Downeks and Quasar RAT Used in Recent Targeted Attacks Against Governments

By Mashav Sapir, Tomer Bar, Netanel Rimer, Taras Malivanchuk, Yaron Samuel, Simon Conant

Published: 2017-01-30 · Archived: 2026-04-05 20:22:15 UTC

Palo Alto Networks Traps Advanced Endpoint Protection recently prevented recent attacks that we believe are part of a campaign linked to DustySky. DustySky is a campaign which others have attributed to the Gaza Cybergang group, a group that targets government interests in the region.

This report shares our researchers' analysis of the attack and Remote Access Tool (RAT). We also discovered during our research that the RAT Server used by this attacker is itself vulnerable to remote attack, a double-edged sword for these attackers.

Attack

The initial infection vector in this attack is not clear, but it results in installing the "Downeks" downloader, which in turn infects the victim computer with the "Quasar" RAT.

Downeks uses third party websites to determine the external IP of the victim machine, possibly to determine victim location with GeoIP. It also drops decoy documents in an attempt to camouflage the attack.

Quasar is a .NET Framework-based open-source RAT. The attackers invested significant effort in attempting to hide the tool by changing the source code of the RAT and the RAT server, and by using an obfuscator and packer.

Detection

Unit 42 researchers observed the Quasar RAT being prevented from executing on a Traps-protected client in September 2016. We observed these Quasar samples:

File Name: f-secure.exe
SHA256: 99a7cb43fb2898810956b6137d803c8f97651e23f9f13e91887f188749bd5e8f
Note: connects to hnoor.newphoneapp[.]com

File Name: HD_Audio.exe
SHA256: 0c4aa50c95c990d5c5c55345626155b87625986881a2c066ce032af6871c426a
Note: connects to manual.newphoneapp[.]com

File Name: HD_Audio.exe
SHA256: 86bd78b4c8c94c046d927fb29ae0b944bf2a8513a378b51b3977b77e59a52806
Note: crashes upon execution

File Name: sim.exe
SHA256: 723108103ccb4c166ad9cdff350de6a898489f1dac7eeab23c52cd48b9256a42
Note: connects to hnoor.newphoneapp[.]com

Further research found other Quasar examples, an attack earlier in the month 2016 on the same target:

SHA256: 1ac624aaf6bbc2e3b966182888411f92797bd30b6fcce9f8a97648e64f13506f

We found the same Quasar code in an additional attack on the same day, but upon a different target. A second Quasar sample was also observed attacking this new victim:

SHA256: 99a7cb43fb2898810956b6137d803c8f97651e23f9f13e91887f188749bd5e8f

We do not have detailed visibility into the specific host attacked, and have not been able to reproduce the second stage of the attack in our lab. However, based upon the timeframe of subsequent telemetry we observe, we understand the attack chain as follows:

1. The initial dropper (which varies across attacks) is delivered to the victim via email or web:

File Name: Joint Ministerial Council between the GCC and the EU Council.exe"
SHA256: 0d235478ae9cc87b7b907181ccd151b618d74955716ba2dbc40a74dc1cdfc4aa

2. The initial dropper, upon execution, extracts an embedded Downeks instance:

File Name: ati.exe
SHA256: f19bc664558177b7269f52edcec74ecdb38ed2ab9e706b68d9cbb3a53c243dec

3. Downeks makes a POST request to dw.downloadtesting[.]com, resulting in the installation of the Quasar RAT on the victim machine.
4. Additional Downeks downloaders connecting to the previously-observed server dw.downloadtesting[.]com were also found in this attack:

SHA256: 15abd32342e87455b73f1e2ecf9ab10331600eb4eae54e1dfc25ba2f9d8c2e8a

SHA256: 9a8d73cb7069832b9523c55224ae4153ea529ecc50392fef59da5b5d1db1c740

Further research identified dozens of Dowenks and Quasar samples related to these attackers. All included decoy documents written in Arabic (all related to Middle Eastern politics) or Hebrew. Most of them use the same mutex structure, share the same fake icon and unique metadata details, file writes, registry operations, and fake common program metadata, as seen in DustySky samples.

The Downeks downloader and Quasar C2 infrastructures are each self-contained and independent of each other. However, we did find a single shared IP address demonstrably connecting the Downeks downloader and Quasar C2 infrastructure s. The below chart (Figure 1) shows Quasar infrastructure (top), Downeks (bottom), and the shared IP link.

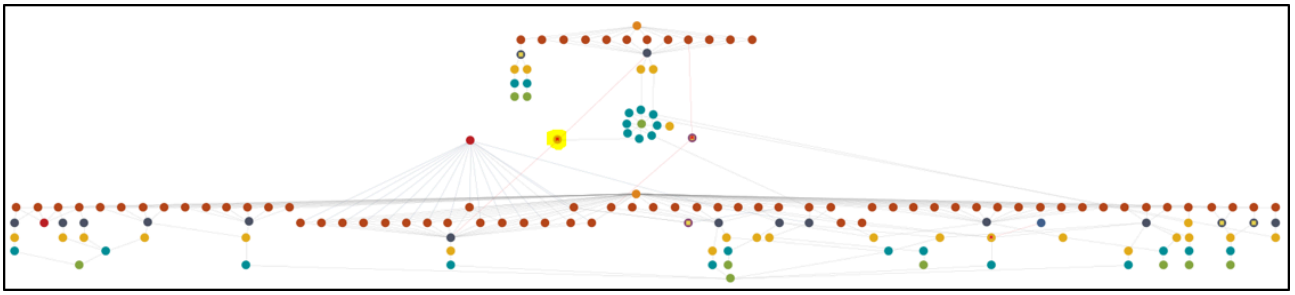

Figure 1- Quasar and Downeks

Charting the samples and infrastructure clearly shows the separate Downeks campaigns, and infrastructure links (Figure 2):

