FEB
12

Triggers as a Windows persistence mechanism -- an example

@keydet89 posed the following question on Twitter:

property on an object. The serialization is base64 encoded, gzipped bytes of the original binary file.

Obviously you need a way to deserialize this data in order to analyze it. There's a new analysis script for this in Analysis\Deserialize-KansaField.ps1. More details on this in a later post.

-JSONDepth:
Another optional parameter that takes an int32 argument. Because we've added support for JSON, we had to add a parameter for specifying the depth of JSON's serialization. The default for Kansa is 10, which is sufficient for many things, but if you need to go deeper, you can.

-UseSSL:
If this optional switch is present, Kansa.ps1 will call New-PSSession with the -UseSSL switch and remote sessions will use Secure Sockets Layer, **iff** you've done the legwork to push certificates. More on this in a later post.

-Authentication:
Previous releases of Kansa only supported Kerberos (non-delegated, network) authentication, which is the safest authentication mechanism, if you're concerned about credential theft when running investigations across large numbers of hosts, which is something you should be concerned about. The trouble with only supporting Kerberos authentication is that it meant Kansa could not work in many scenarios, like authenticating against a local administrator account in cases where you either don't want to use a forest admin credential or you're investigating a machine that is not domain joined.

-Authentication is an optional parameter, acceptable arguments are "Basic", "CredSSP", "Default", "Digest", "Kerberos", "Negotiate" and "NegotiateWithImplicitCredential". Remember, some of these will put your credentials at risk for harvesting by attackers. The default is still Kerberos.

-Port:
And finally, for those running with PowerShell endpoints on alternate ports, you can use this optional

parameter with an argument to specify the remote management port to use on the remote host(s).

The first change above related to Kansa's output, makes the framework more forensically sound because we've eliminated a few cases where Kansa would previously write data to disk on remote hosts, potentially overwriting deleted data on those systems.

The second set of changes, relating to authentication and transport make the framework more capable for non-domain scenarios. One thing I can now do with the framework that I couldn't do before is run it against Azure VMs to collect data from the cloud... fluffy, fluffy clouds.

There are lots of other changes in Stafford as well and not all of them were committed by me. @z4ns4tsu made significant contributions to this release, including Modules\Disk\Get-MasterFileTable.ps1, a tool for parsing the MFT from target systems. Because it uses Win32 device namespace to access the file system in the raw, it bypasses file locks, attributes and access control lists.

@JValdezjr1 also contributed changes, including some code in kansa.ps1 that will save target lists to a file when Kansa builds the target list dynamically by querying AD for the list of hosts in the domain.

Kansa is maturing and this release is the most capable version yet. Please give it a try and if you run into problems, I'm the primary developer, so I'm sure there are bugs, open an issue at https://github.com/davehull/Kansa/issues and let me know what problems you're encountering.

I'll be following up in a few days with a post that demonstrates some of the new features in this release.

Happy hunting!

Posted 14th August 2015 by davehull

0 Add a comment

JUL
11

Cracking repeating XOR key crypto

My last post here, XOR'd play: Normalized Hamming Distance, was a lengthy bit about the reliability of Normalized Hamming Distance to determine the size of a repeating XOR key that was used to encrypt a string of text and was based on my experience working on the Matasano Crypto Challenges at cryptopals.com.

After a few weeks of focusing on other things, I returned to that effort and finished problem six from challenge set one, which says to use Normalized Hamming Distance to determine the probable key size and then use English letter frequency analysis to determine the probable bytes that make up the key. That's the short version.

The script I wrote for challenge applies some extra math to the problem of determining key size, because I found normalized Hamming Distance alone to not be very reliable. However, even with the extra math, the problem is still one of probability, not certainty. If you have no idea about the key size, the problem is even more expansive, though not completely open ended because if it's repeating key, it has to be no more than half the size of the encrypted byte stream.

Let's look at an example. Below I have already populated two variables, $plaintext and $key and am using XOR-Encrypt.ps1 to encrypt the $plaintext value with the $key and return a base64 encoded string. In the second command, I'm populating the $ciphertext variable with that base64 encoded string.

Click to enlarge

Now that we have a string of plaintext encrypted with a repeating XOR key, let's try cracking it.

.

Click to enlarge

In the screenshot above, I've run the Crack-XORRepeatingKeyCrypto.ps1 script with about the minimum set of command line parameters. The VERBOSE output line tells me that I didn't provide a MaxKeySize value, so the script has calculated the number of bytes in the ciphertext and has set the MaxKeySize to half of that value. If we're dealing with repeating key XOR crypto, the key must be at most half the size of the ciphertext, otherwise the key can't repeat (completely).

The script applies normalized Hamming Distance to find the top n most likely key sizes, five is the default, which you can see listed towards the bottom of the screen. Next, the script calculates the *most frequently occurring greatest common denominator*, hereafter MFOGCD, of those top n probable key sizes. In my testing, calculating the MFOGCD of the top n normalized average Hamming Distance values can be used to more accurately find the correct key size. This is a good example, as you can see, the correct key size is 30, but its normalized average Hamming Distance is fourth in the list of the top five.

My script has a programmatic bias towards smaller key sizes, especially where they are the MFOGCD.

This formula returns the correct key size more than 90% of the time in my testing where I was able to control for the MaxKeySize, whereas normalized average Hamming Distance alone only returns the correct key size about 40% of the time where MaxKeySize is controlled. If MaxKeySize is unknown, your mileage may vary.

After the script determines the most probable key size, it transposes the ciphertext into blocks aligned on the key size byte boundaries. In other words, if the script determines the key size is 30 bytes, the first block will be comprised of ciphertext bytes 0, 29, 59, 89, 119, etc. The next block begins with ciphertext byte 1, then 30, 60, 90, 120, etc. This process repeats until all ciphertext bytes have been allocated to their respective blocks.

Next the script moves into the actual brute-force phase, XORing each byte of the transposed blocks against ASCII printable characters. That's the default, you can apply all bytes 0x00 - 0xFF, via the -includeNonPrintable command line switch. If you think this takes longer, you're right.

After each transposed block is XOR'd with each byte, the English letter frequency calculation is applied and the byte that results in the highest score, is deemed to be the most probable byte for that position in the key and the process repeats for each byte in the key.

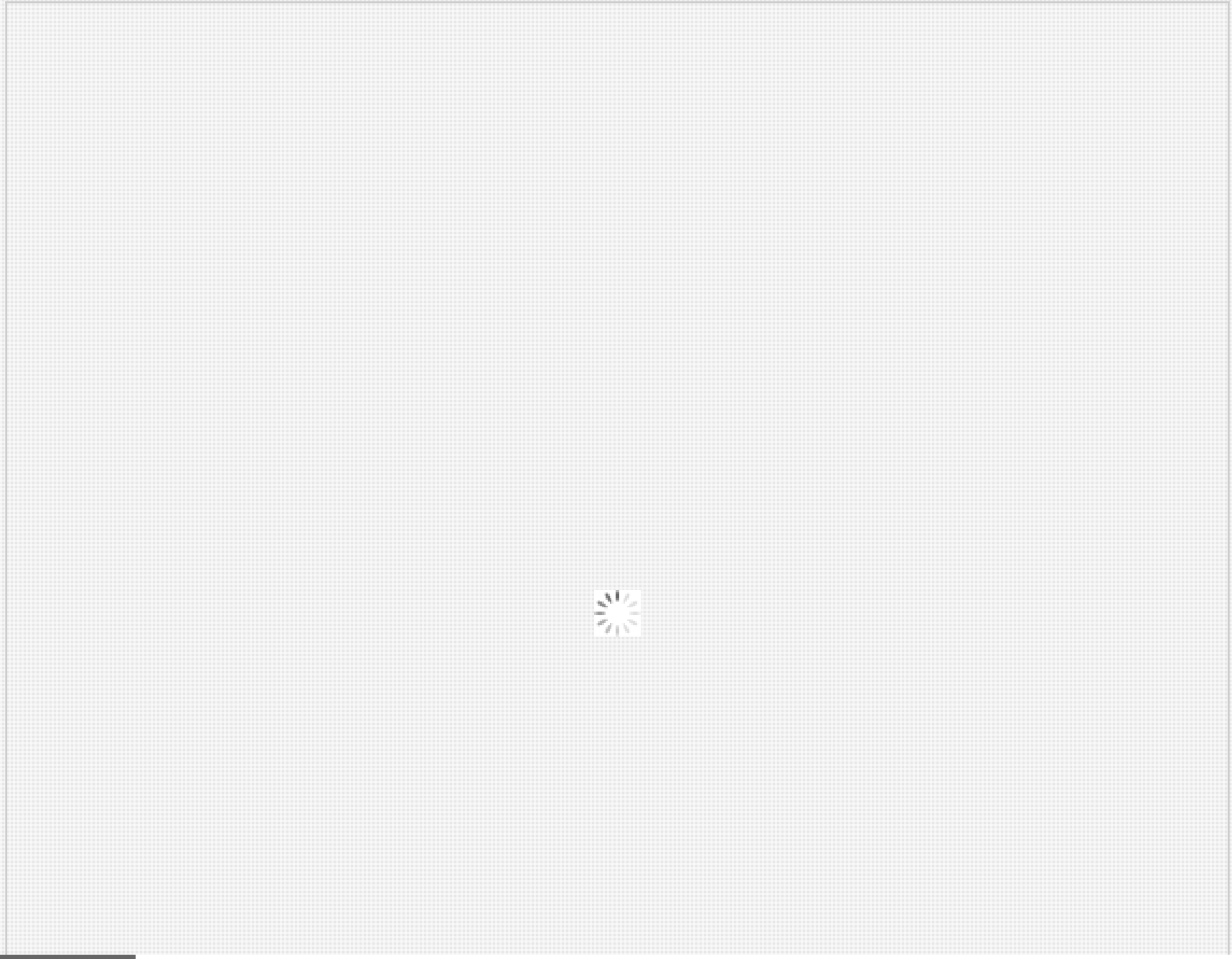
Once all the likely key bytes have been determined, the original ciphertext is XOR'd against that key and the likely plaintext is returned. You can see this above in the ProbableDecryptedValue property of the output. At first glance, the output looks pretty good, but if you read it carefully, you'll see it's not exactly correct. Remember we're dealing with probabilities.

If you look at that ProbableKey property, you can see the probable key and you may be able to guess which byte is incorrect. You could use the XOR-Decrypt.ps1 script, which can take a user supplied key string and decrypt the ciphertext if you want to be more exact. Without a key, XOR-Decrypt.ps1, assumes a single byte key and attempts to brute force it, but I digress.

So that's the script when it's working well, but the probabilities sometimes don't come out so nicely.

