

Extracting Hancitor's Configuration with Ghidra part 1

 medium.com/@crovax/extracting-hancitors-configuration-with-ghidra-7963900494b5

Crovax

January 21, 2022



Crovax

Dec 27, 2021

.

7 min read

Topics covered

- Locate the decryption function
- Determine the decryption algorithm
- Build Ghidra script to decrypt the configuration file
- Build yara detection rule (In part 2)

Hancitor Overview

Hancitor is an older malware loader that is known to drop additional malware on to the system once infected (cobalt strike and some ransomware variants). The entry point is from your typical malspam campaign, that contains a link to an office doc or already has one embedded in the email.

The original dll is usually packed, and once unpacked, will execute the hancitor payload and reach out to one of its hard coded c2's.

| **Original packed samples:**

<https://www.malware-traffic-analysis.net/2021/06/17/index.html>

| **Unpacked Sample:**

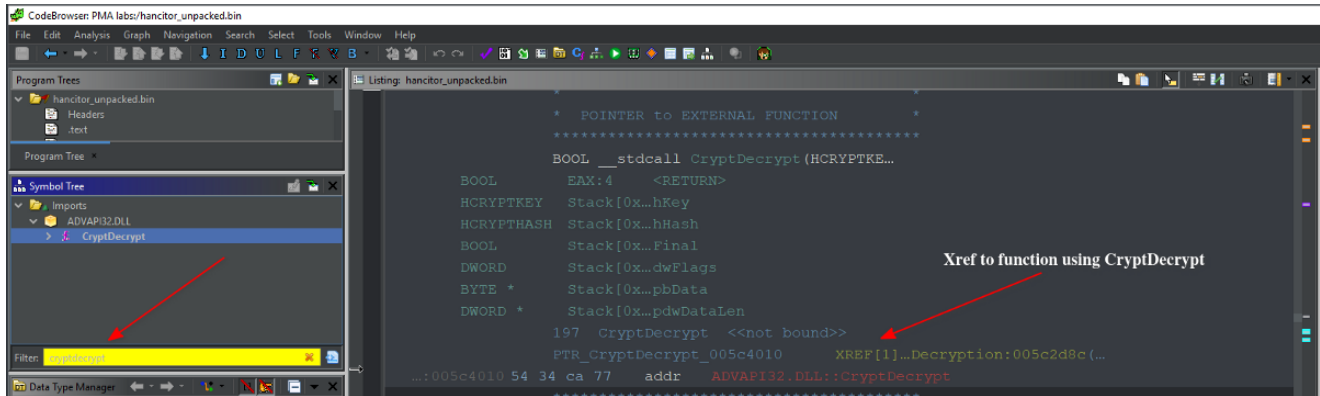
SHA-256

882b30e147fe5fa6c79b7c7411ce9d8035086ad2f58650f5d79aadfb2ffd34f4

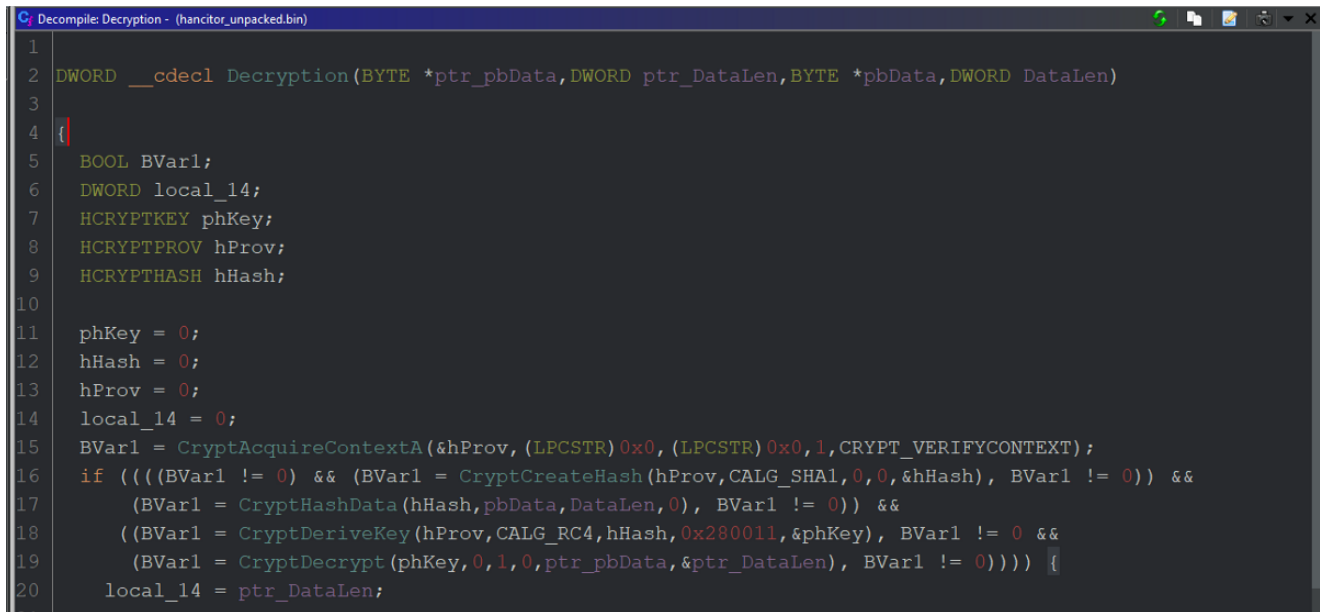
Locating the decryption function

Hancitor leverages the Windows native Cryptographic Service provider (CSP) context to perform its decryption routine. Knowing that, locating the decryption function will be relatively easy.

One way, is to load the binary into Ghidra and look at import table list for one of the cryptographic functions (CryptDecrypt, CryptAcquireContextA etc) then follow the referenced function from there.



Now that we have located the function leveraging the CryptDecrypt routine, we can get a better picture of how the decryption process works. Below is the decompiled view of the function performing the decryption routine. We'll step through each function to determine how the decryption logic works then start programming it out.



CryptAcquireContextA