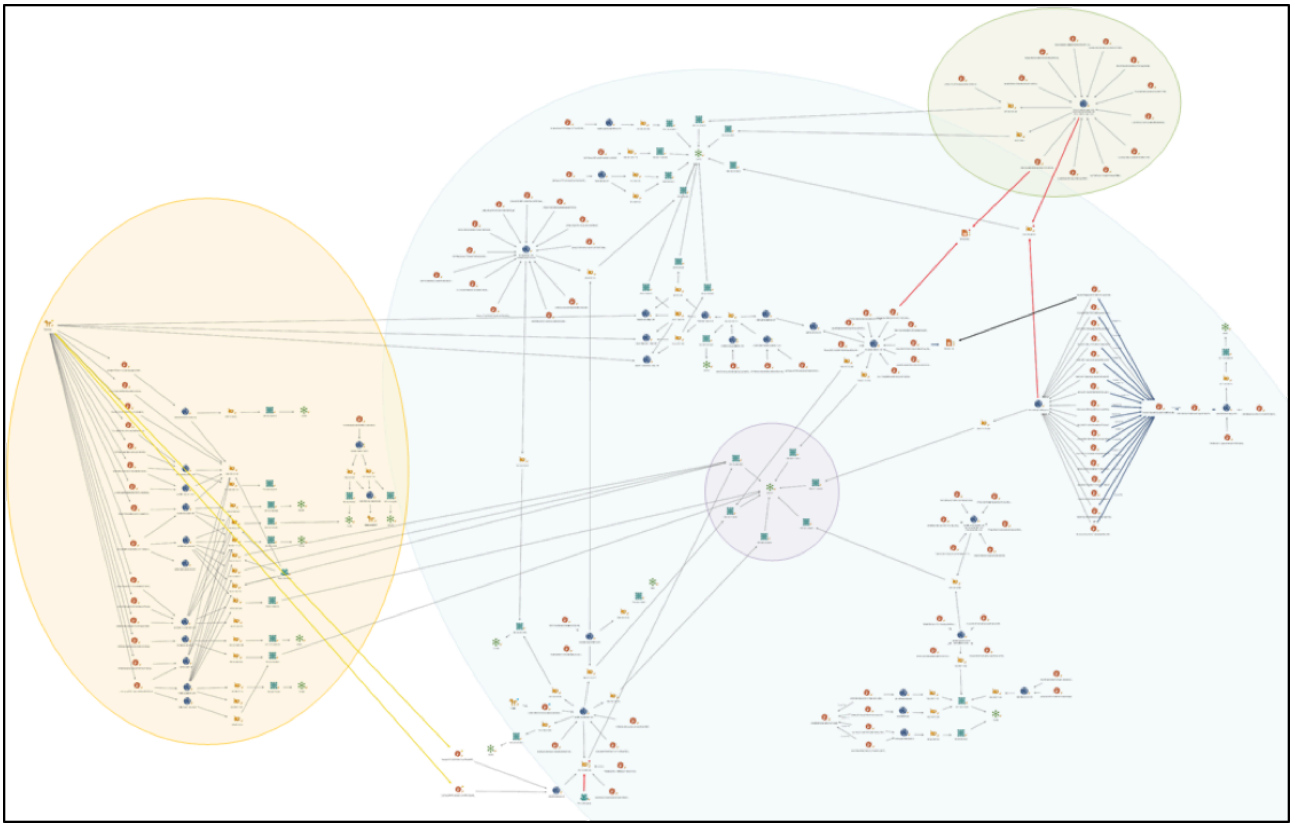

Figure 2- Infrastructure Patterns and Connections

In Figure 2, top-right (green) has the Quasar infrastructure (Figure 3), with a link to the Downeks infrastructure. Left (yellow) is DustySky infrastructure (Figure 4) and the links to this Downeks campaign. As well as similarities in the code, decoys and targets, we also identified C2 infrastructure links between DustySky and this campaign. The remainder is sub-campaigns of Downeks samples, their infrastructure, their links – and a favored ISP (center) (Figure 5).

Figure 3- Quasar

Figure 4- DustySky

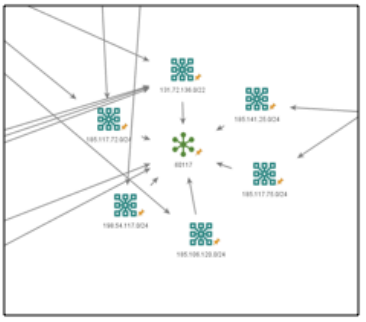

Figure 5- Favorite ISP

The timing of the attacks is commensurate with the Middle-Eastern working week (Figure 6):

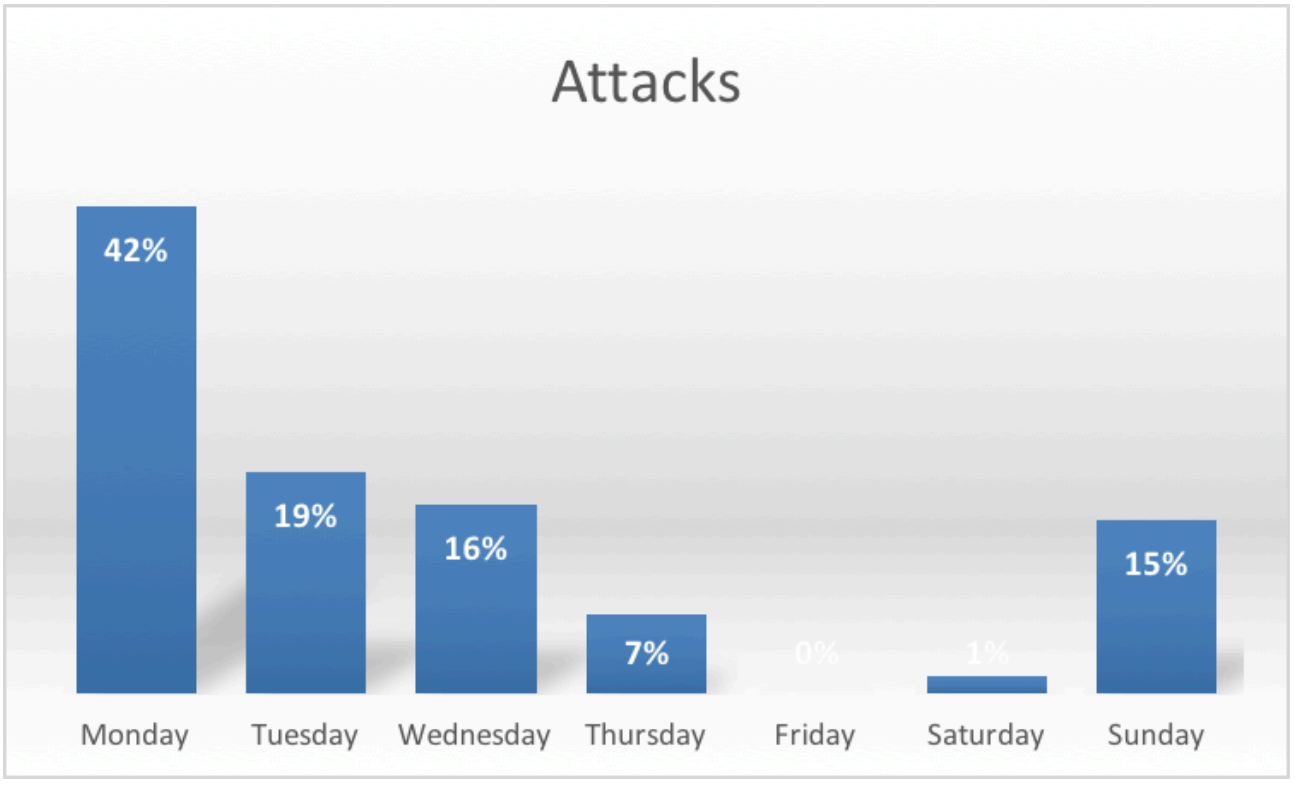

Figure 6- Attacks by day-of-the-week

The sample build days-of-the-week follow an almost identical pattern (Figure 7):

