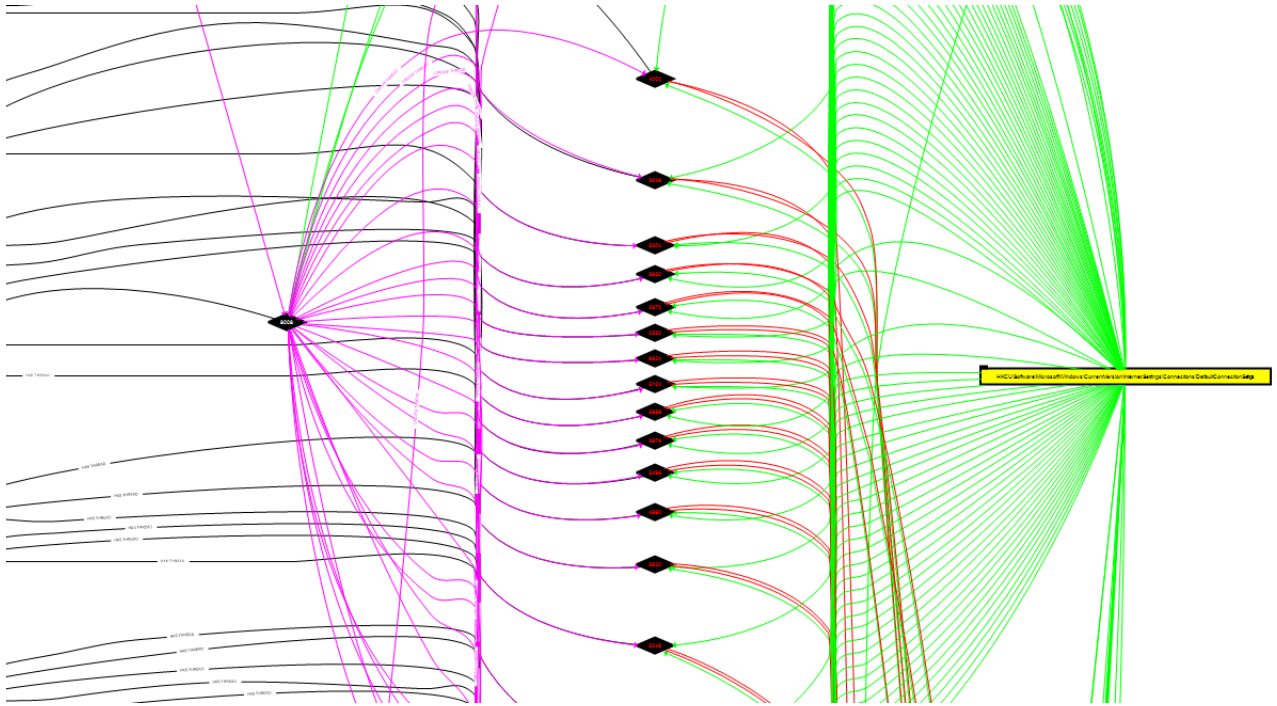


Moving on down the graph, we can see another read attempt on another registry key relating to proxies:

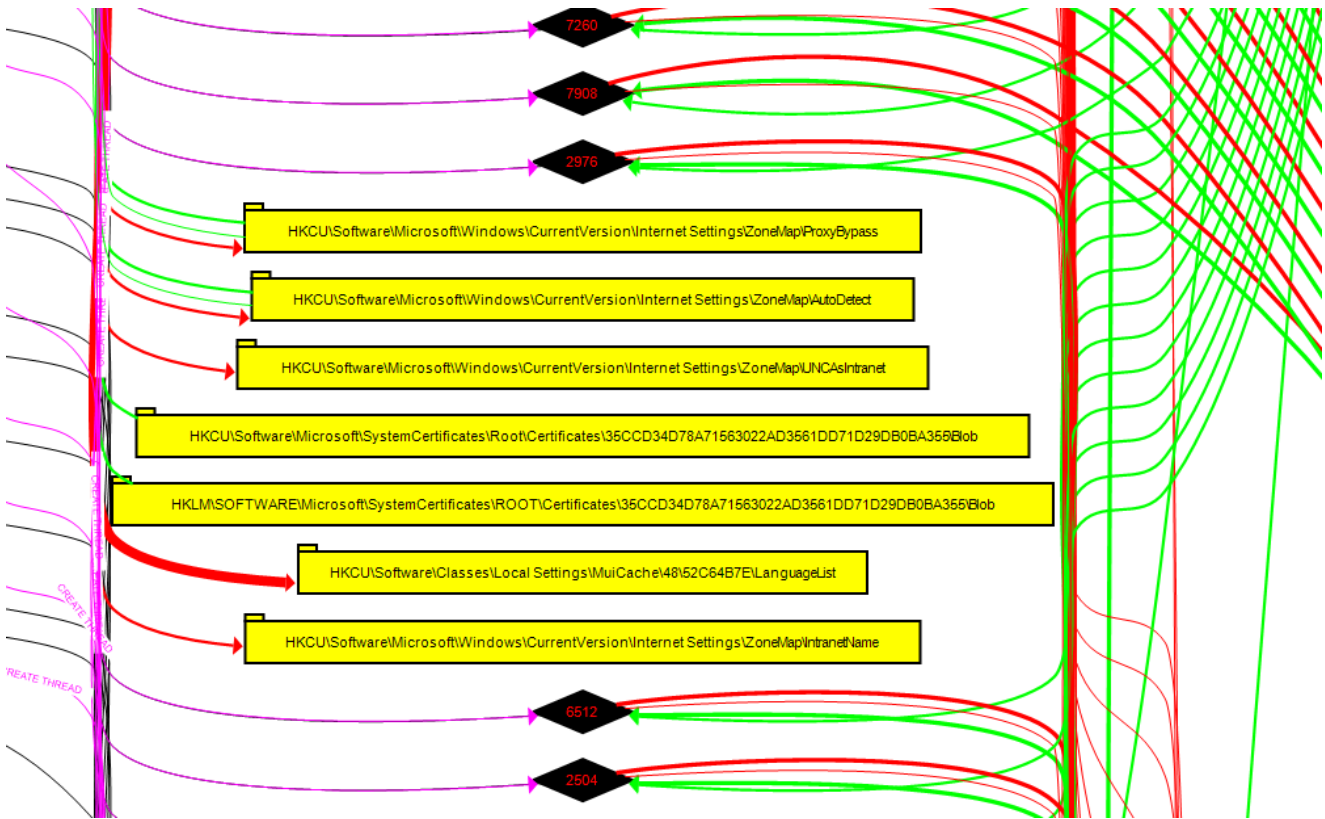


One interesting thing I'd like to point out is the Badger is leveraging a bunch of ThreadCreates and ThreadOpens to potentially confuse AV or EDR.

Zooming out, all the black diamonds are all new threads and Thread ID Numbers.



Scrolling down a bit more, this pattern continues. More Threads being created to read registry keys relating to proxies:



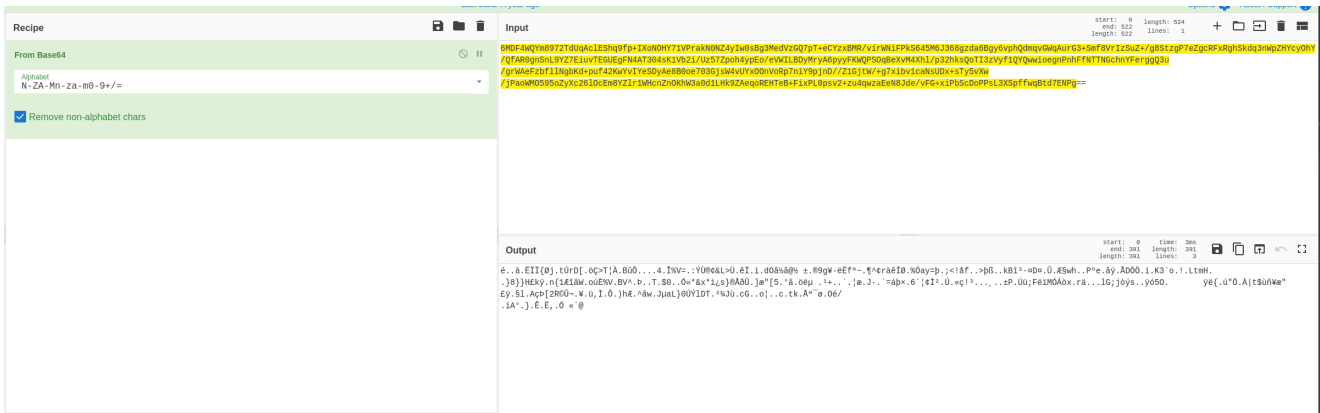
Back to iNetSim

Now that we know a bit more about what the program is trying to do, let's go back to iNetSim and read the POST data from the Web Server.