The CryptAcquireContextA function initiates the call to the Cryptographic Service Provider (CSP) to determine what kind of CSP to use for the CryptoAPI functions. This function is straightforward as its role is just to initiate the CSP context for use. We can determine which CSP is going to be used by the first 'PUSH 0x0' instruction. Since its a null value being used, the default selection is made (native windows cryptographic service provider).

```

:005c2cd0 55          PUSH   EBP
:005c2cd1 8b ec         MOV    EBP, ESP
:005c2cd3 83 ec 14      SUB    ESP, 0x14
:005c2cd6 c7 45 f4 00... MOV    dword ptr [EBP + phKey], 0x0
:005c2cdd c7 45 fc 00... MOV    dword ptr [EBP + hHash], 0x0
:005c2ce4 c7 45 f8 00... MOV    dword ptr [EBP + hProv], 0x0
:005c2ceb c7 45 f0 00... MOV    dword ptr [EBP + local_14], 0x0
:005c2cf2 c7 45 ec 11... MOV    dword ptr [EBP + local_18], 0x2...
:005c2cf9 68 00 00 00... PUSH   CRYPT_VERIFYCONTEXT           ; DWORD dwFlags for CryptAcquireContextA
:005c2cfe 6a 01        PUSH   0x1                          ; DWORD dwProvType for CryptAcquireConte...
:005c2d00 6a 00        PUSH   0x0                          ; LPCSTR szProvider for CryptAcquireCont...
:005c2d02 6a 00        PUSH   0x0                          ; LPCSTR szContainer for CryptAcquireCon...
:005c2d04 8d 45 f8     LEA   EAX=>hProv, [EBP + -0x8]
:005c2d07 50          PUSH   EAX                          ; HCRYPTPROV * phProv for CryptAcquireCo...
:005c2d08 ff 15 20 40... CALL  dword ptr [->ADVAPI32.DLL::Cryp... ; = 77c69143
:005c2d0e 85 c0        TEST  EAX, EAX
:005c2d10 75 0a        JNZ   LAB_005c2d1c
:005c2d12 e9 89 00 00... JMP   LAB_005c2da0

```

CryptCreateHash

When the CryptCreateHash function is called, it creates the hashing object to be used in the cryptographic routine, and determines the type of hashing algorithm to be used. Once successful, it returns a handle to the object, for subsequent calls to the other cryptographic functions.

