Network Traffic Analysis Exfil (Hard)

In this challenge the user was provided a single .pcap file that contained network traffic. Based on this network capture they needed to:

1. Figure out which IP was leaking the flag

2. Figure out which IP was receiving the flag

3. Figure out what the flag was Tools I Used wireshark tshark xxd Frhed (didn't use this, personally, but included for reference) Process OK, so I can't deny, this is the one challenge that took me the longest. I spent nearly the entire day (yes, including during classes) trying to figure this one out.

I spent a solid 3 to 4 hours trying to extract SSL keys, another 2 to 3 hours trying to extract and reassemble the bittorrent data, and at least as much telling myself "Just because the web says its not possible doesn't mean its not possible". Well, I can confirm, if you read it on the internet; its true. (OK, I can't really confirm it; but if I was under this mind set, I'd have lost a lot less time). I searched both content, and columns for SKY and NCL which would be keywords for flags; I even converted it to hex and searched for the hex values of SKY and NCL .

I used wireshark to extract all files from the HTTP streams (nothing of use), and countless other things until I was sitting in my last class of the day and ready to give up. (Quick joke, why is what your trying to find always in the last place you look? Because once you find it, you stop looking!) Before I gave up, I decided that I would go though each and every packet one at a time, painstakingly slow and looking at the hex value of each and every packet. (Hey, I had nearly an hour left in class, yes I have a midterm coming up, but this was driving me crazy) 4315, 4315 packets later.... I saw something, at first I wasn't sure what I saw, but I saw it. This IMCP packet was DIFFERENT than all of the rest! Every IMCP packet I had seen up to this point looked like this: But this one.... This one didn't.... Having gone down one too many rabbit holes; and (thankfully) not having found the mad hatter yet; I decided to take this slow.

First thing I did was add the "data" field to be displayed as a column, Now, I can see the data field displayed in the main view on wireshark. Since it was an ICMP packet that was different, lets set a filter and look at only ICMP packets, and once we set that filter lets scroll down to this packet we just found. OK, so its NOT just this packet that's different! AND the other "odd" packet we can quickly see is from/to the same host! Looking at the info, we can see that its a ping request and ping response. So if we use our knowledge of ping the recipient of the request, will simply return the data in the reply. So lets filter this a little more, and specify which sender and receiver we are looking for specifically and only look at the ping request 's NONE of this data looks the same! Mh.....

If we look at the first two packets, what is in the data field? Having just done another CTF, and having looked deeply at PNG files; I know this is EXACTLY how a PNG file starts; (kind of given away by the first 4 HEX groups in the first packet) But, lets assume you don't know this, lets download an open a random PNG in a hex editor (or create one in paint). OK, To me, that confirms it! There is PNG data hidden in these ping requests. Lets figure out what this PNG is an image of! First, lets File->Export Specified Packets to a new .pcap file so we don't need to worry about anything besides what we care about. Lets call it pingpng.pcap OK, now we need to extract the data field! OK, well that's closer, but not exactly what we need. We know binary files don't have new lines in them.... OK! That should work! Lets pipe it to a file flag.png

Now, lets open that file up and see what we've got! Erm.... ok, so maybe that doesn't work. I forgot that a binary file need to be saved as binary, not as hex... Lets pass our hex string though xxd and then save the file! tshark -r pingpng.pcap -T fields -e data.data
89504e470d0a1a0a0000000d49484452000002590000013008020000000b89f31f00000001735247
4200aece1ce90000
000467414d410000b18f0bfc6105000000097048597300000ec300000ec301c76fa8640000111149
444154785eeddd4d
6ee3bab606d03790d77ca3492b6349a3c652408652c8509219dc46508d204080fbe4c4a9d8fc9148
89725cb5d7023ba7
62cbb424ee8f947cecfff92f00c4260b01884e1602109d2c04203a59084074b21080e8642100d1c9
4200a29385004427
0b01884e1602109d2c04203a59084074b21080e8642100d1c94200a293850044270b01884e160210
9d2c04203a590840
74b21080e8642100d1c94200a293850044270b01884e1602109d2c04203a59084074b21080e86421
00d1c94200a29385
0044270b01884e1602109d2c04203a59084074b21080e8642100d1c94200a293850044270b01884e
1602109d2c04203a tshark -r pingpng.pcap -T fields -e data.data | tr -d '\n' tshark -r pingpng.pcap -T fields -e data.data | tr -d '\n' | xxd -r -p > flag.png PERFECT