All of the POST data is stored in `/var/lib/inetsim/postdata/*`. I hope that helps someone in the future... :)

```
2022-07-09 15:59:10 [14992] [https 443 tcp 14352] [192.168.128.12:51876] disconnect
2022-07-09 15:59:10 [14992] [https 443 tcp 14352] [192.168.128.12:51876] connect
2022-07-09 15:59:10 [14992] [https 443 tcp 14352] [192.168.128.12:51876] recv: POST /login HTTP/1.1
2022-07-09 15:59:10 [14992] [https 443 tcp 14352] [192.168.128.12:51876] recv: 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
2022-07-09 15:59:10 [14992] [https 443 tcp 14352] [192.168.128.12:51876] recv: Host: 159.65.186.50
2022-07-09 15:59:10 [14992] [https 443 tcp 14352] [192.168.128.12:51876] recv: Content-Length: 1072
2022-07-09 15:59:10 [14992] [https 443 tcp 14352] [192.168.128.12:51876] recv: Cache-Control: no-cache
2022-07-09 15:59:10 [14992] [https 443 tcp 14352] [192.168.128.12:51876] recv: (POSTDATA)
2022-07-09 15:59:10 [14992] [https 443 tcp 14352] [192.168.128.12:51876] info: POST data file e3b0c44298fc1c149afb4fc8996fb92427ae14e4649b93ca495991b7852b055 already exists
2022-07-09 15:59:10 [14992] [https 443 tcp 14352] [192.168.128.12:51876] info: Request URL: https://159.65.186.50/login
2022-07-09 15:59:10 [14992] [https 443 tcp 14352] [192.168.128.12:51876] info: No matching file extension configured. Sending default fake file.
2022-07-09 15:59:10 [14992] [https 443 tcp 14352] [192.168.128.12:51876] send: HTTP/1.1 200 OK
2022-07-09 15:59:10 [14992] [https 443 tcp 14352] [192.168.128.12:51876] send: Server: Inetsim HTTPs Server
2022-07-09 15:59:10 [14992] [https 443 tcp 14352] [192.168.128.12:51876] send: Connection: close
2022-07-09 15:59:10 [14992] [https 443 tcp 14352] [192.168.128.12:51876] send: Content-Type: text/html
2022-07-09 15:59:10 [14992] [https 443 tcp 14352] [192.168.128.12:51876] send: Content-Length: 258
2022-07-09 15:59:10 [14992] [https 443 tcp 14352] [192.168.128.12:51876] send: Date: Sat, 09 Jul 2022 19:59:10 GMT
2022-07-09 15:59:10 [14992] [https 443 tcp 14352] [192.168.128.12:51876] info: Sending file: /var/lib/inetsim/http/fakefiles/sample.html
2022-07-09 15:59:10 [14992] [https 443 tcp 14352] [192.168.128.12:51876] stat: 1 method=POST url=https://159.65.186.50/login sent=/var/lib/inetsim/http/fakefiles/sample.html postdata=/var/lib/inetsim/http/postdata/9939a68a8f819353530e167c4eb5eb8c8f2ff
9eb879f6d218ea6642208
2022-07-09 15:59:10 [14992] [https 443 tcp 14352] [192.168.128.12:51876] disconnect
C
remux@remux:~/p01$ sudo cat /var/lib/inetsim/http/postdata/9939a68a8f819353530e167c4eb5eb8c8f2ff9eb879f6d218ea6642208
MTZU9V9tW9M9P9d9W9E9t9D9u9c9L9E9h9p9f9t9X9OH9Y979V9p9k9M9Z949J9f9S9B9m9Q9z9p9T9c9Z9d9P919I9W9L9P9S9E9N9J9S9G9Z9d9B9y9v9h9Q9m9G9u9r9C9S9f9W9J9Z9S9z9r9g9S9z9r9Z9g9F9R9g9h9k9q9m9Z9h9y9c9y9
9f9F9a9u9S9p9L9V9E9t9U9S9p9M9A9T9S9k949K9Z9J9U9S79z9d9y9g9S9f9R9x9G9k9R9y9M9M9L9B9Z9h9c9Q9T939Y9f9T9Q9c9u9g9P9h9F9T9M9W9h9F9r9g9g9
9r9m9e9z9b9f9l9g9k9p9u9f9429w9V9Y9e9S9y9A9e9B9o9E979S9J9h9V9Y9d9v9o9p9T9V919D9J9T9W9+9g979k9v9i9c9a9S9U9x9+9T9y9S9w9J9P9a9w9O9S9o9Z9x9C969l9o9c9E9m9Z9L9M9h9c9Z9k9h939a9d9L9H9Z9A9e9q9R9H9t9e9F9x9P9L9B9v9z9+9u949q9z9A9e9N9J9d9/V9F9+9x9P9
9c9o9P9h9L9X9p9r9W9b9t979E9p9--remux@remux:~/p01$
```

Let's bring the input into CyberChef and decode the Base64.



Searching for Encryption in APIMonitor

Interesting! The POST Data is encrypted. I think I know a trick or two that could help us decode this. To do so, we'll need to hop into API Monitor and hook into the process and observe the API Calls the badger is performing. We're looking for a call to Microsoft's Cryptographic API or a call to the HTTP APIs as we know some cryptographic function performs before the POST data is sent...

Time	Process	Operation	Address	Value
290875	wininet.dll	memcpy	{ 0x0000000002877a10, 0x00007ffce1a7c60, 5 }	
290876	wininet.dll	memcpy	{ 0x0000000002877a15, 0x00007ffce1a7c5c, 3 }	
290877	wininet.dll	memcpy	{ 0x0000000002877a18, 0x00000000026ab5c, 13 }	
290878	wininet.dll	memcpy	{ 0x0000000002877a25, 0x000000000285ebc, 8 }	
290879	KERNELBASE.dll	RtlUTF8ToUnicodeN	(NULL, 0, 0x00000000059be7b8, "https://159.65.186.50/admin", 30)	
290880	KERNELBASE.dll	RtlUTF8ToUnicodeN	("D", 60, 0x00000000059be7b8, "https://159.65.186.50/admin", 30)	
290881	KERNELBASE.dll	memcpy	{ 0x00000000059be49a, 0x00007ffce2f2d8708, 10 }	
290882	KERNELBASE.dll	memcpy	{ 0x000000000287b10, 0x00000000059be49a, 56 }	
290883	KERNELBASE.dll	RtlUnicodeToUTF8N	(NULL, 0, 0x00000000059be830, "https://159.65.186.50/admin", 56)	
290884	KERNELBASE.dll	RtlUnicodeToUTF8N	("", 28, 0x00000000059be830, "https://159.65.186.50/admin", 56)	

Pre-Call Value	Post-Call Value
ination: 0x0000000002878140 "D"	0x0000000002878140 "https://159.65.186.50/admin"
ByteCount: 60	60
ialByteCount: 0x00000000059be7b8 = 0	0x00000000059be7b8 = 60
0x0000000002877a10 "https://159.65.186.50/admin"	0x0000000002877a10 "https://159.65.186.50/admin"
nt: 30	30
	STATUS_SUCCESS

Find dialog box showing search criteria: `RtlUTF8ToUnicodeN`. The search is performed in the `Down` direction.

By searching for a common Windows API (RtlUTF8ToUnicodeN), we can quickly find where some data conversion is taking place to give us a good starting point of reference.

