# ArechClient; Decoding IOCs and finding the onboard browser extension

By Jason Reaves

Published: 2025-03-13 · Archived: 2026-05-07 02:19:36 UTC

4 min read

Mar 13, 2025

By: Jason Reaves

Found myself analyzing ArechClient recently and needed a way to automatically decode out the C2 info from lots of samples. By the end of our work we also discovered that the browser extension being delivered by ArechClient is on board the client itself.

The sample I started with:

```
96ba74cc27b44547d23fe1fa550fc59b4f340dbbca8472d9b4698576751bb189
```

A very good technical writeup on ArechClient was done back in 2023 by dr4k0nia[1,2]. Using that information I was able to quickly find the function responsible for decoding strings:

```
        internal static string c(int num, int num2, int num3)
        {
                num += 593;
                Assembly executingAssembly = Assembly.GetExecutingAssembly();
                num2 -= 331;
                Stream manifestResourceStream = executingAssembly.GetManifestResourceStream("resourc
                int num4 = num ^ num2;
                num4 = num4 * 17 / 27;
                manifestResourceStream.Seek((long)(7 + num4), SeekOrigin.Begin);
                byte[] array = new byte[8];
                manifestResourceStream.Read(array, 0, 4);
                int num5 = (BitConverter.ToInt32(array, 0) ^ 2100157544) - 100;
                manifestResourceStream.Read(array, 0, 4);
                int num6 = BitConverter.ToInt32(array, 0) - 5 ^ 485648943;
                manifestResourceStream.Seek((long)num5, SeekOrigin.Begin);
                array = new byte[num6];
                manifestResourceStream.Read(array, 0, num6);
```

```
                for (int i = 0; i < array.Length; i++)
                {
                        array[i] = (byte)((int)array[i] ^ num3);
                }
                return Encoding.UTF8.GetString(array);
        }
```

A single byte XOR, and the offset and key are passed as parameters. Using different keys for various strings makes it a bit annoying for automating but makes no difference in detection using YARA:

```
rule archeclient
{
    strings:
    $a1 = "https://pastebin.com/raw/" xor
    $a2 = "CatalinaGroup" xor
    $a3 = "Cookies" xor
    condition:
    all of them
}
```

While I let that run gathering up more samples, let’s look at writing a decoder.

So the decoding method means if we want to stay static then we’ll either need to delve into decompiling every binary or look at bruting, decompiling them would require a bit much for my framework so I stick to the bruting realm using a regex for the IP and a quick shortcut check for pastebin for the URL:

```
import re
import sys

data = open(sys.argv[1], 'rb').read()
ips = []
for i in range(255):
  temp = bytearray(data)
  for j in range(len(temp)):
    temp[j] ^= i
  t = re.findall(b'''[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}''', temp)
  ips += t

  if b'pastebin' in temp:

    tt = re.findall(b'''https?://[\x20-\x7e]+''', temp)
    ips += tt

print(ips)
```

This works but is very slow because we are bruting files that are just under 1mb in size, speeding this up for finding pastebin atleast we can utilize YARA in python:

```
rule_source = '''
rule archeclient
{
    strings:
    $a1 = "https://pastebin.com/raw/" xor
    $a2 = "CatalinaGroup" xor
    $a3 = "Cookies" xor
    condition:
    all of them
}
'''

def yara_scan(raw_data, rule_name):
    xor_keys = []
    yara_rules = yara.compile(source=rule_source)
    matches = yara_rules.match(data=raw_data)
    for match in matches:
        if match.rule == 'archeclient':
            for item in match.strings:
                if item.identifier == rule_name and len(item.instances) > 0:
                    xor_keys.append((item.instances[0].xor_key, item.instances[0].offset))
    return xor_keys
```

The code above will allow us to recover the the offset and xor key for the pastebin URL much easier, this doesn't help with finding IP addresses though. Since most of the strings are all stored in a giant blob we can use the offset to narrow the scope for bruting the IPs:

```
data = open(sys.argv[1], 'rb').read()
tet = yara_scan(data, '$a1')
if len(tet) > 0:
    ips = []
    rough_off = tet[0][1]
    newdata = data[rough_off-10000:rough_off+10000]
    pb_xor_key = tet[0][0]
    temp = bytearray(newdata)
    for i in range(len(temp)):
        temp[i] ^= pb_xor_key
    tt = re.findall(b'''https?://pastebin.com/raw/[\x20-\x7e]{8}''', temp)
    ips += tt

    for i in range(255):
        temp = bytearray(newdata)
        for j in range(len(temp)):
```

```
            temp[j] ^= i
        t = re.findall(b'''[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}''', temp)
        ips += t
    print(ips)
```