We open that file up and see a SKY-AAAA-#### flag! That's the format we want! So, looking back at the first packet in wireshark.... We can see that the source (IP leaking the flag) is 192.168.64.137 and the dest nd the destination (IP receiving the flag) is 138.197.108.176 All 3 parts of this challenge!

Log Analysis Employee Activity (Hard)

In this challenge the user is given a zip file containing three separate log files email.log , rdp.log , http.log . Based on these three log files they need to:

1. Figure out when files were exfiltrated from the network

2. Figure out the size (KB) of the files exfiltrated

3. Find a possible insider threat Tip Work smarter not harder;

I can tell you that you will not be sitting in front of a computer reading each and every entry in a log file. Use the tools you have available to generate information from the logs into something you can easily read and review. Use that data to find things that stand out from a "normal" baseline. Tools I used python notepad++ (I did try using things such as splunk, and a few other log viewers, but none preformed how I wanted them to as such they have been ignored.) Process This task was by far the most daunting to me when I first looked at it; 50253 entries for rdp.log 272773 entries for email.log 1303021 entries for http.log It actually looked bad enough that I didn't want to complete it until it was the last task I had to do. So, for problems like this I try to always keep in mind, that just because there are a lot of entries, doesn't mean its a lot of data, let alone meaningful data. So lets start with the file that has the least amount of entries, and figure out what we have. rdp.log OK, so in the first 10 entries what do we know? We know that there are entries for when someone logs into the RDP server, and we know when a file is transferred as well as who transferred said file.

With this little bit of information, lets see who has logged into the server. In order to do this I wrote a python script and had it output the data to me.

[01/Dec/2015:5:23:01 -0500] - Successful RDP Authentication to Domain Host for user "advertising"
[01/Dec/2015:5:44:21 -0500] - Successful RDP Authentication to Domain Host for user "secretary"
[01/Dec/2015:6:06:22 -0500] - Successful RDP Authentication to Domain Host for user "michelle"
[01/Dec/2015:6:14:29 -0500] - Successful RDP Authentication to Domain Host for user "parker"
[01/Dec/2015:6:23:52 -0500] - Successful RDP Authentication to Domain Host for user "office"
[01/Dec/2015:6:33:43 -0500] - Successful RDP Authentication to Domain Host for user "anatol"
[01/Dec/2015:7:04:15 -0500] - File Transfer (41542KB) of document type .docx across network requested by "advertising" [01/Dec/2015:7:07:41 -0500] - Successful RDP Authentication to Domain Host for user "hardison" [01/Dec/2015:7:11:13 -0500] - Successful RDP Authentication to Domain Host for user "daniel" [01/Dec/2015:7:17:24 -0500] –

File Transfer (17169KB) of document type .zip across network requested by "advertising" f=open("rdp.log", "r") lines = f.readlines() users = {} for line in lines: userS = line.find("\"")+1 userE = line.find("\"",userS) user = line[userS:userE] if not user in users: users[user] = 1 print() print(users

That output doesn't look the best; but at least now we know what to expect. And there is nothing in there stands out right away; lots of user accounts, and lots of "service" accounts, but nothing that says LOOK AT ME So lets take this a step further; which of these users have downloaded a file, and what was the size of that file? Lets also make our life a bit easier and output in a CSV format...

