StealC Malware Analysis Part 3

Archived: 2026-04-05 15:51:46 UTC

06 - Analysis of StealC (Stage 4)

In the first article of the series, we saw how to unpack the first stage `pkr_ce1a` manually and using an emulator (MIASM). In the second article we have extracted the C2 of the loader and unpacked the last stage using MIASM. Now let's take a look at the **StealC** malware and recover some IOCs.

Sample information

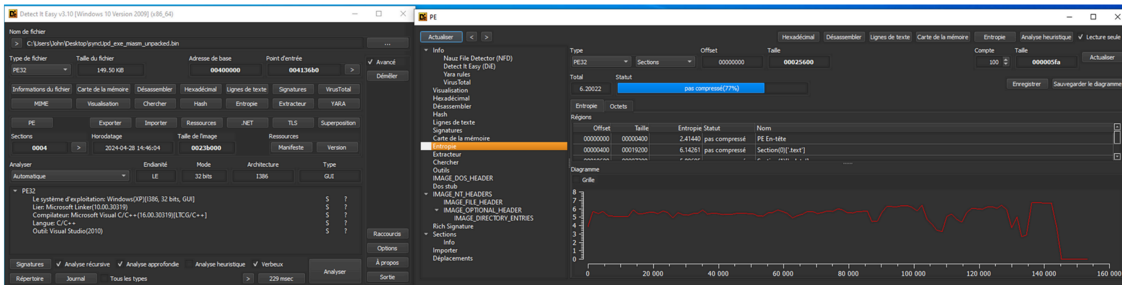
Below is the information concerning the last stage we are going to analyze:

Type	Data
SHA256	18f53dd06e6d9d5dfe1b60c4834a185a1cf73ba449c349b6b43c753753256f62
SHA1	a952b1ed963346d5029a9acb3987d5e3a65c47a3
MD5	8022ef84cfe9deb72b1c66cd17cac1cd
File size	153 088 bytes
First seen	N/A
MIME type	application/x-dosexec
imphash	1ef0d6e4c3554a91026b47d9a27bf6db
ssdeep	3072:ivyLlG8KPgpJSG61doHN4NoQiUukOoy9bzyRy2GxhGJuU:ivyhJryZoIohvkOpt+M2GzAu

At the time of writing, the sample has not been made public on sample-sharing platforms such as VirusTotal.

Detect the sample family

First, we can look at the entropy of the file to see if it is packaged:



Detect It Easy - Entropy of StealC sample (stage 4)

The latter is not packaged, as its entropy is not high.

We don't know which family of malware it is yet. A loader? A beacon? To find out, we can run a Yara scan. If you don't have a Yara rules database, you can find some on GitHub repositories such as [Yara-Rules]() (no longer maintained), or on specialized platforms such as [Malpedia]().

If you run a scan on all Malpedia's Yara rules, you should come across a positively matching rule named `win_stealc_auto` :

```
 $ yara -s win.stealc_auto.yar syncUpd_exe_miasm_unpacked.bin
win_stealc_auto syncUpd_exe_miasm_unpacked.bin
0x135df:$sequence_0: FF 15 38 4F 62 00 85 C0 75 07 C6 85 E0 FE FF FF 43
0xde4c:$sequence_1: 68 6F D7 41 00 E8 EA 82 00 00 E8 65 E7 FF FF 83 C4 74
0xc760:$sequence_2: 50 E8 3A 9A 00 00 E8 D5 F2 FF FF 83 C4 74
0xc81c:$sequence_2: 50 E8 DE 40 FF FF E8 D9 EC FF FF 83 C4 74
0xc8ca:$sequence_2: 50 E8 70 98 00 00 E8 EB FC FF FF 83 C4 74
0xaf51:$sequence_3: E8 4A B2 00 00 E8 D5 E4 FF FF 81 C4 80 00 00 00 E9 53 03 00 00
0xb03c:$sequence_3: E8 5F B1 00 00 E8 EA E3 FF FF 81 C4 80 00 00 00 E9 68 02 00 00
0xd558:$sequence_4: 50 E8 42 8C 00 00 E8 DD F3 FF FF 81 C4 84 00 00 00
0xd5d4:$sequence_4: 50 E8 C6 8B 00 00 E8 61 F3 FF FF 81 C4 84 00 00 00
0xd650:$sequence_4: 50 E8 4A 8B 00 00 E8 E5 F2 FF FF 81 C4 84 00 00 00
0x124c7:$sequence_5: E8 44 25 FF FF 83 C4 60 E8 7C E2 FF FF 83 C4 0C
0x10692:$sequence_6: E8 09 5B 00 00 E8 A4 4A FF FF 83 C4 18 6A 3C
0x149d6:$sequence_7: FF 15 88 50 62 00 50 FF 15 20 50 62 00 8B 55 08 89 02
0x149dc:$sequence_8: 50 FF 15 20 50 62 00 8B 55 08 89 02
```