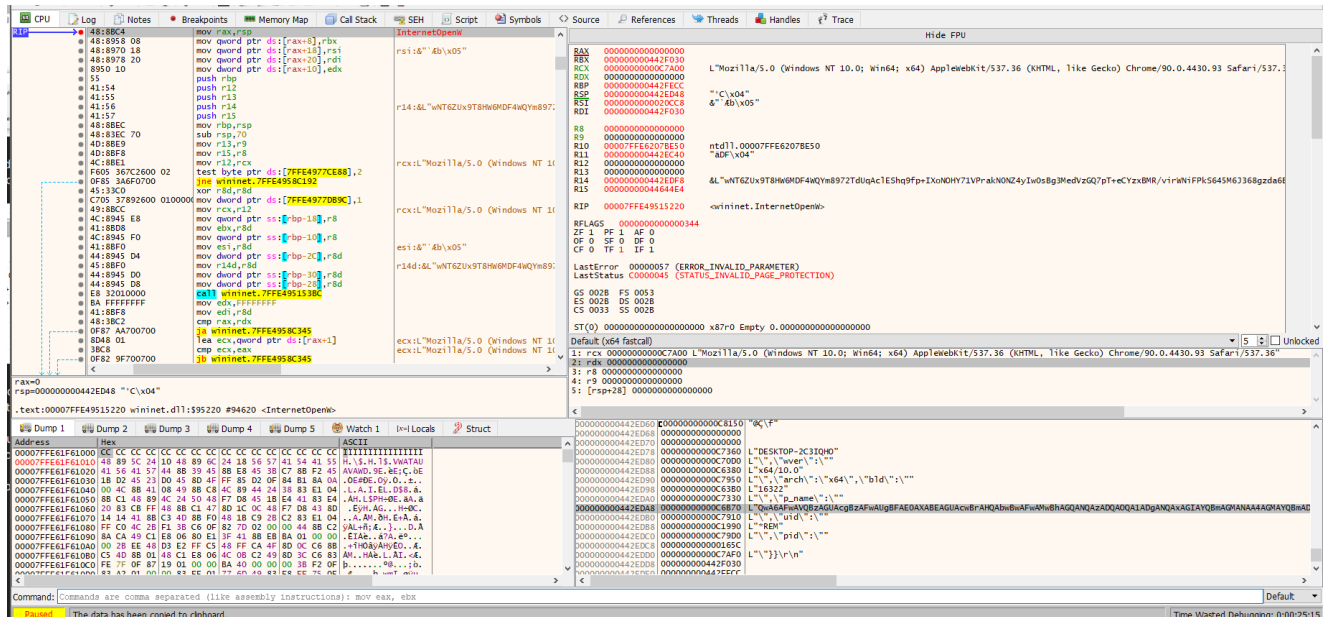
	NTSTATUS	Return	STATUS_SUCCESS	
Call Stack: RtlUTF8ToUnicodeN (Ntdll.dll)				
#	Module	Address	Offset	Location
1	KERNELBASE.dll	0x00007ffce2e7...	0x4d18e	MultiByteToWideChar + 0x25e
2	wininet.dll	0x00007ffccdf...	0x213d9	InternetCreateUrlW + 0x3d89
3	wininet.dll	0x00007ffccdf...	0x22826	HttpOpenRequestW + 0x12a6
4	wininet.dll	0x00007ffccdf...	0x21953	HttpOpenRequestW + 0x3d3

Looking at the CallStack, we see some lovely Windows API calls that look very close to what we need. Since some sort of technique is being used to dynamically resolved the APIs needed is being used, let's back off of APIMonitor and move over to a Debugger.

Pivoting to x64Dbg

I have setup x64Dbg to use counter-antidebugging techniques using ScyllaHide, so if there are any techniques implemented, we won't have to worry about them.

After letting the program run for a while, I set a breakpoint on a couple of the common HTTP APIs. We got a hit on InternetOpenW; in my surprise, in the stack window, here we are. We have the unencrypted data starting at us!



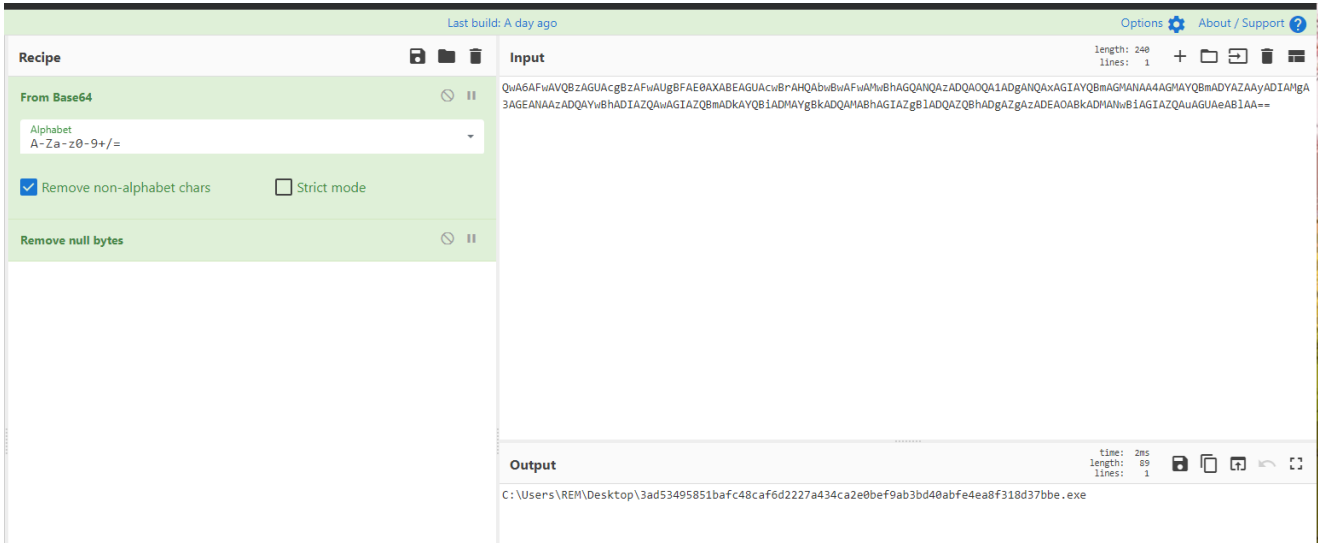
It appears to be some JSON that looks like so:

```

"desktop-2c3Iqh0",
  "wver": "x64/10.0",
  "arch": "x64",
  "bld": "16322",
  "p_name": "<base64 blob>",
  "uid": "REM",
  "pid": ""
}

