Analyzing a Brute Ratel Badger

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

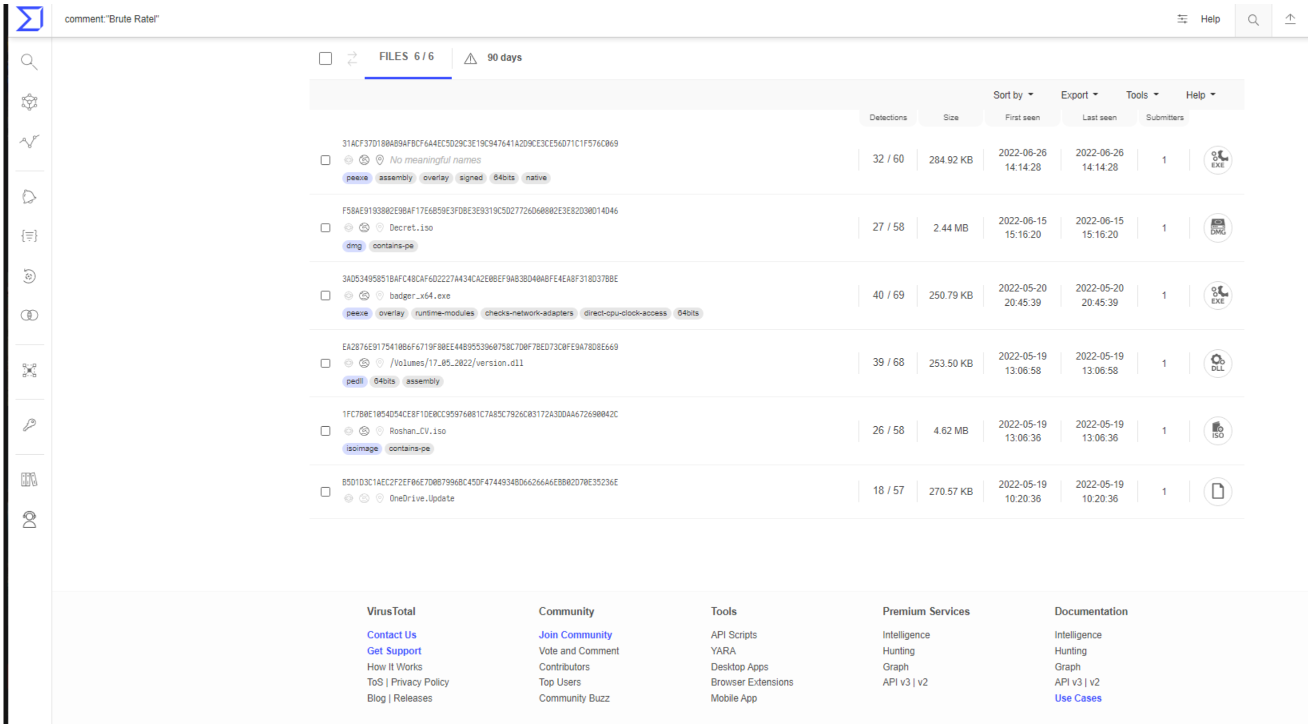
09 Jul 2022

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 exaclty feasable. 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

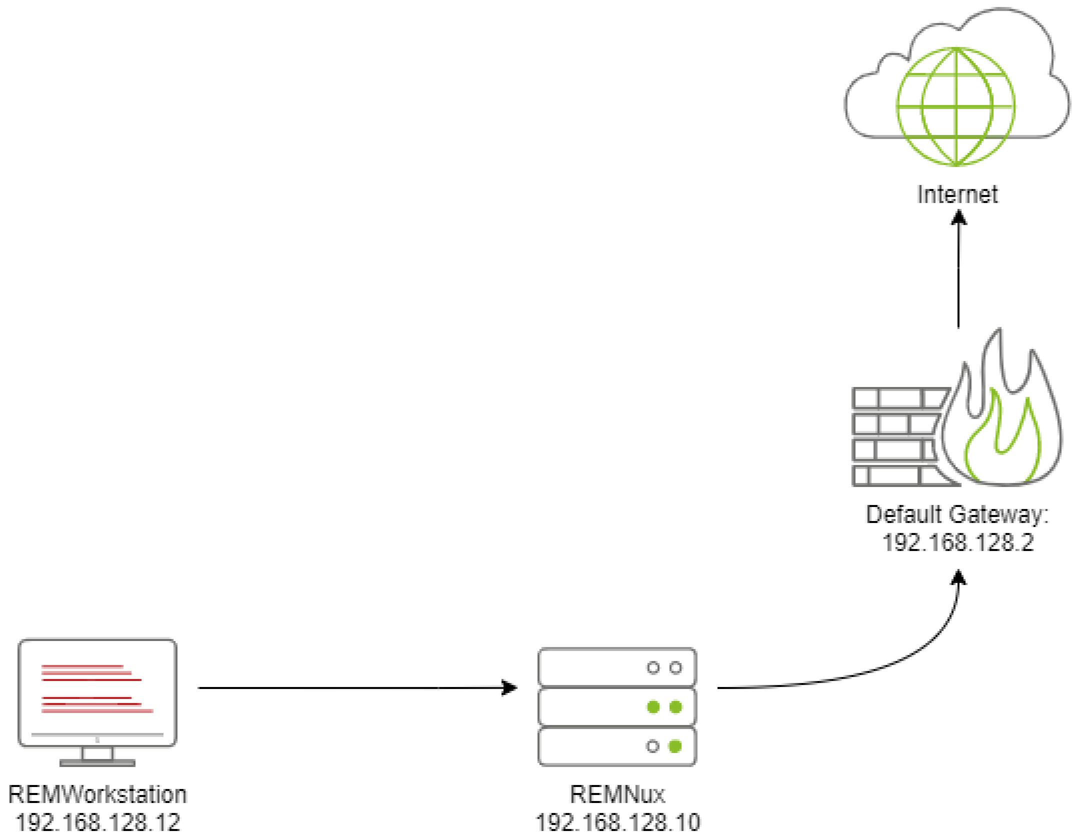
How can we do this exactly? Fortunately, I have a VirusTotal Enterprise license! This means we can pull down (download) a publicly tagged “Brute Ratel” sample from the community. To do so, we’re going to use a search for something like `Comment:"Brute Ratel"` and see if we get any hits…