It seems that the executable matches 9 out of 10 sequences, which is a very good score. We can therefore strongly assume that this is the final **StealC** malware. Malpedia has [a dedicated page about it]().

Our aim is to recover C2 from the malicious program, and several methods can be used:

- Sandbox
- Emulation
- Static analysis

In this article, we will use the **Static** method.

Automate string decryption

Open the sample with your favorite disassembler. By wandering through the various functions, you should be able to identify methods calling `sub_4043b0` which seems to take 3 parameters: a sequence of bytes, a string, then an integer (which seems to correspond to the length of the string):

```
sub_402130:
   0 @ 00402147  data_624b24 = sub_4043b0(&data_41e23c, "OFCJ5RMVF94M8EN", 0xf)
   1 @ 00402160  data_624db8 = sub_4043b0(&data_41e24c, &data_41e0f4, 2)
   2 @ 00402179  data_624b10 = sub_4043b0(&data_41e254, &data_41e250, 2)
   3 @ 00402192  data_624d54 = sub_4043b0(&data_41e25c, &data_41e258, 2)
   4 @ 004021ab  data_624d50 = sub_4043b0(&data_41e264, &data_41e260, 2)
   5 @ 004021c4  data_624be0 = sub_4043b0("p15cC>1t-&9!B1", "7TA31QR5IBKD1B", 0xe)
   6 @ 004021dd  data_624a00 = sub_4043b0("xV ,\t&)9#$.s", "49AHEOKKBVW2", 0xc)
   7 @ 004021f6  data_624da4 = sub_4043b0("U6%K:", "9EQ9Y36P", 8)
   8 @ 0040220f  data_624e10 = sub_4043b0(&data_41e2cc, "FZ4YVFXEWL", 0xa)
   9 @ 00402228  data_624a60 = sub_4043b0(&data_41e2e8, "XBW3FCWIZDOX", 0xc)
  10 @ 00402241  data_624a4c = sub_4043b0(&data_41e304, "RPH8RNOJMQG", 0xb)
  11 @ 0040225a  data_624aec = sub_4043b0(&data_41e318, "L6EEL", 5)
  12 @ 00402273  data_624cb8 = sub_4043b0(&data_41e338, "V90GY07UGG8NMQE3ARHP", 0x14)
  13 @ 0040228c  data_624b30 = sub_4043b0(&data_41e364, "RXXKZN9G4K6A7KW0MI", 0x12)
  14 @ 004022a5  data_624d84 = sub_4043b0(&data_41e384, "91RW2KOB66M", 0xb)
  15 @ 004022be  data_624d28 = sub_4043b0(&data_41e3a0, "NLO6Z48LQQK10", 0xd)
  16 @ 004022d7  data_624bac = sub_4043b0(&data_41e3c0, "05J8QC3OR1SQ", 0xc)
  17 @ 004022f0  data_624ae0 = sub_4043b0(&data_41e3dc, "745N6YQD4", 9)
  18 @ 00402309  data_624dd8 = sub_4043b0(&data_41e3fc, "R9RDIQYO61YA156A", 0x10)
  19 @ 00402322  data_6248b0 = sub_4043b0("=&G1[F-w", "QU3C86T6", 8)
  20 @ 0040233b  data_624d7c = sub_4043b0(&data_41e438, "Q19D8U24X6J624", 0xe)
  21 @ 00402354  data_624a20 = sub_4043b0(&data_41e45c, "HWE9VA4QG99MQDWA5", 0x11)
  22 @ 0040236d  data_624c08 = sub_4043b0("9""*>", "UQVXRQVQ", 8)
  23 @ 00402386  data_624e00 = sub_4043b0(&data_41e494, "M93M4W9IVNH", 0xb)
  24 @ 0040239f  data_6248bc = sub_4043b0(&data_41e4b8, "78KSBAUAEWDYHXAYPHG0", 0x14)
  25 @ 004023b8  data_624928 = sub_4043b0(&data_41e4e0, "FTTJAADMN9D40", 0xd)
  26 @ 004023d1  data_624aac = sub_4043b0(&data_41e508, "PC6J7TX6QQ3ILWZ47TBP", 0x14)
  27 @ 004023ea  data_624d30 = sub_4043b0(&data_41e530, "WBNPBSB6IXFS", 0xc)
  28 @ 00402403  data_624978 = sub_4043b0("-\Xfva758", "J81UDOSYT", 9)
  29 @ 0040241c  data_624900 = sub_4043b0("A)!(qh`+><", "4ZDZBZNORP", 0xa)
  30 @ 00402435  data_6249d8 = sub_4043b0("3164:j`g+>\", "PCODNYRIOR0", 0xb)
  31 @ 0040244e  data_624b1c = sub_4043b0("6G( TcP)\", "X3LL8M4E0", 9)
  32 @ 00402467  data_624c94 = sub_4043b0(&data_41e5b0, "SBN52MKZA3XR", 0xc)
  33 @ 00402480  data_624c14 = sub_4043b0("s:?1@Q", "0HZP44O5V", 9)
  34 @ 00402499  data_6249c8 = sub_4043b0(&data_41e5e8, "GDG2G61OCMSJ1", 0xd)
  35 @ 004024b2  data_624b88 = sub_4043b0("d+61*1", "6NZTKBO1H", 9)
  36 @ 004024cb  data_624924 = sub_4043b0(&data_41e628, "V4J7VV6DSQJEM8GDWPSL", 0x14)
  37 @ 004024e4  data_624c04 = sub_4043b0("K"&)W-", "8QEH9K", 6)
  38 @ 004024fd  data_62494c = sub_4043b0(&data_41e660, "MZLTNR49RMTI", 0xc)
  39 @ 00402516  data_624d20 = sub_4043b0(&data_41e678, "LGR94C", 6)
  40 @ 0040252f  data_624de8 = sub_4043b0(&data_41e688, "AWUPSPY", 7)
  41 @ 00402548  data_6248d0 = sub_4043b0(&data_41e698, "V5PJFZB", 7)
  42 @ 00402559  HLOCAL result = sub_4043b0(&data_41e6ac, "7KJG18C2KMQ", 0xb)
  43 @ 00402561  data_624bf0 = result
  44 @ 00402567  return result
```