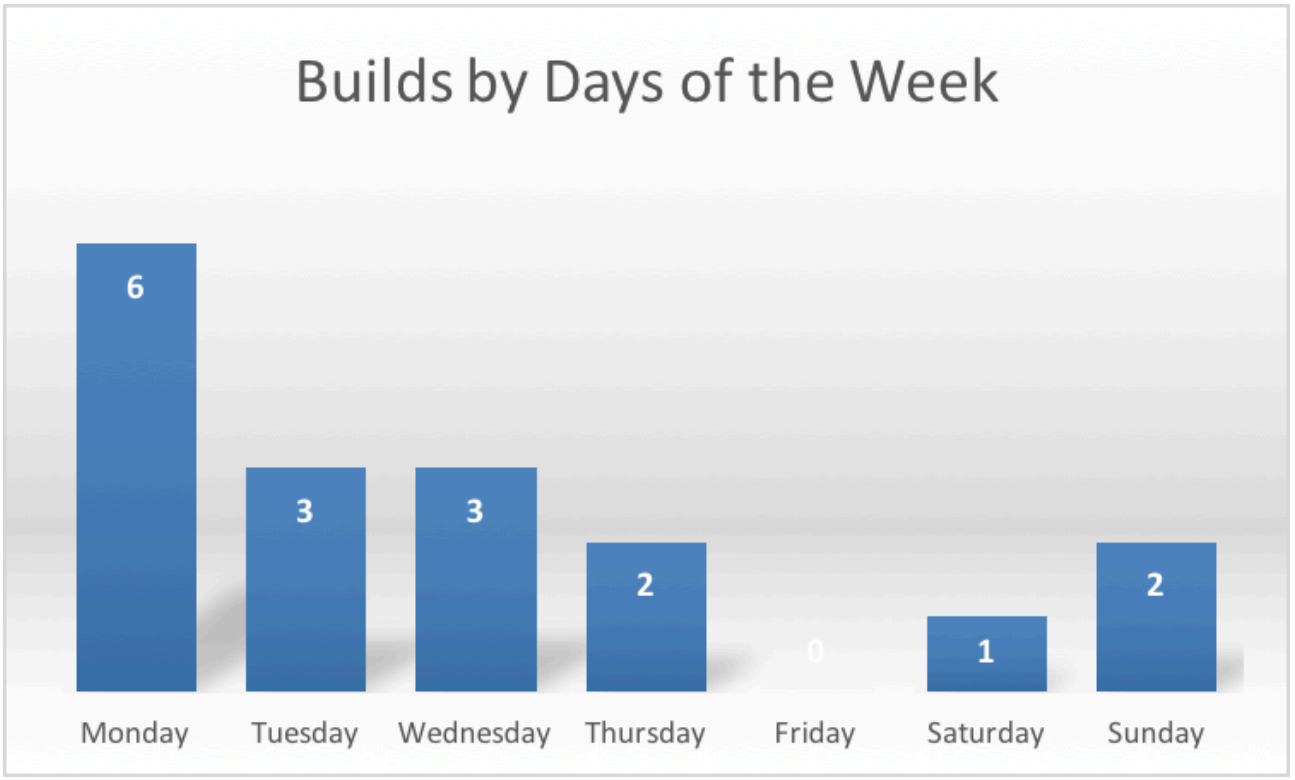

Figure 7- Builds by day-of-the-week

We saw five samples built on the same date in December 2015, and six on the same date in January, further solidifying the link between each sample.

Quasar

We analyzed a Quasar sample we found that was communicating with an active C2 server at the time of analysis:

SHA256: 4393ff391396cdfd229517dd98aa7faecad04da479fe8ca322f035ceee363273

Quasar is a publicly-available commodity RAT, an evolution of his earlier xRAT, by German developer "MaxXor". This sample is a modified version of Quasar, most likely forked from open source version 1.2.0.0 on GitHub. The client was likely built using the Quasar server client builder. We observed the following customizations:

C2 server:

app.progsupdate[.]com, which resolved to 185.141.25[.]68), over port 4664.

Quasar mutex name:

VMFvdCsC7RFqerZinfV0sxJFo

Keylogger log location:

Users\hJTQwqwwSCkZU\AppData\Roaming\GoogleDesktop\<date>

The malware uses fake version information to appear as a Microsoft update program, as well as Google Desktop once unpacked.

Property	Value
FileDescription	updata microsoft programe
FileVersion	3.10.2.0
InternalName	micr data
LegalCopyright	microsoft C 2011
OriginalFilename	micr data
ProductVersion	3.22.1.20
Assembly Version	3.22.1.20
CompanyName	microsoft
ProductName	microsoft data

Packer

This sample is packed by “Netz”, a simple .NET Framework packer which stores the original executable compressed (zlib) as a resource. At runtime, the packer decompresses the resource and uses Reflection to load the assembly, find its Entry point, and Invoke it. Extracting the payload is straight forward – we simply dump the resource and decompress it. After decompilation, the packer looks like this:

```
public static int Main(string[] args) {
try {
NetzStarter.InitXR();
AppDomain.CurrentDomain.AssemblyResolve += new
ResolveEventHandler(NetzStarter.NetzResolveEventHandler);
return NetzStarter.StartApp(args);
```

Find the resource and call InvokeApp:

```
public static int StartApp(string[] args) {
byte[] resource = NetzStarter.GetResource("A6C24BF5-3690-4982-887E-
```

```
11E1B159B249");
return NetzStarter.InvokeApp(NetzStarter.GetAssembly(resource), args);
```

Get the assembly object by decompressing the resource and loading it with Reflection:

```
private static Assembly GetAssembly(byte[] data) {
MemoryStream memoryStream = (MemoryStream) null;
memoryStream = NetzStarter.UnZip(data);
memoryStream.Seek(0L, SeekOrigin.Begin);
return Assembly.Load(memoryStream.ToArray());
}
```

And finally, find the entry point and invoke it:

```
private static int InvokeApp(Assembly assembly, string[] args) {
MethodInfo entryPoint = assembly.EntryPoint;
ParameterInfo[] parameters1 = entryPoint.GetParameters();
object[] parameters2 = (object[]) null;
if (parameters1 != null && parameters1.Length > 0)
parameters2 = new object[1]{ (object) args };
object obj = entryPoint.Invoke((object) null, parameters2);
```

Extracting produces:

SHA256: c931de65d9655a772d23e4227a627a1140d8d3c4912ca71c324421b13efa1a02

Property	Value
Comments	
CompanyName	Google
FileDescription	Google Desktop was a computer program with desktop search ...
FileVersion	5.0.0.0
InternalName	Google Desktop
LegalCopyright	©2016 Google
LegalTrademarks	©2016 Google
OriginalFilename	Google Desktop
ProductName	Google Desktop
ProductVersion	5.9.1005.12335

This layer uses obfuscation in an attempt to avoid detection/analysis.

Obfuscation

We discovered that the sample was obfuscated using .NET reactor. It is possible to decompile the deobfuscated sample and retrieve most of the original source code but not enough to compile it easily.

After deobfuscation we extracted:

SHA256: d773b12894d4a0ffb0df328e7e1aa4a7112455e88945a10471650e503eecdb3d

Property	Value
Comments	
CompanyName	Google
FileDescription	Google Desktop was a computer program with desktop search ...
FileVersion	5.0.0.0
InternalName	Google Desktop
LegalCopyright	©2016 Google
LegalTrademarks	©2016 Google
OriginalFilename	Google Desktop
ProductName	Google Desktop
ProductVersion	5.9.1005.12335

Quasar Code

After decompiling the sample, we were able to document the modifications from the open-source Quasar.

Settings

The configuration of Quasar is stored in the Settings object, which is encrypted with a password which is itself stored unencrypted.

```
1  public static class Settings {
2  public static string VERSION;
3  public static string HOSTS;
4  public static int RECONNECTDELAY;
5  public static string PASSWORD;          // password for encryption of communication
6  public static Environment.SpecialFolder SPECIALFOLDER;
7  public static string DIR;
8  public static string SUBFOLDER;
9  public static string INSTALLNAME;
10 public static bool INSTALL;
```

```
11  public static bool STARTUP;
12  public static string MUTEX;
13  public static string STARTUPKEY;
14  public static bool HIDEFILE;
15  public static bool ENABLELOGGER;
16  public static string ENCRYPTIONKEY;     // Encryption password of the settings
17  public static string TAG;
18  public static string LOGDIRECTORYNAME;
19  public static bool HIDELOGDIRECTORY;
20  public static bool ISCHECKIP;
21  public static int INSTARTUPFOLDER;
```

Modifications:

- The ISCHECKIP and INSTARTUPFOLDER are not found in open source Quasar samples.