Suprise Suprise, 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

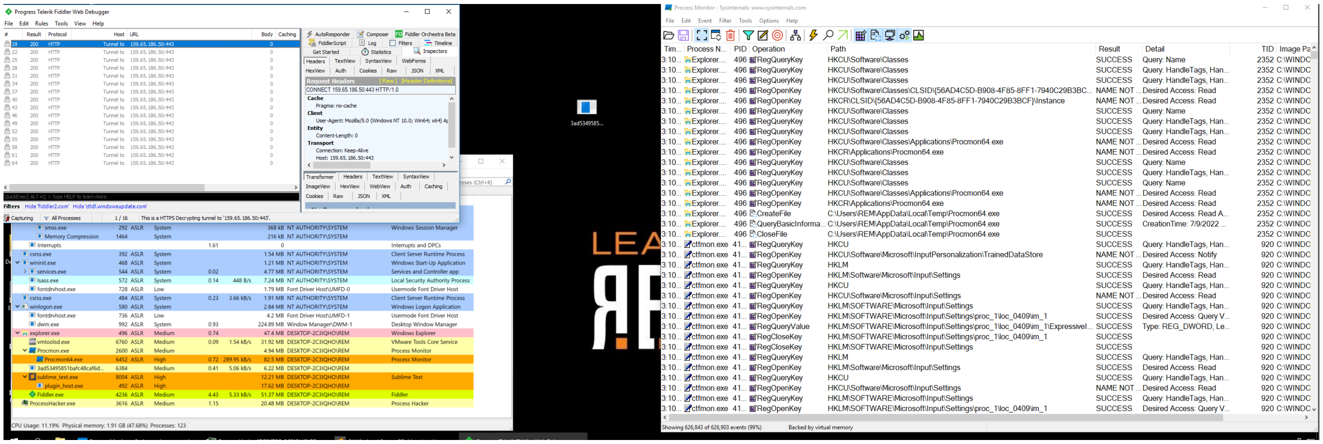
In addition:

- 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

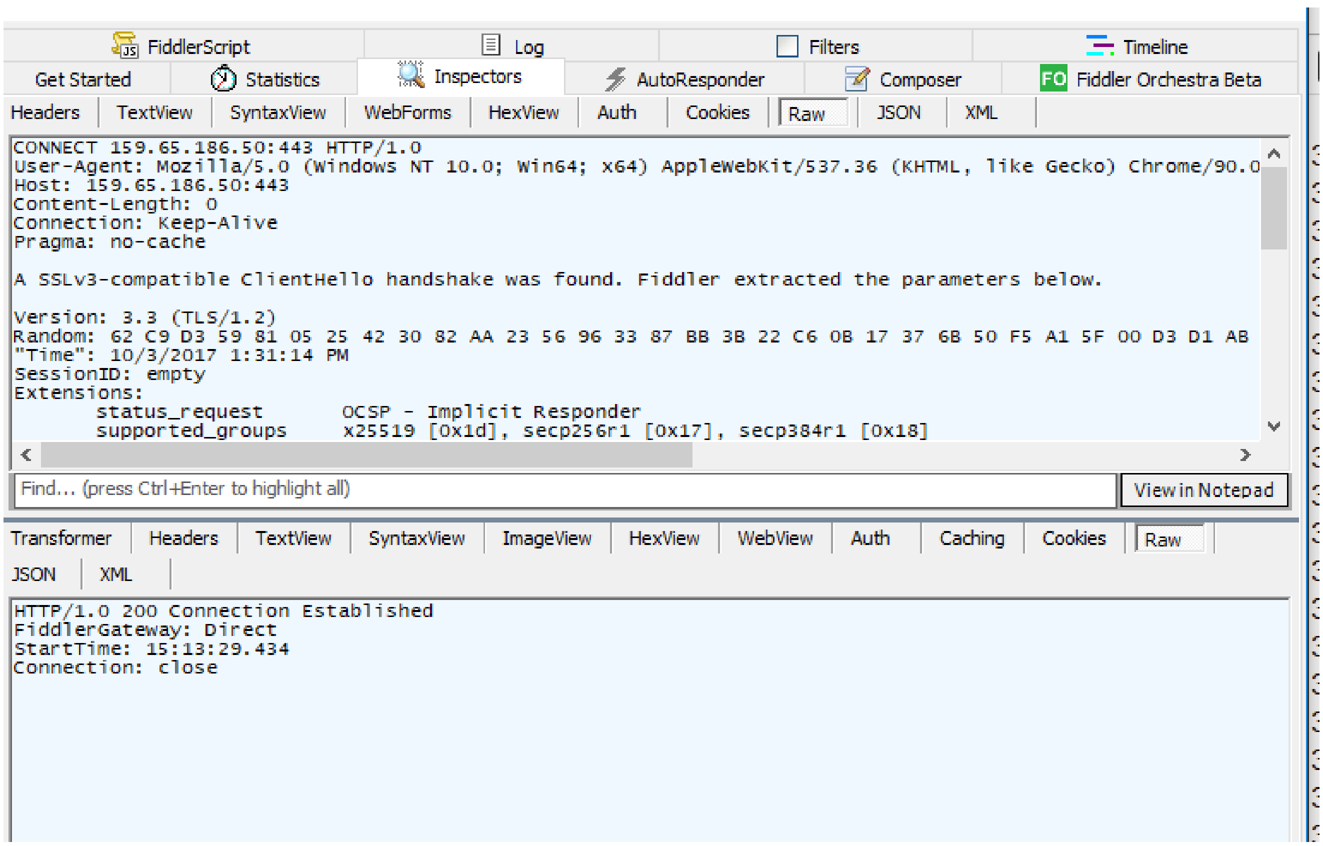
And thats our lab!

Dynamic Analysis - Malware Detonation

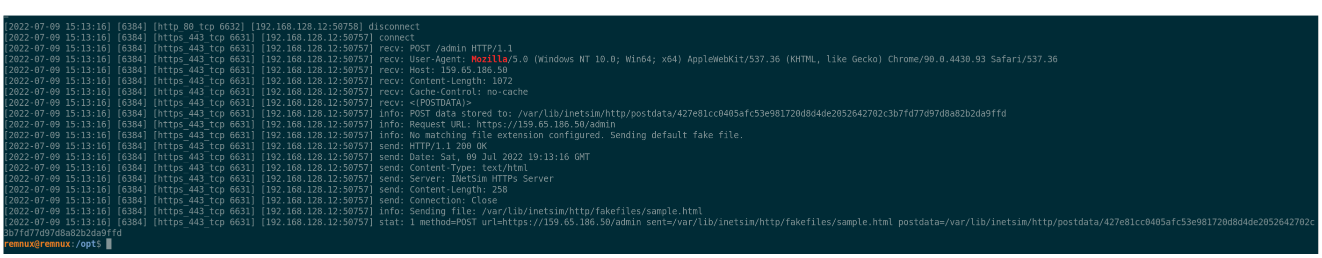
Now that we have our sample acquired, and you're familiar with my lab setup, let's double click some EXEs!



