Orcus RAT Technical Malware Analysis and Configuration Extraction

By hardee

Published: 2023-02-13 · Archived: 2026-04-05 18:40:12 UTC

Our malware analysts are always on the lookout for and researching various malicious samples. This time we came across Orcus RAT in ANY.RUN online malware sandbox and decided to perform a technical malware analysis. In this article, you will learn how this RAT stores and protects its configuration and how to write the memory dump extractor in Python.

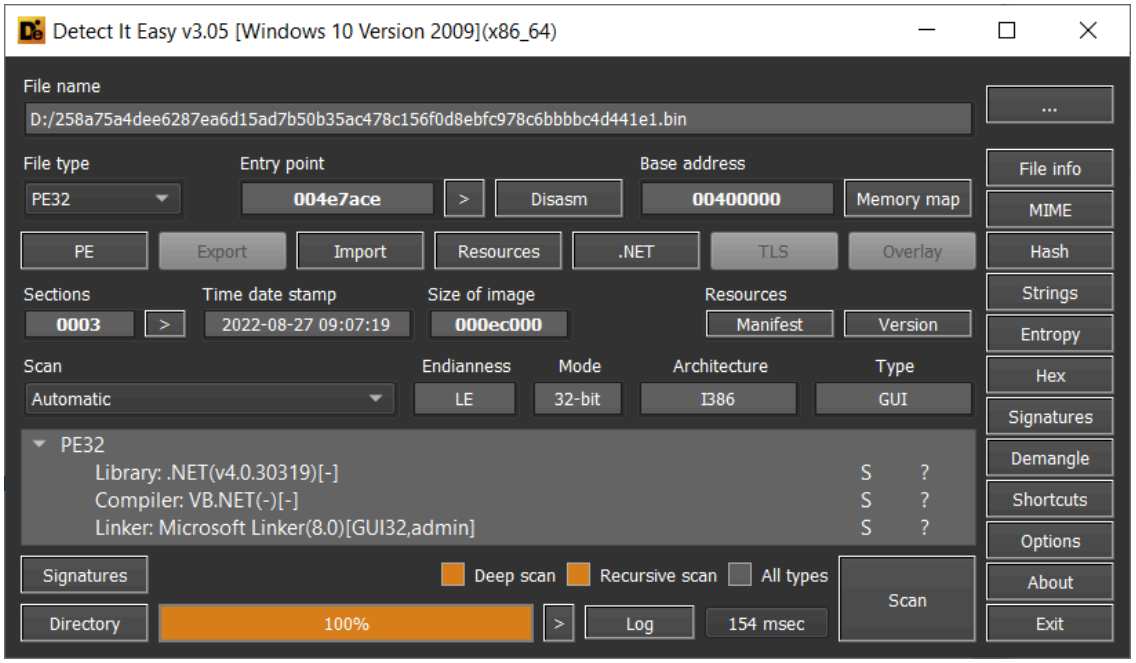
What is Orcus RAT?

Orcus is a Remote Access Trojan with some distinctive processes. The RAT allows attackers to create plugins and offers a robust core feature set that makes it quite a dangerous malicious program in its class.

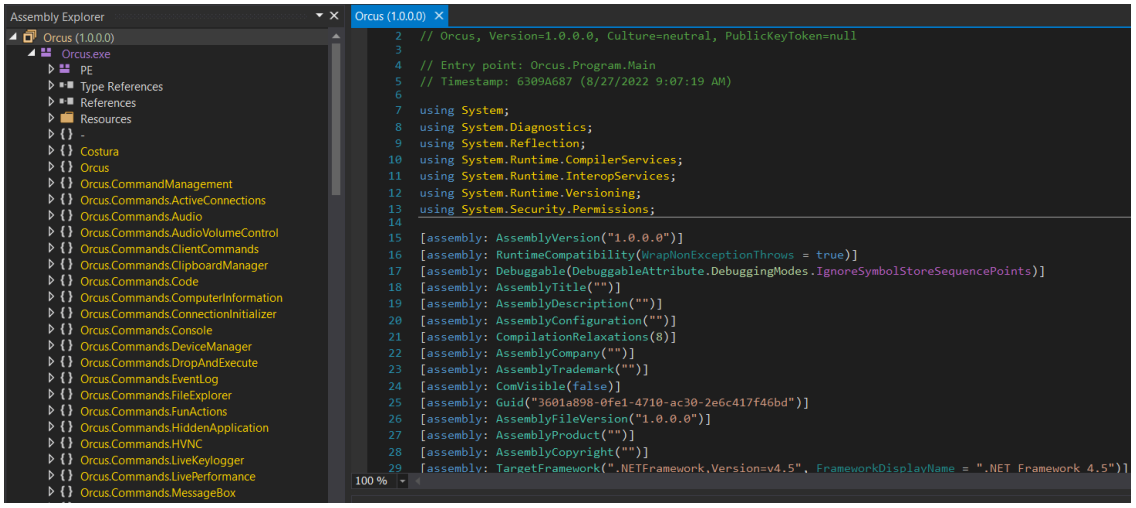
The sample for the malware analysis has been obtained from the ANY.RUN database. You can find it and follow along:

SHA-256	258a75a4dee6287ea6d15ad7b50b35ac478c156f0d8ebfc978c6bbbbc4d441e1

We downloaded the Orcus RAT sample and opened it in DiE to get basic information:



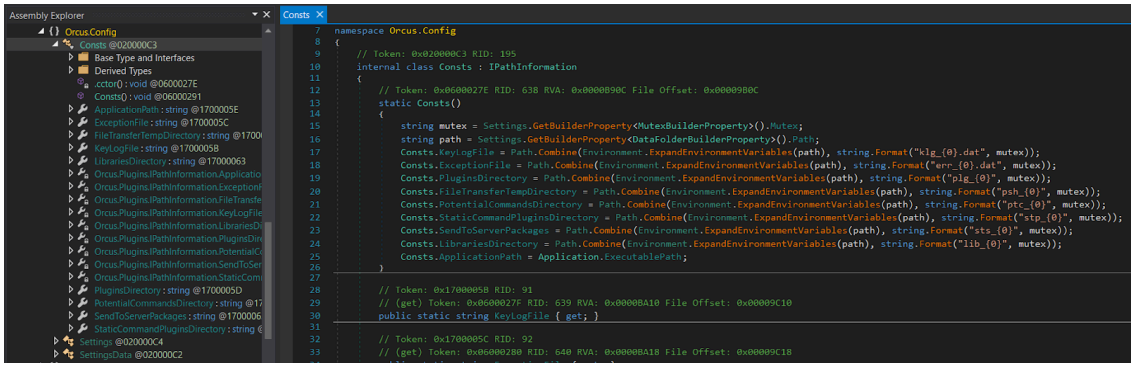
The DiE results show that we are dealing with a .NET sample. And it's high time to start malware analysis of Orcus. For this matter, DnSpy comes in handy.



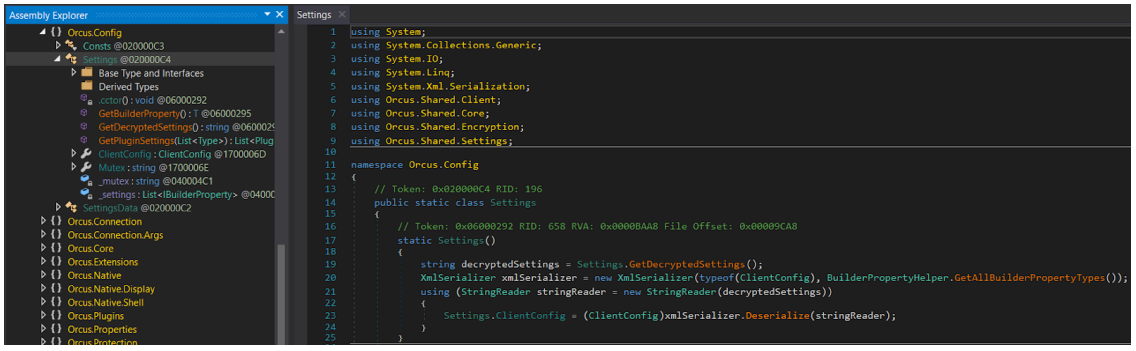
Orcus RAT classes overview

Our primary research goal is to find the RAT configuration. The first destination point is malware classes. While going through them, we bump into a namespace called Orcus.Config, and it contains the following classes:

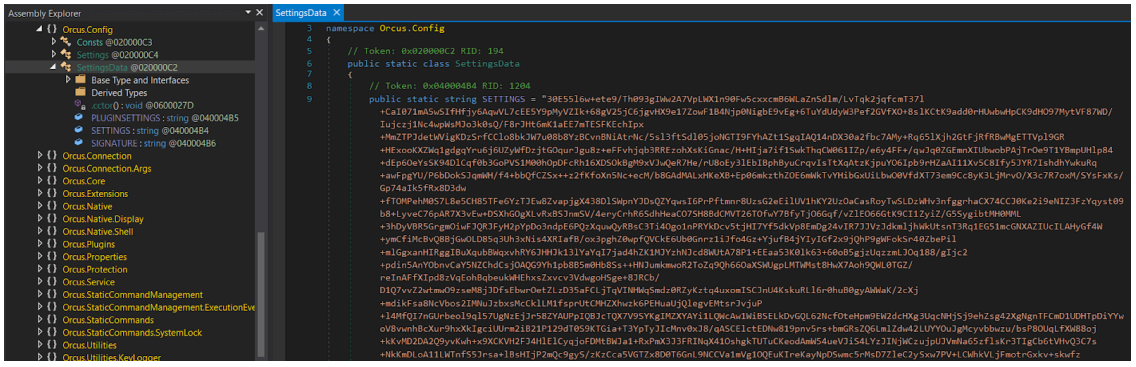
- **Consts** include information about the different files and directories that Orcus RAT uses. For example, the path to the file where user keystrokes are saved or to the directory where the plugins used by a sample reside.