{'advertising': 1, 'secretary': 1, 'michelle': 1, 'parker': 1, 'office': 1, 'anatol': 1, 'hardison': 1, 'daniel': 1, 'alexeev': 1, 'webmaster': 1, 'tyrome': 1, 'abramov': 1, 'kapt': 1, 'bobrov': 1, 'con': 1, 'gordoon': 1, 'judy': 1, 'bogdanov': 1, 'glavbuh': 1, 'connor': 1, 'accounting': 1, 'kevin': 1, 'andre': 1, 'afanasev': 1, 'dir': 1, 'box': 1, 'belousov': 1, 'support': 1, 'joe': 1, 'home': 1, 'marketing': 1, 'cody': 1, 'chris': 1, 'andreev': 1, 'jane': 1, 'davella': 1, 'aravind': 1, 'amara': 1, 'george': 1, 'gregory': 1, 'smith': 1, 'john': 1, 'anna': 1, 'sam': 1, 'biggie': 1, 'manager': 1, 'testing': 1, 'monkey': 1, 'export': 1, 'director': 1, 'chen': 1, 'buhgalteria': 1, 'agata': 1, 'aleksander': 1, 'account': 1, 'shawn': 1, 'ashton': 1, 'oracle': 1, 'billing': 1, 'bill': 1, 'mchams': 1, 'baranov': 1, 'avdeev': 1, 'info': 1, 'fabrika': 1, 'spencer': 1, 'aleksandrov': 1, 'uploader': 1, 'user': 1, 'james': 1, 'mail': 1, 'anya': 1, 'dragon': 1, 'business': 1, 'michael': 1, 'corp': 1, 'aksenov': 1, 'blinov': 1, 'jack22': 1, 'hr': 1, 'jerome': 1, 'dan': 1, 'design': 1, 'agafonov': 1, 'moscow': 1, 'mike': 1, 'kadry': 1, 'personal': 1, 'contact': 1, 'economist': 1, 'sophie': 1, 'beckt': 1, 'arhipov': 1, 'alla': 1, 'max': 1, 'ryan': 1, 'holding': 1, 'administrator': 1, 'andrey': 1, 'kruds': 1, 'mcdonald': 1, 'avto': 1, 'post': 1, 'alexe': 1, 'nate': 1, 'root': 1, 'gus': 1, 'kang': 1, 'jacky': 1, 'rbury': 1, 'jack': 1, 'christina': 1, 'climb': 1, 'mysql': 1, 'machin': 1, 'fin': 1, 'chin': 1, 'sales': 1, 'finance': 1, 'birns': 1, 'contactus': 1, 'bank': 1, 'carly': 1, 'email': 1, 'rachel': 1, 'art': 1, 'company': 1, 'admin': 1, 'mahe': 1, 'reklama': 1, 'elliot': 1, 'elsayyad': 1, 'DomainAdmin': 1} f=open("rdp.log", "r") lines = f.readlines() downloadCount = {} downloadSize = {} for line in lines: userS = line.find("\"")+1 userE = line.find("\"",userS) user = line[userS:userE] if line.find("File Transfer") >=0: if user in downloadCount: downloadCount[user] += 1 else: downloadCount[user] = 1 sizeS = line.find("File Transfer (") sizeE = line.find("KB)",sizeS) size = line[sizeS+15:sizeE] if user in downloadSize: downloadSize[user] += int(size) else: downloadSize[user] = int(size) print() for user in downloadCount:

print(user,",",downloadCount[user],",",downloadSize[user]) advertising , 283 , 6758281 anatol , 281 , 6921332 office , 275 , 7026142 secretary , 284 , 7267088 bogdanov , 214 , 5327223 daniel , 220 , 5205694 abramov , 236 , 5940932 amara , 249 , 6078850 andreev , 271 , 6807159 jane , 270 , 6649115 accounting , 280 , 7073163 aravind , 251 , 5939495 parker , 240 , 6093411 alexeev , 281 , 6895573 glavbuh , 221 , 6061639 gordoon , 249 , 6223940 chris , 265 , 6166097 support , 249 , 5968564 john , 208 , 5183835 mchams , 258 , 6738844 info , 295 , 7307765 aksenov , 306 , 7698868 gregory , 309 , 8194829 sam , 285 , 7009697 export , 232 , 6079126 uploader , 261 , 6500878 aleksander , 295 , 7064252 hr , 242 , 6174430 buhgalteria , 245 , 6132043 director , 243 , 6249107 testing , 247 , 6188442 corp , 278 , 6862581 dragon , 285 , 7241892 george , 206 , 5324441 marketing , 277 , 6818838 chen , 287 , 7474269 avdeev , 285 , 6877407 business , 330 , 8160602 box , 262 , 6714210 con , 254 , 6368860 moscow , 289 , 7202527 judy , 279 , 7024600 dir , 242 , 5890868 jerome , 219 , 5958193 agafonov , 254 , 6392452 belousov , 269 , 6788251 personal , 235 , 5784504 contact , 259 , 6837404 joe , 286 , 6756662 aleksandrov , 277 , 7087649 bill , 279 , 7107200 shawn , 280 , 7517926 anya , 244 , 6192642 spencer , 309 , 8307911 arhipov , 260 , 6537768 design , 301 , 7362756 andrey , 267 , 6795352 alla , 270 , 6667208 home , 272 , 6718106 manager , 274 , 6779785 billing , 279 , 6938128 biggie , 253 , 6342653 ryan , 248 , 5914636 account , 290 , 7215264 smith , 254 , 6450081 anna , 252 , 6103888 kevin , 281 , 7125205 nate , 275 , 7174250 kadry , 270 , 6797417 afanasev , 254 , 6405375 post , 289 , 7391345 rbury , 270 , 6732566 gus , 208 , 5050376 kapt , 251 , 6223817 sophie , 279 , 7244533 holding , 247 , 6012998 kruds , 245 , 6250085 blinov , 239 , 5818175 kang , 258 , 6395043 tyrome , 262 , 7183771 mysql , 205 , 5388969 alexe , 305 , 7823089 andre , 299 , 6884470 chin , 259 , 6365770 cody , 274 , 6780733 machin , 261 , 6414218 mcdonald , 232 , 6051607 ashton , 268 , 6839693 root , 278 , 6921954 james , 276 , 6712194 economist , 246 , 6294699 christina , 260 , 6312628 monkey , 250 , 6680266 jack , 292 , 7369098 agata , 290 , 7341106 climb , 263 , 6502922 beckt , 259 , 6627648 oracle , 273 , 6465941 jacky , 218 , 5618580 fin , 265 , 6550640 administrator , 246 , 6073092 mail , 235 , 6073431 michelle , 272 , 6753394 connor , 213 , 5193074 contactus , 262 , 6923876 jack22 , 276 , 7322146 carly , 262 , 6524788 birns , 255 , 6314319 finance , 239 , 5593175 bank , 280 , 7017253 baranov , 232 , 5706589 dan , 305 , 7186094

Off the bat; nothing looks too weird; so lets open that up in excel and sort the data; Lets start with who has the most file downloads.

OK, well we have two users that have downloaded; 330 files total; followed closely behind by a user with 329; none of that seems too abnormal.

Now lets checkout who has the largest File Size Total. OK, now that STANDS OUT DomainAdmin downloaded one file, and that single file was larger than anyone else's total download size!

Lets open the rdp.log file and take a closer look at that. Lets search for requested by "DomainAdmin" because that is how the file download entries look including the username.

Ok, now that is a red flag for sure! DomainAdmin logged in at 2330, at least an hour an a half after anyone else, downloaded the file, and then disconnected a full five and a half hours before the next user logged in. If we simply search for "/2016:23" we should be able to find all other events that happened at 11PM for the rest of that year. How many results do we get?

OK, so this looks like the RDP log; who has sent an email; its size; and if it had an attachment. Lets start simple again; which users are sending emails, how many, and how many have attachments?

Again, lets open this up in Excel and see what we can find. Mhhh, Nothing stands out here, lets sort by emails high to low, and then low to high.

