

# Paperduck Paperboy Malware

Hans Amelang

<sup>1</sup> The University of Memphis, Memphis TN 38152, USA  
hsmelang@memphis.edu

**Abstract.** Word processing files are ubiquitous in computer usage but can be made insecure. Some file types, like PDFs, can be made to trigger embedded scripts to perform unexpected – and sometimes unwanted – actions.

**Keywords:** malware, HID, PDF security, embedded actions, keystroke injection, Rubber Ducky

## 1 Introduction

### 1.1 Malware from the project perspective

Oftentimes, malware is “all about making the computer do something ordinary but very very fast and in a repetitive way” [1]. Changing this definition slightly, we can look at malware, many times, as making the target machine do something ordinary, but automatically and without warning or indication that the machine is performing the malicious action. One of the chief purposes of the modern computer is to transfer information or files to another computer, but in the case of malware, the computer may be transferring things we don’t want transferred to or from it.

Keystroke injection attacks are a form of cyber-attack that “involves injecting malicious commands or keystrokes into a target system to perform unauthorized actions or gain elevated privileges” [13]. Given the goal of writing a malware for this project and recognizing the ubiquity of document files, I have chosen the document file as an attack vector, essentially the means by which I will deploy my malware. In this project, I intend to automate the transfer of a file – in this instance, the completed project paper – to the professor’s computer using embedded actions in a word processing file and keystroke injection in order to hide the actions taking place behind the scenes. If absolutely successful, my malware will not only perform the malicious action of downloading a payload without the user’s permission, but it will also evade detection techniques – both from the user and from security software.

The following sections of this report detail more of the background for the project, the rationales behind the project, the design and implementation of the project and the results and my conclusions.

## **2 Related Work**

### **2.1 Work done in reference to embedded actions and keystroke injection**

The concepts of embedded actions and keystroke injection are not new – in fact, getting this attack to work on a system guarded by modern malware detection software may prove to be an impediment to this project. That said, the inspiration for the project came largely from the concept of the “bad USB” or USB Rubber Ducky, a commonly available attack device sold by Hak5 and others. These devices are essentially microprocessors in the form of USB drives that deliver a binary injection, triggered by the insertion of the USB drive into the target machine, which activates a script that acts as a “Human Interface Device” (HID). This essentially acts as a computer user at the keyboard and deploys a script of keyboard actions as a payload [2]. While the Rubber Ducky is certainly not the only attack vector open for keystroke injection attacks, my brief research indicates that it is the most popular. Even a basic search query of “keystroke injection” on your favorite search engine yields an apparent majority of Rubber Ducky results, leading me to believe this is a good point from which to start learning about keystroke injection and deploying it effectively.

Expanding on the Rubber Ducky idea further and removing the need for a physical device for this project, I would like to exploit embedded actions in a word processing/file processing program to deliver the injection exploit as opposed to the USB device. Many document processing programs (Word, Acrobat, etc) use macros or embedded actions to run processes behind the normal user operation of the program. This has been well documented in many case studies of actual attacks, but Cynet explores this in an article focused on Microsoft Office macros [3] and Didier Stevens discusses using embedded actions in PDFs on his blog [4]. My goal for this project is to use a payload injection like a Ducky Script, combined with embedded actions in a PDF file to download the final version of this paper to the user’s computer, taking the place of a malicious payload.

## **3 Rationales**

### **3.1 Why is this applicable to what we are studying here?**

Though this type of attack is not specifically network focused, it will illustrate the ability of a computer program to use the internet in a manner unintended by the user – again, a perfectly normal action, but not authorized or expected by the user. A successful attack will cause the victim computer to download at least one payload file from an online repository.

### **3.2 What resources will I need to complete this project?**

Nothing more should be needed for this project than my laptop, an internet connection, and the development of the deliverables listed above. I will likely use a service like Dropbox to host the downloadable report file. I will be referencing numerous print

sources and websites to learn the scripting, injection coding, and creation of embedded actions in Acrobat. The first few of these are indicated in the “References” section of this report, which will be added to as the project develops. Attacks code and commands will be created in either a word processing application or VS Code and should not require additional hardware or materials.