- **Settings** contain wrapper methods for decrypting the malware configuration and its plugins.

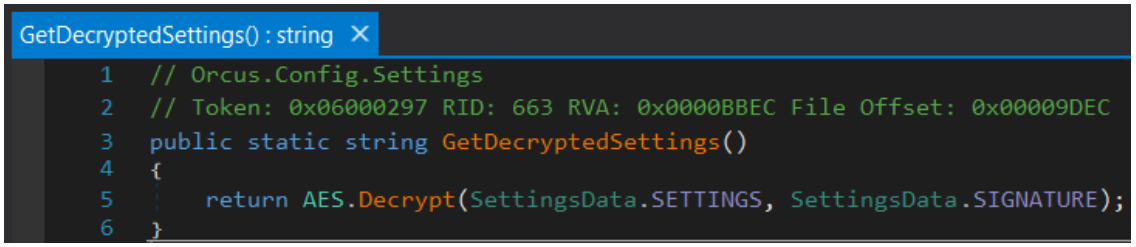


- **SettingsData** is a static class only with the encrypted malware and plugin configuration fields.

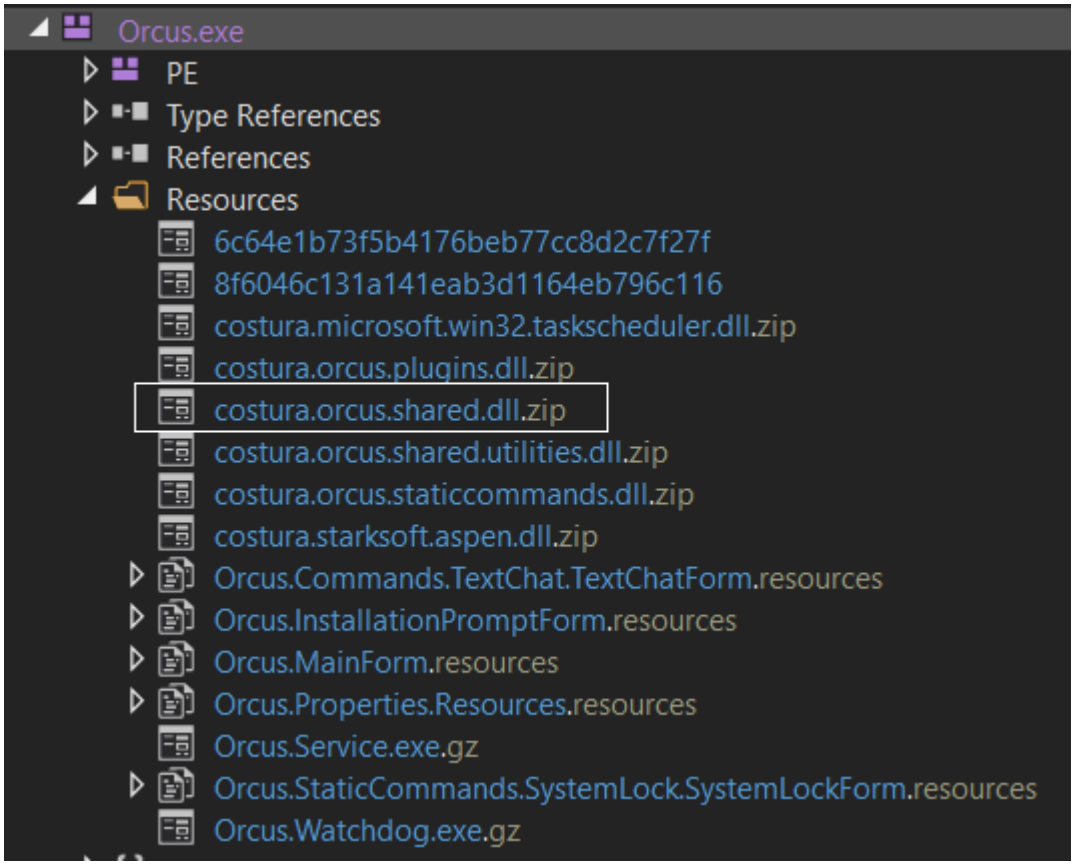


Orcus malware resources

Inside the **Settings** class, we see the **GetDecryptedSettings** method. Later, it calls out the **AES.Decrypt**. After noticing it, we can suppose that the AES algorithm encrypts the malware configuration:

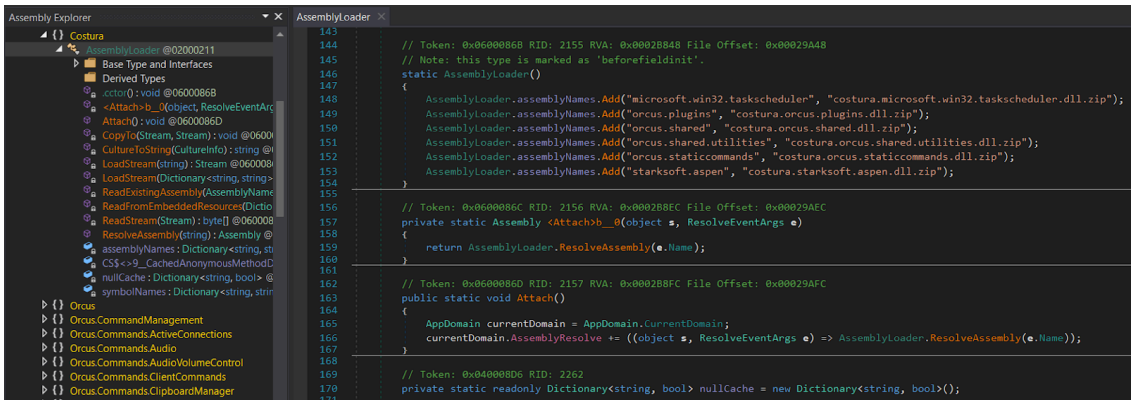


The **AES** class is imported from the **Orcus.Shared.Encryption**. The only problem is that the assembly doesn't contain such a namespace. To find it, we can go to the Orcus RAT resources:



We seem to have found an assembly **orcus.shared.** But what is this **costura** prefix? And why is the assembly stored with a .zip extension? We extracted this resource and tried to unpack it. Unfortunately, it was a miss – despite the .zip extension, this resource is not an archive.

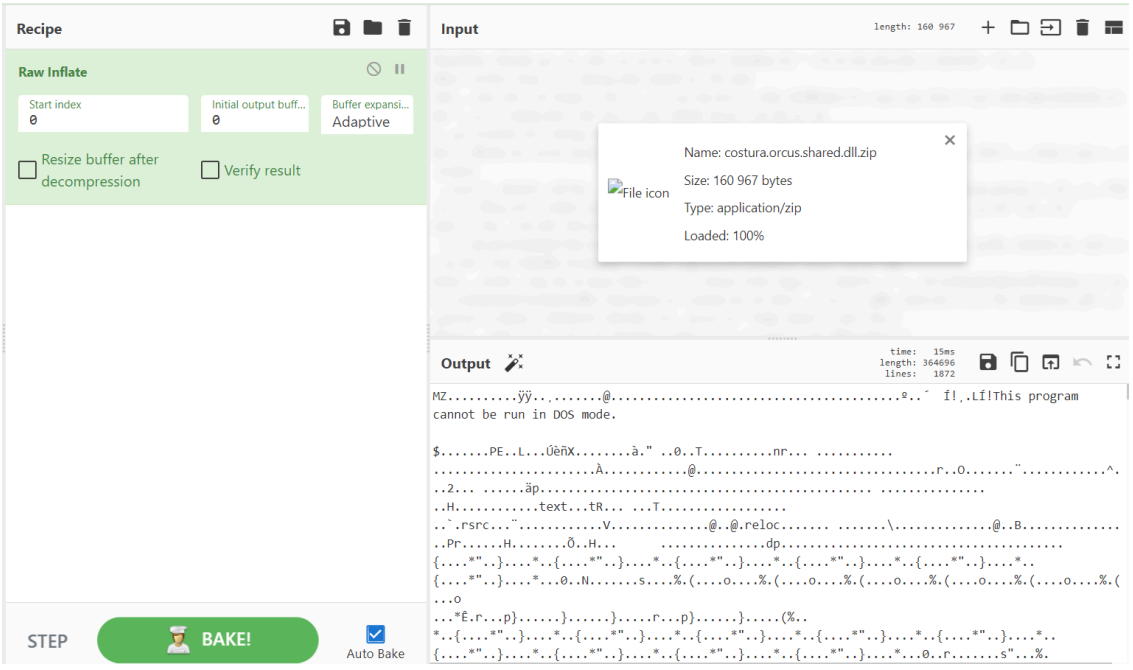
Realizing that, at some point, this assembly must be loaded into the application, we make a decision to look for another place where this happens. Of course, keeping that strange costura prefix in mind. And it didn't take us long – we have found the **Costura** namespace that contains the **AssemblyLoader** class. It is supposed to load the assemblies packed in Orcus resources.