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 suprisingly bare. Let's pivot over to iNetSim and see whats going on over there.



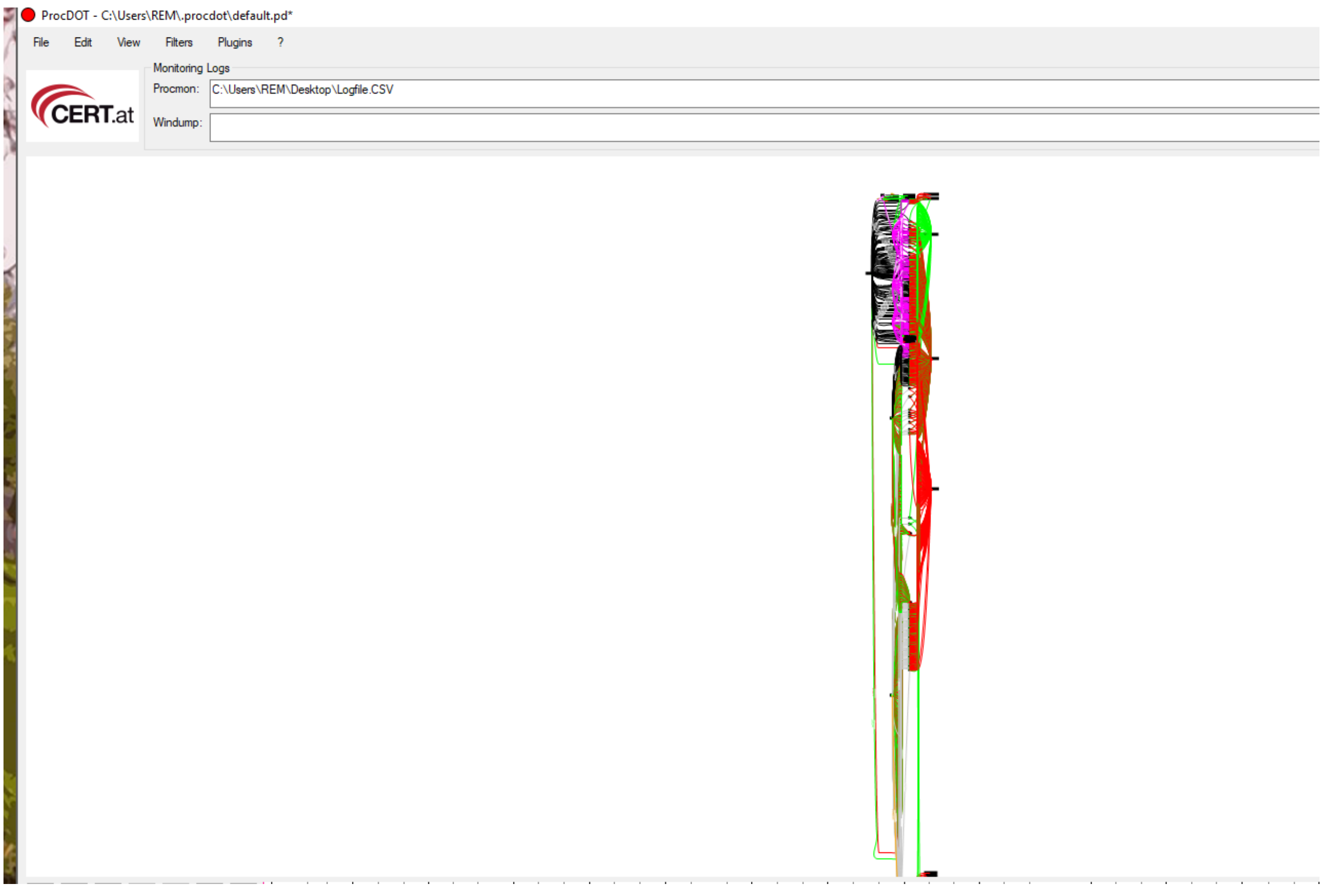
On that side, we can see a little bit more. The file that the “Badger” requested is `/admin`, and there is also some POST data that we missed!

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.

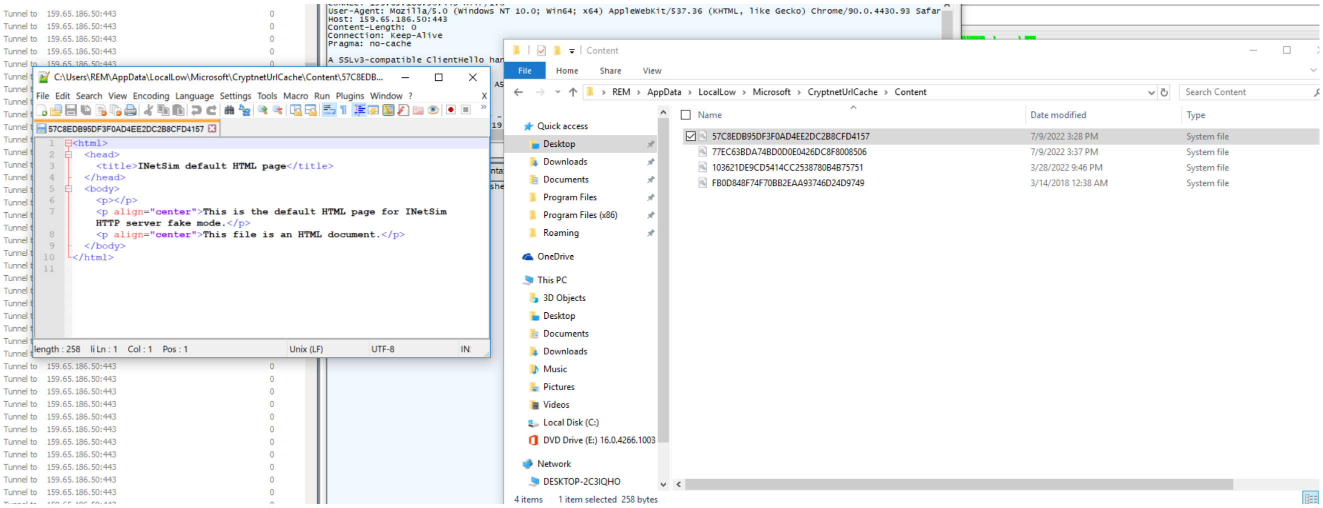


Starting out, this is an absolutely massive graph. Let’s start from the top and work our way down.