Here's another example:

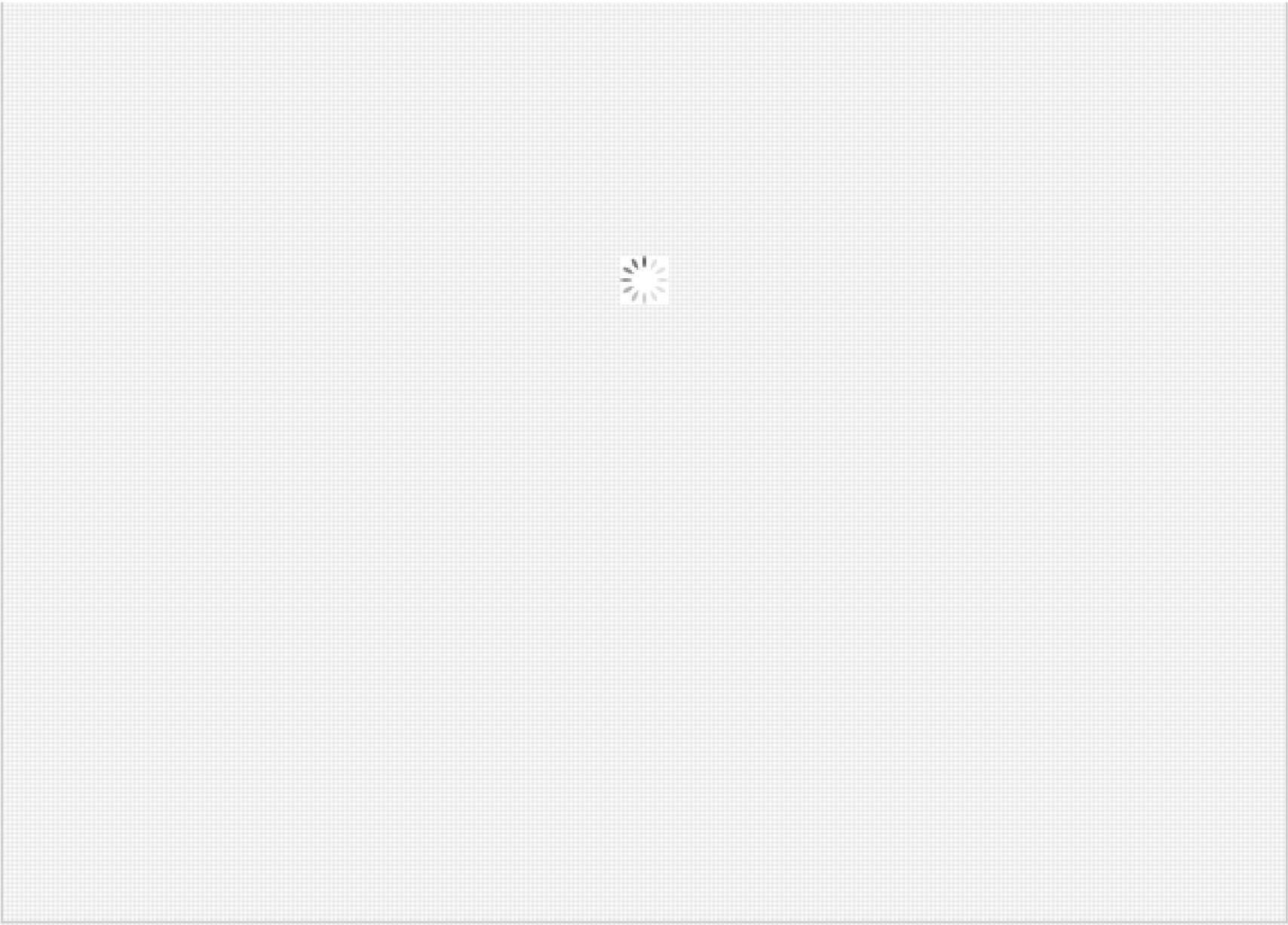
.



Click to enlarge

If you look at the ProbableDecryptedValue property here, you can plainly see this does not look like English plaintext. What happened? The script worked as written. It calculated the MFOGCD among the top five most probable key sizes, but that value, 16, was not also listed in the top five most probable key sizes, so it didn't try 16 as a key size. Instead, it selected the smallest key among the keys with the smallest normalized average Hamming Distance, 416 and that turned out to be horribly wrong, but all is not lost.

One approach that I've tried at this point, is to run the script again, but add the -MaxKeySize parameter with an argument matching the smallest key size from the above run, 48. Here's the output of that run:



Click to enlarge

This looks more like English plaintext, but it's still not perfect. Note the ProbableKeySize property is 16 bytes, that was the MFOGCD from the previous run so I'm not surprised to see it is the probable key

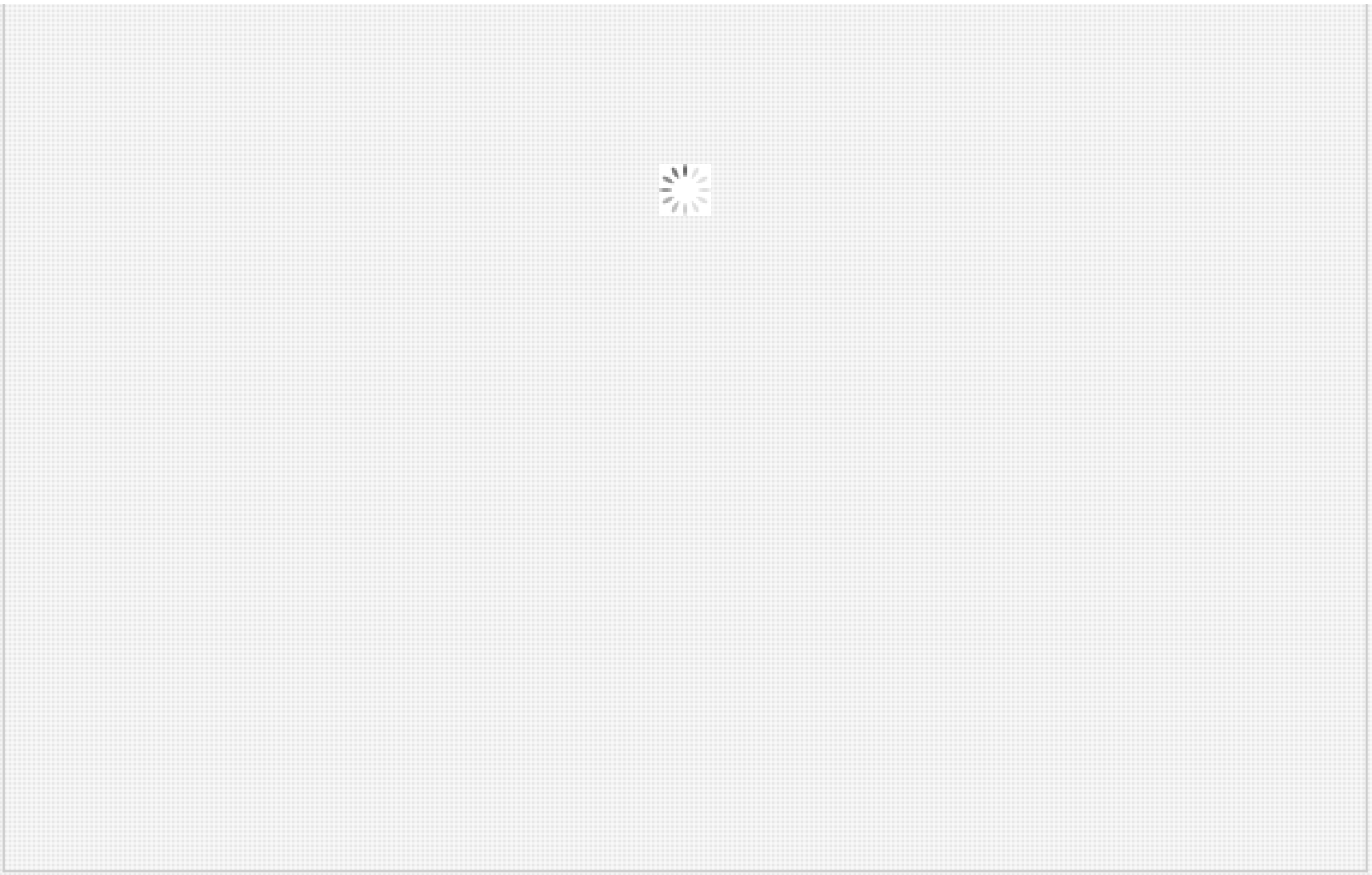
size. 48 still has the lowest normalized average Hamming Distance, but remember, the script has a bias towards smaller keys, especially when those keys are also the MFOGCD in the list of the top n ProbableKeySizes.

Our key is obviously not 100% correct, but again there may be enough of the correct value there that you can make some guesses and use the XOR-Decrypt.ps1 script to try them out until you have the correct key.

What happens if you run the script again and pass -MaxKeySize as 16, unfortunately, if you do that, the most probable key size becomes two because that is the MFOGCD and that yields even more inaccurate results.

What's the practical use for this script? There is some malware that uses repeating XOR key encryption because it's easy to implement. You may be able to apply it in your analysis. Does it work against code?

Here's an example:



Click to enlarge

Crack-XORRepeatingKeyCrypto.ps1's accuracy could be improved at a performance cost. If you look at XOR-Decrypt.ps1, you'll see that it has multiple scoring functions, a sort of mini-neural network. One for English letter frequency, one for English bigram frequency and one for English trigram frequency. All of the values returned from those functions factor into determining the most probable key for XOR-Decrypt.ps1, but because of the way Crack-XORRepeatingKeyCrypto.ps1 works, determining the most

probable byte for each position of the key against the transposed blocks, bigrams and trigrams can't be used because we're not looking at the bytes in context, so bigrams and trigrams won't follow their natural frequency of occurrence.

Crack-XORRepeatingKeyCrypto.ps1 could factor in bigram and trigram scores when it moves to the stage of decrypting the original ciphertext, using what it has calculated as the most probable key, but that would require maintaining a list of the top n most probable keys and trying each of them, scoring them and choosing the one with the best score.

Maybe if time permits, I'll trying making this change, but I'm afraid of what it may do to the run time.

Lastly, the script includes a .SYNOPSIS section with help and examples, like any PowerShell script should. So do a `Get-Help -Full .\Crack-XORRepeatingKeyCrypto.ps1 | more` for additional information.

Posted 11th July 2015 by davehull

0 Add a comment

JUN
7

XOR'd play: Normalized Hamming Distance

I've been playing around with the matasano crypto challenges for my own edification. Let me say up front, I'm a noob when it comes to crypto. I've used gpg, pgp, OpenSSL, etc. as a consumer of crypto products for a long time, but I've never really peeled back the onion and honestly, I'm not deep enough into Matasano's challenges to really have increased my understanding of modern crypto much at all.

My point is, I'm learning and I may say some things here out of ignorance. Take that as a disclaimer.

Also, this post is going to be ridiculously long. TLDR; In my testing, normalized Hamming Distance alone accurately returned the correct repeating XOR key size in about 40% of cases, but there's a simple calculation you can add that can increase this to over 90%.

In the challenges, Matasano introduces Hamming Distance, something I'd read about previously, but only in passing. Hamming Distance is the measure of difference between two strings. For example, the Hamming Distance between "fuse" and "fuel" is two because two characters would have to be changed to make the two strings the same.