Cryptography

The sample we analyzed is using RijndaelManaged with ECB mode and PKCS7 padding. The key is the SHA256 hash of the hard-coded password. The password of the sample we analyzed is:

“6y7u^Y&U6y7u^Y&U6y7u^Y&U”

Although at first glance this appears somewhat complex, it is in fact a rather simple, repeated keyboard sequence. We observe similar keyboard patterns in other samples: “567%^&”, “zxc!@#ASD”.

```
1  public static void SetDefaultKey(string key) {
2  byte[] bytes = Encoding.UTF8.GetBytes(key);
3  AES._defaultKey = SHA256.Create().ComputeHash(bytes);
4  private static void EncDec(Stream src, Stream ds, bool encDec, byte[] key) {
5  RijndaelManaged rijndaelManaged = new RijndaelManaged();
6  rijndaelManaged.Key = key;
```

```
7   rijndaelManaged.Mode = CipherMode.ECB;
8   rijndaelManaged.Padding = PaddingMode.PKCS7;
9   rijndaelManaged.BlockSize = AES.BlockSize * 8;
10  if (encDec) {
11  ICryptoTransform encryptor = rijndaelManaged.CreateEncryptor();
12  CryptoStream cryptoStream = new CryptoStream(ds, encryptor, CryptoStreamMode.Write);
13  long position = src.Position;
14  AES.CopyTo(src, (Stream) cryptoStream, 8192);
15  cryptoStream.FlushFinalBlock();
16  } else {
17  ICryptoTransform decryptor = rijndaelManaged.CreateDecryptor();
18  AES.CopyTo(new CryptoStream(src, decryptor, CryptoStreamMode.Read), ds, 8192);
19  }
20  }
21
```

Modifications:

- Uses SHA256 instead of MD5 to create the key.
- Uses RijndaelManaged instead of AES for encryption. (with ECB mode, which is considered weak).

Serialization

Quasar contains the NetSerializer library that handles serialization of high level IPacket objects that the client and server use to communicate. The serialization assigns unique IDs for serializable objects types. The open source and several other samples we found give a dynamically-assigned 1 byte ID at compile time. The sample we analyzed changed that behavior and hard-coded DWORD for each object type. This is a better implementation, as it allows servers and clients from different versions to communicate with each other to some extent.

```
    private static void initTypeMap() {
    Exts.dict_0.Add(typeof (object), -737641570);
    Exts.dict_0.Add(typeof (GetPasswordsResponse), -692037318);
```

```
Exts.dict_0.Add(typeof (List<string>), 1046249082);

Exts.dict_0.Add(typeof (int), -118636331);

Exts.dict_0.Add(typeof (string[]), -2103720204);

Exts.dict_0.Add(typeof (string), 1236129805);
```

Version

The sample we analyzed is most likely forked from open source quasar 1.2.0.0. We find multiple file/object names hinting at the version, but must compelling:

- Quasar version 1.1.0.0 names the encryption module name space "Encryption", while subsequent Quasar versions use "Cryptography" – which we observe in this sample.
- Quasar version 1.3.0.0 changed the encryption key generation, and stopped saving the password in the sample. There are more indications as well, such as names of objects, files etc.

Other samples we analyzed had different combinations of modification to cryptography and serialization.

The C2 server

Our decompilation of the serialization library was not complete enough to allow simple recompilation. Instead, we downloaded and compiled the 1.2.0.0 server of the open-source Quasar RAT, having determined that this seemed likely the most similar version. The out-of-the-box server could not communicate with the client sample owing to the previously documented modifications that we had observed. We incorporated those changes into our build, discovering that this worked for most sample versions with almost no further modification.

Both the client and the server use the same code to serialize and encrypt the communications. Instead of compiling a different server for each client, our server uses the code from within the client to communicate with it. Using Reflection, the server can load the assembly of the client to find the relevant functions and passwords.

Load the client assembly:

```
private static System.Reflection.Assembly assembly =

System.Reflection.Assembly.LoadFile(@"C:\Users\user1\Desktop\Quasar\ServerVersionLo

adClient\resource.bin.open.exe");
```

Encryption:

Rather straight forward, as the server version uses the same API as the sample client.

Get the AES class:

```
private static Type tAES = assembly.GetType("_.Cr.Crp.AES");
```

Getting the setDefKey, encrypt and decrypt methods:

```
private static System.Reflection.MethodInfo[] mi = tAES.GetMethods();
private static System.Reflection.MethodInfo setDefKey = mi[1]; // this one is used
to set the current encryption key (IE sha256 of the password stored in Settings)
//tAES.GetMethod("Encrypt"); doesn't work, because its ambiguous as it is overridden, so I choose the right ones directly
private static System.Reflection.MethodInfo encMIBuf = mi[4];
private static System.Reflection.MethodInfo decMIBuf = mi[6];
```

Replace the server functions:

```
public static void SetDefaultKey(string key)
```

with

```
public static void setDefKey.Invoke(null, new object[] { key });
```

```
public static byte[] Encrypt(byte[] input)
```

with

```
public static byte[] encdata = (byte[])encMIBuf.Invoke(null, new object[] { input
});
```

```
public static byte[] Decrypt(byte[] input)
```

with

```
public static byte[] data = (byte[])decMIBuf.Invoke(null, new object[] { input });
```

Serialization:

This was more complex. Both the client and server uses the same API, but the client serializer cannot serialize server objects, because they are not the same as their "mirrored" objects inside the client. In some cases these objects are completely different, for example the server commands to get the file system.

Our solution is to:

1. Translate on the fly the objects the server send to mirrored matching client objects (will not work if client doesn't have this object, or renamed it).
2. Copy the content from the server object into the new client object (will not work if client implementation is different).
3. Serialize the client object (which will be later encrypted and sent).
4. Deserialize the decrypted response into another client response object.
5. Translate the client response object into the server version of the client response object.
6. Copy the contents from the client response object into the translated server object.
7. Return the translated object.

```
1  public static void SerializeWrapper(Stream stream, object data)
2  {
3  System.Reflection.Assembly assembly =
4  System.Reflection.Assembly.LoadFile(@"C:\Users\user1\Desktop\Quasar\ServerVersionL
5  oadClient\resource.bin.open.exe");
6  Type serializerType = assembly.GetType("_.Cr.NetSerializer.Serializer");
7  System.Reflection.PropertyInfo serInstanceProp = serializerType.GetProperty("Instace");
8  object serInstance = serInstanceProp.GetGetMethod().Invoke(null, new object[] { });
9  System.Reflection.MethodInfo serializeMet = serializerType.GetMethod("Serialize");
10 Type typeOfData = data.GetType(); string typeOfDataFullName = typeOfData.FullName;
11 string typeOfDataFullNameNew = typeOfDataFullName.Replace("xServer.Core", "_.Cr");
12 Type packType = assembly.GetType(typeOfDataFullNameNew);
   object pacTypeInstance = packType.GetConstructor(new Type[] { }).Invoke(new object[] { });
```

```
13  // now try to copy data into the instance
14  foreach (FieldInfo fieldOfClient in packType.GetFields())
15  {
16  string fieldName = fieldOfClient.Name;
17  FieldInfo fieldOfServer = typeOfData.GetField(fieldName);
18  PropertyInfo PropOfServer = typeOfData.GetProperty(fieldName);
19  object serverValue = null;
20  if (fieldOfServer != null) {
21  serverValue = fieldOfServer.GetValue(data);
22  } else if (PropOfServer != null) {
23  serverValue = PropOfServer.GetValue(data,null);
24  } fieldOfClient.SetValue(pacTypeInstance, serverValue);
25  }
26  foreach (PropertyInfo fieldOfClient in packType.GetProperties())
27  {
28  string fieldName = fieldOfClient.Name;
29  FieldInfo fieldOfServer = typeOfData.GetField(fieldName);
30  PropertyInfo PropOfServer = typeOfData.GetProperty(fieldName);
31  object serverValue = null;
32  if (PropOfServer != null) {
33  serverValue = PropOfServer.GetValue(data, null);
34  }
35  else if (fieldOfServer != null) {
36  serverValue = fieldOfServer.GetValue(data);
37  }
38  fieldOfClient.SetValue(pacTypeInstance, serverValue,null);
```

```
39  }
40  serializeMet.Invoke(serInstance, new object[] { stream, pacTypeInstance });
41  }
42  public static object DeserializeWrapper(Stream stream) {
43  System.Reflection.Assembly assembly =
44  System.Reflection.Assembly.LoadFile(@"C:\Users\user1\Desktop\Quasar\ServerVersionL
45  oadClient\resource.bin.open.exe");
46  Type serializerType = assembly.GetType("_.Cr.NetSerializer.Serializer");
47  System.Reflection.PropertyInfo serInstanceProp = serializerType.GetProperty("Instace");
48  object serInstance = serInstanceProp.GetGetMethod().Invoke(null, new object[] { });
49  System.Reflection.MethodInfo DeserializeMet = serializerType.GetMethod("Deserialize");
50  object ob = DeserializeMet.Invoke(serInstance, new object[] { stream });
51  Type typeOfPacket = ob.GetType(); string typeOfPacketFullName = typeOfPacket.FullName;
52  string typeOfPacketFullNameNew = typeOfPacketFullName.Replace("_.Cr", "xServer.Core");
53  System.Reflection.Assembly currentAssembly = Assembly.GetExecutingAssembly();
54  Type packTypeServ = currentAssembly.GetType(typeOfPacketFullNameNew);
55  object pacTypeInstance = packTypeServ.GetConstructor(new Type[] { }).Invoke(new object[] { });
56  // now try to copy data into the instance
57  foreach (FieldInfo fi in typeOfPacket.GetFields()) {
58  string fieldName = fi.Name;
59  FieldInfo fiServ = packTypeServ.GetField(fieldName);
60  if (fiServ != null) {
61  object clientSentValue = fi.GetValue(ob);
62  fiServ.SetValue(pacTypeInstance, clientSentValue);
63  }
64  }
    foreach (PropertyInfo fi in typeOfPacket.GetProperties()) {
```

```
65  string fieldName = fi.Name;
66  PropertyInfo fiServ = packTypeServ.GetProperty(fieldName);
67  if (fiServ != null) {
68  object clientSentValue = fi.GetValue(ob,null);
69  fiServ.SetValue(pacTypeInstance, clientSentValue,null);
70  }
71  }
72  return pacTypeInstance;
73  }
74
75
```