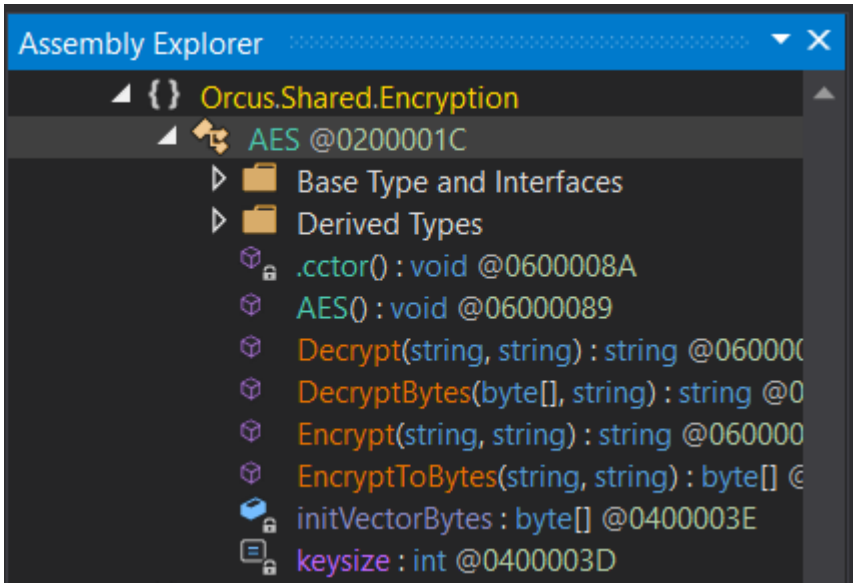
Inside the AssemblyLoader class, we have caught how assemblies are loaded from resources:

```
// Costura.AssemblyLoader
// Token: 0x06000866 RID: 2150 RVA: 0x0002B658 File Offset: 0x00029858
private static Stream LoadStream(string fullname)
{
    Assembly executingAssembly = Assembly.GetExecutingAssembly();
    if (fullname.EndsWith(".zip"))
    {
        using (Stream manifestResourceStream = executingAssembly.GetManifestResourceStream(fullname))
        {
            using (DeflateStream deflateStream = new DeflateStream(manifestResourceStream, CompressionMode.Decompress))
            {
                MemoryStream memoryStream = new MemoryStream();
                AssemblyLoader.CopyTo(deflateStream, memoryStream);
                memoryStream.Position = 0L;
                return memoryStream;
            }
        }
    }
    return executingAssembly.GetManifestResourceStream(fullname);
}
```

After repeating this operation with [CyberChef](), we got an unpacked assembly.



To avoid any second thoughts, we upload the unpacked assembly to DnSpy. Hopefully, it can confirm or deny our assumption about the encryption algorithm used by the Orcus RAT.



This class contains methods for encrypting and decrypting data, as well as an initialization vector field for the AES algorithm and a field with the key length. We are not really interested in the **encryption** process, but the data **decryption** is exactly what we need:

```
public static string Decrypt(string cipherText, string passPhrase)
{
    if (string.IsNullOrEmpty(cipherText))
    {
        return string.Empty;
    }
    return AES.DecryptBytes(Convert.FromBase64String(cipherText), passPhrase);
}
public static string DecryptBytes(byte[] cipherText, string passPhrase)
{
    byte[] bytes = new PasswordDeriveBytes(passPhrase, null).GetBytes(32);
    string @string;
    using (RijndaelManaged rijndaelManaged = new RijndaelManaged())
    {
        rijndaelManaged.Mode = CipherMode.CBC;
        using (ICryptoTransform cryptoTransform = rijndaelManaged.CreateDecryptor(bytes, AES.initVectorBytes))
        {
            using (MemoryStream memoryStream = new MemoryStream(cipherText))
            {
                using (CryptoStream cryptoStream = new CryptoStream(memoryStream, cryptoTransform, CryptoStreamMode.Read))
                {
                    byte[] array = new byte[cipherText.Length];
                    int count = cryptoStream.Read(array, 0, array.Length);
                    @string = Encoding.UTF8.GetString(array, 0, count);
                }
            }
        }
    }
    return @string;
}
```

Orcus RAT data decryption

We have found out the following information concerning data decryption:

1. **Base64** is applied to the encrypted data besides the AES algorithm.
2. The exact encryption type is AES256-CBC.
3. We identified how the encryption key is derived.