For the crypto challenges, Hamming Distance is calculated at the bit level rather than at the character level as above, so to get the Hamming Distance bit-by-bit, we convert the strings to a byte arrays, then convert each byte to bits, then count the number of differences:

```
PS> GetByteArray "fuse" | ForEach-Object { GetBits $_ }
01100110
01110101
01110011
01100101
PS> GetByteArray "fuel" | ForEach-Object { GetBits $_ }
01100110
01110101
01100101
01101100
```

Obviously there's no difference in the bits for the first two bytes. Let's stack the bit strings for the last two bytes one after the other, it'll make counting the differences easier. I've highlighted the columns

with differences:

```
01110011 s  01100101 e
01100101 e  01101100 l
```

In total then, there are five different bits between the two strings. The Hamming Distance then is five. Perceptive readers may notice that if the bytes above are XOR'd against one another, the result is that five bits are set to 1:

```
    01110011 s      01100101 e
XOR 01100101 e  XOR 01101100 l
-------------   -------------
    00010110        00001001
```

So we can use XOR for each pair of bytes, count the 1 bits and the result is the Hamming Distance. Incidentally, Hamming Distance, when calculated at the bit level as above, is equivalent to calculating the Index of Coincidence, which is useful to cryptographers attempting to crack polyalphabetic substitution ciphers such as Vigenère cipher or repeating XOR key encryption, which is what the initial set of challenges are built around.

Repeating XOR key encryption is a simple encryption scheme and not one that is terribly secure. Here's an example of how it works. Let's take our "fuse fuel" and XOR it with the key "few."

Converting our strings to bytes:

```
102:117:115:101:032:102:117:101:108  f:u:s:e: :f:u:e:l
102:101:119  f:e:w
```

In repeating XOR, since our key is shorter than our plaintext, we repeat the key, so the bytes line up as

follows:

```
102:117:115:101:032:102:117:101:108  f:u:s:e: :f:u:e:l
102:101:119:102:101:119:102:101:119  f:e:w:f:e:w:f:e:w
```

Now we XOR the values together and the result in bytes is:

```
000:016:004:003:069:017:019:000:027
```

Notice wherever the same bytes lined up with each other above, they cancel each other out, this is also true for the bits, to see this in action, consider the second byte pair:

```
117 in bits 01110101          01110101
101 in bits 01100101  XOR'd: 01100101
                             --------
                             00010000 in decimal 016 as above.
```

The challenge says that Normalized Hamming Distance can be used to calculate probable XOR key size. An algorithm is given, but there's a little ambiguity in the directions, so you still have to figure out the details. Essentially, you take the first n bytes of the ciphertext and calculate the Hamming Distance of those bytes against the next n bytes and divide the result by the key size. Whichever size yields the smallest Normalized Hamming Distance is probably the key size.

Let's walk through this, but for the sake of demonstration, we'll modify our plaintext to the following:

```
"fuse fuel for falling flocks"
```

Yes, it's a silly string, but I think it will make explaining things easier due to it's length. XOR encrypted with our key, we get the following byte array:

```
000:016:004:003:069:017:019:000:027:070:003:024:020:069:017:007:009:027:015:011:0
16:070:003:027:009:006:028:021
```

If we apply Matasano's algorithm, it's not actually their algorithm, but I'm calling it that here because lazy, we read a couple bytes of our ciphertext, then the next two bytes and get the Hamming Distance. We can repeat this up to the end of the ciphertext and average all the values together, divide by the number of bytes (two initially), that's the "normalization" part. We repeat this for three bytes, then four, then five and so on, up to 14 bytes for this string, since it's length is 28 bytes and we need to difference two strings of bytes.

Let's look at a few examples, I'm going to refer the Hamming Distance as HD, Average Hamming Distance as AvgHD and Normalized Average Hamming Distance as NAvgHD. Let's do some math:

```
   000:016    004:003    069:017    019:000    027:070    003:024
HD 004:003 HD 069:017 HD 019:000 HD 027:070 HD 003:024 HD 020:069
---------- ---------- ---------- ---------- ---------- ----------
         4          4          6          4          7          9
```

If you doubt any of the above is correct, convert the bytes to bits, XOR them against each other and count the 1s:

```
    00000000:00010000 (000:016)
XOR 00000100:00000011 (004:003)
------------------------------
    00000100:00010011 (four ones) for an HD of 4
```

If we keep going like this across the entire 28 byte string, we'll calculate the HD for 14 pairs of bytes, sum up the 14 values and get an AvgHD of 5.61538461538462, divide that by two, because we were

using two byte pairs and you get a NAvgHD of 2.80769230769231.

Now we repeat this, but we do it for byte pairs of three bytes each:

```
   000:016:004    003:069:017    019:000:027    003:024:020
HD 003:069:017 HD 019:000:027 HD 003:024:020 HD 069:017:007
-------------- -------------- -------------- --------------
             9              6              7              8
```

Again all we're doing is XORing together the corresponding pairs, then counting the 1s in the results. If we do this for the first 13 three byte pairs, add up the results and divide by the number of triplet pairs we'll get an AvgHD of 7.625, which is actually slightly higher than the AvgHD we had with our two byte pairs, but we're interested in he NAvgHD, so we divide that number by three, the length of our byte pairs and we get a NAvgHD of 2.54166666666667, which is lower than our NAvgHD for a two byte key.

We can repeat this process for byte pairs up to a length of 14 bytes, the first half of the ciphertext against the second half. The results look like this:

```
CalcKeySize            AvgHD           NAvgHD
-----------            -----           ------
          2 5.76923076923077 2.88461538461538
          3            7.625 2.54166666666667
          4 12.3333333333333 3.08333333333333
          5            13.25            2.65
          6 14.6666666666667 2.44444444444444
          7 19.6666666666667 2.80952380952381
          8             22.5           2.8125
          9               20 2.22222222222222
         10               26              2.6
```

11	24	2.18181818181818
12	26	2.16666666666667
13	34	2.61538461538462
14	38	2.71428571428571

Sorted by NAvgHD, because the byte pair size with the smallest NAvgHD is probably our key size shows:

CalcKeySize	AvgHD	NAvgHD
12	26	2.16666666666667
11	24	2.18181818181818
9	20	2.22222222222222
6	14.6666666666667	2.44444444444444
3	7.625	2.54166666666667
10	26	2.6
13	34	2.61538461538462
5	13.25	2.65
14	38	2.71428571428571
7	19.6666666666667	2.80952380952381
8	22.5	2.8125
2	5.76923076923077	2.88461538461538
4	12.3333333333333	3.08333333333333

But our key was "few," that's three bytes, why isn't it at the top of the list? Matasano's instructions do state the value with the lowest NAvgHD is probably the key size, they don't say it will be the key size. In their example, the value with the lowest NAvgHD is the key size, but in my testing, the NAvgHD is not always the key size, as you can clearly see above.

Let's dive into this a bit more.

The HDs for byte pairs of nine bytes:

```
   000:016:004:003:069:017:019:000:027
HD 070:003:024:020:069:017:007:009:027
---------------------------------------
                                     17

   070:003:024:020:069:017:007:009:027
HD 015:011:016:070:003:027:009:006:028
---------------------------------------
                                     23

 AvgHD: (17 + 23) / 2 = 20
NAvgHD:         20 / 9 =  2.22222222222222
```

If you compare the AvgHD for smaller key sizes with the AvgHD for the larger key sizes, you'll see the smaller ones have lower AvgHDs, this makes sense if you think about the process of comparing sets of larger numbers with sets of smaller numbers, you're more likely to have larger differences as the upper bound on your numbers increases -- consider the possible differences between numbers 01 and 09 and between 001 and 999.

What really throws things off is the normalization, when we divide AvgHD by the key size, but you may have noticed that most of the values with lower NAvgHDs are multiples of the actual key size. In my testing across 20+ different plaintext samples, this trend appears again and again when calculating Hamming Distance to find the repeating XOR key size.

Matasano's instructions say, "the KEYSIZE with the smallest normalized edit distance is probably the key." Edit distance is another way of saying Hamming Distance, apparently. They go on to say, "You

could proceed perhaps with the smallest 2-3 KEYSIZE values. Or take 4 KEYSIZE blocks instead of 2 and average the distances."

In their example, the key size with the smallest normalized HD is in fact the key, I thought maybe that was because the maximum key size they ask you to try is 40 bytes and the correct key size is not something that divides evenly into 40, but even increasing the maximum key size to a couple multiples over the correct key size does not cause a larger key size to rise to the top, but again, in my tests over 20+ different sample plaintexts encrypted thousands of times with random keys from 2 to 100 bytes, that's the exception.

For my testing, admittedly limited given the infinite number of possible plaintext messages, I took 20 different samples of text from various sources, blogs, books, both technical and fiction -- most ranging from a paragraph to a page in length, sorry I won't be more precise, again, lazy. I ran these texts through a script that would generate a random key from 2 to 100 bytes in length, random bytes in the range 0x00 through 0xFF, the encrypted text was then passed through the HD algorithm as above and since I knew the actual key size, I set an upper bound on the key sizes I would try to be three times the actual key size. I selected the top five values in terms of lowest NAvgHD for each ciphertext. I ran this test 7758 times.

In 38% of the cases, the key size with the lowest NAvgHD was the correct key size.
In 33% of the cases, the key size with the second lowest NAvgHD was the correct key size.
In 28% of the cases, the key size with the third lowest NAvgHD was the correct key size.

During my experimentation, the correct key size landed outside of the top three values a few times, but very rarely and I was still tweaking the algorithm to get it correct as I made a fence post error that took me some time to spot and resolve. I'm not counting those values in my final tally. The numbers above are from the last tests I ran after getting all bugs worked out of the algorithm, but your mileage may vary.

Still, I found the above numbers discouraging. If I'm going to write a tool for brute force decrypting XOR

encrypted data and the first step in the process is to get the correct key size, I want the algorithm to be more reliable than ~40%. I could do what Matasano suggests and simply try the first three values, but I suspected work could be done to improve this.

I believe I was right, but again, my sample size is small, but the results across that sample size are promising and the solution was simple, so simple that you've probably already thought of it. I'm sure many others have, but I found no references to it, possibly because search engines fail, or I fail at using them, or because my solution is bogus and just happened to work for my test cases and methodology.

Here's what I did, I saved the top five results -- the five most probable XOR key sizes based on NAvgHD -- then calculated the greatest common denominator (GCD) for the first two key sizes, then the first and third key sizes, then the second and third key sizes. The results were assigned to variables, GCD12, GCD13 and GCD23.

If the following conditions are met, GCD12 is returned as the most probable key size:

1. GCD12 is not one and
2. GCD12 is a key size in the top n key sizes and
    1. GCD12 is equal to GCD13 and GCD23 or
    2. GCD12 is equal to either the first or second most probable key size
    3. If the above conditions are not met, the script returns the key size with the smallest NAvgHD as the most probable key size.

How well does this algorithm work? In the testing above, all 7758 cases, I applied this algorithm and the correct key size was returned as the most probable key size 100% of the time.