OK, so just looking at this, we know something is off and weird; Where is the URL for the request? Even just getting the index page should result in a "GET \ HTTP/1.1" but none of these entries have any thing like that! Because of this fact alone, I'm going to assume that the http.log is a red herring and go back to the the email.log 33.85.183.129

- - [01/Jan/2016:1:00:00 -0500] "GET HTTP/1.1" 200 1513 "-" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322)" 5.128.198.159 - - [01/Jan/2016:1:00:01 -0500] "GET HTTP/1.1" 404 1236 "-" "Mozilla/5.0 (X11; Linux i686) AppleWebKit/534.24 (KHTML, like Gecko) Chrome/11.0.696.50 Safari/534.24" 110.198.136.12 - - [01/Jan/2016:1:00:02 -0500] "GET HTTP/1.1" 200 790 "-" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322)" 152.144.158.47 - - [01/Jan/2016:1:00:05 -0500] "GET HTTP/1.1" 200 2088 "-" "Mozilla/5.0 (X11; Linux x86_64; rv:6.0a1) Gecko/20110421 Firefox/6.0a1" 133.214.142.129 - - [01/Jan/2016:1:00:12 -0500] "GET HTTP/1.1" 200 1102 "-" "Mozilla/5.0 (X11; Linux x86_64; rv:6.0a1) Gecko/20110421 Firefox/6.0a1" 148.243.30.40 - - [01/Jan/2016:1:00:16 -0500] "GET HTTP/1.1" 200 1073 "-" "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322)" 83.20.185.93 - - [01/Jan/2016:1:00:17 -0500] "GET HTTP/1.1" 200 1541 "-" "Mozilla/5.0 (X11; Linux i686) AppleWebKit/534.24 (KHTML, like Gecko) Chrome/11.0.696.50 Safari/534.24" 112.143.173.161 - - [01/Jan/2016:1:00:18 -0500] "GET HTTP/1.1" 200 608 "-" "Mozilla/5.0 (X11; Linux i686) AppleWebKit/534.24 (KHTML, like Gecko) Chrome/11.0.696.50 Safari/534.24" 210.169.191.144 - - [01/Jan/2016:1:00:51 -0500] "GET HTTP/1.1" 200 1828 "-" "Mozilla/5.0 (X11; Linux i686) AppleWebKit/534.24 (KHTML, like Gecko) Chrome/11.0.696.50 Safari/534.24" 141.140.228.49 - - [01/Jan/2016:1:00:54 -0500] "GET HTTP/1.1" 200 852 "-" "Mozilla/5.0 (X11; Linux i686) AppleWebKit/534.24 (KHTML, like Gecko) Chrome/11.0.696.50 Safari/534.24" So, one thing we didn't look at before what what types of files were being sent... Lets take a look at what file types are being sent.

:O rar Now that's a red flag if I've ever seen one! Who would ever use a rar file for work, if they weren't a hacker! exe That's interesting as well! Why would anyone be sending exe files over email? Lets write another script and find out which users send what types of files.

Lets run that and open the output up in Excel. OK, so based on this, I think that maybe people other than just hackers use rar files.... But look at that, exe doesn't look to be a popular file type to send or receive! Lets sort the exe column high -> low. OK, now that stands out! dan was the only user to send an exe file via email; lets find that entry and take a closer look!

OK, now lets look at the data exfiltration entry we found earlier because maybe they correlate. OK, now that is fishy for sure! The day before data was exfiltrated from the network; dan sent an exe file, and its the only exe file we have in the email log. dan has to be our insider threat! (He should have zipped it first to help cover his tracks....)

Malware Sample (Hard)

In this challenge the user is given a binary (32-bit and 64-bit versions available) and they need to

1. Figure out how long the malware stays dormant

2. Find the hidden flag in the malware

3. Figure out the domain the malware will contact on 2025-05-07 11:00:00 UTC 4. Figure out how many datagram packets the malware sends ou

t Tools I used Binary Ninja (Using Ghidra may be possible) gdbgui (Easier than plain old gdb) Wireshark Process First thing first, try running the program and see what happens. OK, so we know how to enter the UID provided now; lets try that again.