Binary Ninja - Function with lot of call with strings arguments

If we unpack the `sub_4043b0` function, we can see that the malware uses a military encryption algorithm (`xor`). We can rename the function and its variables:

```
004043b0  HLOCAL simple_crypto_xor(void* encoded_bytes, char* key, int32_t key_len)

004043c0      HLOCAL result = LocalAlloc(uFlags: LMEM_ZEROINIT, uBytes: key_len + 1)
004043cf      *(result + key_len) = 0
004043b0
004043ea      for (int32_t i = 0; i u< key_len; i += 1) {
0040441a          *(result + i) = *(encoded_bytes + i) ^ key[modu.dp.d(0:i, strlen(_Str: key))]
004043ea      }
004043b0
00404425      return result
```

Function that XOR strings

The function we just renamed `simple_crypto_xor` is used by two other functions in the program:

```
Code References                                                                              {351}
  sub_402130                                                                                  {43}
    |← 0040213f data_624b24 = simple_crypto_xor(&data_41e23c, "OFCJ5RMVF94M8EN", 0xf)
    |← 00402158 data_624db8 = simple_crypto_xor(&data_41e24c, &data_41e0f4, 2)
    |← 00402171 data_624b10 = simple_crypto_xor(&data_41e254, &data_41e250, 2)
    |← 0040218a data_624d54 = simple_crypto_xor(&data_41e25c, &data_41e258, 2)
    |← 004021a3 data_624d50 = simple_crypto_xor(&data_41e264, &data_41e260, 2)
    |← 004021bc data_624be0 = simple_crypto_xor("p15cC>1t-&9!B1", "7TA31QR5IBKD1B", 0xe)
    |← 004021d5 data_624a00 = simple_crypto_xor("xV ,\t&)9#$.s", "49AHEOKKBVW2", 0xc)
    |← 004021ee data_624da4 = simple_crypto_xor("U6%K:", "9EQ9Y36P", 8)
    |← 00402207 data_624e10 = simple_crypto_xor(&data_41e2cc, "FZ4YVFXEWL", 0xa)
    |← 00402220 data_624a60 = simple_crypto_xor(&data_41e2e8, "XBW3FCWIZDOX", 0xc)
    |← 00402239 data_624a4c = simple_crypto_xor(&data_41e304, "RPH8RNOJMQG", 0xb)
    |← 00402252 data_624aec = simple_crypto_xor(&data_41e318, "L6EEL", 5)
    |← 0040226b data_624cb8 = simple_crypto_xor(&data_41e338, "V90GY07UGG8NMQE3ARHP", 0x14)
    |← 00402284 data_624b30 = simple_crypto_xor(&data_41e364, "RXXKZN9G4K6A7KW0MI", 0x12)
    |← 0040229d data_624d84 = simple_crypto_xor(&data_41e384, "91RW2KOB66M", 0xb)
    |← 004022b6 data_624d28 = simple_crypto_xor(&data_41e3a0, "NLO6Z48LQQK10", 0xd)
    |← 004022cf data_624bac = simple_crypto_xor(&data_41e3c0, "05J8QC3OR1SQ", 0xc)
    |← 004022e8 data_624ae0 = simple_crypto_xor(&data_41e3dc, "745N6YQD4", 9)
    |← 00402301 data_624dd8 = simple_crypto_xor(&data_41e3fc, "R9RDIQYO61YA156A", 0x10)
    |← 0040231a data_6248b0 = simple_crypto_xor("=&G1[F-w", "QU3C86T6", 8)
    |← 00402333 data_624d7c = simple_crypto_xor(&data_41e438, "Q19D8U24X6J624", 0xe)