To determine what hashing algorithm is being used, we can look at the 'AlgId' value that will be pushed on to the stack. In this case, we see the 'PUSH CALG_SHA1' instruction is being used. So, we know that the hashing algorithm is SHA1.

```

LAB_005c2d1c                                XREF[1]...005c2d10(j)
...:005c2d1c 8d 4d fc     LEA   ECX=>hHash, [EBP + -0x4]
...:005c2d1f 51          PUSH   ECX                          ; HCRYPTHASH * phHash for CryptCreateHash
...:005c2d20 6a 00        PUSH   0x0                          ; DWORD dwFlags for CryptCreateHash
...:005c2d22 6a 00        PUSH   0x0                          ; HCRYPTKEY hKey for CryptCreateHash
...:005c2d24 68 04 80 00... PUSH   CALG_SHA1                    ; ALG_ID AlgId for CryptCreateHash
...:005c2d29 8b 55 f8     MOV    EDX, dword ptr [EBP + hProv]
...:005c2d2c 52          PUSH   EDX                          ; HCRYPTPROV hProv for CryptCreateHash
...:005c2d2d ff 15 0c 40... CALL  dword ptr [->ADVAPI32.DLL::C... ; = 77c6deb6
...:005c2d33 85 c0        TEST  EAX, EAX
...:005c2d35 75 04        JNZ   LAB_005c2d3b
...:005c2d37 eb 67        JMP   LAB_005c2da0
...:005c2d39 eb          ??   EBh
...:005c2d3a 65          ??   65h   e

```

CryptHashData

This function is going to hash the data passed to it, using the previously specified algorithm(SHA1). But what is being hashed and how do we figure it out?

First we need to look back at the arguments being passed into the function (I have labeled it "Decryption") to see what data is being passed and where its located.

by looking at the arguments and their position in which they're being passed in, we need to find out what the data is expecting and where is it located. In order to find this information we can follow the data that is being passed into the function.

Hint: I have already labeled it 😊

```
1
2 DWORD __cdecl Decryption(BYTE *ptr_pbData,DWORD ptr_DataLen, BYTE *arg_ptr_key, DWORD arg_key_length)
3
4 {
5     BOOL BVar1;
6     DWORD local_14;
7     HCRYPTKEY phKey;
8     HCRYPTPROV hProv;
9     HCRYPTHASH hHash;
10
11     phKey = 0;
12     hHash = 0;
13     hProv = 0;
14     local_14 = 0;
15     BVar1 = CryptAcquireContextA(&hProv, (LPCSTR)0x0, (LPCSTR)0x0, 1, CRYPT_VERIFYCONTEXT);
16     if (((BVar1 != 0) && (BVar1 = CryptCreateHash(hProv, CALG_SHA1, 0, 0, &hHash), BVar1 != 0)) &&
17         (BVar1 = CryptHashData(hHash, arg_ptr_key, arg_key_length, 0), BVar1 != 0)) &&
18         (BVar1 = CryptDeriveKey(hProv, CALG_RC4, hHash, 0x280011, &phKey), BVar1 != 0) &&
19         (BVar1 = CryptDecrypt(phKey, 0, 1, 0, ptr_pbData, &ptr_DataLen), BVar1 != 0))) {
20
21 }
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
```

```
1
2 BYTE * Config_Decryption(void)
3
4 {
5     if (ptr_alloc_mem == (BYTE *)0x0) {
6         DAT_005c5000 = 0;
7         ptr_alloc_mem = (BYTE *)Heap_Alloc(0x2000);
8         mov_encrypted_data(ptr_alloc_mem, &ptr_encrypted_data, 0x2000);
9         Decryption(ptr_alloc_mem, 0x2000, ptr_to_key, 8);
10    }
11    return ptr_alloc_mem;
12 }
13
```

Address	Disassembly	Comment
...:005c5005 60	??	60h
...:005c5006 70	??	70h p
...:005c5007 80	??	80h
...:005c5008 32	??	32h 2
...:005c5009 32	??	32h 2
...:005c500a 2e	??	2Eh
...:005c500b 31	??	31h 1
...:005c500c 31	??	31h 1
...:005c500d 00	??	00h
...:005c500e 00	??	00h
...:005c500f 00	??	00h
...
...:005c5010 b3	ptr_to_key db	B3h
...:005c5011 03	db	3h
...:005c5012 18	db	18h
...:005c5013 aa	db	AAh
...:005c5014 0a	db	Ah
...:005c5015 d2	db	D2h
...:005c5016 77	db	77h
...:005c5017 de	db	DEh

As you can see, the third argument being passed is a pointer to the 'key' that's going to be hashed, and the argument being passed to the right of it (8) is the key length.

so we know the following:

Key = b'\xb3\x03\x18\xaa\x0a\xd2\x77\xde'

Key length = 8 bytes

CryptDeriveKey

This next part is going to be a bit tricky. The CryptDeriveKey is going to accept a few parameters:

hProv = handle to the cryptographic service provided being used.