### **3.3 How am I suited to solve this problem?**

Though I have never created an exploit or malware (at least not on purpose), I have introductory study on several programming languages and scripting environments through my coursework at the University of Memphis. I am familiar with the basic concepts of vulnerabilities, exploits, and payloads due to the same course of study, as well. The purpose for this project, aside from demonstrating the problem of injection attacks and their preventative measures, is to develop further skills necessary to perform offensive and defensive tasks to broaden my experience in cybersecurity.

## **4 Design and Evaluation**

### **4.1 Original Plan**

I originally envisioned this exploit as consisting of three components:

1. Compromised PDF/.doc file: this would be a seemingly innocent file that contained a malicious script. This is what the victim would download and open.
2. Malicious script: this would be the actual code executed to carry out the exploit, embedded in the compromised file. The script would download the final copy of my project report paper, mimicking a payload.
3. Payload: the unintentionally downloaded final copy of my project report paper.

That said, I very quickly decided against this approach to save time and my own limited intellectual resources. To get a workable attack together in time to meet the deadlines of this project, I had to switch gears and look at alternative attack vectors.

### **4.2 Shifting Gears**

After doing further research on the topic, most every article I found led me to believe that embedding a self-executing file into a PDF or other document file would be extremely difficult without doing a ton more research on the topic and figuring out how to get around modern security measures. As an alternative course of action, I decided to proceed with designing an executable and attempting to obfuscate it as a PDF file. With that said, I endeavored to continue making my exploit from DuckyScript.

A quick note before continuing – the files for this project, and the repository from which its “payloads” are downloaded, are housed at <https://github.com/hanselopolis/Paperboy> and subsequent branches. This link is publicly accessible, and I have made no efforts to hide its contents outside of making them read-only. This is, of

course, not the norm for malware in the real world, but obfuscation of source repositories, etc, are beyond the scope of this project and hence not considered.

**Ducky Script.** As a reminder, DuckyScript is the scripting language behind Hak5’s previously mentioned USB Rubber Ducky [2]. The idea with the language is that it translates keystrokes to actions so that the attacker can script out an attack via keyboard action. As an example, in my attack I wanted to launch PowerShell in a Windows environment – making this a Microsoft Windows specific attack. To do this, assuming I didn’t use a pinned icon in my taskbar, the easiest thing to do would be to search for “PowerShell” and launch it. Translated to pure keyboard action, I would press the “Windows” key, then type “powershell”, and then hit the “enter” key. The Ducky Script, then, would look like this:

```
WINDOWS
DELAY 100
STRING powershell
ENTER
```

Note that in the above, items in all caps represent the actual key being pressed or a variable being entered. `WINDOWS` corresponds to the “Windows” key being pressed, `STRING` corresponds to typing a string, etc. Also note that line two, `DELAY 100`, is used to allow 100ms for the search/action menu to open before continuing to the next line and “typing” powershell. This is a common occurrence with Ducky Scripts, as there must be time allowed for windows to open, actions to execute, etc, before keystrokes can reliably be injected and applied to the intended environmental item.

The problem now became how to get this script to execute. As designed, Ducky Script is meant to be compiled as a binary and loaded onto an armed “bad USB” device to be used in USB drops and other similar attacks. This USB device is actually a microprocessor with embedded code enclosed in a “thumb drive” style housing. However, I wanted to eschew hardware and be able to launch this attack purely from a file which could be distributed by download or social engineering methods. Thus, I had to be able to translate the Ducky Script to an executable. This turned out to be a simple two-part process – convert the created script to Python code, then convert the Python code to an .exe file [5].

The first step was to create the Ducky Script and convert it to Python. The video in [5] does this via a python program called “ducky2python”, but I had issues getting this to work properly via the command line application. Luckily, I was able to find an online converter of the same name (likely the same code running in the background) [6]. This converter allows you to past in your Ducky Script and convert it over to Python via a web application. I then copied that output code over to an instance in VSCode for easier editing and testing. With this I created an initial script to open PowerShell as an operating environment and then download and open a payload.