Recall that in my test script, I'm calculating HDs for all key sizes up to three times the size of the actual key. This does impact the outcome of the script. As a test of this, I changed the script to calculate HDs

for all combinations up to four times the actual key size. The results came out as follows across 2574 test runs:

The value with the best NAvgHD was the correct key size 893 times or 34% of the time.
The value with the second best NAvgHD was the correct key size 714 times or 27% of the time.
The value with the third best NAvgHD was the correct key size 537 times or 20% of the time and the value with the fourth best NAvgHD was the correct key size 430 times or 16% of the time.

All percentages have been rounded down, stop nitpicking my numbers.

As you can see increasing the trial key sizes by one more multiple of the actual key size causes the HD algorithm to be come less accurate and unfortunately in real-world cases, we have no knowledge of the actual key size.

But how did my algorithm enhancements fair in this test run? Out of the 2574 test runs, my script selected the correct key size 2108 times or 81% of the time, not bad, but I believed I could do better. I also wanted to add more sample texts. So I did.

I downloaded a bunch of English language books from http://www.gutenberg.org/ and refactored my plaintext function to return random samples from these texts. I did go through each of the files and remove the Project Gutenberg licensing information. I also removed some books that were framed in boxes made up of ASCII characters like pipes, dashes and underlines. I initially ran with these books in, but my numbers were skewed, so I investigated and took them out. I also removed duplicate texts and I only left in a few of the books from the Bible as I thought they may have an undo influence on my results. In the end, I had 80 books total, comprised of 353422 non-blank lines, 3646553 words and 20415200 characters.

The average input to my XOR encryption was 196 words comprised of 1160 characters.

For expediency, I reduced the maximum encryption key size to 40 bytes on my initial run. The minimum key size was 2 bytes. The key sizes to be tried was set to 4 times the actual key size and I opted to save the top six most probable key sizes for further review.

I did make a significant change to the script. In the previous testing, I'd calculated the GCDs for the first three most probable key sizes. I modified the script to calculate the GCDs for every combination of the top n most probable keys. Since I was extracting the top six keys in this run, I was calculating the GCDs for those six key sizes in every possible combination. I also added a parameter for MaxNAvgHD -- the highest NAvgHD I would accept for the probable key size in cases where it could not be clearly determined, the default is 3.5.

For evaluating the top six (in this case) most probable keys, I put the data through these steps:

1. Calculate the GCDs across every pair of values in the top n values and add them to a hashtable with a count of frequency of occurrence. Each time a given GCD appears, increment its counter in the hashtable.
2. Get the most frequently occurring GCD (MFGCD) from the hashtable.
3. If the MFGCD is in the list of the top n most probable key sizes and its NAvgHD is less than the MaxNAvgHD, then return MFGCD as the most Probable Key Size.

If the conditions above are met, the likelihood that MFGCD is the correct key size is high. I haven't done the analysis to see what percentage of cases this is true for, but that analysis could be done. If those conditions are not met, do the following:

1. Take the smaller of the first two most probable key sizes as a possible key size.
2. Take the most probable key size with the smallest NAvgHD as a possible key size.
3. If the values from 1 and 2 are the same, return that value as the most probable key size.
4. If the values from 1 and 2 are not the same, if the first value has a NAvgHD below the MaxNAvgHD, return it as the most probable key size, else return value 2.

This last set of conditions gives more weight to the smaller of the top two values in the list of probable key sizes, but there's no guarantee that the smaller value is more likely to be the correct key size. However, the next step after attempting to calculate the correct key size is to break the encryption via brute force and it's going to be less expensive to try smaller keys than larger ones, so favoring the smaller key size when you're less certain as to the correctness of either key seems like the right thing to do.

Given these changes in my script, what was the outcome? In this test run there were 3897 samples encrypted with random keys from 2 to 40 bytes.

The script returned the correct key size as the most probable key size 97% of the time.
The key size with the lowest NAvgHD was the correct key size 29% of the time.

Out of the 114 cases where the returned probable key size was not the actual key size, there were 79 cases where the top six most probable key sizes didn't even contain the correct key size. I found this to be the most troubling outcome of my testing, but diving into the data deeper, there was an explanation for 97% of these failures.

In 77 of these 79 cases, the key was equal to or longer than half the length of the plaintext, meaning the encryption key was not a repeating XOR key at all, some portion of the key may have repeated, but not the entire key. In fact, in 16 cases, the key was longer than the plaintext entirely, meaning the encryption key was effectively a one-time pad.

In the other two cases, I'm not sure what went wrong. In one, the key size was 14 bytes and the plaintext length was 35 bytes, the value with the lowest NAvgHD was seven and that was the most probable key size as returned by the script. It is a factor of 14, but that's little consolation. According to the output, 12, 11 and 17 all had lower NAvgHDs than 14. In the other failed case, the actual key size was 2 bytes and the plaintext was 2457 bytes in length. The script showed that a key size of three bytes had the lowest NAvgHD with 4, 7, 8, 6 and 5 rounding out the list. Clearly the most common GCD here would be two, but alas two was not in the list of most probable keys and so could not be returned as the most

probable key size.

What of the other 35 cases where the correct key size was in the list of top six probable key sizes, but was not the most probable key size returned by the script? The results came out like this:

```
ActualKeySize ProbableKeySize Top6KeySizes
------------- --------------- ------------
18            4               18:4:17:22:14:32
34            31              34:31:35:30:29:6
37            35              37:35:39:17:24:22
8             6               8:6:4:3:7:5
24            14              24:14:22:10:12:18
40            28              40:28:32:35:14:56
18            6               18:6:14:22:17:27
25            10              50:25:40:11:10:29
29            25              29:25:33:8:30:12
22            19              22:19:32:36:33:12
18            9               36:18:9:27:3:45
37            18              37:18:33:39:35:4
19            15              19:15:23:32:4:28
28            25              28:25:35:19:33:17
35            25              35:25:33:15:11:31
12            6               24:12:6:30:18:25
28            25              28:25:9:32:34:17
15            6               15:6:12:18:13:16
32            23              32:23:9:54:27:41
24            21              24:21:27:35:26:29
28            18              28:18:10:19:12:21
```

```
6               3               6:3:9:8:4:5
25              11              25:11:36:16:14:9
18              15              18:15:28:25:24:33
26              15              26:15:11:8:19:16
28              22              28:22:12:7:25:13
21              10              21:10:23:31:27:16
23              16              23:16:29:31:24:15
24              22              24:22:14:11:10:30
21              6               42:21:48:55:6:15
15              6               15:6:18:9:12:17
29              28              29:28:12:14:19:23
15              12              15:12:10:4:11:2
21              3               21:3:18:9:24:6
12              4               12:24:4:8:16:20
```

Again, if the scripts most frequently occurring GCD calculations don't point out a clear winner, it puts more weight on smaller probable key sizes that having NAvgHD value below a user supplied threshold with 3.5 being the default value. This could probably use refining, but again, the script has a much higher probability of returning the correct key size (around 97% in my testing) than NAvgHD alone (around 40% or less in my testing).

It would be easy to have the script return multiple ProbableKeySizes for cases where it can't clearly pick a winner and have that list sorted by key size in ascending order before entering the brute-force decryption phase of its work.

As I said at the start of this post, I'm a crypto-noob and that's an insult to noobs everywhere. My methods may be fraught with errors and there are lots of smart people in the world who may skim this and laugh, but I wanted to share my analysis and the results because I found it interesting and even fun.

My code for this is written in PowerShell and is available at https://github.com/davehull/MCC/blob/master/sets/1/XOR-Test.ps1. For testing, I do something like:

```
1..100 | Foreach-Object { .\XOR-Test.ps1 -MinKeySize 2 -MaxKeySize 60 -top 5 -MaxNAvgHD 3.4 | Export-Csv -NoTypeInformation .\output_$_.csv }
```

The above will run the script 99 times. It will select some portion of text from text files in .\texts\, sold separately. That text will be encrypted with a random byte key of two bytes on the first pass, that ciphertext string will be passed to the code that attempts to calculate the key size, then the next iteration begins, selecting a new text value, picking a random three byte key and so on, saving the output to CSV files that you can analyze later using Excel or PowerShell.

Update: Running the same analysis again for 60 byte max key size, discarding the cases where the actual key size could not repeat because it was more than half the size of the plaintext, the script returned the correct key size as the most probable key size 99.5% of the time. By comparison, the key size with lowest NAvgHD was the correct key size 31% of the time.

YMMV.

Posted 7th June 2015 by davehull

MAR
30

Kansa: Get-LogparserStack.ps1

Kansa is an incident response framework written in PowerShell, useful for data collection and analysis. Most of the analysis capabilities in Kansa require Logparser, which is a very handy tool for creating SQL-like queries over data sets that may be comprised of a single file or many files.

Because adversaries usually want to leave a small footprint, one technique for finding them is frequency analysis -- looking for outliers across many systems. This technique has been written about before. As such, most of the analysis tools in Kansa are scripts that stack-rank or perform frequency analysis of specific fields in a given data set. Some examples include:

- Get-ASEPImagePathMD5Stack.ps1
- Get-ASEPImagePathLaunchStringMD5UnsignedStack.ps1
- Get-ASEPImagePathLaunchStringMD5UnsignedTimeStack.ps1
- ...

And the list goes on. These script names are fairly descriptive, but they are a mouthful and they are not very flexible as they contain hardcoded Logparser queries with set field names.

Kansa needed a more flexible stack-ranking solution and now it has one.

Get-LogparserStack.ps1 can be used to perform frequency analysis against any delimited file or set of files, so long as the set all has the same schema and the same header row across each file. Unlike all other Kansa utilities, Get-LogparserStack.ps1 is interactive. After reading the first two lines of each input file and confirming that they all have the same header row, the script prompts the user for the field she wishes to pass to Logparser's COUNT() function, then the script prompts the user for the fields she wishes to GROUP BY.

Below is a screen shot of the script in action against a small set of Autorunsc data from five systems. The frequency analysis is against the "Image Path" field with both "Image Path" and MD5 being added to the GROUP BY clause. As you can see in the screen shot, the resulting query quickly bubbles up an

outlier, a dll from one system does not match the same dll from the other four systems.