AlgId = algorithm for which the key is to be generated. In this case, its going to be RC4.

hHash or **hBaseData** = handle to the hash object that points to the data.

dwFlags = this is going to be key length that is going to be used. Which is the lower 16 bits of the value being passed. In our case that value is 0x00280011, so we only want the lower values or 0x0028 (we can truncate the leading zeros). So by dividing $0x28 / \text{key length}(8)$ that was being passed to the CryptHashData function, we know how many bytes of the RC4 key is going to be used to decrypt the configuration data.

RC4 key: $0x28 / 8 = (5 \text{ bytes})$

```
LAB_005c2d57 XREF[1]..005c2d51(j)
...:005c2d57 8d 45 f4 LEA EAX=>phKey, [EBP + -0xc]
...:005c2d5a 50 PUSH EAX
...:005c2d5b 8b 4d ec MOV ECX, dword ptr [EBP + local_18]
...:005c2d5e 51 PUSH ECX
...:005c2d5f 0b 55 fe MOV EDX, dword ptr [EBP + hHash]
...:005c2d62 52 PUSH EDX
...:005c2d63 68 01 68 00... PUSH CALG_RC4
...:005c2d68 8b 45 f8 MOV EAX, dword ptr [EBP + hProv]
...:005c2d6b 50 PUSH EAX
...:005c2d6c ff 15 10 40... CALL dword ptr [->ADVAPI32.DLL::CryptDerive...
...:005c2d72 85 e0 TEST EAX, EAX
...:005c2d74 75 04 JNZ LAB_005c2d7a
...:005c2d76 eb 28 JMP LAB_005c2da0
...:005c2d78 eb ?? EBH
...:005c2d79 26 ?? 26h &
```

```
5 BOOL BVar1;
6 DWORD local_14;
7 HCRYPTKEY phKey;
8 HCRYPTPROV hProv;
9 HCRYPTHASH hHash;
10
11 phKey = 0;
12 hHash = 0;
13 hProv = 0;
14 local_14 = 0;
15 BVar1 = CryptAcquireContextA(&hProv, (LPCSTR)0x0, (LPCSTR)0x0, 1, CRY
16 if (((BVar1 != 0) && (BVar1 = CryptCreateHash(hProv, CALG_SHA1, 0,
17 (BVar1 = CryptHashData(hHash, arg_ptr_key, arg_key_length, 0), B
18 ((BVar1 = CryptDeriveKey(hProv, CALG_RC4, hHash, 0x280011, &phKey)
19 (BVar1 = CryptDecrypt(phKey, 0, 1, 0, ptr_pbData, &ptr_DataLen), B
20 local_14 = ptr_DataLen;
```

CryptDecrypt

This function is straightforward, it accepts the data to be decrypted, the key used, and the length of the data as parameters. To verify, we can once again look at the arguments that are being passed to the function.