The converted Python script makes use of the `pyautogui` and `time` libraries for implementation. `Pyautogui` allows for the injection of keystrokes while the `time` library allows for implementation of sleeps, etc., for window timing. Once the initial Ducky Script was converted to Python, I continued testing and development in Python to avoid multiple conversions as iterations developed. Development in VSCode allowed for easy

iteration and code execution. In its final attack form, the script opened PowerShell and downloaded two components from my github repository for the attack [7], a .vsb Windows error message and a PDF file. The PDF was intended to eventually be the final version of this paper, acting as a final attack payload. In a real attack, this PDF could be replaced with any sort of malicious script, another executable, etc.

As this was created from Ducky Script and meant to download my final paper as a payload, I dubbed it the “Paperduck” attack. With this initial script created and converted to Python for continued development, the next step was to convert the Python code over to an .exe file for “one click” execution. I also wanted to be sure that the code would run on largely any Windows machine without having to have a Python environment running. Most sources that I found on converting Python to executables centered on one of two methods, either via a command line tool called “pyinstaller” or via a graphical user interface called “auto-py-to-exe” [8]. I opted for the latter as it seemed to provide ease of flexibility and I found it to be a relatively intuitive product after figuring out where it stored everything by default – via error messages while attempting to start on command line.

At this point I had a successful keystroke injection payload that ran from an .exe file. As it is a keystroke injection attack, the attack runs in what is essentially real time. To this end, while the malware operates quickly it is visually readily apparent what is occurring as the attack executes. As such, the attack is not at all obfuscated or hidden, so to remedy this I decided to try to make a stealthier attack, this time strictly as a PowerShell script.

**PowerShell Script.** As the previous keystroke injection was largely formulated on PowerShell commands, converting the actions to a PowerShell script was easy – the initial iteration merely downloaded the two files from the Ducky Script attack, then opened the .vsb error to indicate that the user had downloaded the malware and next step (check for the PDF in the user’s downloads folder). The original PowerShell script in its raw .ps1 and converted .exe forms, and the accompanying payloads can be found in my github repository on the main branch for the attack [9]. At this point I began referring to this malware as “Paperboy” in order to separate the new PowerShell script origin from the old Ducky Script attack.

The new PowerShell script operated much more quickly, and less apparently than the Ducky Script based keystroke injection attack. Once the initial phase of this development was done, I moved on to, as with the keystroke injection script, convert the PowerShell script over to an executable to make it a little more universal. After a little research, I found a source recommending a tool called “ps2exe” [10], which was corroborated in another source [11] that also contained some details on obfuscation techniques for .exe files. I found the GUI version of ps2exe and downloaded it, which made quick work of converting the PowerShell script to an .exe. After testing the .exe, I was prepared to move on to obfuscation.

### 4.3 Obfuscation

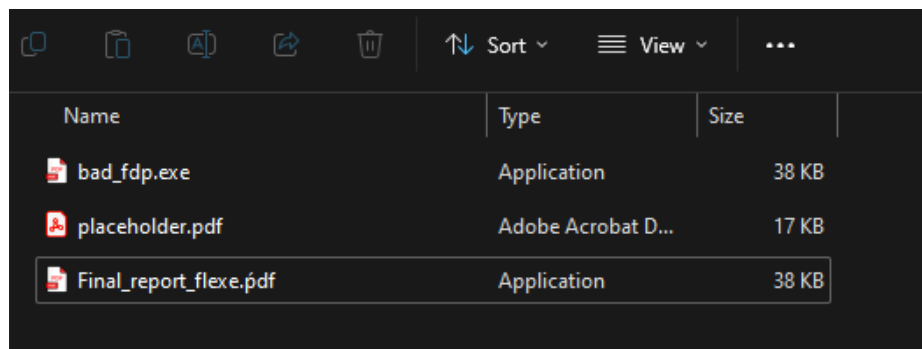
The next phase of development for the malware was obfuscation – hiding the nature of the file. The purpose of this is twofold: first, we must make the malware file appear as a normal, trusted file to the victim. Second, we must manipulate the file in such a way that we can try to get around security software. The former is somewhat easy to do, the latter not so much.

The first thing to do is to make the file look like a normal PDF. Taking tips from [11], the first thing I did was import an .ico when compiling the .ps1 to an .exe in ps2exe, thus giving it the quick glance appearance of a PDF. This was relatively simple, as .ico files are readily available on the internet for downloading. As I wanted to make the icon switch somewhat noticeable, I imported a low quality .png file for a PDF icon and converted it to an .ico file using an online tool (iconconverter.com) and imported it during the .exe compilation in ps2exe.