    |← 0040234c data_624a20 = simple_crypto_xor(&data_41e45c, "HWE9VA4QG99MQDWA5", 0x11)
    |← 00402365 data_624c08 = simple_crypto_xor("9""*>", "UQVXRQVQ", 8)
    |← 0040237e data_624e00 = simple_crypto_xor(&data_41e494, "M93M4W9IVNH", 0xb)
    |← 00402397 data_6248bc = simple_crypto_xor(&data_41e4b8, "78KSBAUAEWDYHXAYPHG0", 0x14)
    |← 004023b0 data_624928 = simple_crypto_xor(&data_41e4e0, "FTTJAADMN9D40", 0xd)
    |← 004023c9 data_624aac = simple_crypto_xor(&data_41e508, "PC6J7TX6QQ3ILWZ47TBP", 0x14)
    |← 004023e2 data_624d30 = simple_crypto_xor(&data_41e530, "WBNPBSB6IXFS", 0xc)
    |← 004023fb data_624978 = simple_crypto_xor("-\Xfva758", "J81UDOSYT", 9)
    |← 00402414 data_624900 = simple_crypto_xor("A)!(qh`+><", "4ZDZBZNORP", 0xa)
    |← 0040242d data_6249d8 = simple_crypto_xor("3164:j`g+>\", "PCODNYRIOR0", 0xb)
    |← 00402446 data_624b1c = simple_crypto_xor("6G( TcP)\", "X3LL8M4E0", 9)
    |← 0040245f data_624c94 = simple_crypto_xor(&data_41e5b0, "SBN52MKZA3XR", 0xc)
    |← 00402478 data_624c14 = simple_crypto_xor("s:?1@Q", "0HZP44O5V", 9)
    |← 00402491 data_6249c8 = simple_crypto_xor(&data_41e5e8, "GDG2G61OCMSJ1", 0xd)
    |← 004024aa data_624b88 = simple_crypto_xor("d+61*1", "6NZTKBO1H", 9)
    |← 004024c3 data_624924 = simple_crypto_xor(&data_41e628, "V4J7VV6DSQJEM8GDWPSL", 0x14)
    |← 004024dc data_624c04 = simple_crypto_xor("K"&)W-", "8QEH9K", 6)
    |← 004024f5 data_62494c = simple_crypto_xor(&data_41e660, "MZLTNR49RMTI", 0xc)
    |← 0040250e data_624d20 = simple_crypto_xor(&data_41e678, "LGR94C", 6)
    |← 00402527 data_624de8 = simple_crypto_xor(&data_41e688, "AWUPSPY", 7)
    |← 00402540 data_6248d0 = simple_crypto_xor(&data_41e698, "V5PJFZB", 7)
    |← 00402559 HLOCAL result = simple_crypto_xor(&data_41e6ac, "7KJG18C2KMQ", 0xb)
  sub_402590                                                                                 {308}
    |← 0040259f data_624d34 = simple_crypto_xor(&data_41e6d0, "4UNEM48WCGN06G652VO0MW", 0x16)
    |← 004025b8 data_6249f8 = simple_crypto_xor(&data_41e700, "RKA7LH59ILH9Y5KQTVKSX", 0x15)
    |← 004025d1 data_624b20 = simple_crypto_xor(&data_41e72c, "4UUM95DKG6WDICDEPS", 0x12)
    |← 004025ea data_624a2c = simple_crypto_xor("UT%Y7?Cbqe", "11C8BS7SAU", 0xa)
    |← 00402603 data_624d70 = simple_crypto_xor(&data_41e770, "MC2Q49X0PULM1ZCPV5P3HLE", 0x17)
    |← 0040261c data_6249c0 = simple_crypto_xor(&data_41e79c, "DS1V6TCOR4S7EDGJIP", 0x12)
    |← 00402635 data_624974 = simple_crypto_xor(&data_41e7bc, "MY4HP9M5V7", 0xa)
    |← 0040264e data_6249a8 = simple_crypto_xor(&data_41e7d4, "CCHNA7Z7", 8)
    |← 00402667 data_624d14 = simple_crypto_xor(&data_41e7ec, "CO0GNG3RJ17", 0xb)
```