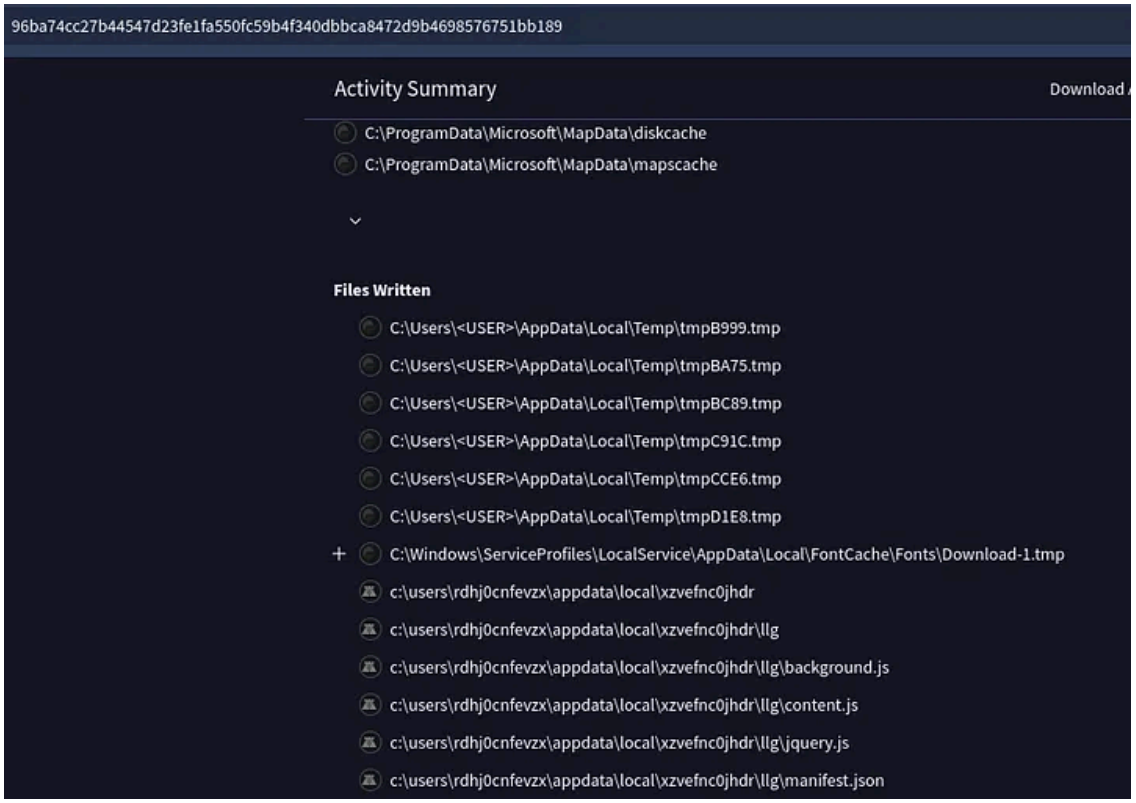
Decoded IOCs for ArechClient samples can be found at the bottom.

## Get Jason Reaves's stories in your inbox



A previous blog detailed that ArechClient was being leveraged for delivering a browser extension[3], looking at the sandbox report for our sample it appears the same thing is happening.

Press enter or click to view image in full size




Since the C2 resides in the content.js we can enumerate samples on VirusTotal:

```
content:{7370796a735f726566726573684576656e747328636c69656e74496429}
```

Taking the first 48 samples found and writing a quick one-liner to rip out the C2s:

```
grep "server =" * |awk -F'"' '{print $2}' |awk -F'/' '{print $3}' |sort |uniq -c
```

However while looking at the pcap for this sample I noticed something strange, for one there was not enough traffic before the extension traffic started to justify downloading the extension from the C2. After checking in the same section of the ArechClient binary we were able to see that it is actually onboard and decoded in the same way as the C2 info earlier but still needs to be base64 decoded:

```
CmFzeW5jIGZ1bmN0aW9uIGh0dHBHZXQodGhlVXJsKQp7CiAgICBsZXQgcmVzcG9uc2UgPSBhd2FpdCBmZXRjaCh0aGVVcmwpOwog
ewoJIm1hbmlmZXN0X3ZlcnNpb24iOiAyLAoKCSJuYW1lIjogIkdvb2dsZSBEb2NzIiwKCSAgICJkZXNjcmlwdGlvbiI6ICJFZGl0
dmFyIHNlcnZlciA9ICJodHRwOi8vXl5eXl4mJiYmJioqKioqKiYmJl5eXiUlJV5eXiZeJV4mJV4mJiZeJiomXiZeJl4mXio6OTAwL
```