Communication

Our sample communicates with app.progsupdate[.]com, which resolved to 185.141.25[.]68, over TCP port 4664.

Architecture

This is the communication architecture between quasar client and server (Figure 8):

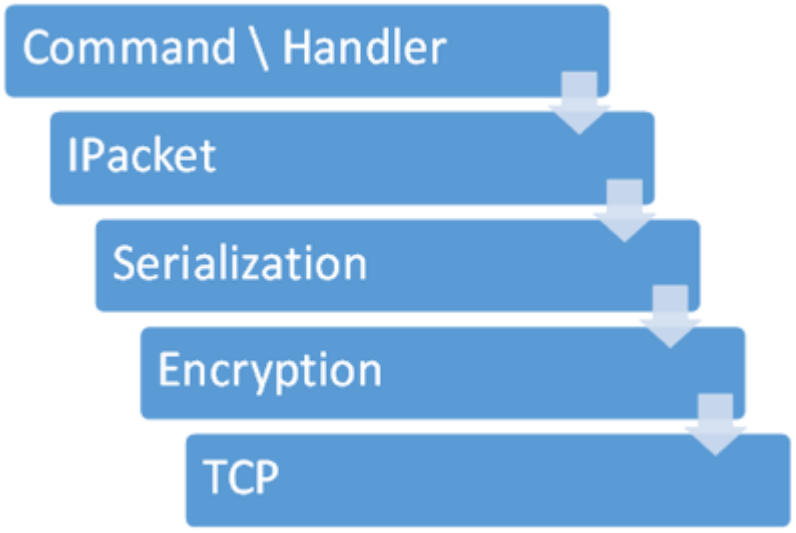

Figure 8- Communication Architecture

1. The server sends a command. for example, “Get System Information”.
2. The command is translated to an IPacket of type GetSystemInfo.

3. The packet is serialized into a stream of bytes.
4. The stream of bytes is encrypted (in some versions there is also optional compression step).
5. The stream of bytes is sent over TCP to the client.
6. The client receives and decrypts the packet.
7. The client deserializes the packet into IPacket GetSystemInfo.
8. The relevant handler of the client is called, collects the system information and sends it back inside IPacket of GetSystemInfoResponse.

Each of these layers seems to be different to some extent in the various samples we found. The IPacket, Serialization and Encryption framework code is shared between the client and the server, therefore we can use it with Reflection. However the Server handlers and command function are not, so we cannot create a completely perfect simulation.

Initial handshake

After the TCP handshake completes, the server starts another handshake with the client by sending packets in the following order (Figure 9):

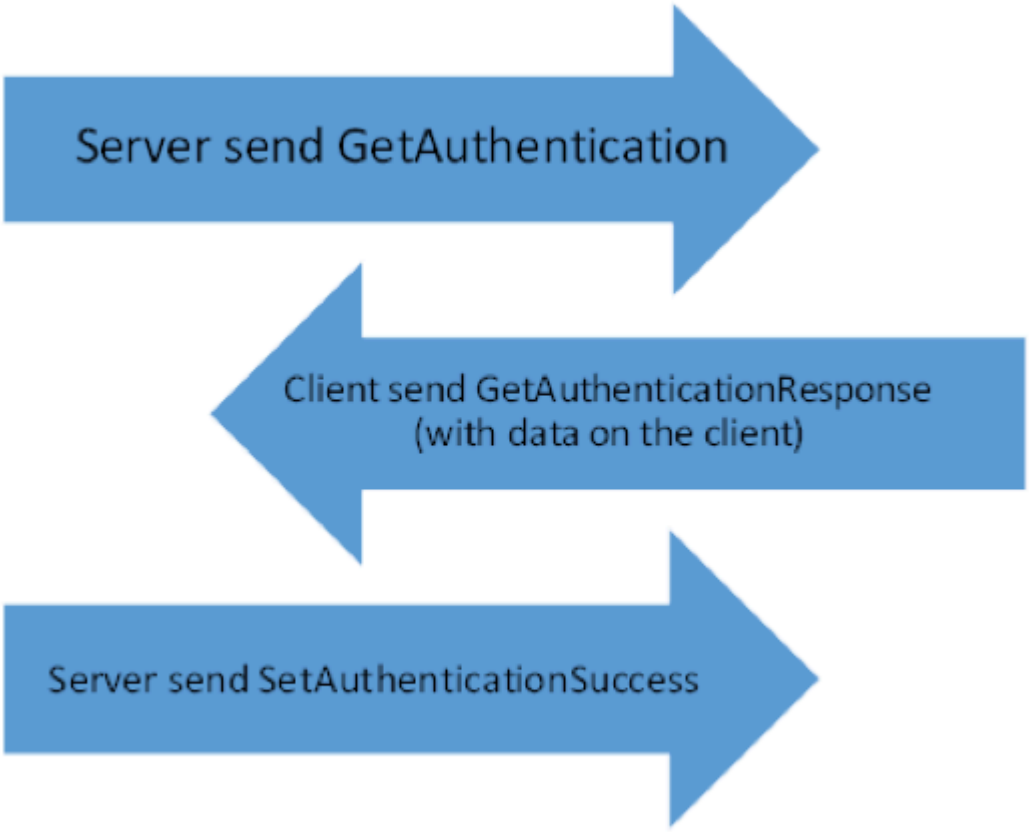

Figure 9- Initial Handshake

The client returns data to the server about the victim computer, which is displayed in the server GUI (Figure 10):

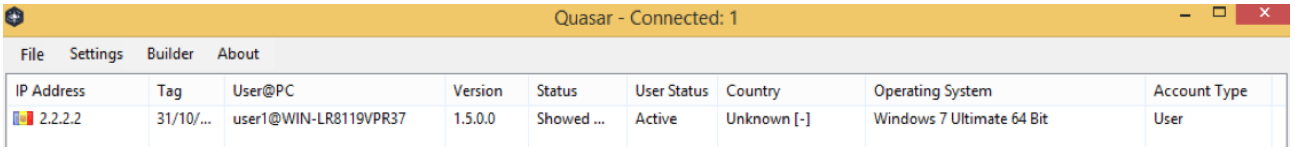

Figure 10- Quasar RAT Server GUI

The server and client then enter into a keep-alive mode, where the attacker can send commands to the client and receive further responses.