```

The Base64 glob is still relatively interesting to me, p_name, could this mean program_name? Let's decode it!



It appears so! I set a BreakPoint earlier in the stack and let the execution flow to see if I could extract any more information from the Badger, doing so did yeild some extra results!



We have an auth token now and a more complete JSON blob.

```
{
  "cds": {
    "auth": "2K4TBS7L9GK2C205"
  },
  "mtdt": {
    "h_name": "DESKTOP-2C3IQH0",
    "wver": "x64/10.0",
    "arch": "x64",
    "bld": "16322",
    "p_name": "<base64 blob>",
    "uid": "REM",
    "pid": ""
  }
}
```

Unfortunately, our analysis stops here as we don't have a live C2 server to observe interactions with. Though, we could explore *how* the badger interacts with the C2 server if we carefully observe how the badger parses the response from the C2 server. There is definitely some hardcoded commands that we would be able to use to manipulate the badger itself with iNetSim.

I would have liked to have caught the Windows API that actually encodes/encrypts this data, so I could write a small decoder for the information if you have the badger; but it appears that wasn't meant for tonight :(

Basic Static Analysis

So, this section is going to be much shorter than the last, as I've already found the interesting C2 related data; Now, we're going to play an interesting game of "How good is Brute Ratel's Obfuscation Techniques"! The answer isn't very good.

To start, we're going to chuck the EXE into Cyberchef and look at some of the clear text ASCII values.

HTTP Request Information

So, right off the bat, it's not looking so good. We can see a **lot** of interesting strings; we can see a lot of the HTTP POST information broken up into various strings. For example:

- /logi
- AppleWeb
- Kit/537
- 65.186.5
- 159
- 443

Some of these strings are incredibly meaningful! For example, putting together the bits 159.65.186.50 gives away our command and control server, and 443 gives away the port! How interesting...

Windows APIs

Looking a little bit lower, we can see some of the Windows APIs the program uses as well. They appear to be jumbled up, but still readable to the human eye.

```
.strtokPH, rncmp>.PH, en6.stPH, 3.str1PH, ..strcpyPH, rcat_sPH, and*.stPH, ntF&.srPH, #.spriPH, ..signalPH, reallocPH, ran
d..PH, mcmp..PH, _sy.mePH, mbstowcsPH, llocü.PH, aceø.maPH, ß.isspPH, Ê.fwritePH, ¼.freePH, callocPH, atol..PH, ctime..PH,
ort..asPH, mp..abPH,
_wcsicPH, wprintfPH, ÿ.vsnPH, snprintfPH, ocké._vPH, Ê._unlPH, Â._ultoAPH, _time64PH, _lock¶.PH, me64..PH, _localtiPH, t
term..PH, .._iniPH, ¼_errnoPH, time64PH, it§_cPH, _amsg_exPH, funcyPH, T__iob_PH, trlenWPH, ywO.lsPH, I.lstrcpPH, lQueryPH
, Ö.VirtualPH, rotectPH, VirtualPPH, lterÖ.PH, eptionFiPH, ndledExcPH, e³.UnhaPH, sGetValuPH, ead¼.TlPH, inateThrPH, ..Term
PH, eProcessPH, TerminatPH, Sleep..PH, ilter..PH, ceptionFPH, andledExPH, r.SetUnhPH, UnwindPH, lVirtualPH, ryÖ.RtPH, ctio
nEntPH, ookupFunPH, tÏ.RtlLPH, reContexPH, RtlCaptuPH, TableÇ.PH, FunctionPH, Æ.RtlAddPH, CounterPH, formancePH, QueryPer
PH, lFreek.PH, Ê.LocaPH, calAllocPH, onå.LoPH, calSectiPH, aveCritiPH, ionØ.LePH, icalSectPH, lizeCritPH, |.InitiaPH, ReAl
locPH, i.HeapPH, HeapFreePH, Alloce.PH, _..HeapPH, ickCountPH, e..GetTPH, sFileTimPH, temTimeAPH, ..GetSysPH, ssHeapPH, Get
ProcePH, ressÏ.PH, tProcAddPH, eWÆ.GePH, uleHandlPH, ..GetModPH, tErrorPH, v.GetLasPH, readIdPH, urrentThPH, d-.GetCPH, Pr
ocessIPH, tCurrentPH, ess).GePH, rentProcPH,
(.GetCurPH, LibraryPH, e»).FreePH, eeConsolPH, on..FrPH, calSectiPH, terCritiPH, ion?.EnPH, icalSectPH, leteCritPH, ..DeP
PH, .I.PH, .I.PH, øH.PH, êH.PH, àH.PH, ÒH.PH, ÌH.PH, ÂH.PH, .H.PH,
~H.PH ¶H.PH .H.PH .H.PH .H.PH ~H.PH tH.PH fH.PH \H.PH RH.PH HH.PH @H.PH 6H.PH .H.PH $H.PH .H.PH .H.PH .H.PH ôG.
```

- VirtualProtect
- GetLastError
- GetModuleHandleW
- GetProcAddress

The more you keep looking, the more you see the pattern.