Snippet from base64 decoded content.js:

```
var server = "http://^^^^^&&&&&******&&&^^^%%%^^^&^%^&%^&&&^&*&^&^&^&^*:9000/";
var iddd = '$%^&#^&@#@&#@&#&*&*&*^#*@&#*&@#*@&#*#&*@#';

var debug = 1;
var currLoc = "";

(async function () {

        var clientId = iddd;
        urlChangeAllert();

        spyjs_refreshEvents(clientId);

})()

function urlChangeAllert(){
```

## IOCs

Pastebin URLs:

```
https://pastebin.com/raw/DWCCqGB0
https://pastebin.com/raw/ENEhYpTW
https://pastebin.com/raw/Ld9GfkdJ
https://pastebin.com/raw/NH3NwEpf
https://pastebin.com/raw/UPxYyFp8
https://pastebin.com/raw/cLika3dt
https://pastebin.com/raw/eB8bmiVA
https://pastebin.com/raw/eKZZkcvz
https://pastebin.com/raw/gxDS2LkW
https://pastebin.com/raw/hN7U6xHt
https://pastebin.com/raw/nJqnXW3u
https://pastebin.com/raw/ravLcZ9f
```

```
https://pastebin.com/raw/ubFNTPjt
https://pastebin.com/raw/wikwTRQc
https://pastebin.com/raw/yZ0ektdh
```

Pastebin C2s:

```
92.255.57.31
92.255.57.32
144.76.103.92
92.255.85.23
109.120.186.139
45.141.84.208
185.147.124.181
92.255.57.75
45.118.248.29
85.209.11.243
194.26.135.180
91.202.233.18
80.64.30.2
```

IPs from binaries:

```
109.120.186.139
185.147.124.179
185.147.124.181
194.26.135.180
213.109.202.229
45.118.248.29
45.141.84.168
45.141.84.208
45.141.87.124
45.141.87.50
62.84.98.67
80.64.30.2
81.19.135.11
91.202.233.18
92.255.57.31
92.255.57.32
92.255.57.75
92.255.85.23
92.255.85.36
```

Browser Extension package on VT:

```
7b9cc025a1bc19552ec3a69872ddc9dbf1233ab779f081aaf75b902435f6477e
```

Browser Extension C2s:

```
109.107.182.209:9000
109.234.34.3:9000
144.76.163.55:9000
152.89.198.51:9000
176.9.66.115:9000
178.63.51.126:9000
185.147.124.181:9000
185.42.12.247:9000
185.42.12.64:9000
185.42.12.85:9000
185.73.125.96:9000
213.109.202.15:9000
34.141.167.33:9000
34.89.247.212:9000
45.118.248.29:9000
45.141.84.208:9000
45.141.87.124:9000
45.141.87.215:9000
45.141.87.55:9000
45.88.104.78:9000
45.92.179.249:9000
5.42.67.10:9000
77.246.107.149:9000
80.64.30.2:9000
81.19.135.11:9000
91.202.233.18:9000
91.215.85.23:9000
91.215.85.66:9000
91.240.118.154:9000
92.255.57.31:9000
92.255.57.75:9000
92.255.85.23:9000
95.216.24.238:9000
```

YARA:

```
rule archeclient
{
    strings:
    $a1 = "https://pastebin.com/raw/" xor
    $a2 = "CatalinaGroup" xor
```

```
    $a3 = "Cookies" xor
    condition:
    all of them
}

rule spyjs_extension
{
strings:
$a1 = "spyjs_saveData"
$a2 = "spyjs_refreshEvents"
condition:
all of them
}
```

References

1: https://dr4k0nia.github.io/posts/Analysing-a-sample-of-ArechClient2/

2: https://www.sentinelone.com/blog/reverse-engineering-walkthrough-analyzing-a-sample-of-arechclient2/

3: https://malwr-analysis.com/2025/02/18/arechclient2-malware-analysis-sectoprat/

---