RAT commands

The attacker can issue commands (not all commands appear in different samples) through the Quasar server GUI for each client:

- Get system information
- Get file system
- Upload / download / execute files
- Startup manager
- Open task manager
- Kill / start processes
- Edit registry
- Reverse Proxy
- Shutdown / restart the computer
- Open remote desktop connection
- Observe the desktop and actions of active user
- Issue remote mouse clicks and keyboard strokes
- Password stealing
- Retrieve Keylogger logs
- Visit website
- Display a message box

Our server build was able to successfully execute most of the commands.

The file system commands underling handlers and IPacket were modified to support more features, so these commands don’t work out of the box and required manual implementation from us.

A Double-Edged Sword…

With further analysis of the Quasar RAT C2 Server, we uncovered vulnerabilities in the server code, which would allow remote code execution. This might allow a second attacker to install code of their choice – for example, their own Quasar RAT – on the original attacker’s server. We refer to this (somewhat ironic) technique as a “Double Edged Sword Attack”. We did not apply this to any live C2 servers – we only tested this with our own servers in our lab.

In the lab, we changed our Quasar RAT source code to use the known encryption key, and to send fake victim IP address, City, Country code, Flag, and Username. The Quasar server does not verify the RAT data, and displays this data in the RAT Server GUI when the RAT is executed and connects to the server. We found this could be used to supply compelling “victim data” to convince the attacker to connect to this “victim” via the GUI.

Quasar server includes a File Manager window, allowing the attacker to select victim files, and trigger file operations – for example, uploading a file from victim machine to server. Uploaded files are written to the server sub directory “clients\user_name@machine_name_ipaddress”.

Quasar server does not verify that the size, filename, extension, or header of the uploaded file is the same as requested. Therefore, if we convince the attacker to request the file “secret_info.doc (20KB)”, we can instead return to the server any file of our choice, of any size or type.

When the Quasar server retrieves the name of the uploaded file from the victim, it does not verify that it is a valid file path. Therefore sending the file path "..\..\ secret_info.doc " will result in writing our file instead to the same directory as the Quasar server code.

Quasar server does not even verify that a file was requested from the victim. Immediately when the File Manager window is opened by the attacker, the Quasar server sends two commands to the RAT: GetDrives and listDirectory (to populate the list of the victim’s files in the RAT Server GUI). We can respond to those commands by instead sending two files of our choice to the Quasar server. Again, we control the content of the file, the size and the path and filename.

Quasar is a .NET Framework assembly, loading multiple DLLs upon launch, for example “dnsapi.dll”. Quasar server is vulnerable to a simple DLL hijacking attack, by using this technique to replace server DLLs.

When the attacker restarts the Quasar application, our uploaded “dnsapi.dll” will instead be loaded. Through this vector, we could drop our own Quasar client on the attacker’s server and execute it. Our Quasar RAT will connect to our own (secured, of course) Quasar server, allowing us to control that attacker’s server with his own RAT. We can also replace “shfolder.dll” (and add a DLL export proxy to avoid a crash), which is loaded whenever the attacker clicks the builder tab – allowing us to infect the server while it runs, without the need to wait for application restart.

Downeks

Although Downeks has been publicly examined to some extent, our analysis found several features not previously described.

Earlier Downeks samples were all written in native code. However, among our Downeks samples, we found new versions apparently written in .NET. We observe many behavioral similarities and unique strings across both the native-Downeks versions, and the new .NET Downeks versions. Almost all of the strings and behaviors we describe in this analysis of a .NET version are also present in the native version.

We observed these samples deployed only against Hebrew-speaking targets.

Downeks.NET – “SharpDownloader”

Downeks .NET internal name is “SharpDownloader”, “Sharp” may be a reference to the language it was written in – C#.

As seen in previous Downeks versions, it uses masquerades with icons, filenames and metadata imitating popular legitimate applications such as VMware workstation (Figure 1) and CCleaner, or common file formats such as DOC and PDF.

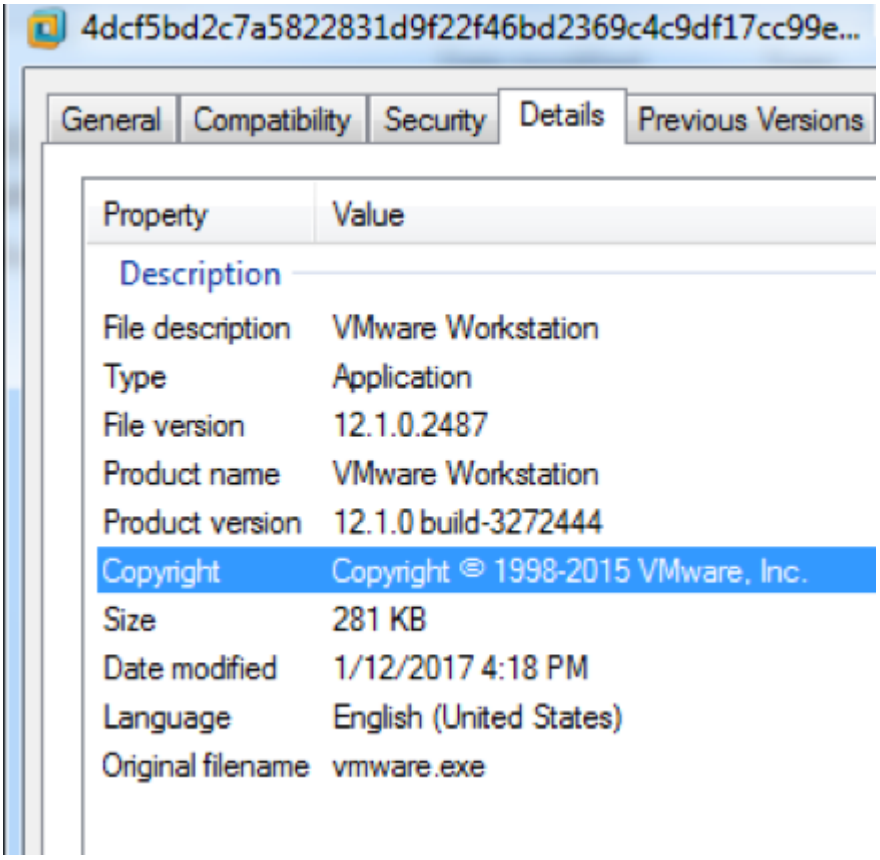

Figure 11 - Application metadata masquerading as VMWare Workstation

All 3 samples were compiled with the same timestamp. Downeks.NET is obfuscated using “Yano” and can be easily de-obfuscated using the de4dot utility.

SHA256: 4dcf5bd2c7a5822831d9f22f46bd2369c4c9df17cc99eb29975b5e8ae7e88606
SHA256: 905f6a62749ca6f0fd33345d6a8b1831d87e9fd1f81a59cd3add82643b367693
SHA256: c885f09b10feb88d7d176fe1a01ed8b480deb42324d2bb825e96fe1408e2a35f

Communication

Downeks is a backdoor with only very basic capabilities. It communicates with the C2 server using HTTP POST requests.

It runs in an infinite loop, in each iteration it requests a command from the C2, and then it sleeps for a time period it receives in the C2 response (defaulting to 1 second if no sleep-time sent).

The data that is sent in the POST is serialized with json, which is then is encrypted, and finally encoded in base64. The json format is typically {“mth”:”some_method”, “data”:”some_encrypted_data”}. The C2 server responds using the same format and serialization/encryption/encoding.