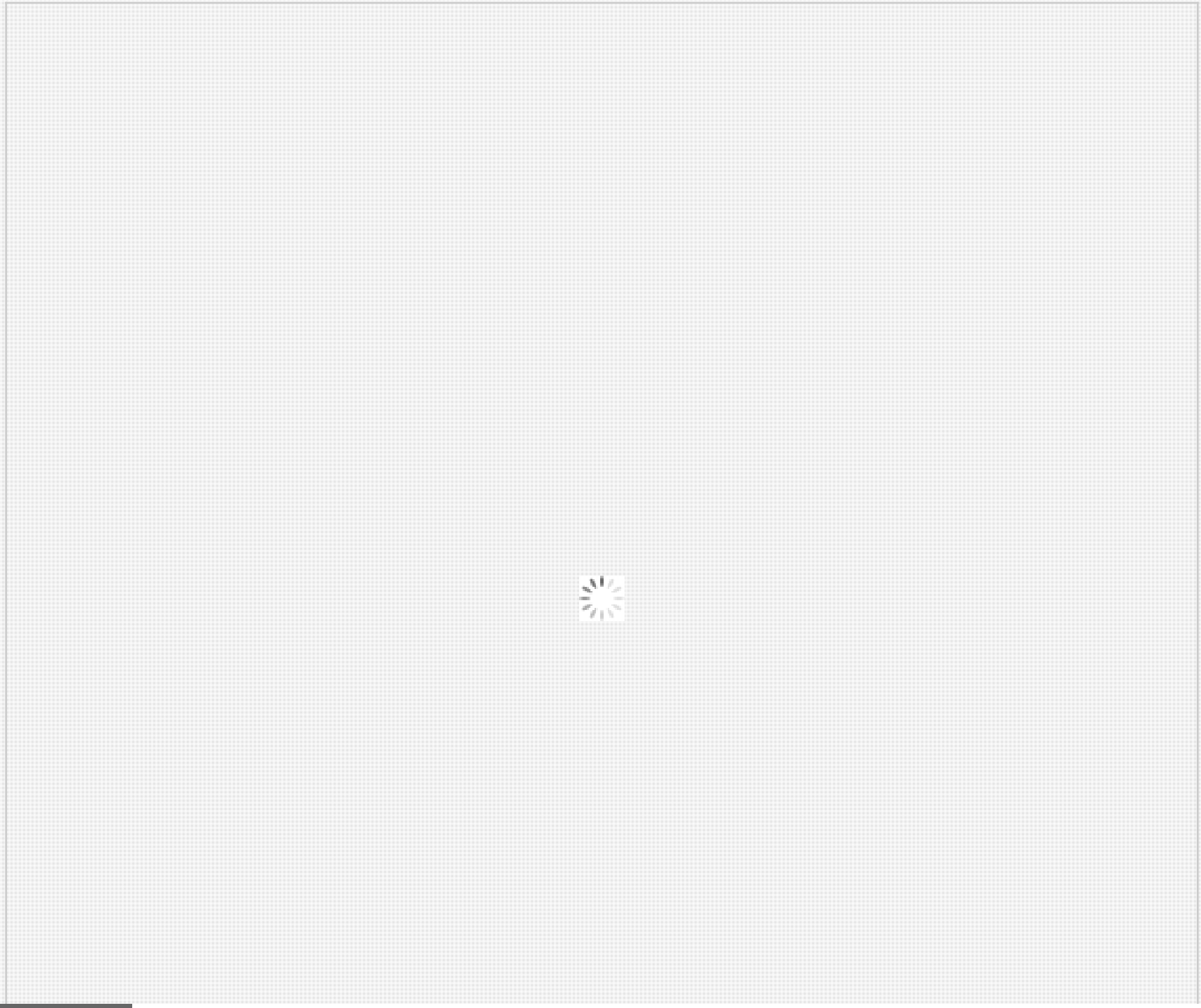


Figure 1: Get-Logparser.ps1 quickly shows that one dll is not like the others.

Get-LogparserStack.ps1 is a new utility and as such, may mature a bit in time. One potential feature would be to make it non-interactive, so it can be scripted.

As with nearly all of the scripts that make up Kansa, Get-LogparserStack.ps1 can be used in conjunction with Logparser.exe outside the framework to perform frequency analysis of any data set, providing the schemas match and each file in the set has a header row.

If you use it and encounter any bugs, please open an issue in Kansa's GitHub page.

Posted 30th March 2015 by davehull

Labels: autoruns, frequency analysis, Kansa, least frequent occurrence, Logparser, stack ranking, stacking

0 Add a comment

MAR

Kansa: Get-AutorunscDeep.ps1 -- Taking Autorunsc to 11

25

I wanted to put up a quick post about a new Kansa collector I recently added -- Get-AutorunscDeep.ps1. Sysinternals' Autoruns is a great utility for finding auto-start extension points in Windows and one I've blogged about a number of times.

Kansa has had a collector that wraps around Autorunsc.exe from Sysinternals almost since I began writing Kansa in March of 2014. It's such a great little utility for enumerating so many persistence mechanisms in Windows, though by no means, does it cover all of them.

Get-AutorunscDeep.ps1 goes to 11. How so? There are two ways that Get-AutorunscDeep.ps1 improves on Autorunsc.exe alone.

1. Get-AutorunscDeep.ps1 includes code written by my friend @z4ns4tsu (who will be speaking at the SANS 2015 DFIR Summit) that calculates the Shannon Entropy of Autorunsc's Image Path property. As many of you well know, packed binaries are common in malware families and those binaries have higher entropy than many legit binaries, so knowing a file's entropy can be a useful lead generation tool when dealing with large amounts of data.
2. Get-AutorunscDeep.ps1 goes a step further than Autorunsc.exe alone for common interpreters that execute scripts such as cmd.exe, PowerShell.exe or wscript.exe that may call .bat, .ps1 or .vbs files repsectively. Below in Figure 1, I've run the latest version of Autorunsc.exe and saved the output to a csv file, then loaded that csv file into a PowerShell variable called $data, then I'm dumping the contents of $data for any entry that calls a PowerShell script:

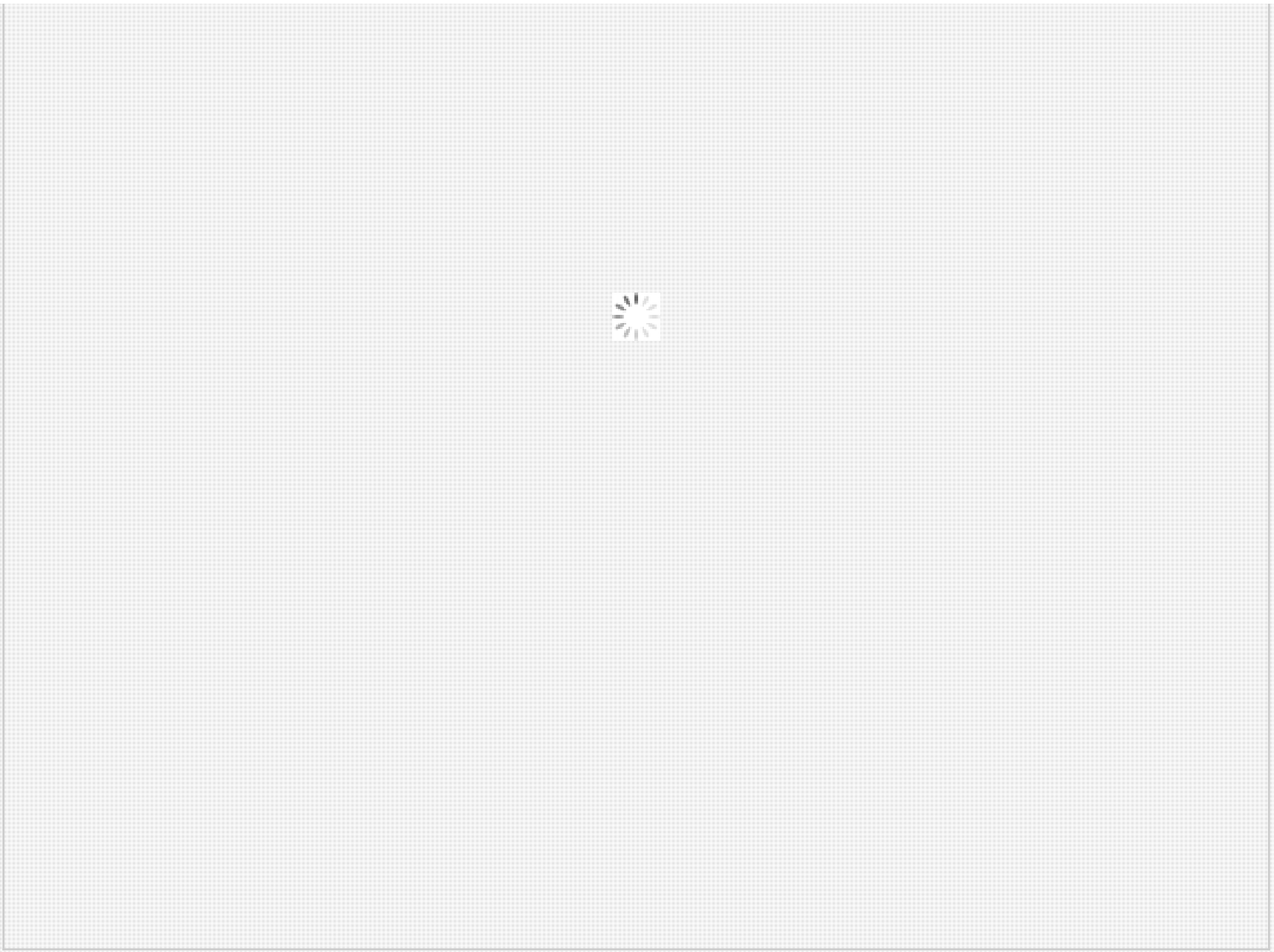


Figure 1: Output of Autorunsc.exe for a PowerShell ASEP (Click to enlarge)

Note that in Figure 1, Autorunsc.exe provides hashes of the PowerShell.exe binary itself. I've worked investigations where adversaries have taken existing ASEP entries and modified the scripts that are

called, planting their own malicious code in those scripts. If you've got an ASEP that runs via PowerShell, cmd.exe or wscript.exe on hundreds or thousands of hosts and you use Autorunsc.exe to collect that data, a few of the scripts called by that interpreter could have been modified by an attacker and you'd have no visibility into it via Autorunsc.exe alone.

Get-AutorunscDeep.ps1 solves this problem, in most cases, by adding a hash of the script called by the interpreter. So for ASEPs like cmd.exe, PowerShell.exe and wscript.exe that call scripts, you'll see the hash of the binary itself as Autorunsc.exe calculates it, and you'll get the hash of the script because Get-AutorunscDeep.ps1 will calculate it for you, assuming it can find and read the script. Below in Figure 2, is an example of what this looks like, the data was collected from the same host as above, but remotely using Kansa:

Figure 2: Output of Get-AutorunscDeep.ps1 for the same PowerShell ASEP (Click to enlarge)

Note the red marks in Figure 2 and apologies, I'm not a graphic artist. Get-AutorunscDeep.ps1 has added an MD5 hash for the Get-AutorunscDeep.ps1 script that the PowerShell executable in the above Scheduled Task is calling. You can also see the ShannonEntropy value for PowerShell.exe, another feature of Get-AutorunscDeep.ps1's output. Now consider the visibility this gives you if you have the same ASEP script deployed across thousands of hosts. Use Kansa's Get-AutorunscDeep.ps1 collector, in conjunction with Sysinternal's Autorunsc.exe, and you'll quickly be able to find versions of scripts that have been modified.

Posted 25th March 2015 by davehull

Labels: asep, autoruns, dfir, entropy, Get-AutorunscDeep.ps1, IR, Kansa, persistence

JUL
14

Kansa: Passing arguments to collector modules

In my previous post on Kansa's automated analysis, I mentioned there was another improvement I made to the framework that I would cover in a future post. I thought at that time, that Kansa was at a point where I could go into some details about the new feature, but as it turns out, it wasn't quite ready.