Let’s discuss this stage, this one is definitely interesting. To generate the key for a given string, Orcus uses the **PasswordDeriveBytes** class, which is based on the **PBKDF1** algorithm from Microsoft. The malware uses the default settings: it means that the number of iterations for key generation will be 100, and the hashing algorithm will be **SHA1**.

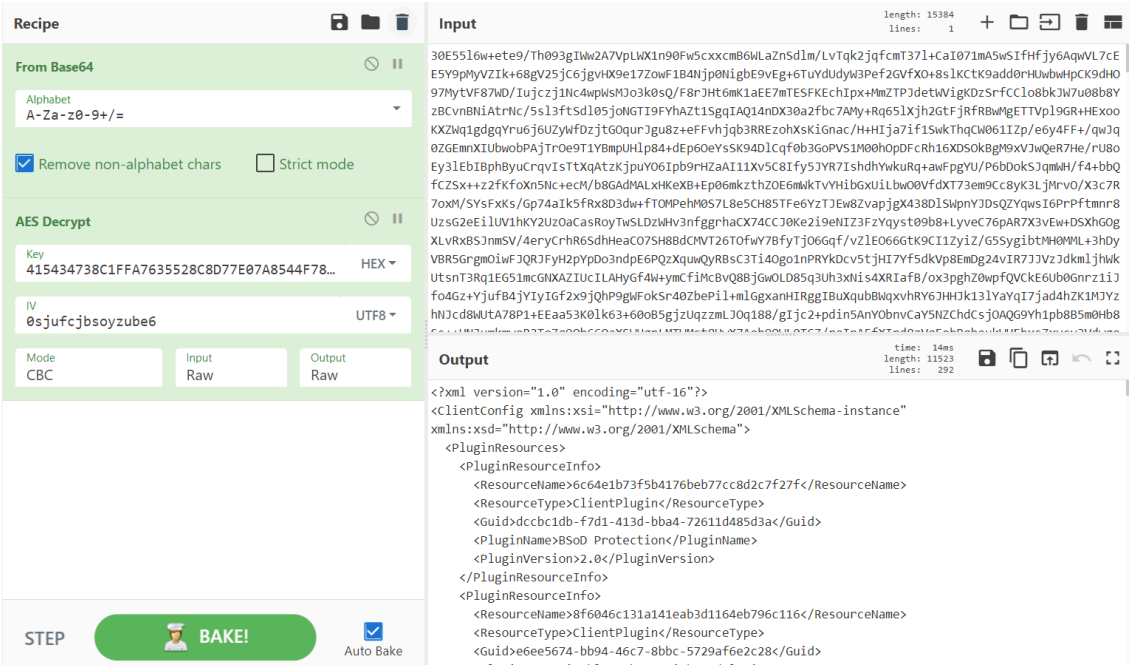
```
public PasswordDeriveBytes(string strPassword, byte[]? rgbSalt, CspParameters? cspParams) :
    this(strPassword, rgbSalt, "SHA1", 100, cspParams) { }
```

Are you wondering how it’s done? Here is a scenario:

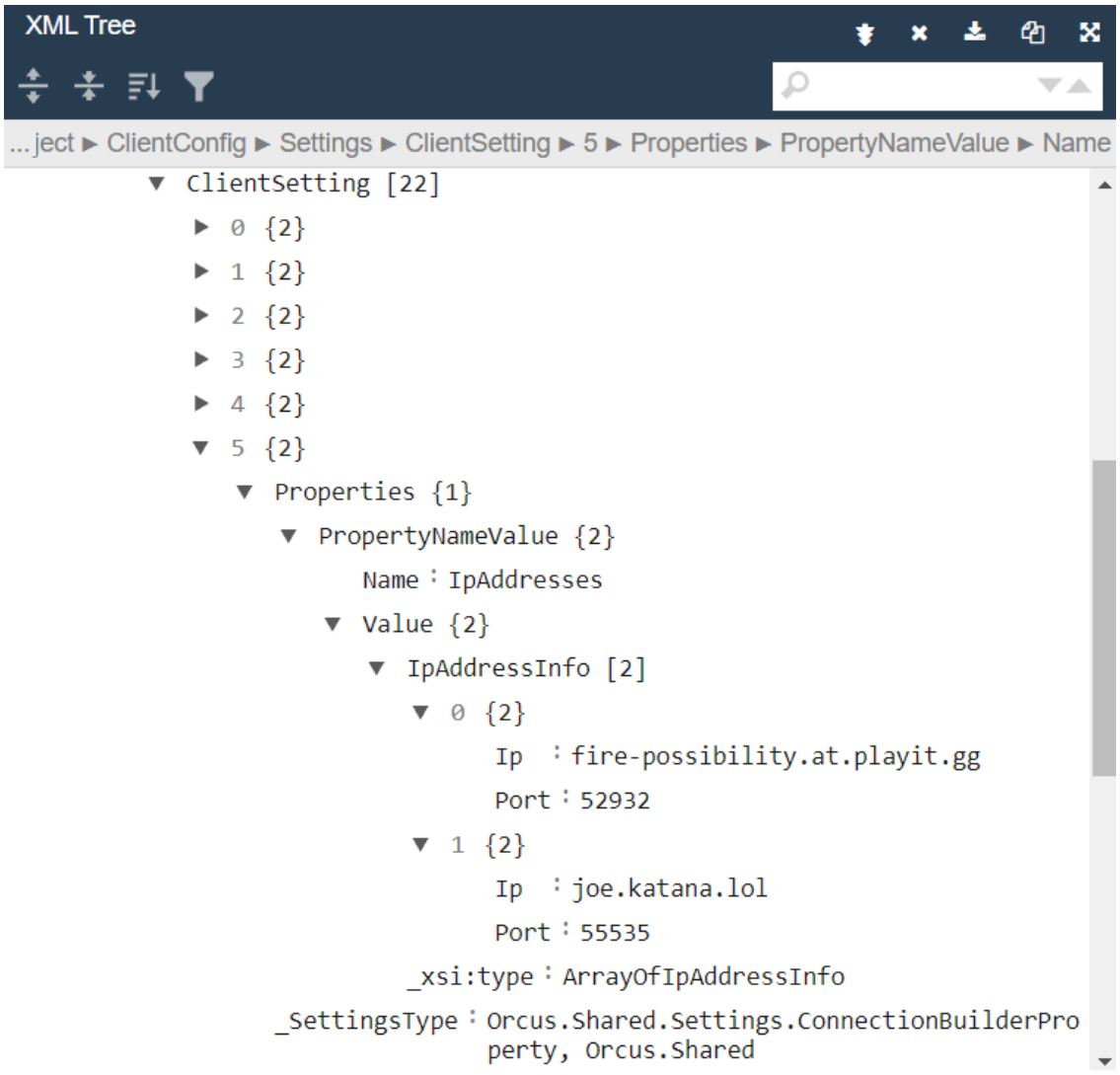
The first 20 bytes proceed as usual, then a byte counter is added to each hashed byte of the inherited string from the 20th to the last byte. Taking it into account, we implemented this in Python:

```
# microsoft's pbkdf1 implementation
def __pbkdf1(self, password, salt, count=32, iterations=100) -> bytes:
    def do_hash(data) -> bytes:
        _sha1 = hashlib.sha1()
        _sha1.update(data)
        data = _sha1.digest()
        return data

    if len(salt) > 0:
        password = password + salt

    for i in range(iterations - 1):
        password = do_hash(password)

    ret = do_hash(password)

    i = 1
    while len(ret) < count:
        ret += do_hash(bytes(str(i), encoding="ascii") + password)
        i += 1

    return ret[:count]
```

Knowing the correct key, you can decrypt the data using CyberChef.



As a result of decoding, we get the malware configuration in the XML format.