Binary Ninja - XREF of simple_crypto_xor

We can then rename the variables according to the bytes decoded with our disassembler's API. First, we'll try to recover the address of the function that performs the XOR operations, based on the calls to the `LocalAlloc` and `strlen` functions:

```
 def get_most_called_function_sorted():
    funcs = bv.functions
    call_function_counter = {}
    for func in funcs:
        callers = func.callers
        #print("\nFunction {} is called from {} known locations.".format(func.name, len(callers)))
        call_function_counter[func] = len(callers)
    return sorted(((v, k) for k, v in call_function_counter.items()), reverse=True)

def get_xor_str_func():
    most_called_functions  = get_most_called_function_sorted()
    for func in most_called_functions:
        if func[0] < 260:
            # xor func must have more than 100 callers
            break
        if not func_has_call_func_by_name(func[1], "LocalAlloc"):
            continue
        if not func_has_call_func_by_name(func[1], "strlen"):
            continue
        return func[1]
    return None

def func_has_call_func_by_name(func, func_name: str):
    for inst in func.mlil.instructions:
        if not isinstance(inst, Localcall):
            continue
        if str(inst.dest) == func_name:
            return True
    return False
```

Once we have the address of our `simple_crypto_xor` function, we can try to identify the functions that call it, retrieve the arguments sent as parameters and then decode them. Once decoded, we can then rename the destination variables to make them easier to read:

```
 [...]

xored_str = None
def xorme(secret_string, key, key_len):
    final = ""
    for i in range(0, key_len):
```

```
        final = f"{final}{chr(secret_string[i] ^ key[i])}"
    return final

def visitor_xored_func(_a, inst, _c, _d) -> bool:
    global xored_str
    if isinstance(inst, Localcall):
        ptr_secret_string = int(inst.params[0].value)
        key = inst.params[1].string[0].encode()
        key_len = int(inst.params[2].value)
        secret_string = bv.read(ptr_secret_string, key_len)
        xored_str = xorme(secret_string, key, key_len)

def main():
    global xored_str
    xor_func = get_xor_str_func()
    if not xor_func:
        print("Error: StealC xor func not found")
        return
    print(f"Xor function at @{xor_func.address_ranges}")
    xor_func_callers = []
    callers = xor_func.callers
    for caller in callers:
        if caller in xor_func_callers:
            continue
        xor_func_callers.append(caller)
        for inst in caller.hlil.instructions:
            xored_str = None
            inst.visit(visitor_xored_func)
            if xored_str:
                try:
                    dst = inst.dest.get_expr(0).get_int(0)
                    var = bv.get_data_var_at(dst)
                    print(f"New string identified : {xored_str}")
                    xored_str_cleaned = xored_str.replace("/", "-").replace("%", "").replace(":", "").replace("
                    var.name = f"str_{xored_str_cleaned}" # Rename variable

                except:
                    pass
main()
```