The second thing to do here is to somehow change the extension so that the file looks like a PDF as opposed to an .exe. Again, referring to [11], I made use of a technique called “right to left override” (RTLO), in which we change the order of letters to make them read in the desired order. In my case, I named the exploit file “Final\_report\_flfdp.exe”, but made a few notable changes.

First, I changed the “p” character before the period separator to “p̂” – the idea being that we use something that looks like a standard character but is not to try to fool a malware scanner as to the file name but not make the difference noticeable to the user. I chose the “p̂” character so that the character change was apparent for academic purposes, knowing that there are other characters available that look like a standard “p” but are not. The second change was the insertion of a RTLO character (Unicode 202E from the system Character Map) between the last “l” and the “f”. The character is invisible to the user but has the effect of reversing all characters after the RTLO character. Thus, the file name reads as “Final\_report\_flexe.p̂df”. The obfuscation is nowhere near perfect but may fool a casual victim. Note from the system screen capture below that the system is not fooled as to the nature of the file – it still shows as an application – but the casual user may still be fooled.

As to being able to evade malware detection software, Windows Defender put up several alerts during the development of the .exe, especially in the “middle” stages of development, but eventually was somewhat compliant. With the malware uploaded to a github repository, I asked a trusted student to download the malware and try to run it after providing to them full knowledge of what it is intended to do and the circumstances around its development, to test the obfuscation efficacy externally. They reported back that Windows Defender also posted an alert when they downloaded the malware, but the download was allowed and they were able to open the file. I also attempted to email the file to a non-student, again providing full disclosure beforehand as to the function and intent of the file, but my browser and email client would not allow the file to be attached to an email due to security reasons.



Name	Type	Size
bad_fdp.exe	Application	38 KB
placeholder.pdf	Adobe Acrobat D...	17 KB
Final_report_flexe.pdf	Application	38 KB

**Fig. 1.** Screenshot of the host folder of the malware. Note the different file extensions and the file types being displayed.

I also attempted to embed a mechanism to cover the malware’s tracks by deleting the malware file via running a batch file separately from the main attack script, leaving the downloaded PDF “payload” intact. Some forms of the batch files are included in the github repos on both the main and Paperduck branches but were ultimately unsuccessful. I was able to get the batch files to delete themselves, but not the executables that launched them. This is, of course, since the batches were launched by the executables as child processes, and not as independent processes. In some test instance using the keystroke injection attack, I was able to set a long enough delay on the batch file timing so that the batch file would delete both the Ducky Script executable and the batch file, but then the problem would be to get clear the Recycle Bin to further remove traces of the attack files. The solution for this would likely be to run the removal and Recycle Bin clearing as scheduled jobs in PowerShell, but that would require privilege escalation to create the scheduled jobs, which would add a whole other element to this attack.

At this point, I consider my ultimate obfuscation goals unachieved. With more time, I would research more and attempt to get around the security measures. One of the sources that corroborated the ps2exe conversion usage recommended changing the code over to something like a base64 before .exe compilation to further obfuscate the code [12], as well as using archiving software like WinRAR. This would allow for automatically opening multiple files within the archive when expanding (double clicking) the archive – potentially achieving my original design goal, at least in perception by the victim, of the benign PDF opening a malicious code. If combined with successful file name obfuscation, this could be effective. When starting on the WinRAR path, however, I received constant alerts from Windows Defender, so I did not go further down that path.

## 5 Evaluations and Conclusions

Ultimately, I think this project was at least partially successful. I was able to accomplish the main goals of the malware, although it could be argued as to how successful the attack would be without further exploitation and obfuscation work.

**Goal 1: Keystroke Injection Attack.** Developing a simple keystroke injection attack was the first goal of the project. Inspired by Hak5's "Rubber Ducky" device, I was able to adapt and modify their Ducky Script as the development environment for my successful keystroke injection attack. Running via an executable, the attack opens PowerShell, downloads multiple payload files, and activates one of them, causing a custom alert to be displayed to the victim user. This file is called "Paperduck\_script.exe", located at [7], and can be downloaded and opened.