Download and Execute

As described in earlier analyses, Downeks’ main purpose is as a downloader. Unfortunately, we were unable to get any C2 servers to issue download commands to any samples that we tested in our lab.

The download is initiated upon receiving json with a “download” command, which includes the URL of the file to be downloaded. Downeks can also be instructed to execute binaries that already exist on the victim machine. After successful execution, Downeks returns the results to the C2 server.

Downeks also has a self-update capability, if instructed by the C2.

Screen Capture

Downeks can be instructed with the “img” command to capture the victim screen and transmit it back to the C2. The parameters “wth” and “qlt” specify “width” and “quality”.

Appdata

Downeks .NET creates a file in the “Appdata” directory, based on certain properties of the machine. During our analysis, Downeks created a file in “Appdata\Roaming” containing only “SD{new line} 0” (“SD” possibly for “SharpDownloader”).

Although this file itself is not particularly interesting, the older (native) Downeks versions also creates a file in Appdata\Roaming, with identical data.

The filenames across the two variants bear striking similarities. The .NET variant creates “1FABFBFF0000065132F71D94”, while the native version creates “000206511FABFBFF”. We observed the string “1FABFBFF0000065132F71D94” in memory during debugging of the native variant (Figure 12). This is a pseudo-unique ID for each machine, based on install date taken from the registry, volume serial number, OS version and service pack, Processor architecture, and computer name.

Figure 12 - Machine ID in memory

Installed Antivirus check

Downeks enumerates any antivirus products installed on the victim machine and transmits the list to the C2. It constructs this list using the WMI query:

“SELECT displayName FROM AntivirusProduct”

Persistence

Downeks achieves host persistence through either the registry “run” key or with a shortcut in the start-up folder.

External IP

In another similarity between both variants, Dowenks assesses the victim’s external IP using an HTTP request to http://www.myexternalip.com/raw.

Other commands

Downeks can be instructed by the C2 to perform a few other commands:

- Check if the computer name and user name, or external IP address, is in a provided list and if so, display a message box with a message as defined by the C2.
- Kill any running process and attempt to delete the associated executable.
- “Setup” command – sends various info about the machine with each iteration of the C2 communications loop.

Encryption keys

Downeks has static encryption keys hardcoded in the code. These keys are initialized in the “Defaults” class constructor, suggesting that the author of this malware has great affection for stackoverflow:

```
static Defaults()
{
    ResEncKey = Strings.Get(0x1524);       // resolves to “$t2ck0v3rFl0w”
    RarPass = Strings.Get(0x1539);         // resolves to “123456”
    ServerTransKey = Strings.Get(0x1542);  // resolves to “P@$sw0rD$nd”
    DataEncKey = Strings.Get(0x1553);      // resolves to “$t@k0v2rF10w”
    ConnRequestKey = Strings.Get(0x1564);  // resolves to “1q@W3e$RQ!w2E#r4”
}
```

Typos

We observed some typos in the code, such as “responce” (“response”) and “GroubID” (“GroupID”) in this version.

Coverage & IoCs

Palo Alto Networks customers are protected from Downeks and Quasar used in this attack:

- WildFire properly classifies these Downeks and Quasar samples as malicious.
- Traps detects and blocks malicious behavior exhibited by new, unknown Quasar samples.
- C2 servers associated with this activity are blocked through Threat Prevention DNS signatures.
- AutoFocus customers can monitor this activity using the Downeks and QuasarRAT tags.

A list of Indicators of Compromise can be found in Appendix C - IoCs.

Appendix A - IoCs

C2 Domains

downloadtesting.com
gameoolines.com
onlinesoft.space
newphoneapp.com
gamestoplay.bid
smartsftp.pw
galaxysupdates.com
galaxy-s.com
datasamsung.com
progsupdate.com
topgamse.com
bandtester.com
speedbind.com
ukgames.tech
wallanews.publicvm.com
wallanews.sytes.net
noredirecto.redirectme.net
dynamicipaddress.linkpc.net
downloadlog.linkpc.net
havan.qhigh.com
kolabdown.sytes.net
rotter2.publicvm.com
ftpserverit.otzo.com
webfile.myq-see.com
downloadmyhost.zapto.org
help2014.linkpc.net
safara.sytes.net
exportball.servegame.org
viewnet.better-than.tv

down.downloadoneyoutube.co.vu
netstreamag.publicvm.com
hostgatero.ddns.net
subsidiaryohio.linkpc.net
helpyoume.linkpc.net

Quasar & Downeks SHA256s

3053e1e8df7e525ef98a77190cefce258aea365e2998425ecc8e139230680156
f19bc664558177b7269f52edcec74ecdb38ed2ab9e706b68d9cbb3a53c243dec
0d235478ae9cc87b7b907181ccd151b618d74955716ba2dbc40a74dc1cdfc4aa
96c1346351a53c865afef5e087a8cbcf8e28d652fbc083a93399a8b94328d456
15abd32342e87455b73f1e2ecf9ab10331600eb4eae54e1dfc25ba2f9d8c2e8a
b66e5af52fd4d802f64788692b3eafe6b5ff61cea09c06a237a96b6cdb90b41a
55a7ca1e5ed2d74c7eb6ab6a985c1d369157a91275f575967aefb7ddb3388e0c
9a8d73cb7069832b9523c55224ae4153ea529ecc50392fef59da5b5d1db1c740
39bdeaded0f919caa6697ae1ae4953de1c7afa79905939dbbd8c647a84f6cd07
0e41c3611da6e3a2b0dd0d43b9ce0b3f3405472efa5760767719cc82692afb7b
bc7c3f687d0589a4db53475bc65056a628b52aa27f84c1d76b9fe686d495df27
0d1aa670df8ae1379d6997c9dc8b40c893ee395c3d45b84c2ad1732e86973143
8ac7138215b2500d0737b483b9194419c0e0248014147e84f43b1e2b409184cf
8ca99455d244fab2701beb5127f94745154e03ac1231a58f8bd2cd01732a341b
a149340f920888256902e28e4c5d8587fed3037682e875ed1fdf6a3213c50e92
78aaed20914d3895708985aee089a464b31e11eb3b3e90b530dcebbe10e915ec
704b19e0460a0fa7d952ba6feb5eadb9054895d1d753df72faf6f470446a0519
118d0bd8ec35b925167c67217d2fe06ac021ce253f72d17f1093423b8f9b4a2b
68ec3588735341566e9736b897aac06affb4a4808b05ceffb72384e77ea04b2c
d3c710eaaf849598fa486823da42bdce03ea3c9421c3936e3330e98b34e4ef47
e6e9f7b0449976537d9276192e5767c9909cd34df028a8bf1cac3dbe490f0e73
75336b05443b94474434982fc53778d5e6e9e7fabaddae596af42a15fceb04e9
4dcf5bd2c7a5822831d9f22f46bd2369c4c9df17cc99eb29975b5e8ae7e88606
b8a3096a396c28462c0d168d97e28573e0e6d272bbc1dd2432e7effe098bd979
905f6a62749ca6f0fd33345d6a8b1831d87e9fd1f81a59cd3add82643b367693
c885f09b10feb88d7d176fe1a01ed8b480deb42324d2bb825e96fe1408e2a35f
0c4aa50c95c990d5c5c55345626155b87625986881a2c066ce032af6871c426a
1ac624aaf6bbc2e3b966182888411f92797bd30b6fcce9f8a97648e64f13506f
1acffe68fcdc301b8ab7640eda75ff82788b2f93d869e421e28bacbba93b76d1
723108103ccb4c166ad9cdff350de6a898489f1dac7eeab23c52cd48b9256a42
99a7cb43fb2898810956b6137d803c8f97651e23f9f13e91887f188749bd5e8f
86bd78b4c8c94c046d927fb29ae0b944bf2a8513a378b51b3977b77e59a52806
3243292E46A198BD83E0DCE58258312852C99217187E6D5399066189FEB2677B
9b8d8780454708b950459d43161097ac72f62ff349bc8f379b5b2216bc9ae935
3619b12b11cda6e87644d3316355d99ee5fa5407aa8a8f107aa1058e33b19bf6