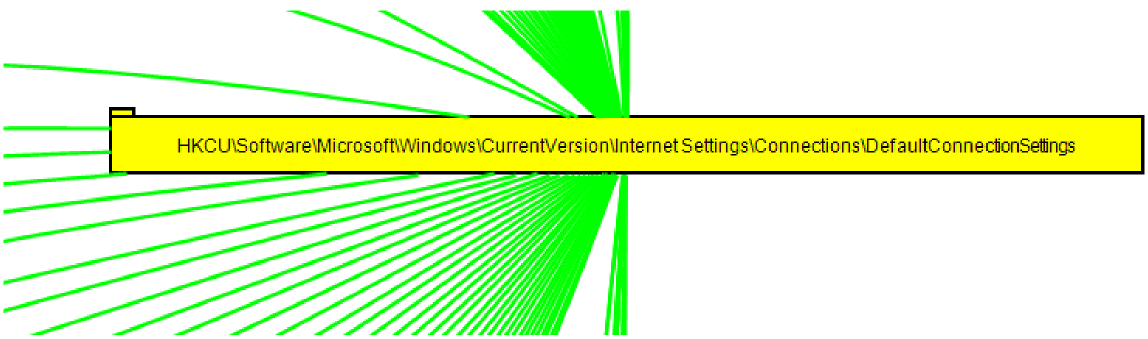
At the top:

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 foudn 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.

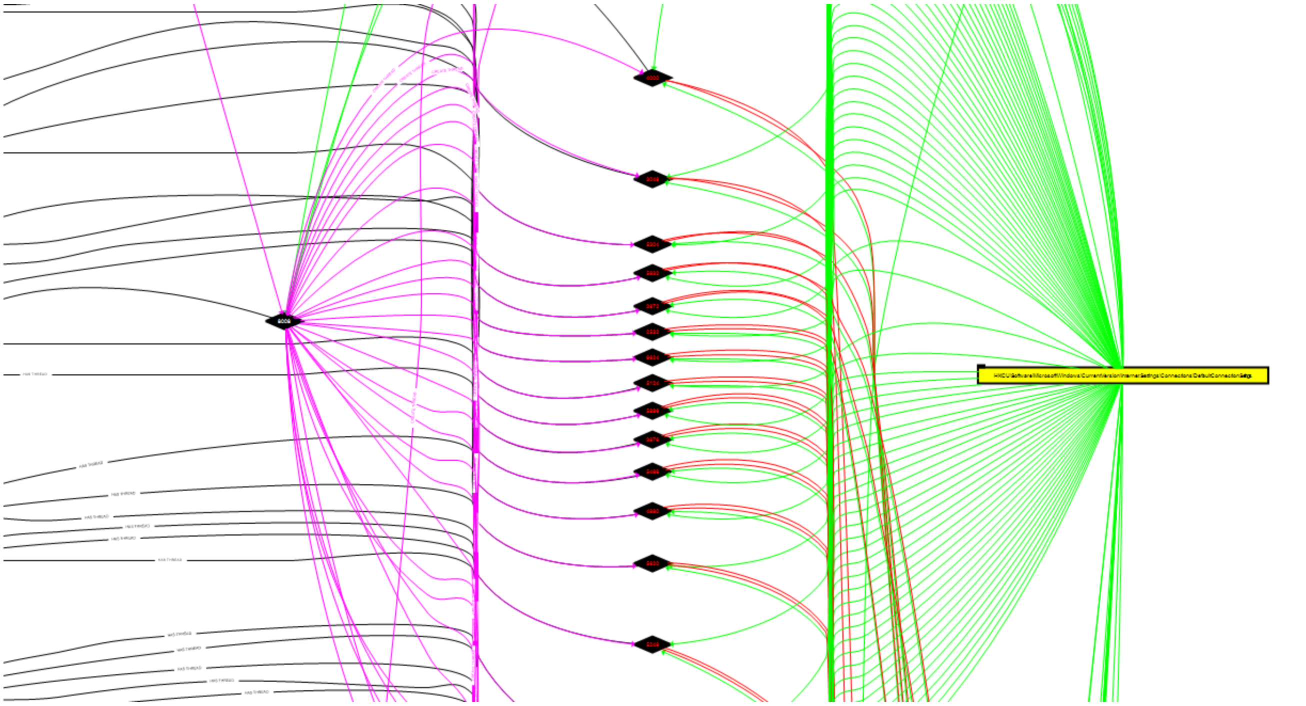


Moving on down the graph, we can see another read attempt on another registry key relating to proxies:

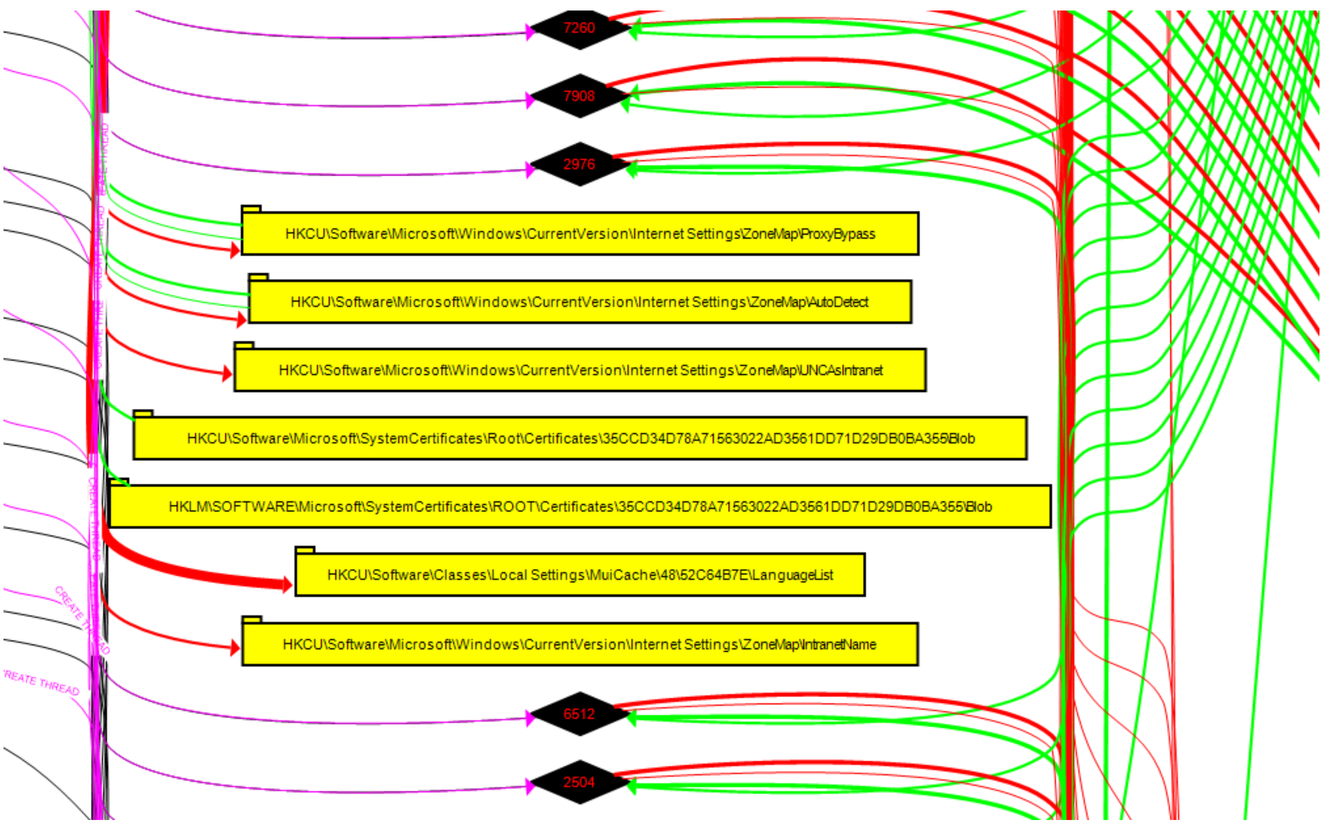


One interesting thing I'd like to point out is the Badger is leveraging a bunch of ThreadCreates and ThreadOpens to potentially confuse AV or EDR.

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



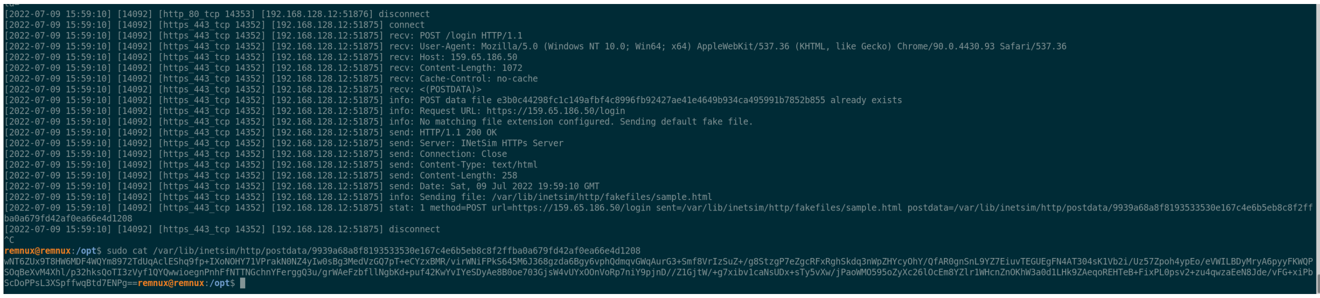
Scrolling down a bit more, this pattern continues. More Threads being created to read registry keys relating to proxies:



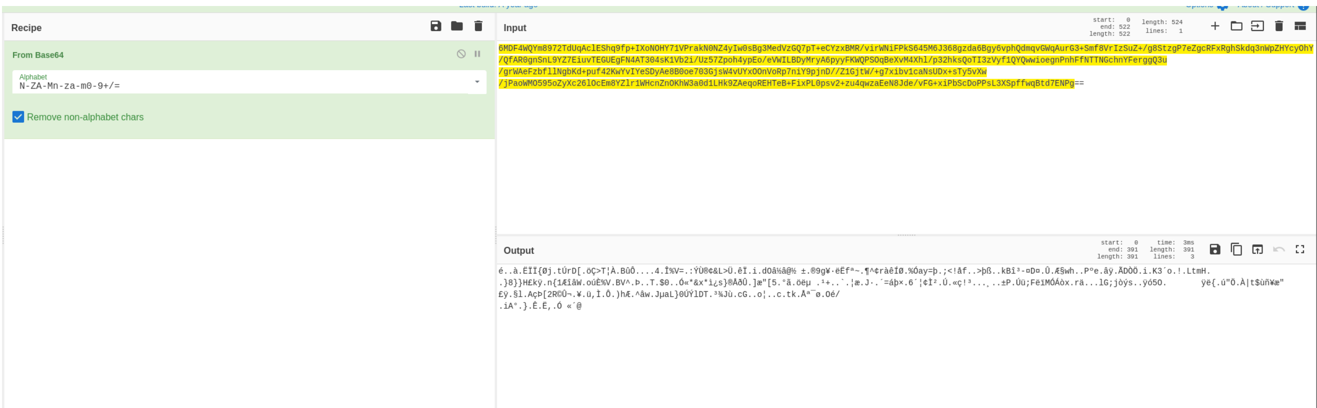
Back to iNetSim

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… :)