Previously, some of Kansa's collector modules would need to be edited or customized prior to being run. Disk\Get-File.ps1, for example, was one that could acquire a specific file from target machines, but users would have to edit the collector to specify the file they wanted to acquire. Obviously that was less than ideal, so I did some work that would allow users to specify those kinds of things via command line arguments. In my limited testing, this worked... but, my testing was limited.

This week I had a pull request submitted by @z4ns4tsu for a collector module called Get-FilesByHash.ps1 that would allow investigators to take a cryptographic hash (MD5, SHA1, etc.) of a known suspect file, then search for files with that same hash across many machines in the environment. The module was the first that would take multiple arguments, the search path, the hash and the hash type; this is where Kansa had an issue. It couldn't pass multiple arguments to collectors, but after a couple nights of work, now it can.

I also added a few arguments to Get-FilesByHash.ps1, including a file extension regex so the script doesn't hash every single file looking for matches, instead, it will only hash those files with extensions that match the provided file extension regex, the default regex is \.(dll|exe|ps1|sys)$, this greatly

reduces the number of files that will be hashed. I also added two more arguments that limit the files that will be hashed based on minimum and maximum file size in bytes.

Here's a command line example showing how this module can be used:

```
.\Kansa.ps1 -ModulePath ".\Modules\Disk\Get-FilesByHash.ps1
BF93A2F9901E9B3DFCA8A7982F4A9868,MD5,C:\Windows\System32,\.exe$" -target
localhost -Verbose
```

Above Get-FilesByHash.ps1 will search for any files with the MD5 hash of BF93A2F9901E9B3DFCA8A7982F4A9868, in or below the C:\Windows\System32 path and ending with an extension of .exe., Notice that the arguments to Get-FilesByHash.ps1 are not named parameters. Named parameters are not supported for remoting, so they must be positional also note that they are comma separated and the whole module and arguments are quoted.

As with other modules, you can use the .\Modules\Modules.conf file to pass arguments to Get-FilesByHash.ps1 (or any other module that takes arguments) via the conf file itself. Here's the entry for the module above taken from the conf file:

```
Disk\Get-FilesByHash.ps1 BF93A2F9901E9B3DFCA8A7982F4A9868,MD5,C:\Windows\System32
```

Note the absence of quotes in the configuration file, and I've also omitted the regex extension argument.

Adding the ability to pass parameters to modules meant I could remove several collectors from Kansa that were written to acquire specific event logs, each one collecting a specific log file, instead, now Kansa has one collector written to generically collect any Windows event log and the specific log is simply passed as an argument. Here's the relevant section of the .\Modules\Modules.conf file:

```
Log\Get-LogWinEvent.ps1 Security
Log\Get-LogWinEvent.ps1 Microsoft-Windows-Application-Experience/Program-
Inventory
Log\Get-LogWinEvent.ps1 Microsoft-Windows-Application-Experience/Program-
Telemetry
Log\Get-LogWinEvent.ps1 Microsoft-Windows-AppLocker/EXE and DLL
Log\Get-LogWinEvent.ps1 Microsoft-Windows-AppLocker/MSI and Script
Log\Get-LogWinEvent.ps1 Microsoft-Windows-AppLocker/Packaged app-Deployment
Log\Get-LogWinEvent.ps1 Microsoft-Windows-Shell-Core/Operational
Log\Get-LogWinEvent.ps1 Microsoft-Windows-TerminalServices-
LocalSessionManager/Operational
Log\Get-LogWinEvent.ps1 Microsoft-Windows-TerminalServices-
RemoteConnectionManager/Operational
```

Above we have a single collector, Log\Get-LogWinEvent.ps1, that replaced nine collectors because it accepts an argument specifying which log to collect.

As you can see, being able to pass command line arguments to collectors is a big benefit, just be mindful that they are positional, not named parameters and as a result, if you want to accept all the default arguments but the last one, you still have to specify every argument, supplying the default values for every argument except the one you want to modify.

You can find more information about Kansa and the latest release at https://github.com/davehull/Kansa/releases.

Posted 14th July 2014 by davehull

Labels: Kansa, Kansa collector command line arguments, Kansa collectors

0 Add a comment

JUL
4

Kansa: Automating Analysis

Kansa, the PowerShell based incident response framework, was written from the start to automate acquisition of data from thousands of hosts, but a mountain of collected data is not worth bits without analysis, thus analysis has been part of the framework from almost the beginning as may be seen here in this commit from 2014-04-18.

Data collection has been configurable via the Modules.conf text file since the beginning and the project has been packaged with a default Modules.conf file with the order of volatility applied. Users could edit the file, comment and uncomment lines to disable and enable modules, customizing data collection.

After Kansa completed its collection, users could cd into the newly created output directory and then into the specific module directory and run the analysis script of their choosing to derive some intelligence from the collected data. For example, a user might run Kansa's Get-Autorunsc.ps1 collector to gather Autoruns data from a hundred hosts that should have identical or very similar configurations.

Following the data collection, they could cd into the new output directory's Autorunsc subdirectory, then run

```
Get-ASEPImagePathLaunchStringMD5Stack.ps1
```

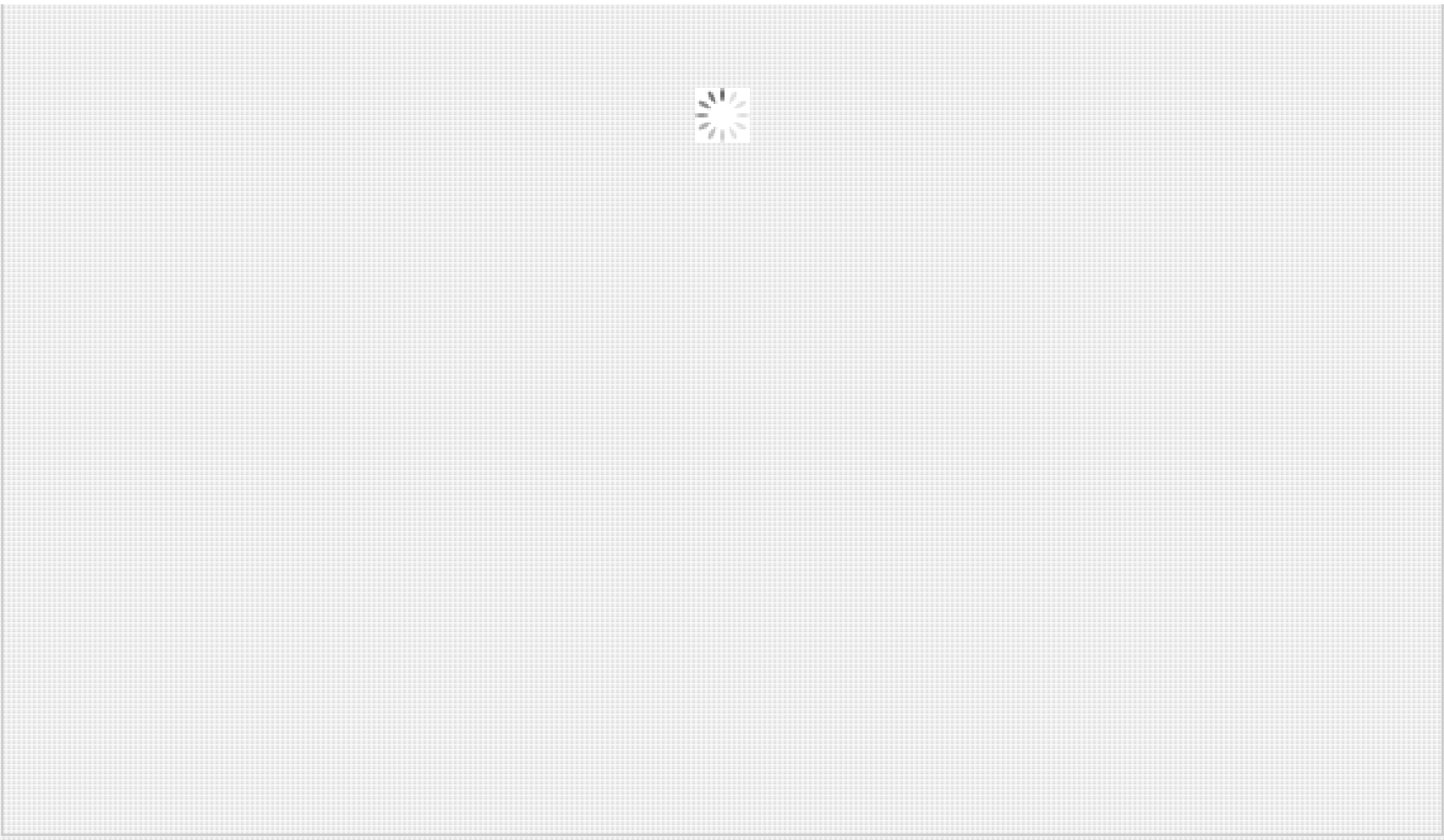
which would return a listing of Autoruns aggregated by path to executable, command line arguments

and MD5 hash of the executable or script all in ascending order, so any outliers would be at the top of the list and these entries may warrant further investigation.

This was all well and good, but with more than 30 analysis scripts, analysis of the collected data was becoming cumbersome. It was begging for automation. So, I added it.

There is now an Analysis.conf file in the Analysis folder that works in much the same way as the Modules\Modules.conf configuration file. Every analysis script has an entry in the configuration file and you can edit the file and comment out the scripts you don't want to run or uncomment the ones you want to run. Then when you run Kansa, simply add the -Analysis flag and after all the data is collected, Kansa will run each analysis script for you and save the output to a new folder under the time stamped output folder called AnalysisReports.

Below is a sample listing:



Click for original size image

In the top directory listing of the output directory, you can see the normal output file structure, one folder per module, this was obviously a very limited data collection with Autorunsc, File, Netstat and PrefetchListing modules being used. Error.Log contains information about errors encountered during runtime. What's new here is the AnalysisReports directory.

The bottom directory listing shows the contents of the AnalysisReports path. Each of these are TSV files containing summary data of the collected data with the file names reflecting the name of the script that produced the data set. And the beauty of this is, it's fully automated when you run Kansa with the -

Analysis flag and you've configured the Analysis\Analysis.conf file.

I've made some other improvements to Kansa in the last couple weeks, but I'll save that for the next post. For now, I wanted to share the automated analysis piece. I'm pretty psyched about it because it's a big time-saver and it puts Kansa in a position where it can easily produce alerts based on the presence of a quality or quantity on which an analysis script is written to trigger.

Posted 4th July 2014 by davehull

Labels: analysis, automated analysis, automation, Kansa

JUN
18

Kansa: Get-LogUserAssist.ps1

Tonight I pushed the latest collector to Kansa, Get-LogUserAssist.ps1. This is probably the most complicated collector I've written for Kansa. It has several moving parts and there were some obstacles to overcome.