Well that was interesting.... It took what seemed like a minute before that getaddrinfo line was printed... My first thought? Let me count this.... 1, 2, 3, 4, ... 55, 56, Done! Mhh, This isn't scientific in the least bit. Maybe I could use a stopwatch, that's better at keeping time than I am counting out loud to myself. But, will I really be able to stop/start the stopwatch right when I need to? Lets think about this; I don't know how the binary works yet, so lets take a look at it in a disassembler. Lets open up Binary Ninja, and take a look at what's going on.

Honestly; I don't like assembly; so lets change to a higher level view. From the bottom right corner, we'll select Options->Medium Level IL

Ah, Much better! Not much is being done here, so lets double click on the blue main which is the first thing that the program does call wise.

OK, so in the top, we see the main function (variable initializations and definitions have been cut out) and the last line there we see an if statement; In the code block on the right we see the error message we received the first time we tried to run the program without a uid ; So obviously we're not running that code; so lets look over at the left hand side.

Oh! A sleep command; well with some programing background, we know that sleep will "pause" the program for some amount of time; Looking up the sleep command, we can see that the argument being passed to sleep is the number of seconds. Now we have two options; we can try to reverse engineer some more of this code; or we can try using a debugger;

Personally, I like to use the debugger to help ensure I don't mess up any code and get an invalid entry. So lets launch gdbgui Note: when launching gdbgui we don't include the uid argument Now; lets navigate to that URL, and take a look at the interface.

Lets start from the bottom, and work our way up to the top. The first thing is the gdb command line, this will be where we enter commands into the debugger Above that we have the output window. Note: Notice how Function "main" not defined is there and in red? gdbgui automatically creates a break point for the main function (discussed more shortly), our file doesn't have a "main" function. Above that on the right; is "internal" information; these are values that involve memory; current location in the

program execution and other such things. (discussed more shortly) To the left of that is the source and assembly viewer; Our program isn't running; so nothing is displayed yet. To the left of that, is the file system; Feel free to hit "hide filesystem" we won't be using it

Finally, to the top right we have some buttons. Lets look at these: For the purpose of this write up I'm only going to be discussing 5 of these buttons The "replay" button; this button will start/restart the program The "play" button will continue execution of the program until the next breakpoint The "pause" button will pause the program execution and allow you to take a closer look the "NI" button, will execute the next instruction (the instruction the arrow is currently on in the assembly view) The "SI" button, will "step into" the current instruction if it is a function call. S

Side Trip for Explaining For the purpose of ensuring everyone is on the same page; lets look at an over simplified explanation of "NI" and "SI". Lets pretend that the above code is being debugged; Our debugger will point to the line we are on, but haven't executed yet; so when we are "on" the first line in main, our variables will look like this: When we hit NI; it will execute the instruction for the line we are on, and the debugger will point to the next line that hasn't been executed yet.

Now our variables will look like this: Now we hit NI again; the debugger will point to the next line and our output will look like this: This is where things get interesting, lets compare NI and SI Hitting NI If we hit NI, the debugger will point to the print line, and our variables will look like: int cube(int y) { int retval = y; retval *= y; retval *= y; return retval; } int main() { int x = 3; x = x*2; x = cube(x); printf("The cubed Value is %d", x); return 0; } x: 0 x: 3 x: 6 HITTING SI However if we hit SI, the debugger will step into the cube function; our variables will look like this:

Notice how we are no longer have x but instead y and retval which are the local variables for the function we "stepped into" Back to the Challenge So; lets start out by hitting the "replay" button and seeing what happens. Oh! we didn't pass it the uid so lets fix that. In the GDB command line lets enter: If we were run hit the "replay" button now, it'll look like nothing is happening, (we may even get an error in the console window about gdb not responding; just give it a second) and then we'll get the following in the output: So that's the same output we got when we ran it in the terminal. Because there is no "main" function, there was no breakpoint set.