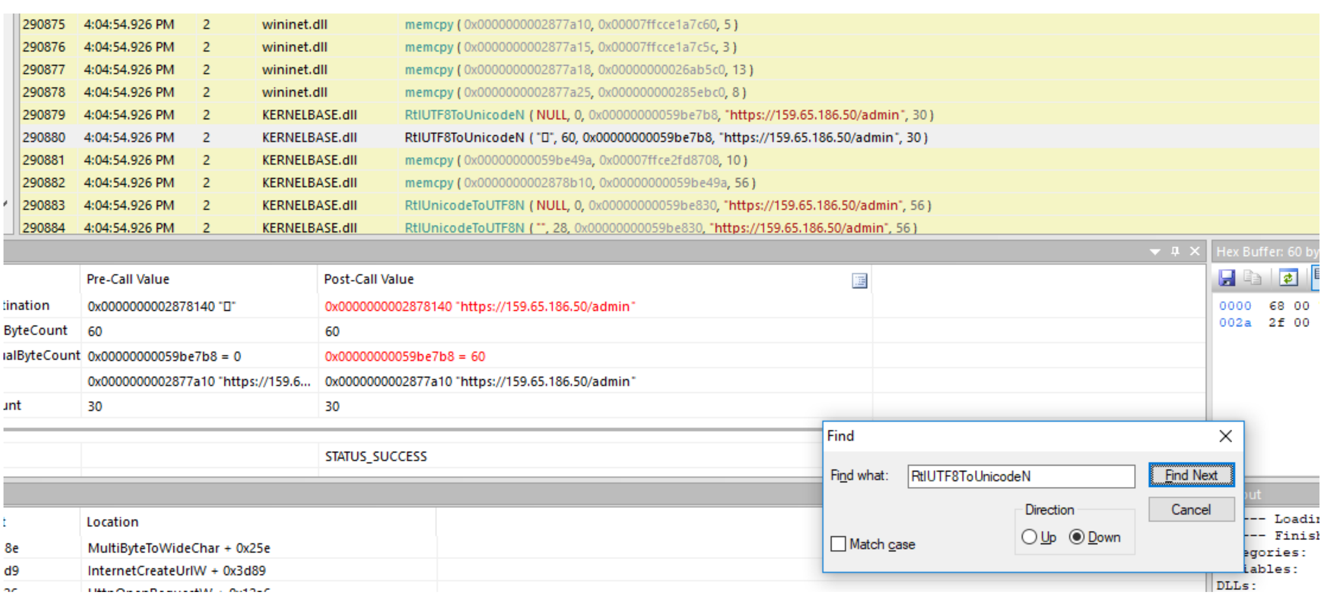


Let's bring the input into CyberChef and decode the Base64.



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…



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)

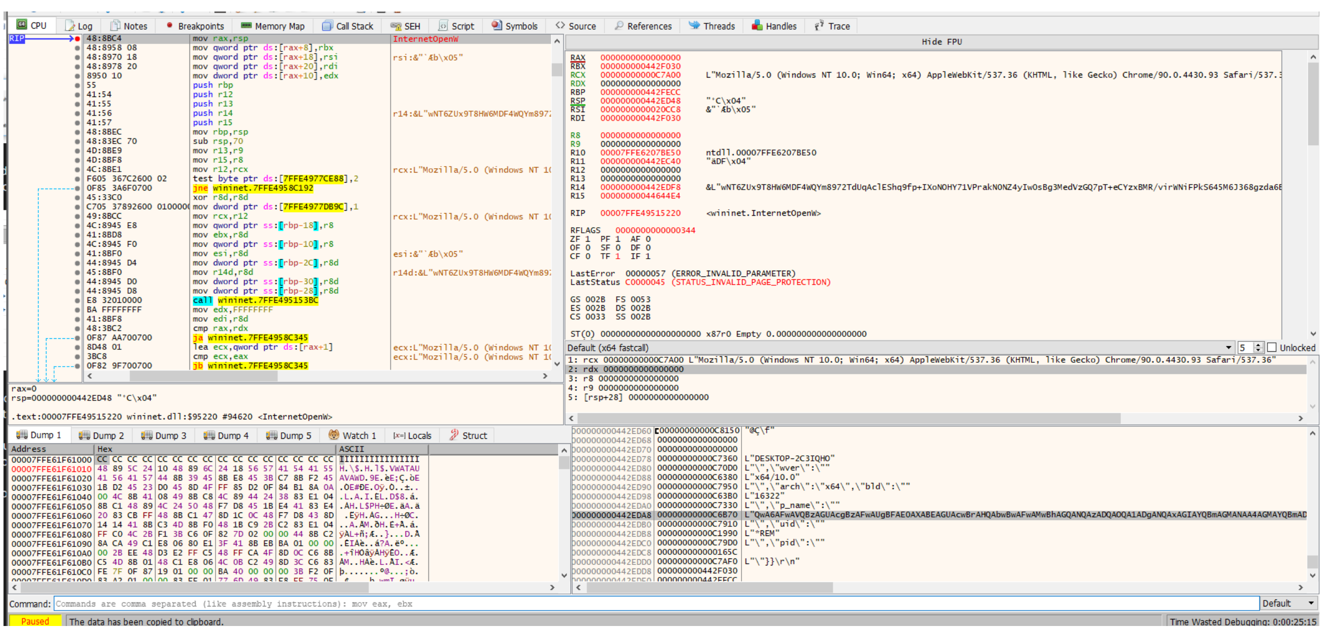
#	Module	Address	Offset	Location
1	KERNELBASE.dll	0x00007ffce2e7...	0x4d18e	MultiByteToWideChar + 0x25e
2	wininet.dll	0x00007ffccdff...	0x213d9	InternetCreateUrlW + 0x3d89
3	wininet.dll	0x00007ffccdff...	0x22826	HttpOpenRequestW + 0x12a6
4	wininet.dll	0x00007ffccdff...	0x21953	HttpOpenRequestW + 0x3d3

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.