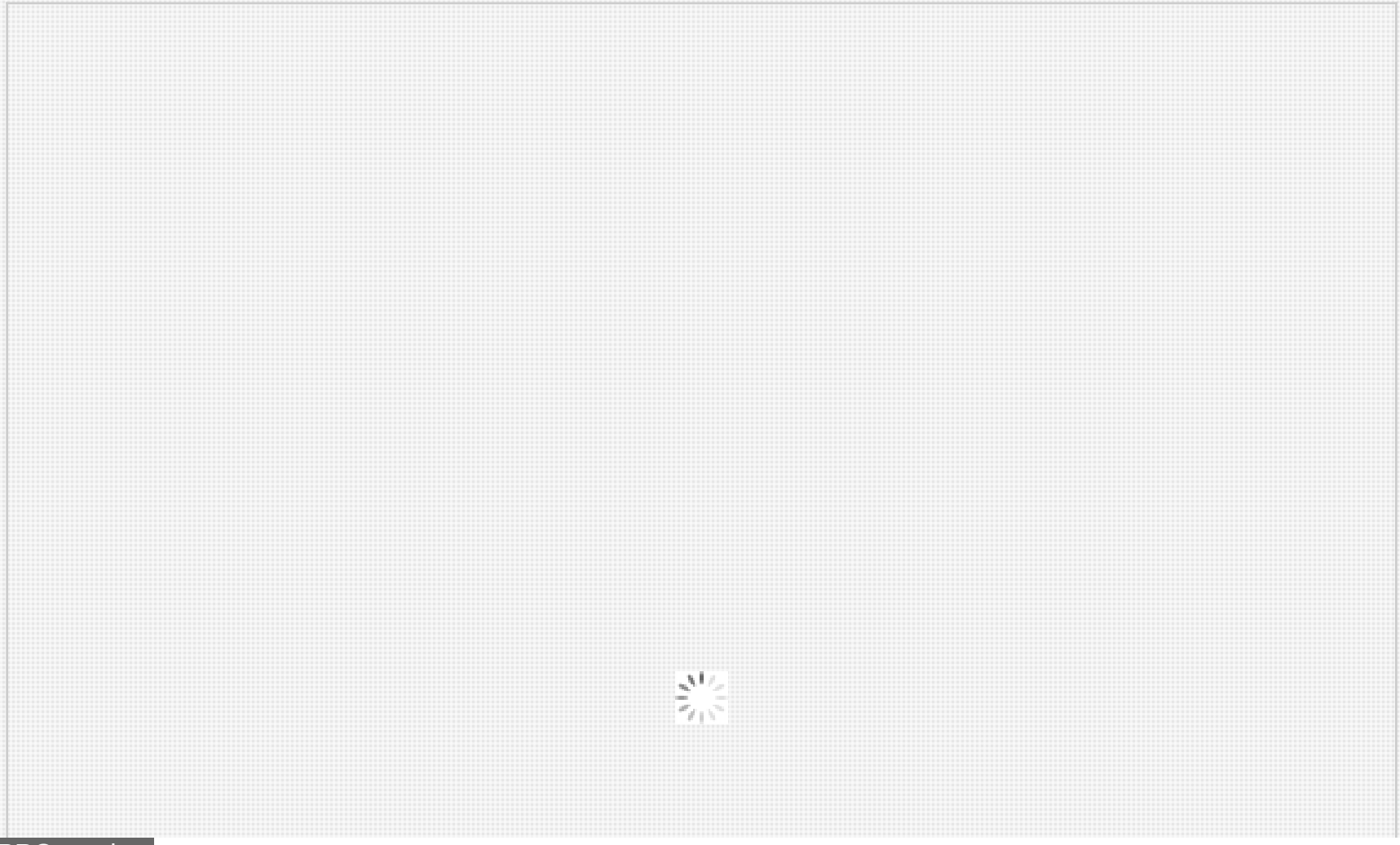
As with most Kansa modules, you can run it stand-alone on your localhost, or through Kansa to collect data from thousands of hosts via Windows Remote Management. To run it against your local system, you should be able to download it from the above link, unblock it either through Explorer by browsing to it, right-clicking on it and unchecking the unblock checkbox under properties somewhere, I don't GUI enough. Or you can download it, open a Powershell prompt to the location where you've downloaded it and do:

```
ls Get-LogUserAssist.ps1 | unblock-file
```

The above may assume you have PS v3, I'm not sure when unblock-file came into being. You should upgrade to PS v3, if you haven't already as it has more whizbang.

Another option is to use Sysinternals Streams.exe -d Get-LogUserAssist.ps1, but I digress.

Here's an example of me running this locally on my laptop:



Click for larger image

When run locally, the script returns Powershell objects. The output directive for this script, tells Kansa to save the output as tab separated values, making for easy import into a database or quick analysis with Logparser. An analysis script for this output is on the Kansa issues list as an enhancement.

If you run this locally and want to massage the output to TSV, at your Powershell prompt, you could do:

```
PS> Get-LogUserAssist.ps1 | ConvertTo-CSV -Delimiter "`t" -NoTypeInformation
```

I don't care for quoted TSV, so I'd go a step further adding:

```
| % { $_ -replace "`"" }
```

to the above and why not write it out to a file that you can load into Excel or a database or query with Logparser? To do that, simply add the below to the above:

```
| Set-Content LocalhostUserAssist.tsv
```

But you don't have to do TSV. You could drop the -Delimiter arg above and default to CSV or instead of using ConvertTo-CSV, use Export-CliXML and you've got XML output for those of you who want a more challenging and slower analysis experience. Zing!

One thing I'm not clear on and may have to research, is why so many of my "counts" are coming up as zero. Did Windows 8 stop incrementing run counts?

This collector starts by enumerating all of the local profiles on the target, then looks in each profile path for an ntuser.dat file. If it finds one, it will try and load that hive. If the hive loads, the script looks for UserAssist and parses it, if found. If UserAssist is not found, it moves on to the next user. If the script was unable to load the hive, it assumes that's because the user is currently logged on and the file is locked, so at that point, it looks in HKEY_USERS for all the loaded hives by SID, resolves those SIDs to usernames and compares them to the username associated with the locked profile. When it finds a match, it looks for UserAssist in the matching HKEY_USERS key by SID. One thing that occurs to me now, based on something I heard @forensic_matt say at this year's SANS DFIR Summit, if the user's account is renamed, this match will likely fail. Something to add to the Kansa issues list. Save for that edge case, this script will pull UserAssist key data for all user accounts on a running system.

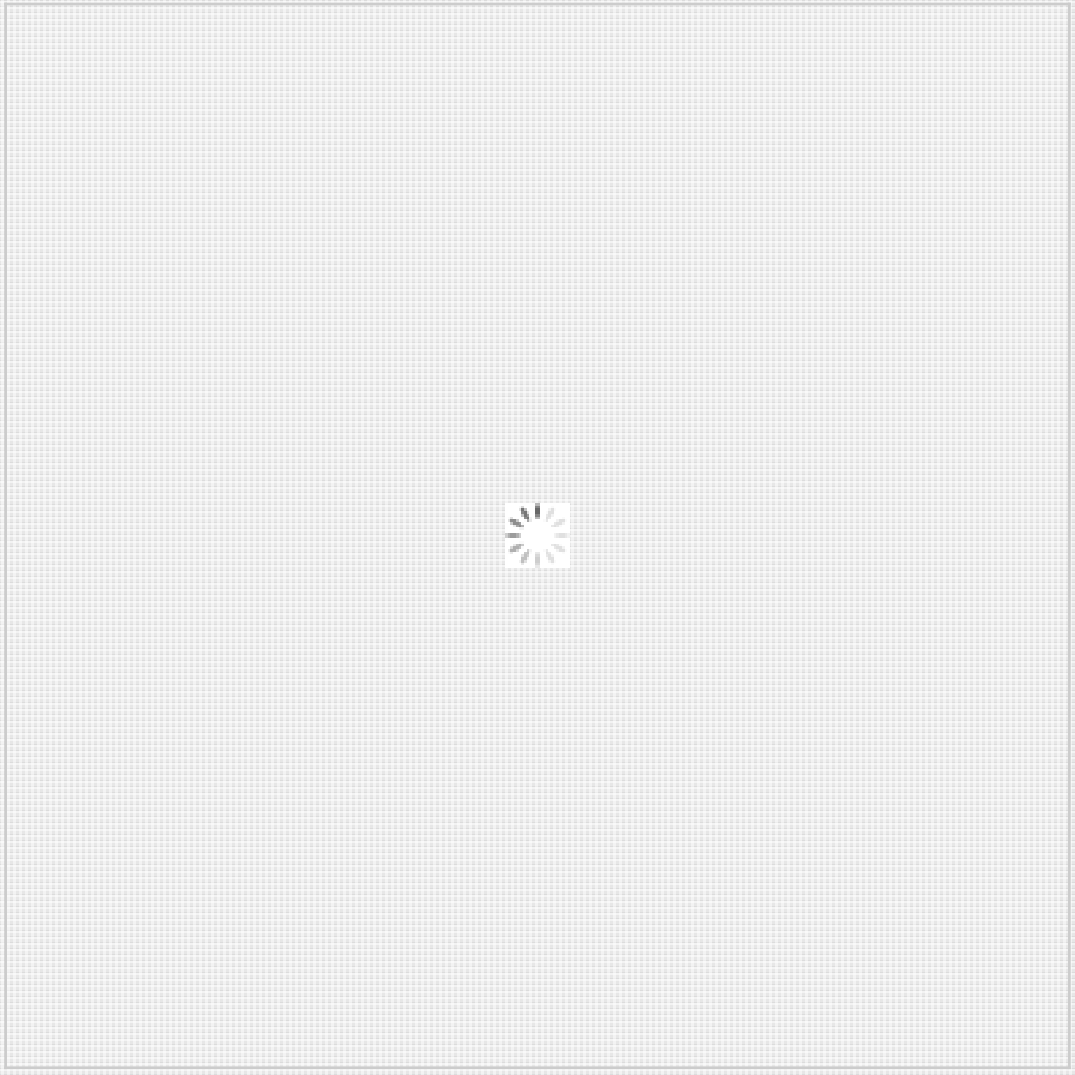
And since it's a Kansa module, you can run it across thousands of hosts easily.

I hope someone finds it useful.

[Update] You may be wondering, "Why is this module under Modules\Log, the Registry is not a log file?"

As Harlan Carvey has rightly pointed out, the Registry sometimes is a log file and in the case of UserAssist, it most certainly is. Hence, I placed it under Modules\Log. You're free to move it elsewhere on your own set up.

[Update 20140623] Confirmed for renamed accounts the module was not able to resolve a loaded SID to an account name, but I've fixed this bug. The script now returns the user account name and the user profile path, so spotting renamed accounts is simple. Here's an example where the Local Administrator account has been renamed to Gomer.



Posted 18th June 2014 by davehull

Labels: Kansa, Registry, UserAssist

MAY
18

Kansa: Powershell profiles potentially hazardous

On the very day I published my previous post, Kansa: Collecting WMI Event Consumer backdoors, Mark Russinovich announced the release of a new version of Autoruns that collects WMI related ASEPs. I had a chance to play around with it on a machine with a WMI Event Consumer, Event Filter and Filter-to-Consumer Binding configured and indeed, Autoruns now picks up the Event Consumers. I still recommend using Kansa's Get-WMIEvtFilter.ps1 and Get-WMIFltConBind.ps1 collector modules to grab the other two essential pieces that make Event Consumer backdoors possible. The Event Filter is the piece that will tell you what triggers the Event Consumer.