This produces the following result:

```
00402590  HLOCAL decode_bytes02()

004025a7      str_http--185_172_128_150 = simple_crypto_xor(encoded_bytes: &data_41e6d0, key: "4UNEM48WCGN06G652VO0MW", key_len: 0x16)
004025c0      str_-c698e1bc8a2f5e6d_php = simple_crypto_xor(encoded_bytes: &data_41e700, key: "RKA7LH59ILH9Y5KQTVKSX", key_len: 0x15)
004025d9      str_-b7d0cfdb1d966bdd- = simple_crypto_xor(encoded_bytes: &data_41e72c, key: "4UUM95DKG6WDICDEPS", key_len: 0x12)
004025f2      str_default100 = simple_crypto_xor(encoded_bytes: "UT%Y7?Cbqe", key: "11C8BS7SAU", key_len: 0xa)
0040260b      str_GetEnvironmentVariableA = simple_crypto_xor(encoded_bytes: &data_41e770, key: "MC2Q49X0PULM1ZCPV5P3HLE", key_len: 0x17)
00402624      str_GetFileAttributesA = simple_crypto_xor(encoded_bytes: &data_41e79c, key: "DS1V6TCOR4S7EDGJIP", key_len: 0x12)
0040263d      str_GlobalLock = simple_crypto_xor(encoded_bytes: &data_41e7bc, key: "MY4HP9M5V7", key_len: 0xa)
00402656      str_HeapFree = simple_crypto_xor(encoded_bytes: &data_41e7d4, key: "CCHNA7Z7", key_len: 8)
0040266f      str_GetFileSize = simple_crypto_xor(encoded_bytes: &data_41e7ec, key: "CO0GNG3RJ17", key_len: 0xb)
00402688      str_GlobalSize = simple_crypto_xor(encoded_bytes: &data_41e804, key: "MOVGN4WJ5D", key_len: 0xa)
004026a1      str_CreateToolhelp32Snapshot = simple_crypto_xor(encoded_bytes: &data_41e82c, key: "2IA8FMOWF7GNIG4YGLYN70U8", key_len: 0x18)
004026ba      str_IsWow64Process = simple_crypto_xor(encoded_bytes: &data_41e858, key: "ISQ53SOX43KF0H", key_len: 0xe)
004026d3      str_Process32Next = simple_crypto_xor(encoded_bytes: &data_41e878, key: "I4P4D408FOMZX", key_len: 0xd)
004026ec      str_GetLocalTime = simple_crypto_xor(encoded_bytes: &data_41e898, key: "LVRGBXC0W5S3", key_len: 0xc)
00402705      str_FreeLibrary = simple_crypto_xor(encoded_bytes: &data_41e8b4, key: "WTSTP6Y88D4", key_len: 0xb)
0040271e      str_GetTimeZoneInformation = simple_crypto_xor(encoded_bytes: &data_41e8d8, key: "8CW0HJMBTE5DCMXISXRV87", key_len: 0x16)
00402737      str_GetSystemPowerStatus = simple_crypto_xor(encoded_bytes: &data_41e908, key: "BRLHO0SY0ZH1TDP74QK3", key_len: 0x14)
00402750      str_GetVolumeInformationA = simple_crypto_xor(encoded_bytes: &data_41e938, key: "9Y1PQH1XGKRR5BBF095WS", key_len: 0x15)

Log   Search log                                                                 All
[Default] New string identified : " & del "C:\ProgramData\*.dll"" & exit
[Default] New string identified : C:\Windows\system32\cmd.exe
[Default] New string identified : https
[Default] New string identified : Content-Type: multipart/form-data; boundary=----
[Default] New string identified : POST
[Default] New string identified : HTTP/1.1
[Default] New string identified : Content-Disposition: form-data; name="
[Default] New string identified : hwid
[Default] New string identified : build
[Default] New string identified : token
[Default] New string identified : file_name
[Default] New string identified : file
[Default] New string identified : message
[Default] New string identified : ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890
```

Binary Ninja - Script that decode xored strings and rename variables

Lists of decoded strings are available here.

Looking at the decoded strings, we learn a lot more about our malware! Some notable features:

- It seems to use the HTTP protocol to communicate
- An IP address is decoded: `185[.]172.128.150`
- A web page: `c698e1bc8a2f5e6d.php`
- A directory on the web server: `/b7d0cfdb1d966bdd/`
- Possibly anti-vm: `VMwareVMware`
- Possible recovery of identifiers in web browsers
- Possible login recovery in software such as Discord, Steam Pidgin, Outlook, Telegram, Tox...
- Recovery of elements of the system configuration of the machine running the program
- Potential recovery of cryptocurrency wallets

From the strings alone, we can deduce that specific method will be loaded into memory and then called. We won't be performing a Stealer (**StealC**) analysis.