Automating the configuration extraction process of Orcus RAT

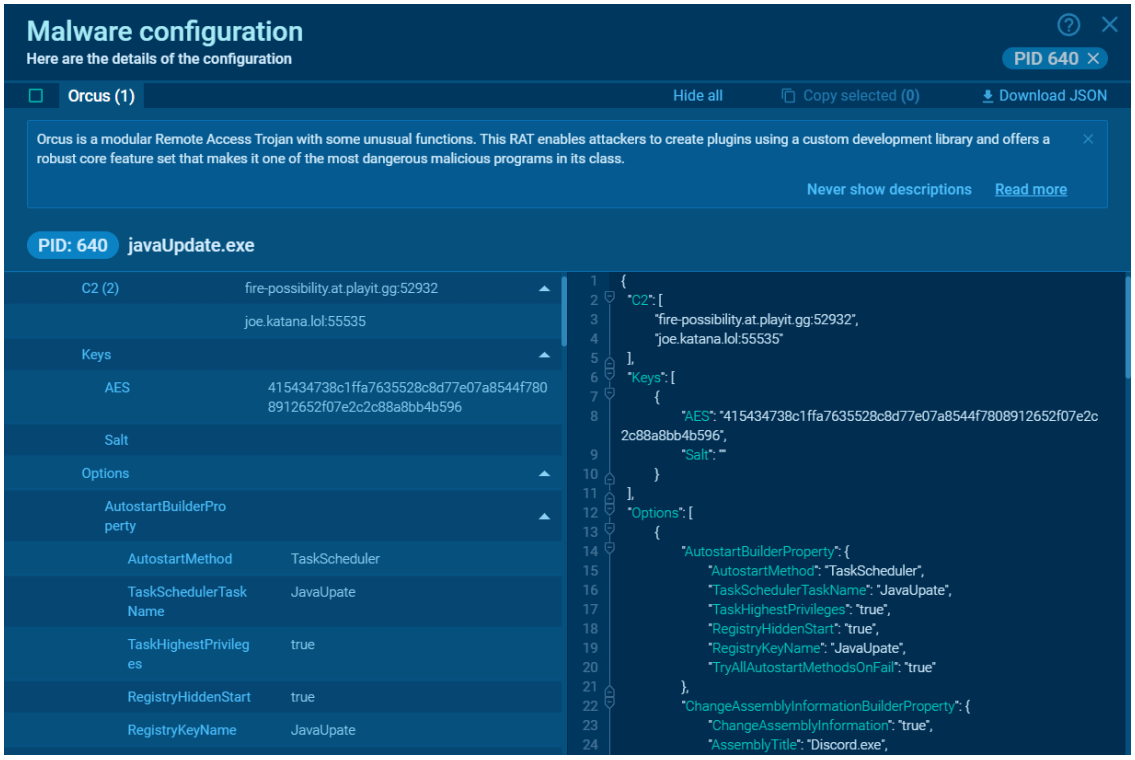
Now, we will write a Python script with the necessary data to decrypt and automate the configuration extraction. After studying some samples, we have seen that the strings with the encrypted data are located one after another in the UserString stream between two other specific UserString objects (the strings “case FromAdministrationPackage.GetScreen” and “klg_”).

Next, using the **dnfile** library, we implement a simple algorithm that iterates through the UserStrings looking for the strings mentioned above. And it’s important to note that the number of received strings between them must be three:

1. The main encrypted configuration of malware
2. The encrypted configuration of the plugins that Orcus uses
3. The key from which the AES key will be generated

```
cfg_data = []
should_collect_data = False
us: dnfile.stream.UserStringHeap = dn.net.metadata.streams.get(b"#US", None)

if us:
    size = us.sizeof()  # get the size of the stream
    offset = 1 # the first entry (the first byte in the stream) is an empty string, so skip it

    while offset < size: # while there is still data in the stream
        # read the raw string bytes, and provide the number of bytes to read (includes the encoded length)
        ret = us.get_with_size(offset)
        if ret is None:
            break

        buf, readlen = ret

        try:
            s = dnfile.stream.UserString(buf) # convert data to a UserString object

            if "klg_" in s.value: break # hit an encapsulating string, leaving..

            if should_collect_data: # collect everything between reference strings
                cfg_data.append(s.value)

            if s.value == "case FromAdministrationPackage.GetScreen":
                should_collect_data = True # we should collect data starting from this line

        except UnicodeDecodeError:
            raise ValueError(f"Bad string: {buf}")

        offset += readlen  # continue to the next entry

if len(cfg_data) != 3:
    raise ValueError("Got invalid cfg data")

return cfg_data
```

You can also always use ANY.RUN service to automatically retrieve the Orcus RAT configuration. It’s a much easier way to analyze a malicious object in a short period of time. For example, the sandbox has already retrieved all data from this Orcus sample, so you can enjoy smooth research.



Conclusion

In this article, we briefly analyzed the Orcus RAT and automated its configuration extraction. The full version of the extractor is available at the link, so don't forget to check it out!

Orcus has become another chapter in our malware analysis series. Read our previous posts about STRRAT and Raccoon Stealer. What should we cover next?

The post blitz survey

- What is Orcus RAT?

Orcus is a Remote Access Trojan that allows attackers to create plugins and offers a robust core feature.

- Where and how does Orcus store additional assemblies?

Orcus RAT stores additional assemblies inside the the malware resources using a 'deflate' algorithm.

- How does Orcus encrypt data?

Orcus RAT encrypts data using the AES algorithm and then encodes encrypted data using Base64.

- How can we decrypt Orcus RAT?

First, you need to generate the key from a given string using Microsoft's PBKDF1 implementation. Second, decode the data from Base64. Finally, apply the generated key to decrypt the data via the AES256 algorithm in CBC mode. As a result of decoding, we get the malware configuration in the XML format.

Reverse Engineer. Malware Analyst at ANY.RUN

hardee

Reverse Engineer, Malware Analyst at ANY.RUN at ANY.RUN | + posts

I contribute to open source from time to time and I am always up for a challenge.

I contribute to open source from time to time and I am always up for a challenge.

Source: https://any.run/cybersecurity-blog/orcus-rat-malware-analysis/