0f8378603e269db16eb7eaca933b587c7de3e914c1d9afaaae688c410befb895
d3066fa4a7a1ef38c753796479768b765c6903ef50c35352e29e79dcd49e4348
39b991838653739eef482af6336fcf03922d7e9d88d17946b688a513dd2bfc34
4393ff391396cdfd229517dd98aa7faecad04da479fe8ca322f035ceee363273
759ae70b035c3bbb6699520db3a55f3947e6ba1b5ce639ec036e3096ee10b26d
17942d9d76dafb64aa0d3ab53c9ee56e5d8bd4477440f06780b70dd4c02af8b8
fea74bf9eed7363f97a09756b4652409cfcd7bbe023383805aec5da7de6310bd
f5413c785770400215c3191ea887517b4380ec81be4e5bdc5aea12bf82f9105d
8cdbea2aea51f73c68adc517eed533802e1f3b2a9ec0b0560b6bb8fc03ac3e4f
dbdc72a7cfbf03599b95d8f1c47e157da34ea5d2f951cf5f49715e8caab58cd4
65986f6f919e9152176a10ae3964fac130ae6195e189453d17306a225022774d
91a4e395d57a52a85a2bda653a0ed796865e8af01c1345dff63469759448daf0
53e82d01dd2502416ad49329e1224a7c4519182186e60f690ecd0cf266f5af5e
575708d3eb23f8111b7174408f05caf6574c5d6782c750562bfb9abe48cb219e
fefa0781e88fa215419b2a1294c8b952b192f8360aeab2f97bbd9cea15fc7338
ea16f0d55918752ad432d0da03a7e39ab9a8442b74ae0bbe724900605a9ba71f
6f6414c8f8a800c769da1f6994cad25757a2928375803a498171db3395183b98
dbeb3c262cc6eefea93846f817e8333ee541ec23d19ffef56a94585e519e6ff1
2ddce8b010f011a04cf24dc8e5932ae13b463dd6a3cb9bf02ae835b04a70d042
182c82100069834ad4a8dadee6874cfb612f0b9babc7cd3ee5d69f16440ad6d7
f772463bafef5f45f675658eee43b6f56911a4f449afb0cc68ac068002a2f875
b30e3dc47848666e71c1f13050a6502b2c2a7a542ee867d152ffb2dd186d7114
e5e4895d2195e14a3a105f3ed73fd49493e9dbdd7dfc6f6616023473fa8f86c6
f4eda40b3f1c77f8f9e02674d93214dd31c13080b034e37b26cc66d744500b1e
95d9a2b664e3e5c1206d94241ded115643aa0452dd3fe3338363ff826260f40c
575b84c2d3bceebfabb2deb289a230f52aca2c504aa854251c1e9057f3f0cf5a
21f09c93325c03940c24d8bd6f33a1a4876bfd5feb8c8bac05b0a359255c0b42
695821451be582d85cd8e42ce4446f131bd474e6c715bdf13fe8bac6de34b2e3
e874deabb7953c2b9b5e67fc08297019bb0171c2fbdbe136b822cee4d43b72e5
6a700aea23f7cb6907e464981a136b0fbfb5a48b910af2f9a44baf98d25f1722
b6adae77a975058720e525a7f6d2451a01fedd3c6cab1515570d8490a8eb4f67
d735c19fc9223e1bc4e625c1f47801d758426fbae89e5086bc56a8d6b1df2011
a66a27d801891e39d3819355366399fabbf2f05327ddb7c7b5d304fabeac7118
cdf4ec8beb3f15d04b54165b53475aa03949a67f9cb1847a749b2fb44a3fe0a4
0045c28ed2a9d98efb798ec59f34b6a3058838f933af7c0dae6482a0e86e37bf
8814fec28ccac77456be73305b32ad5a266a4929203b2acf431759c90fe579bd
f1b682808f1819f0e3d030fce1fd6b1ca95ad052b069e028cd9ed4afd81cd4d6
f361974e6fd6a6d7cfbfafb28159c4f8e514fe6f399788be4daa2449767d5904
11b1088ad962984f6df89ccc6bbc98bf220af952dec0b4622f8453a0a164cb65
Aacf24e288388431b30f8da765b4696975adc9cc0303d285abb05077eda21da2
02bd710d3055076f86116d28427322c9cb623291c6c5a66c1932181fc6558586
5e7d68c53212f6d467533f105d4a067682e28da47304a53c17b056d2b4404f0a

3ff059a53e38f9fcd24e8d6bf008b4e14733db317857764cfcef736119ff26c9
dcc04adf96045e7227a0e1f1d092919276b21035bcb3c5ed462650ef8d2e7aa3
20e3d4c9223955495d00e72e2fedfe825e9fcda57696a255215895cfba490876
6050d4c1efcf8242382293842313f3a93309f1e449197d98c60cec29090c6bff
1d533ddaefc7859a3f6c6751114e895b7aa5935eb0ed68b01ec61aa8560ae3d9
488ba22d6cb8c9b0310c58fa4c4739692cdf45676c3164b357314322542f9dff
7eeeae1f2fc62653593c7ce254e9cf855905035c2e8f8c0588887cb8e99dd770
d2d08bb2707b635617e5bab0fcd033b6f68a753dd2b3897adca1c627758e686b
d30dbf17078a11c32dd23acea42335860e739c9f18bf0ed611132eef4d5cfcb6
7c578dcdcefe78fb1dd51ac611f6450d9eb5be6c5f1e3363f460321a46be4a39
a40627acae6917787e92f9efa85739136c1670dcc5fe66695e105ddd72d7b80a
5668470c92408f4b9f3a659005c2acca9da8df750cc491bffc88ef640474fa4a
d735c19fc9223e1bc4e625c1f47801d758426fbae89e5086bc56a8d6b1df2011
25e6bf67410dffb95c527c19dcff5223dbc3bf4c987650e45fbea1267072e8ff
f53fd5389b09c6ad289736720e72392dd5f30a1f7822dbc8c7c2e2b655b4dad9
2c2ae3f482d9db2541de0d855b5b12cd18028a94887f0c28acf1e2d6a4f3d4ac
a35e2b21f7f770debcffc79eb4834ec8881465df06cee41af705b6ea5d899978
a7aeeead233fcdfe1c7475db982497a82d8ae745ec1c58bd87215e8869c3f9e4
f0e3562d0438695c7f3af0c280968cfc7134b484010d9ba2aceab944b441b127
f5413c785770400215c3191ea887517b4380ec81be4e5bdc5aea12bf82f9105d
29049e2c7671a7c4fc953cb76e539150cc7c80e1b83c19d0894dfa446ce5276e
2eb7aa306551d693691d14558c5dc4f6d80ef8f69cf466149fbba23953c08f7f
dbdc72a7cfbf03599b95d8f1c47e157da34ea5d2f951cf5f49715e8caab58cd4
de3e25a69ba43b9f236e544ece7f2da82a4fafb4489ad2e263754d9b9d88bc5c
bc846caa05939b085837057bc4b9303357602ece83dc1380191bddd1402d4a2b
44b99603dde822b6b86577e64622e9a2f5b76b6d8bd23a3fe1b4d91b73d0230a
bb24105295588d14c4509ec7374fbe6f7a4821cf4e9d9282754dd666ad7a7ea1

Source: http://researchcenter.paloaltonetworks.com/2017/01/unit42-downeks-and-quasar-rat-used-in-recent-targeted-attacks-against-governments