Static sample analysis

You can continue the analysis by tracing function calls back to the decoded strings. For example, our `str_CreateToolhelp32Snapshot` variable appears to be loaded into memory:

```
00416240  int32_t sub_416240()

0041624a      if (data_625074 != 0) {
00416263          data_624f28 = guess_GetProcAddress(data_625074, str_GetEnvironmentVariableA)
0041627b          data_624fa4 = guess_GetProcAddress(data_625074, str_GetFileAttributesA)
00416294          data_624efc = guess_GetProcAddress(data_625074, str_GlobalLock)
004162ac          data_6250ac = guess_GetProcAddress(data_625074, str_HeapFree)
004162c4          data_625094 = guess_GetProcAddress(data_625074, str_GetFileSize)
004162dd          data_624ea0 = guess_GetProcAddress(data_625074, str_GlobalSize)
004162f5          handle_CreateToolhelp32Snapshot = guess_GetProcAddress(data_625074, str_CreateToolhelp32Snapshot)
0041630d          data_625050 = guess_GetProcAddress(data_625074, str_IsWow64Process)
00416326          data_624fd0 = guess_GetProcAddress(data_625074, str_Process32Next)
0041633e          data_624fc8 = guess_GetProcAddress(data_625074, str_GetLocalTime)
00416356          data_6250d0 = guess_GetProcAddress(data_625074, str_FreeLibrary)
0041636f          data_6250b0 = guess_GetProcAddress(data_625074, str_GetTimeZoneInformation)
00416387          data_624eac = guess_GetProcAddress(data_625074, str_GetSystemPowerStatus)
```

Function that use GetProcAddress on de-xored string

We rename the variable to `handle_CreateToolhelp32Snapshot`, we look for cross references to this handle and we can see its call later in the program:

```
00414de0  int32_t* sub_414de0(int32_t* arg1)

00414df4      void var_140
00414df4      sub_416d40(&var_140, &data_41d28a)
00414df9      int32_t var_134 = 0x128
00414e07      int32_t eax = handle_CreateToolhelp32Snapshot(2, 0)

00414e23      if (data_624f08(eax, &var_134) != 0) {
00414e30          while (true) {
```

Usage of methods after renaming them

Now that you've got all the marbles, you should be able to automate the renaming of the various functions loaded in memory, and study the part of the malware you're interested in. Our bndb file is available here.

You'll find here the code used to retrieve the complete C2 used by the **StealC** malware we've just studied.

07 - Conclusion

In this series of articles (part1, part2, part3), we have outlined various techniques for analyzing a malware sample active in 2024. We hope that this article has permit you to understand some of the methods used to:

- identify some packers
- unpack in manual and automated ways
- write detection rules (Yara)
- extract the interesting parts, such as C2, to help you protect your company and those around you.

You have handled a disassembler and a debugger for Stage 1 unpacking. You also got an overview of the MIASM framework and the possibilities offered by its jitter sandbox. You have seen some methods used by malicious actors to make analysis more difficult, using anti-emulation techniques. We've also seen how to automate most of our actions, such as extracting and decrypting shellcode from program resources. You've also had a taste of how to automate tedious tasks such as renaming encrypted variables in your favorite disassembler.

You should be able to set out on your own in search of new samples to analyze, and have the perseverance to overcome the technical problems you'll encounter, given a few liters of coffee and time.

Thanks to zbetcheckin for sharing the initial sample with the community. Thanks also to the MIASM contributors who produced a very practical and functional tool in our case. Thanks to the Binary Ninja developers and community for the various exchanges we were able to have. Thanks to the malware researchers who share their research on threats targeting cyberspace.

We hope this series of articles has motivated you to continue analyzing malicious code! Please don't hesitate to contact us if you'd like us to clarify any aspect of the articles, or if you have any questions we'd be happy to answer. We can also support you in malware analysis, or train you in this area.

We will shortly be publishing an article dedicated to the packer `pkr_ce1a` aka `AceCryptor` which protects malicious binaries like **StealC** seen in these articles. Stay tuned !

Source: https://blog.lexfo.fr/StealC_malware_analysis_part3.html