HTTP POST Data

Interestingly enough, you can actually find a lot of the HTTP POST Data that we had to work oh so hard to reverse engineer to find...

```
PH, ----PH, ----PH, ----PH, ----PH, ----PH, ----PH, ----PH, ----PH, ----PH, ----PH, ----PH, ----PH, ----PH, ----PH, ----PH, ----
PH, ----PH, +--PH, %S
PH,osedPH,] C1PH,
[-PH, ST%SPH, SPOPH, %s%ls%PH, %ls%ls%lsP, P, BFPH, 3#M?:XyMPH, bYXJm/PH, d%lsPH, %ls%PH, s%lsPH, ls%lPH,
%ls%PH, s%lsPH, ls%lPH, %ls%PH, s%lsPH, ls%lPH, %ls%PH, ls, PH, %s%PH, POSTPH, s%lsPH, ls%lPH, %ls%PH, ls%SPH, ls%PH,
%ls%PH, ls%SPH, %SP, P, STPH, %lsPOPH, %lsPH, s%luPH, %S%lPH, S%lsPH, %ls%PH, ls%SPH, V2PH, \CIMPH, ROOTPH, }
PH, ":PH, {"PH, ze":PH, dfsiPH, ", "PH, e": "PH, fnamPH, ", "dPH, ": "PH, hkinPH, : {"cPH, "dt"PH, }, PH, }}
PH, :""PH, pid"PH, ", "PH, d": "PH, ", "uiPH, : ""PH, ame"PH, "p_nPH, ", PH, er": PH, ", "wvPH, : ""PH, ame"PH, "h_nPH, t":
{PH, "mtdPH, }, PH, h": "PH, "autPH, s": {PH, {"cdPH, ": "PH, "bldPH, 64", PH, ": "xPH, archPH, ø", "PH, %SPH, %S
P, PH, Y@PH, SÈBPH, ls%PH, d]
%PH, E: %PH, ed [PH, failPH, oad PH, ownlPH, -] DPH, ls[PH, te
%PH, mplePH, d coPH, nloaPH, DowPH, [+ ]PH, %%)
PH, .2f PH, (%ØPH, s %SPH, tatuPH, ad SPH, wnloPH, ] DoPH, s[+PH, d
%lPH, E: %PH, [- ] PH, -+
PH, ----PH, ----PH, ----PH, ----PH, ----PH, ----PH, ----PH, ----PH, ----PH, ----PH, ----PH, ----PH, ----PH, ----PH, ----PH, ----
```

- arch
- bld
- fname
- h_name

Continuing our search, we may be able to learn more about the badgers capabilities. Looking at the screenshot above, towards the bottom, we can make out "Download Failed". Perhaps this badger has the ability to upload files to the server? Let's keep digging.

```
PH,----PH,+++PH,
%lsPH,: %SPH,adedPH,wnloPH,t doPH,nshoPH,creePH,+] SPH,eb[PH,05e7PH,b351PH,d-5dPH,-9cdPH,452dPH,a4a-PH,b5-
fPH,5be4PH,g1dPH,e/pnPH,imagPH,2d.pngPH,02d%2d%0PH,%d_%02d%PH,%02d%02dP,P,PH,%ls
PH,ng: PH,eryiPH,] QuPH,S[*PH,://%PH,LDAPPH,E,
PH,otDSPH,//roPH,DAP:PH,g: LPH,ndinPH,r biPH,ErroPH,[-]
PH,%lsPH,P://PH,LDAPH,textPH,gConPH,aminPH,ultNPH,defaPH,DSEPH,rootPH,P://PH,LDAPH,gue
PH,taloPH,l CaPH,lobaPH,g: GPH,ndinPH,r biPH,ErroPH,[-] PH,ap
PH,g ldPH,ndinPH,r biPH,ErroPH,[-] PH,GC:PH,ue
PH,alogPH,CatPH,obalPH,: GLPH,yingPH,QuerPH,[*] PH,:XyMBFPH,XJm/3#M?PH,lsbYPH,+
%PH,----PH,----PH,----PH,----PH,----PH,----PH,----PH,----PH,----PH,----PH,----PH,----PH,----PH,----PH,----PH,----
PH,----PH,----PH,G|b|+|b|PH,G|b|D b|PH,G|b|G|b|PH,G|b|D b|PH,G|b|G|b|PH,G|b|D b|PH,D b|G|b|PH,) b|G|b|PH,-
f|b|PH,0f|b|D b|PH,D b|D b|PH,^ b|PH,%d,
PH,ye PH,wn tPH,nknoPH,! UPH,.
[PH,ptorPH,scriPH,y dePH,uritPH, SecPH, -PH,%lu
PH,ow: PH,lu lPH,h: %PH,higPH,es.
PH,xpirPH,er
EPH,NevPH,tSetPH,dLasPH,epwPH,tTimPH,ckouPH,floPH,ogofPH,astLPH,onlPH,tLogPH,lasPH,TimePH,wordPH,PassPH,badPH,
resPH,tExpPH,counPH,
acPH,set.PH,lue PH,o vaPH,IDNPH,ctGUPH,objePH,SIDPH,jectPH,
obPH,%luPH,%lu
PH, - PH,ed
PH,nablPH,nt ePH,ccouPH, - aPH,d
PH,ablePH, disPH,ountPH, accPH, -PH,ns:
PH,ptioPH,nt oPH,ccouPH,+] APH,d
```

Badgers like LDAP!