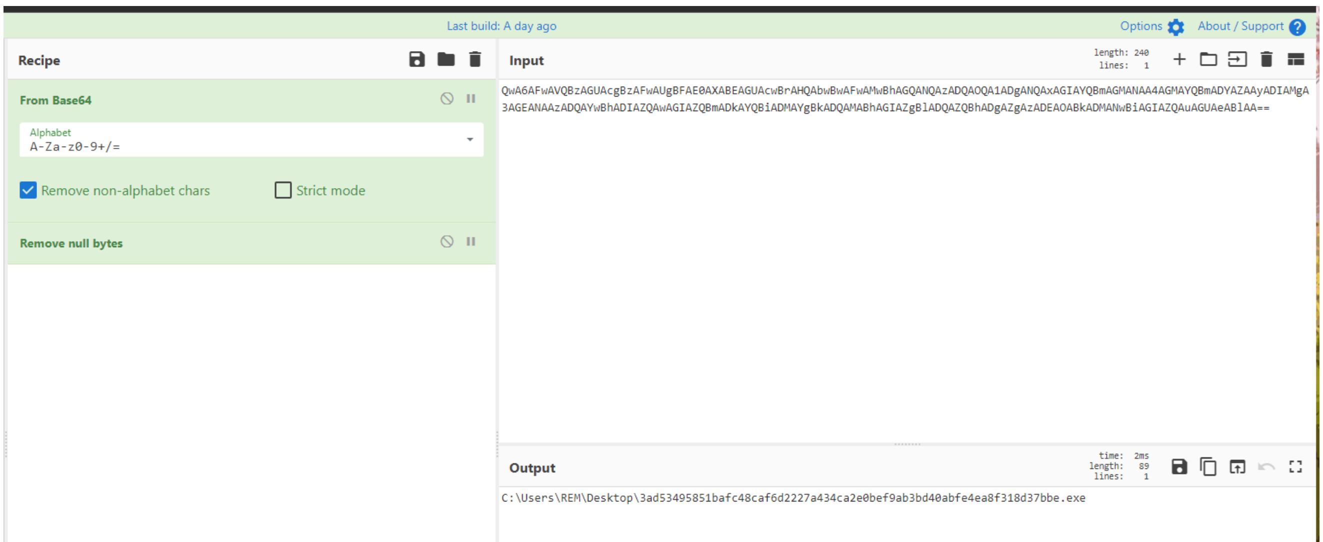
After letting the program run for a while, I set a breakpoint on a couple of the common HTTP APIs. We got a hit on InternetOpenW; in my suprise, in the stack window, here we are. We have the unencrypted data starting at us!



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":"&lt;base64 blob&gt;",
        "uid":"REM",
        "pid":""
        }
}
```

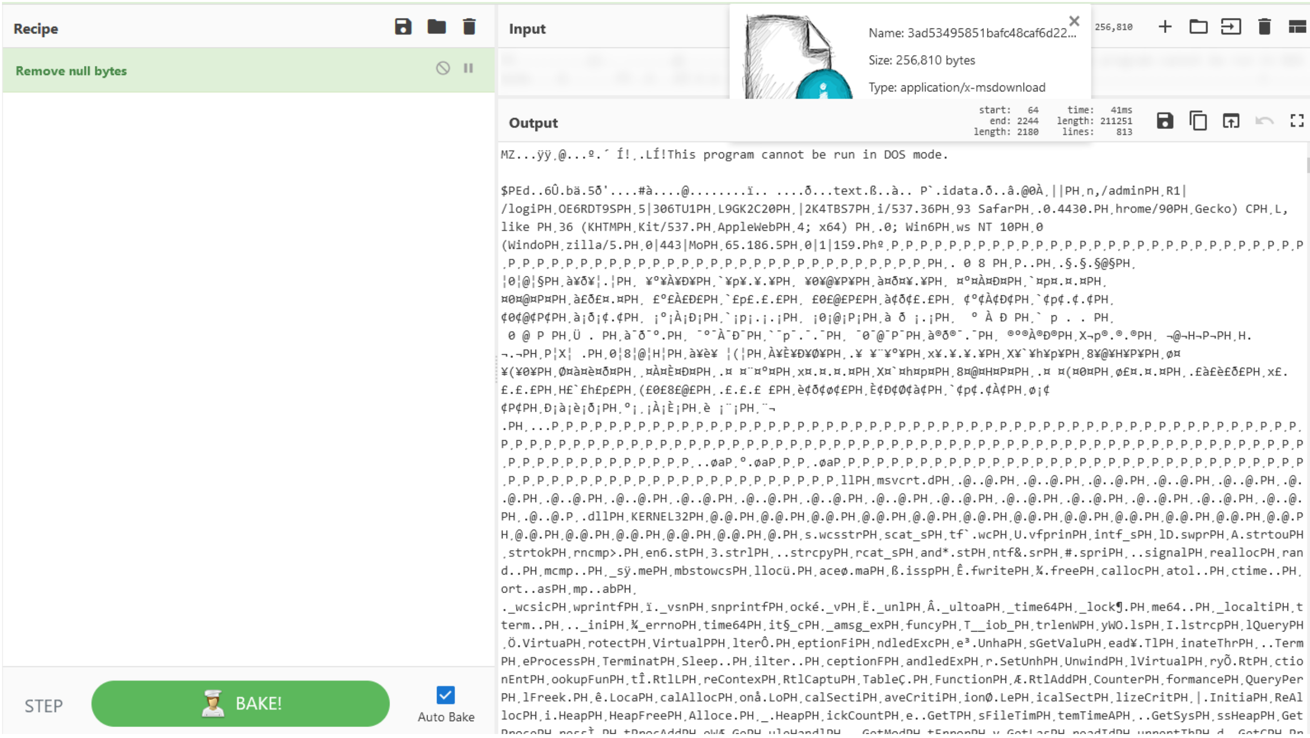
Unfortunately, our analysis stops here as we don't have a live C2 server to observe interactions with. Though, we could explore *how* the badger interacts with the C2 server if we carefully observe how the badger parses the response from the C2 server. There is definately some hardcoded commands that 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.strlPH...strcpyPH.rcat_sPH.and*.stPH.ntf&.srPH.#.spriPH...signalPH.reallocPH.ran
d..PH.mcmp..PH._sÿ.mePH.mbstowcsPH.llocü.PH.aceø.maPH.ß.isspPH.Ê.fwritePH.¾.freePH.callocPH.atol..PH.ctime..PH.
ort..asPH.mp..abPH.
._wcsicPH.wprintfPH.ï._vsnPH.snprintfPH.ocké._vPH.Ë._unlPH.Â._ultoaPH._time64PH._lock¶.PH.me64..PH._localtiPH.t
term..PH..._iniPH.¾_errnoPH.time64PH.it§_cPH._amsg_exPH.funcyPH.T__iob_PH.trlenWPH.yWO.lsPH.I.lstrcpPH.lQueryPH
.Ö.VirtuaPH.rotectPH.VirtualPPH.lterÔ.PH.eptionFiPH.ndledExcPH.e³.UnhaPH.sGetValuPH.ead¥.TlPH.inateThrPH...Term
PH.eProcessPH.TerminatPH.Sleep..PH.ilter..PH.ceptionFPH.andledExPH.r.SetUnhPH.UnwindPH.lVirtualPH.ryÕ.RtPH.ctio
nEntPH.ookupFunPH.tÎ.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.ressÌ.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.ÖH.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. %S
PH.osedPH.] ClPH.
[-PH.ST%sPH.SPOPH.%s%ls%PH.%ls%s%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.%S%P.P.STPH.%lsPOPH.%lsPH.s%luPH.%S%lPH.S%lsPH.%ls%PH.ls%SPH.V2%PH.\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.0","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. (%0PH.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.----
```