In this post I want to cover another "auto start extension point" or ASEP and it happens to be another that is not covered by Autoruns, yet. It also happens to be specific to Powershell. The Windows Powershell Profile is a script that runs, if present, each time a user or SYSTEM opens a Powershell shell. It's akin to a .bash_profile or similar shell profile on *nix systems.

Adversaries can modify an existing Powershell profile for either a user or the default system profile, planting code enabling them to maintain persistence or perform any task that Powershell is capable of given the context of the script (non-administrator users obviously being less capable than administrators or SYSTEM).

Kansa's Get-PSProfiles.ps1 collector will enumerate local accounts on remote systems and check each of them for Powershell profiles. Where Powershell profiles exist, Get-PSProfiles will collect them all in a zip file (it will also check for and collect the default Powershell profile). The zip file will then be sent back to the host where Kansa was run.

Powershell profiles can be located in a few different locations. For user profiles, they are in:

```
$env:userprofile\Documents\WindowsPowershell\Microsoft.Powershell_profile.ps1
```

And the default system profile is in:

```
$env:windir\System32\WindowsPowershell\v1.0\Microsoft.Powershell_profile.ps1
```

User Powershell profiles on XP systems are in a slightly different path and Kansa will not acquire them.

Unfortunately, there's no quick way of analyzing the collected profile scripts for malicious capabilities, at least not that I'm aware of. Analysts will have to spend time reviewing profiles for suspect code. This is a good time to mention that any ASEP script, not just Powershell profiles could be modified by adversaries to perform nefarious actions.

This is another painful reminder of the asymmetry of information security. Adversaries have many places to hide malicious bits and may only need one (or none, if they have a big enough key ring of credentials). Incident responders, depending on the nature of the incident, may have to review every known ASEP.

Enjoy the code review and happy hunting!

Posted 18th May 2014 by davehull

Labels: aseps, Kansa, powershell, powershell profiles

MAY
13

Kansa: Collecting WMI Event Consumer backdoors

In my previous post, Kansa: Service related collectors and analysis, I discussed the Windows Service related collectors and analysis capabilities in Kansa and noted that some of the collected data is not currently collected by Sysinternals' Autoruns.

Today I'll cover another persistence mechanism that Kansa collects, which is not currently collected by Autoruns; namely WMI Event Consumers. That link tells us "Event consumers are applications or scripts that request notification of events, and then perform tasks when specific events occur."
[Update: 2014-05-13] Mark Russinovich released a new version of Autoruns today that reports WMI information. I have not tested it yet. It will be interesting to see if it only reports data form Event Consumers and not the Event Filter, which tells what the trigger is.

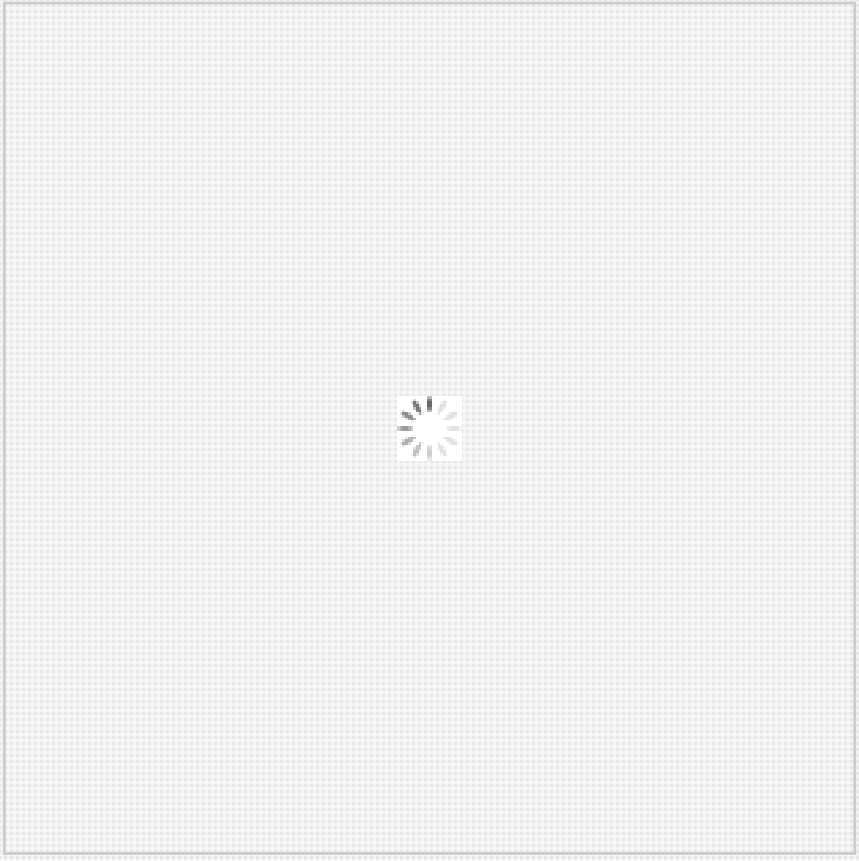
For an event consumer to work, three elements are required:

- An Event Consumer -- this is the piece that performs some action
- An Event Filter -- an event query watching for defined activity -- this triggers the consumer
- A Filter-to-Consumer Binding -- this links the filter to the consumer

In my experience, WMI Event Consumers are not commonly used. So in many situations collecting the data and simply reviewing file sizes can tell you if something is worth investigating further. For example, I recently collected event consumer data from a few thousand hosts. Running the following Powershell command was enough to find which host contained a backdoor running from an event consumer:

```
ls *wmievtconsmr.xml | sort length -Descending | more
```

The output of that command follows, see if you can determine which host had the backdoor installed:

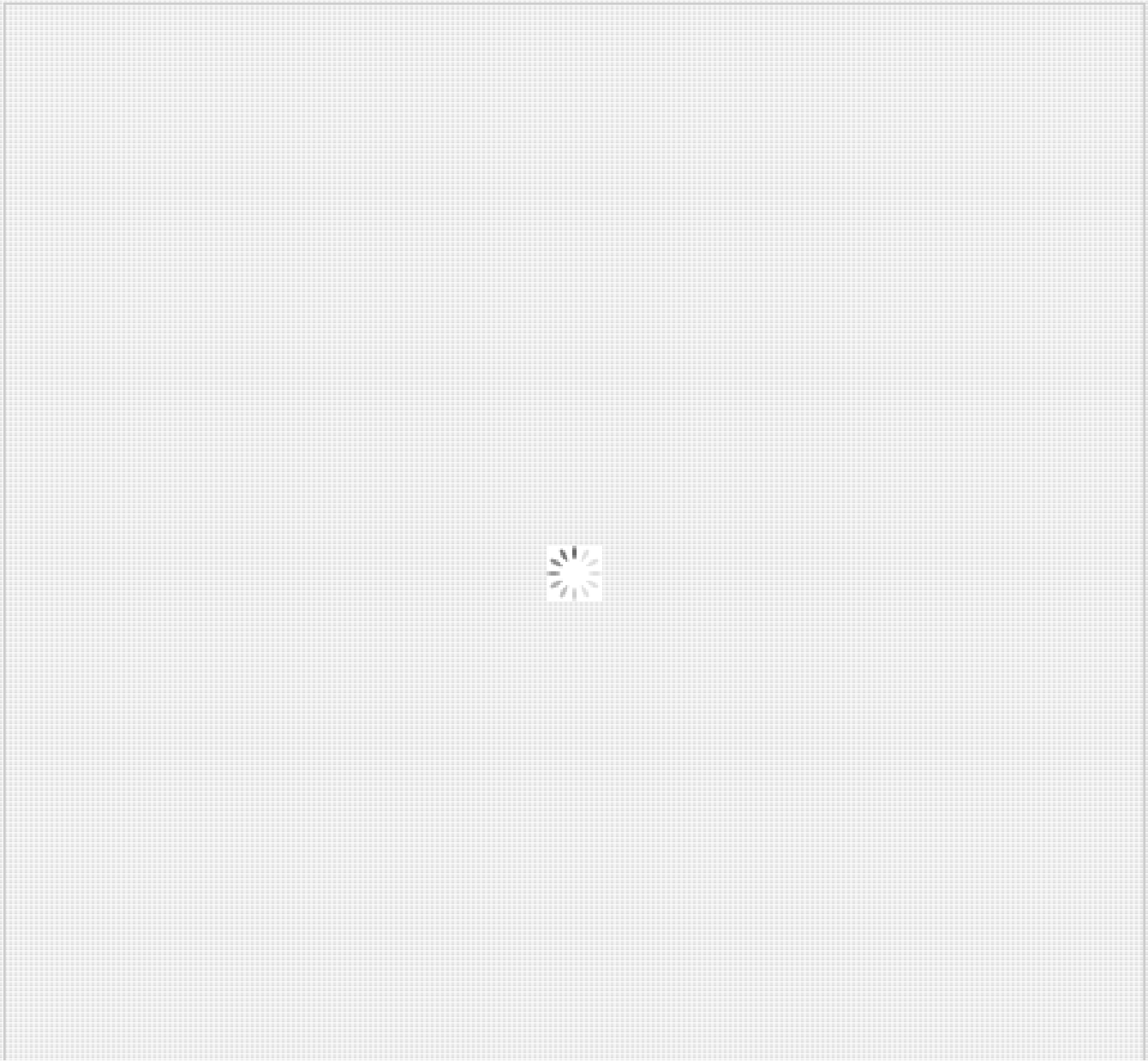


If you guessed DFWBOSSWEE01, congratulations, you may have the skills necessary to find WMI Event Consumer backdoors.

So what's in this file? Since it was collected with Kansa's Get-WMIEvtConsumer collector, which specifies its output should be written to an XML file, we can either open the XML file in a suitable editor or use the Powershell cmdlet Import-Clixml to read the file into a variable and examine the contents via the following commands:

```
$data = Import-Clixml .\DFWBOSSWEE01_wmievtconsmr.xml
$data | more
```

This command returns output like the following:



The most interesting bits above are those in the "CommandLineTemplate" property, which I've redacted a bit, but you can see there's a call to Powershell.exe and a long base 64 encoded string, which in this case was a Powershell encoded command, in essence, a script. We can decode that script via

```
[Convert]::ToBase64String()
```

Doing so would reveal that when this WMI Event Consumer is triggered, it connects to a remote site and downloads another script and runs it.

So how often is it triggered? What triggers it? To answer those questions, you'll have to review the data Kansa collected via Get-WMIEvtFilter.ps1. A consumer by itself is harmless, but if there's an Event Filter and a Filter-to-Consumer binding, then you've got all the ingredients needed for a WMI Event Consumer based backdoor.

Posted 13th May 2014 by davehull

Labels: aseps, backdoor, Kansa, WMI Event Consumer