It looks like the badger uploads PNG/image files to the C2 server. It also makes some queries to LDAP as well and will communicate with the Global Catalog. If it can't, it'll spit out some binding errors.

```
f|b|PH,0f|b|D b|PH,D b|D b|PH,^ b|PH,%d.
PH,ye PH,wn tPH,nknoPH,! UPH,.
[PH,ptorPH,scriPH,y dePH,uritPH, SecPH, -PH,%lu
PH,ow: PH,lu lPH,h: %PH,higPH,es.
PH,xpirPH,er
EPH,NevPH,tSetPH,dLasPH,epwPH,tTimPH,ckouPH,floPH,ogofPH,astLPH,onlPH,tLogPH,lasPH,TimePH,wordPH,PassPH,badPH,i
resPH,tExpPH,counPH,
acPH,set.PH,lue PH,o vaPH,IDNPH,ctGUPH,objePH,SIDPH,jectPH,
obPH,%luPH,%lu
PH, - PH,ed
PH,nablPH,nt ePH,ccouPH, - aPH,d
PH,ablePH, disPH,ountPH, accPH, -PH,ns:
PH,ptioPH,nt oPH,ccouPH,+] APH,d
[PH,:%02PH,%02dPH,02d:PH,2d %PH,d-%0PH,-%02PH,%02dPH,at: PH,res PH,expI PH,ord PH,asswPH, - pPH,s
PH,pirePH,r exPH,nevePH,ord PH,asswPH, - pPH,:
PH,ingsPH,settPH,ire PH, expPH,wordPH,PassPH,+] PH,athPH,ADsPPH,ls
PH,s
%PH,- %lPH,r PH,embePH,: mPH, %lsPH,+]PH,ls:
PH,+] %PH,_SE[PH,FALPH,TRUEP,P,P,P,PH,0lx
PH, 0x%PH,] E:PH,
[-PH,iredPH,requPH,not PH,ing PH,atchPH,SI pPH,] AMPH,
[+PH,AMSIPH,tch PH,o paPH,le tPH,UnabPH,[-] PH,SI
PH,d AMPH,tchePH,] PaPH,
[+PH,ritePH,entWPH,twEvPH,ch EPH, patPH,e toPH,nablPH,-] UPH,e
[PH,WritPH,ventPH,EtwePH,hed PH,PatcPH,+] PH,ed
PH,atchPH,TW pPH,nd EPH,SI aPH,] AMPH,
[+PH,oundPH,ot fPH,v4 nPH,/v3/PH,R v2PH,] CLPH,
[-PH,0727PH,.0,5PH,R v2PH,n CLPH,lu iPH,t v%PH,otnePH,ng dPH,unniPH,+] RPH,d
```

Searching lower down the list, we can see some of the information it collects, like Password Expiration, if the password never expires, and if there is a bad password supplied.

The Badger is Self Aware?

Continuing our string-hunt, here's one of the most interesting sets of strings... Badger itself is embedded as a string in the binary :facepalm:

```
WPH,%1s]PH,ed [PH,nectPH, ConPH,[+]PH,x%x
PH,E: 0PH,[-] PH,%1sPH,%1s\PH,%SPH,
FALSE
PH,sTRUEPH,G%1PH,NNINPH,YRUPH,READPH,UEDPH,QUEPH,BLEDPH,DISAPH,OWNPH,UNKNP,PH,rAtoiPH,mpBadgePH,dgerWcscPH,trcm
pBaPH_BadgerSPH_erMemsetPH_cpyBadgPH_adgerMemPH_WcslenBPH_nBadgerPH_getStrlePH_tchWBadPH_getDispaPH_atchBadPH,d
gerDispPH_lsBaPH,+
%PH,----PH,----PH,----PH,----PH,----PH,----PH,----PH,----PH,----PH,----PH,----PH,----PH,----PH,----PH,----
PH,----PH,+---PH,ls
PH,ls %PH,%-20PH,--
PH,----PH,----PH,----PH,----PH,----PH,----PH,----PH,
---PH,%1sPH,201sPH,
%-PH,%1uPH,] E:PH,
[-PH,%1sPH,251sPH,s %-PH,-201PH,th%PH,
PaPH,----PH,----PH,----PH,----PH,----PH,----PH,----PH,----PH,----PH,----PH,----PH,----PH,----PH,----PH,ls
-PH,ls %PH,%-25PH,01s PH,%-2PH,aresPH,
ShPH,%1sPH,tingPH,meraPH, EnuPH,[+]PH,NamePH,u1e PH,ModPH,NamePH,any PH,CompPH,ionPH,riptPH,DescPH,s
PH,: %1PH,51s PH,%-1PH,
```

I've already loaded up the binary into Ghidra and there's a whole lot of nothing. It seems to be a bit beyond my skill level to reverse engineer in a classic sense, so I'll have to do some more research on my own time to figure out if I can post a followup showing off the actual binary internals.

Misc Findings

Here are some interesting things I found that I wanted to include in the post, but couldn't easily write into the flow of the post. I still think this is worth mentioning.

PUNYCode! The thing I forgot existed?