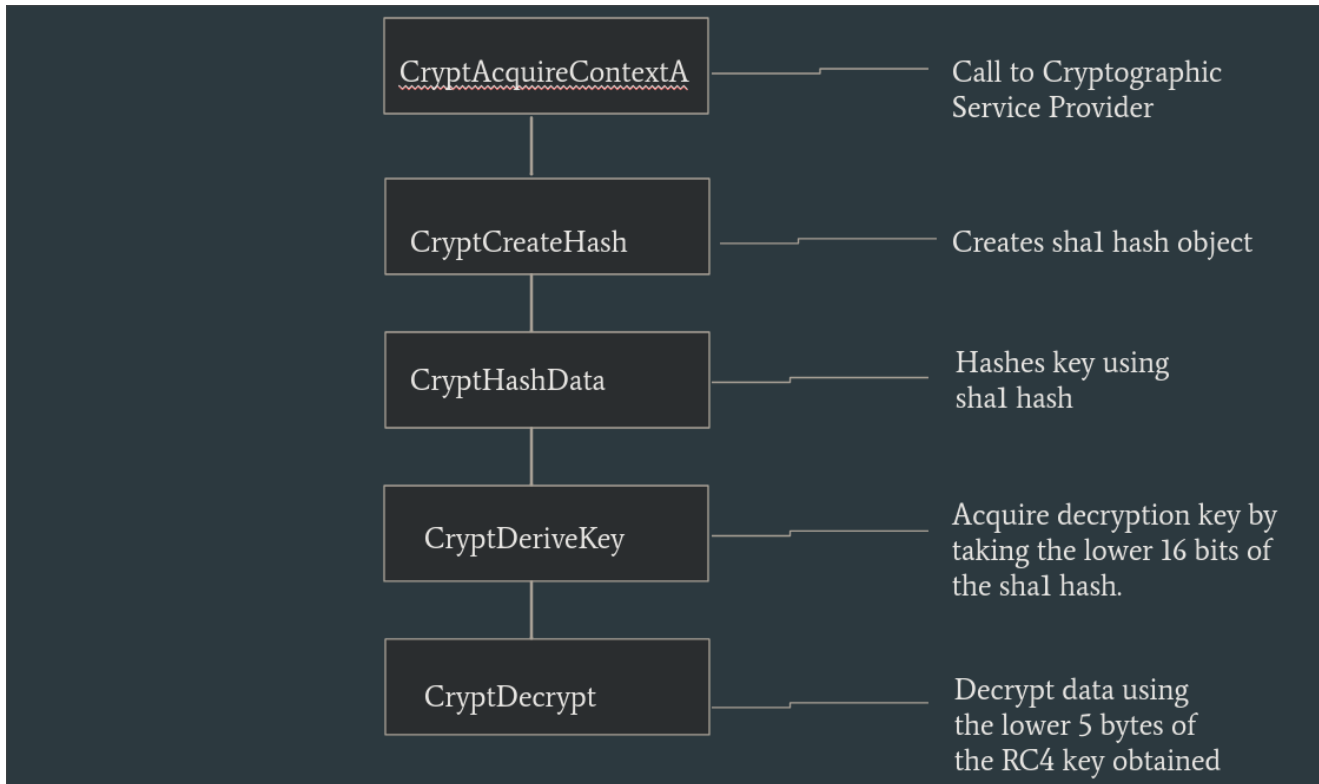
```
LAB_005c2d7a XREF[1]..005c2d74(j)
...:005c2d7a 8d 4d 0c LEA ECX=>ptr_DataLen, [EBP + 0xc]
...:005c2d7d 51 PUSH ECX
...:005c2d7e 8b 55 08 MOV EDX, dword ptr [EBP + ptr_pbData]
...:005c2d81 52 PUSH EDX
...:005c2d82 6a 00 PUSH 0x0
...:005c2d84 6a 01 PUSH 0x1
...:005c2d86 6a 00 PUSH 0x0
...:005c2d88 8b 45 f4 MOV EAX, dword ptr [EBP + phKey]
...:005c2d8b 50 PUSH EAX
...:005c2d8c ff 15 10 40... CALL dword ptr [->ADVAPI32.DLL::CryptDecrypt]; = 77
...:005c2d92 85 e0 TEST EAX, EAX
...:005c2d94 75 04 JNZ LAB_005c2d9a
...:005c2d96 eb 08 JMP LAB_005c2da0
...:005c2d98 eb ?? EBH
...:005c2d99 06 ?? 06h

...:005c25e5 68 00 20 00... PUSH 0x2000
...:005c25e6 68 10 50 5c... PUSH ptr_encrypted_data
...:005c25ef 8b 15 64 72... MOV EDX, dword ptr [ptr_alloc_mem]
...:005c25f5 52 PUSH EDX
...:005c25f6 e8 55 ee ff... CALL mov_encrypted_data
...:005c25fb 83 e4 0c ADD ESP, 0xc
...:005c25fe 6a 08 PUSH 0x8
...:005c2600 68 10 50 5c... PUSH ptr_to_key
...:005c2605 68 00 20 00... PUSH 0x2000
...:005c2606 a1 64 72 5c... MOV EAX, [ptr_alloc_mem]
...:005c260f 50 PUSH EAX
...:005c2610 e8 bb 06 00... CALL Decryption
```

```
4 {
5 BOOL BVar1;
6 DWORD local_14;
7 HCRYPTKEY phKey;
8 HCRYPTPROV hProv;
9 HCRYPTHASH hHash;
10
11 phKey = 0;
12 hHash = 0;
13 hProv = 0;
14 local_14 = 0;
15 BVar1 = CryptAcquireContextA(&hProv, (LPCSTR)0x0, (LPCSTR)0x0, 1, CRY
16 if (((BVar1 != 0) && (BVar1 = CryptCreateHash(hProv, CALG_SHA1
17 (BVar1 = CryptHashData(hHash, arg_ptr_key, arg_key_length, 0)
18 ((BVar1 = CryptDeriveKey(hProv, CALG_RC4, hHash, 0x280011, &phK
19 (BVar1 = CryptDecrypt(phKey, 0, 1, 0, ptr_pbData, &ptr_DataLen)
20
21 BYTE * Config_Decryption(void)
22
23 {
24 if (ptr_alloc_mem == (BYTE *)0x0) {
25 DAT_005c5000 = 0;
26 ptr_alloc_mem = (BYTE *)Heap_Alloc(0x2000);
27 mov_encrypted_data(ptr_alloc_mem, ptr_encrypted_data, 0x2000);
28 Decryption(ptr_alloc_mem, 0x2000, ptr_to_key, 8);
29
30 return ptr_alloc_mem;
31 }
```

based on the value being passed in, we can see the encrypted data length is equal to 0x2000 bytes.