A breakpoint, is simply a point in code execution that the debugger will stop any additional code execution, and allow you to edit and view things that would other wise not be possible while the program is actively executing So we know that the function sleep gets called; lets set that as the first breakpoint: That should produce the following output: Now that we have a breakpoint we should enable it To disable it (not that we want to) we would run: x: 216 y: 6 retval: 0 [Inferior 1 (process 12345) exited with code 01] Usage: /malware-x64 set arg 44g493025d92543099b51201f9938729 [Inferior 1 (process 12346) exited with code 01] getaddrinfo: Name or service not known (gdb) break sleep break sleep Breakpoint 1 at 0x7fffff6b7c10: file ../sysdeps/posix/sleep.c, line 34. (gdb) en 1 We use 1 because that was the breakpoint number provided to use after we created it. Now that we have a breakpoint; lets hit the "replay" button again and see what happens.

OK, so we can now see that error we kept getting involving getaddrinfo lets keep looking though the program and if we see anything that stands were we to follow the program logic if we don't get an error.

(gdb) dis 1 OH, that looks like a flag! and based on it being used twice. If we look at the colors we can tell that the flag is "data_f1a", and if we look at the function its being used in sendto . Looking at the documentation for sendto we can see that the flag is in the position of the "buffer" variable so it is safe to assume that the program is simply sending the flag without any modification.

Two question down, Two to go! So, if we double click on data_f1a we can jump to the location in the binary where that string is stored; lets do that, and see what other strings the binary includes! So looking though this section of the program, we see an error message about system time being out of bounds; (we still don't know what that's about yet), and then its a domain %d.phonehome.cityinthe.cloud Well, that's a domain, but domains don't have % signs in them. Lets click on data_ed6 , and see where it gets called from

OK, so sub_b17 looks like it calls time() lets try to create a break using gdbgui for that function and see what happens. Now lets enable that new breakpoint Since we haven't done anything since we got the sleep time, lets just hit the "play" button so execution continues until the next time we hit a breakpoint. (Yes, it may take a second, because we're going to sleep for those 39 seconds)

So none of this looks useful; but we know what time() will return. At the time of writing we are looking for a value of ~ 1571458399 Lets hit "NI" until we see a value close to that in the registers. That looks like it'll be what we are looking for! (If you didn't notice yet, values highlighted in yellow are values that have changed.) Lets change this value; if we head over to this site we can enter in the future time, and get the unix timestamp we need to change rax to, and we find the value is 1746615600 To change the value of the register rax we will enter the following command:

OK, so there is an if statement that will send us to that error message; lets convert the hex number into a decimal and see what the comparison is. 0x67748580 in hex is equal to 1735689600 in decimal; which is what we want to set our system time to! Based on this information, it looks like we'll never be able to run any of the "valid" code! So lets change this! In Binary Ninja, right click that if line and select Patch->Invert Branch Look at that! the if statement was reversed! Lets save this modified code as a new binary; File->Save Content As and save the binary as a new binary. (malware-x64.2) So in theory we can test this; if we run that program in the terminal, the current time should give us an error, because its not larger than the future date. We didn't get that error last time when we didn't modify the time, so we must be on the right track! Lets close gdbgui and relaunch it with our newly modified binary

Lets set our argument and breakpoints OK, lets hit the "replay button" and find & modify the time again where we need to! Now, here is where some knowledge comes into play. We know that network traffic doesn't use domains, but IP addresses to communicate; so if we open up wireshark we should be able to capture the DNS request of the domain. Once that is done, lets hit the play button and check the output. Well that doesn't sound promising! $ ./malware-x64 44g493025d92543099b51201f9938729 Nice try, your system time is out of bounds. $ gdbgui ./malware-x64.2 (gdb) set arg 44g493025d92543099b51201f9938729 (gdb) break sleep (gdb) en 1 (gdb) break time (gdb) en 2 getaddrinfo: Name or service not known Lets check Wireshark. You can create a filter or search for "phonehome." and sh

Looks like we have our domain! If we look at the DNS responses; we can see that there is no IP address associated with it, which is causing our error with getaddrinfo Now to the final question! If we go back to Binary Ninja; we can look at the other branch from the error we were getting

If we look at it we see two sendto function calls; so, we can make a safe assumption that it sends two datagram packets! And that completes the challenge.