Here is an interesting String Compare after executing a HTTP Request; it appears that this badger is checking to see if some of the response headers contain `xn--`. This may be a sign that a threat actor is spoofing a common domain like `Google.com` to `http://xn--ggle-0nda.xn--om-ubc/`, which displays just like the normal domain does! Browser

settings can be configured to always display xn--, though some by default will render the link as normal. Thanks to @ShitSecure for pointing this out <3

65922	7:24:13.684 PM	6	KERNELBASE.dll	RtlFreeHeap (0x000000002710000, 0, 0x00000000278be30)
65923	7:24:13.684 PM	6	KERNELBASE.dll	RtlFreeHeap (0x000000002710000, 0, 0x0000000027fcd0)
65924	7:24:13.684 PM	6	KERNELBASE.dll	RtlFreeHeap (0x000000002710000, 0, 0x0000000027a2070)
65925	7:24:13.684 PM	6	KERNELBASE.dll	RtlAllocateHeap (0x000000002710000, HEAP_CREATE_ENABLE_EXECUTE 1048576, 304)
65926	7:24:13.684 PM	6	KERNELBASE.dll	RtlFreeHeap (0x000000002710000, 0, 0x0000000044a1830)
65927	7:24:13.684 PM	6	crypt32.dll	wcschr ("inetsim.org", "*")
65928	7:24:13.684 PM	6	crypt32.dll	wcsstr ("inetsim.org", "xn--")
65929	7:24:13.684 PM	6	crypt32.dll	wcsstr ("159.65.186.50", "xn--")
65930	7:24:13.684 PM	6	KERNELBASE.dll	RtlFreeHeap (0x000000002710000, 0, 0x0000000044a1970)
65931	7:24:13.684 PM	6	KERNELBASE.dll	RtlFreeHeap (0x000000002710000, 0, 0x0000000027a8270)
65932	7:24:13.684 PM	6	WINTRUST.dll	malloc (240)

Traffic Generation to windowsupdate.com

Another interesting aspect of this badger is that it periodically reaches out to `ctld1.windowsupdate.com`. I originally thought this was Windows being Windows, but it turns out that this is hardcoded within the binary. This is likely a cloaking mechanism to throw off AV/EDR/Sandboxes.

The screenshot shows a debugger window with the following components:

- Disassembly:**

```

ca11 qword ptr [i:<&RTLUTF8ToUnicodeN
mov ebx, eax
test eax, eax
0F58 EA140500 3B kernelbase.7FEE5E4E682
mov ecx, dword ptr ss:[ebp-21]
884D DF test ecx, ecx
0F84 D0140500 7E kernelbase.7FEE5E4E673
81F8 07010000 3B kernelbase.7FEE5EAFD60
81F9 FFFFFFF7 Cmp ecx, 7FFFFFFF
77 27 JNB kernelbase.7FEE5EAFD1DE
44:8BF9 mov r15d, ecx
E9 10FFFFFF Jmp kernelbase.7FEE5EAFD0D1
89 FFFFFFF7 mov ecx, 7FFFFFFF
48:8BC3 mov rax, rbx
0F1F80 00000000 nop dword ptr ds:[rax], eax
44:3838 cmp byte ptr ds:[rax], r15b
74 18 JB kernelbase.7FEE5EAFD1ED
48:FFC0 tnc rax
48:83E9 01 sub rcx, 1
75 F2 JNB kernelbase.7FEE5EAFD1D0
89 57000000 mov ecx, 57
E8 1802FEFF Jmp <JMP.&RTLRestoreLastWin32Error>
E9 E4FEFFFF Jmp kernelbase.7FEE5EAFD0D1
48:85C9 test rcx, rcx
74 EC JB kernelbase.7FEE5EAFD1DE
BE FFFFFFF7 mov esi, 7FFFFFFF
48:2BF1 sub rsi, rcx
48:FFC6 tnc rsi
48:81FE FFFFFFF7 Cmp rsi, 7FFFFFFF
0F86 4CFEFFFF JB kernelbase.7FEE5EAFD156
E8 D2 Jmp kernelbase.7FEE5EAFD1DE
4C:882D 65681E00 mov r13, qword ptr ds:[7FEE5E3A78]
E9 9EFDFFFF Jmp kernelbase.7FEE5EAFD1B6
49:3BD7 Cmp rdx, r15

```
- Registers:**

```

RAX 0000000000000001 "ctld1.windowsupdate.com"
RDX 0000000000000000
RCX 0000000000000000
RDX 0000000000000000
RBP 00000000621D649
RSP 00000000621D650 &"ctld1.windowsupdate.com"
RST 0000000000000017
RDI 0000000000000001
R8 00000000621D628 "ctld1.windowsupdate.com"
R9 00000000001C880
R10 00000000001C897
R11 0000000000000000
R12 0000000000000000
R13 00000000621D678 &"ctld1.windowsupdate.com"
R14 0000000000000008
R15 0000000000000000
RIP 00007FEE5EAFD188 kernelbase.00007FEE5EAFD188

```
- String:**

```

57: 'w'

```
- Registers (bottom):**

```

RFLAGS 0000000000000344
ZF 1 PF 1 AF 0
OF 0 SE 0 DF 0
CF 0 TF 1 IF 1
LastError 00000000 (ERROR_SUCCESS)
LastStatus C000007C (STATUS_NO_TOKEN)
GS 002B FS 0053
ES 0028 DS 0028
CS 0033 SS 002B
ST(0) 00000000000000000000000000000000 x87:0 Empty 0.00000000000000000000
Default (x64 fastcall)
1: rcx 0000000000000000
2: rdx 0000000000000000
3: r8 00000000621D628
4: r9 00000000001C880 "ctld1.windowsupdate.com"
5: [rsp+20] 0000000000000017

```

I hope you all enjoyed :) ~Ronnie

Comments