- 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,G¦þÿ+¦þÿPH,G¦þÿD þÿPH,G¦þÿG¦þÿPH,G¦þÿD þÿPH,G¦þÿG¦þÿPH,G¦þÿD þÿPH,D þÿG¦þÿPH,) þÿG¦þÿPH,-
£þÿ8¤þÿPH,©¢þÿô¢þÿPH,D þÿD þÿPH,D þÿD þÿPH,^ þÿPH,%d.
PH,ype 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,badP
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.

```
£þÿ8¤þÿPH,©¢þÿô¢þÿPH,D þÿD þÿPH,D þÿD þÿPH,^ þÿPH,%d.
PH,ype 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,expiPH,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,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,%ls]PH,ed [PH,nectPH, ConPH,[+]PH,x%x
PH,E: 0PH,[-] PH,%lsPH,%ls\PH,%SPH,
FALSE
PH,sTRUEPH,G%lPH,NNINPH,YRUPH,READPH,UEDPH,QUEPH,BLEDPH,DISAPH,OWNPH,UNKNP,PH,rAtoiPH,mpBadgePH,dgerWcscPH,trcm
pBaPH,BadgerSPH,erMemsetPH,cpyBadgPH,adgerMemPH,WcslenBPH,nBadgerPH,gerStrlePH,tchWBadPH,gerDispaPH,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, %lsPH,20lsPH,
%-PH, %luPH,] E:PH,
[-PH, %lsPH,25lsPH,s %-PH,-20lPH,th%PH,
PaPH,----PH,----PH,----PH,----PH,----PH,----PH,----PH,----PH,----PH,----PH,----PH,----PH,----PH,ls
-PH,ls %PH,%-25PH,0ls PH,%-2PH,aresPH,
ShPH, %lsPH,tingPH,meraPH, EnuPH,[+]PH,NamePH,ule PH,ModPH,NamePH,any PH,CompPH,ionPH,riptPH,DescPH,s

PH,: %lPH,5ls PH, %-1PH,
```

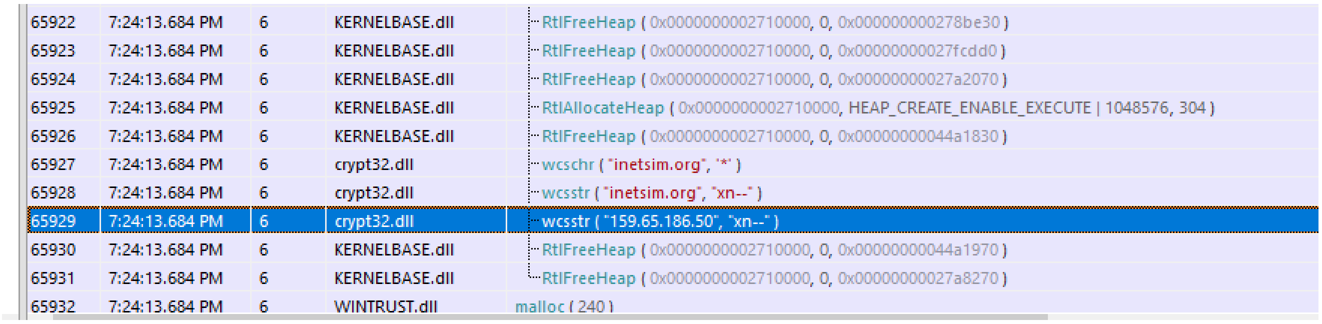
I've already loaded up the binary into Ghidra and there's a whole lot of nothing. It seems to be a bit beyond my skill level to reverse engineer in a classic sense, so I'll have to do some more research on my own time to figure out if I can post a followup showing off the actual binary internals.

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.

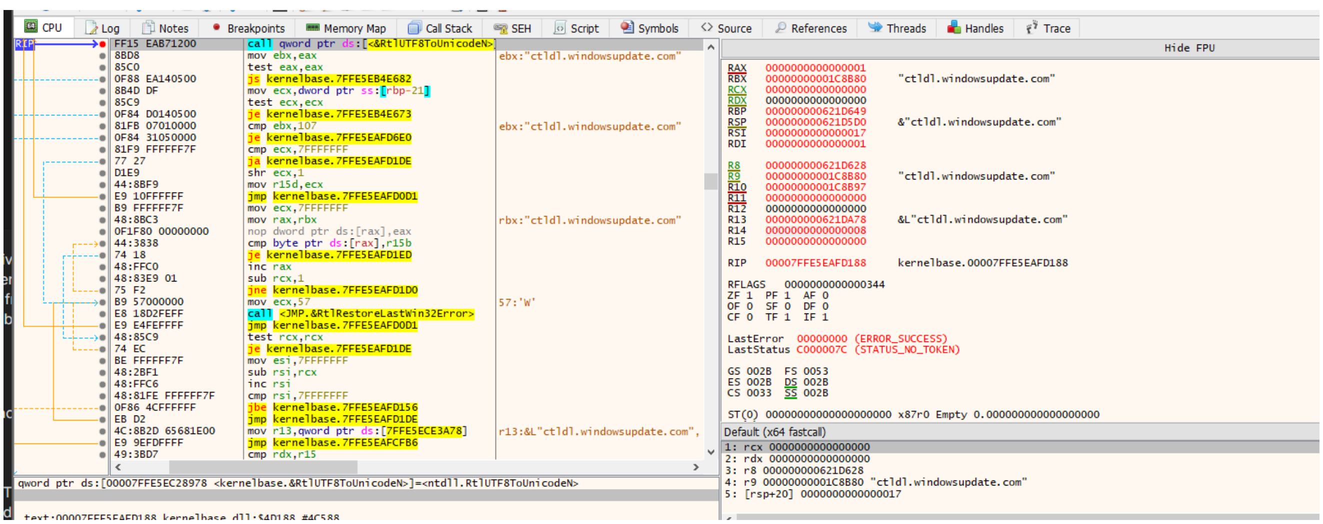
PUNYCode! The thing I forgot existed?

Here is an interesting String Compare after executing a HTTP Request; it asppears 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



Traffic Generation to windowsupdate.com

Another interesting aspect of this badger is that it periodically reaches out to `ctldl.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.



I hope you all enjoyed :) ~Ronnie

Comments