Below is a graphical representation of the actions performed thus far.



Creating the Ghidra script

We now understand how the decryption process works. The next step is to create a script in Ghidra to automate this whole process, so we can extract the build/campaign id and the c2 domains from this sample.

Note: the python script can be downloaded from my github (link [here](#)). Your cursor needs to be pointing to the first address of the encrypted data before running the script. This is due to the 'currentAddress' method.

We know we need to create a sha1 hash of the 8 byte key that's being passed into the Decryption function. So we can copy those bytes out of Ghidra and store them into a variable. The next step is to get a hash (sha1) of the key and extract the first 5 bytes (Key Hash: a956a1e6ff).

Next we need to get the encrypted data using ghidra's getBytes function, and then perform any necessary conversions on the hex values

```

def get_encrypted_bytes():
    get_addr = currentAddress
    get_bytes = list(getBytes(get_addr, 2000))
    converted_bytes = ''
    cByte = ''
    for byte in get_bytes:
        if byte < 0:
            cByte = (0xff - abs(byte) + 1)
        else:
            cByte = byte
        converted_bytes += chr(cByte)

    return converted_bytes

```

We now have our key and the encrypted data we need to decrypt. The last step is to pass these parameters to our rc4 decryption function to perform the remaining steps.

```

def rc4_decrypt(key, data):
    x = 0
    box = range(256)
    for i in range(256):
        x = (x + box[i] + ord(key[i % len(key)])) % 256
        box[i], box[x] = box[x], box[i]
    x = 0
    y = 0
    out = []
    for char in data:
        x = (x + 1) % 256
        y = (y + box[x]) % 256
        box[x], box[y] = box[y], box[x]
        out.append(chr(ord(char) ^ box[(box[x] + box[y]) % 256]))
    return ''.join(out)

```

Now that we have our decrypted data, we just need to format the data we want and discard any null bytes.

```

print 'Current Address:', currentAddress
print 'Key Hash:', binascii.hexlify(key_hash)
get_data = get_encrypted_bytes()
config = rc4_decrypt(key_hash, get_data)
build_id = config.split('\x00')[0]
print 'Build_id:', build_id
for string in config.split('\x00')[1:]:
    if string != '':
        c2 = string
        break
c2_List = c2.split('|')
for c2 in c2_List:
    if c2 != '':
        print 'c2:', c2

```

The output should look something like this 😊

```
ptr_encrypted_data XREF[1]...Config_Decryption:00...
...:005c5018 c0      ??      C0h
...:005c5019 2e      ??      2Eh
...:005c501a b5      ??      B5h
...:005c501b 06      ??      06h
...:005c501c b1      ??      B1h
...:005c501d a4      ??      A4h
...:005c501e ae      ??      AEh

Console - Scripting
Decode_Config.py> Running...
key length 8
Current Address: 005c5018
Key Hash: a956a1e6ff
Build_id: 1706_apkreb6
c2: http://thestaccultur.com/8/forum.php
c2: http://arguendinfuld.ru/8/forum.php
c2: http://waxotheousch.ru/8/forum.php
Decode_Config.py> Finished!
```

Decode_Config.py> Running...key length 8Current Address: 005c5018Key Hash: a956a1e6ffBuild_id: 1706_apkreb6c2: http://thestaccultur.com/8/forum[.]phpc2: http://arguendinfuld.ru/8/forum[.]phpc2: http://waxotheousch.ru/8/forum[.]php

Conclusion

By breaking down the individual functions of the decryption routine we were able to determine how hancitor was decrypting its c2 domain configuration. We then applied the same process in creating a Ghidra script to automatically perform the same steps statically, to reveal the encrypted data.

In part 2, we will take a similar approach and build a yara rule to test our theory and see if we can detect multiple hancitor variants.

As always, Don't expect much, as I have no clue what I'm doing. 😊