**Goal 2: Hide the Attack Executable in a PDF (or Other Document File).** This goal I consider at least partially failed, at least in the sense of the literal goal. The original intent was to hide the attack executable in a PDF or MS Word document, but early research suggested that the most current versions of Adobe Acrobat, MS Word, etc. were no longer susceptible to simple forms of these attacks. Time being in short supply for this project, I decided to forego embedding the attack executable in the document file and instead attempt obfuscation to make my executable appear as though it was a PDF file. In this regard, I think that this goal was partially achieved in that I believe that with another small change or two to the file name formatting the obfuscated file (with special characters and RTLO techniques applied) could viably be employed in a social engineering attack.

That said, I believe further obfuscation techniques need to be researched and employed in order to get the malware to bypass modern security software. Development on my own computer, actions after download on test subject computers, and attempts at propagation by email were at the very least flagged by antimalware software if not outright blocked.

Ultimately, I would like the malware to completely evade antimalware software and checks by email clients, but I believe that would require much more in-depth work. Many of my early research sources also pointed to the ability to use common penetration testing tools like Metasploit/Meterpreter to create malicious PDFs that launched remote shell attacks and the like, and it is possible that I could have gone down this route, as well, to achieve a properly obfuscated attack file. For the time being, I can say that I have learned a lot about malware creation and made aware of the difficulty in pursuing successful malware design from scratch.



## 6 References

1. Medium. “Can Computers Get Sick? (How to Create a Virus in C++),” <https://medium.com/codex/can-computers-get-sick-how-to-create-a-virus-in-c-e09f2a6ad45b>, last accessed 02 February 2024.
2. Hak5. “Hello, World! Getting Started with DuckyScript Payload Development,” <https://docs.hak5.org/hak5-usb-rubber-ducky/ducky-script-basics/hello-world>, last accessed 02 February 2024.
3. CyNet. “Office Macro Attacks,” <https://www.cynet.com/attack-techniques-hands-on/office-macro-attacks/>, last accessed 02 February 2024.
4. Stevens, Didier. “Embedding and Hiding Files in PDF Documents,” <https://blog.didierstevens.com/2009/07/01/embedding-and-hiding-files-in-pdf-documents/>, last accessed 02 February 2024.
5. Pear Crew. “Convert Rubber Ducky Scripts to Programs (video),” <https://www.youtube.com/watch?v=sCLB5vkmksQ>, last accessed 05 April 2024.
6. “Ducky2python,” <https://cedarctic.github.io/ducky2python/>, Cedarctic, last accessed 05 April 2024.
7. Amelang, Hans. “Paperduck,” <https://github.com/hanselopolis/Paperboy/tree/paperduck>. Last edited 14 April 2024.
8. Yash, “How to Turn Your Python Script Into an Executable File,” <https://yash7.medium.com/how-to-turn-your-python-script-into-an-executable-file-d64edb13c2d4>, last accessed 05 April 2024.
9. Amelang, Hans. “Paperboy,” <https://github.com/hanselopolis/Paperboy/tree/main>. Last edited 14 April 2024.
10. Vivekjainmaiet, “Convert PowerShell to EXE in Two Easy Steps,” Medium, <https://medium.com/@vivekjainmaiet/convert-powershell-to-exe-in-two-easy-steps-21cc8cd94c13>, last accessed 01 March 2024.
11. Rothlisberger, Sam. “Embed a Malicious Executable in a Normal PDF or EXE.” Medium, <https://medium.com/@sam.rothlisberger/embed-a-malicious-executable-in-a-normal-pdf-or-exe-81ee5339707e>. Last accessed 02 April 2024.
12. PBER Academy, “PDF Payload (video),” <https://youtu.be/4ZsfFsJfHpY?si=yujERz-ltiwtUJcS>. Last accessed 13 April 2024.
13. Gurcinas, Vitalijus, et al. “A Deep-Learning Based Approach to Keystroke Injection Payload Generation,” Department of Information Systems, Faculty of Fundamental Sciences, Vilnius Gediminas Technical University, LT-10223 Vilnius, Lithuania. <https://doi.org/10.3390/electronics12132894>. Accessed 14 April 2024.