PDF Silent HTTP Form Repurposing Attacks

Web Penetration Testing

Aditya K Sood, SecNiche Security

Dated: May 2, 2009

Acknowledgement

I would like to thank **Adobe PSIRT** team for all discussion and feed back provided in completion of this paper.

[1] Abstract:

This paper sheds light on the modified approach to trigger web attacks through JavaScript protocol handler in the context of browser when a PDF is opened in it. As we have seen, the kind of security mechanism implemented by Adobe in order to remove the insecurities that originate directly from the standalone PDF document in order to circumvent cross domain access. The attack is targeted on the web applications that allow PDF documents to be uploaded on the web server. Due to ingrained security mechanism in PDF Reader, it is hard to launch certain attacks. But with this technique an attacker can steal generic information from website by executing the code directly in the context of the domain where it is uploaded. The attack surface can be diversified by randomizing the attack vector. On further analysis it has been observed that it is possible to trigger phishing attacks too. Successful attacks have been conducted on number of web applications mainly to extract information based on DOM objects. The paper exposes a differential behavior of Acro JS and Brower JavaScript.

[2] History of PDF Attacks

The PDF attacks were actually started with designing of PDF backdoors. The targeted version was version Adobe Reader 7. We have already seen URI handling flaws encompassing command execution in standalone PDF documents. The URI flaw worked fine. The Adobe took stringent steps to patch the stated flaws in an effective manner. There are number of attacks as mentioned below:

1. The PDF backdoors include malicious links and downloading content directly into browsers.
2. The PDF URI handling vulnerability in which protocol handlers are used to execute commands.
3. The direct URI Cross Site Scripting attacks.
4. Other memory corruption flaws.

The PDF provides a functionality which is always interactive. We are mainly concerned with the attacks that require user interaction. The above stated vulnerabilities or class of attacks worked fine with certain Adobe versions. There was no check on the link execution directly from the reader prior to version 8. With the development of version 8 all these vulnerabilities were patched. Even version 7 was incorporated with patches and fixes which gave rise to enhanced Adobe 7 versions. The researchers did a great job previously in giving birth to PDF attacks.

[3] Requirement:

This type of attack persists to fulfill the need for testing web applications. There are number of applications that allow uploading of PDF and provide functionality to view those files directly into browser. We are not aiming at third party attacks that can be launched directly from the web server. But yes, this attack can be used to launch different client side attacks if executed in a right manner. We looked at this attack as a part of web application testing primarily. The JavaScript model supported by Adobe for execution of dynamic content is used from testing perspective. We have successfully examined and executed attacks on the domain where an application allows the uploading of PDF files.

[4] Attack Scenario:

This attack is composed of designing a malicious PDF form with no standard inputs. The POST call in HTTP object is used to dispatch the JavaScript code directly to the domain which hosts that PDF file. No Security check is performed on Inline JavaScript that uses protocol handler.

[5] Adobe Statement:

The security is implemented mainly for cross domain calls. A check is produced when ever a third party URL is contacted. These restrictions work fine. But our analysis proves that Adobe JavaScript model can be used to test web applications (enterprise) through this attack vector.

Adobe stated *"Our position is that, like an HTML page, a PDF file is active content."*

It is right because for JavaScript working the object should be treated as dynamic. But our aim is to demonstrate that the interdependency factor among different software's can raise security concern.

[6] Contradiction:

It is hard to determine that it is a kind of vulnerability in software. The consideration is more of a JavaScript as a functional perspective. The question arises whether a check has to be introduced on JavaScript protocol handlers or not. Is there a need of implementation of certain security checks when a PDF is opened in the browser? But one thing is contemporary right that web applications can be tested through this attack vector. It is an exploitation of JavaScript model by using an attack vector of Form Repurposing. It is simple and easy but effective in number of cases.

[7] The XFA Model – XML Forms Architecture

The PDF forms are based on this model. It is an XML based architecture that supports PDF form based documents through the use of defined templates containing XML code. The prime characteristic is the dynamic reflow, actions and user interaction with the objects in the forms. There is lot of flexibility provided in this model. An intermediate XML Data Package Format is used. It provides flexibility in code. The format allows saving the document in PDF as well as XDP. It is based on the requirement. If the XML form is to be processed by the Acrobat Reader then format used for saving is PDF. If the form is to be used directly on the server then XDP is used. Let's have a look at the generic structure.

```
<?xml version="1.0" encoding="UTF-8"?>
<?xfa generator="AdobeLiveCycleDesignerES_V8.2.1.3144.1.471865" APIVersion="2.8.8118.0"?>
<xdp:xdp xmlns:xdp="http://ns.adobe.com/xdp/" timeStamp="2009-02-27T19:39:37Z">
<template xmlns="http://www.xfa.org/schema/xfa-template/2.4/">
  <?formServer defaultPDFRenderFormat acrobat7.0.5static?>
  <subform layout="tb" locale="en_US" name="form1">
    <pageSet>
      <pageArea id="Page1" name="Page1">
        <contentArea h="10.5in" w="8in" x="0.25in" y="0.25in"/>
        <medium long="11in" short="8.5in" stock="letter"/>
      </pageArea>
    </pageSet>
    <subform h="10.5in" w="8in">
      <field h="11.9052mm" name="Hit_Hard" w="146.0496mm" x="31.75mm" y="73.025mm">
<?templateDesigner isHttpSubmitObject true?>
        <ui><button/> </ui>
        <font typeface="Myriad Pro"/>
        <caption>
          <value>
<text>Definite Gift for You on your Birthday - My Dear</text>
          </value>
<para hAlign="center" marginLeft="0pt" marginRight="0pt" spaceAbove="0pt" spaceBelow="0pt"
textIndent="0pt" vAlign="middle"/>
<font baselineShift="0pt" size="8pt" typeface="Verdana" weight="bold"/>
       </caption>
<border hand="right">
          <fill>
            <color value="212, 208, 200"/>
          </fill>
<?templateDesigner StyleID apbx1?>
</border>
        <bind match="none"/>
        <event activity="click">
<submit format="formdata" target=" " textEncoding="UTF-8"/>
        </event>
      </field>
```

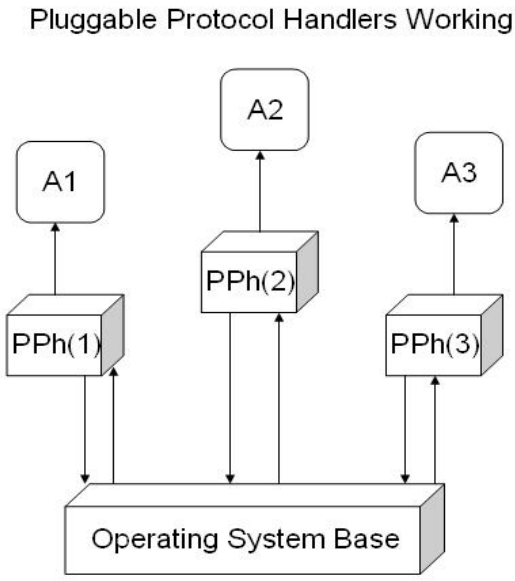
The above presented code shows the composition of PDF form based on XFA Model. The XFA DOM object is used for designing XML document tree and XPath is used to carry the blind XPath queries to be executed through JavaScript. The XFA DOM supports the Load Xml method and other uses apply XPath method. Once the XML object is ready, then the required code is parsed to process these objects and change it into desired document object tree. The overall functionality is based on two objects. There is an interdependency among these objects which primarily sets the functioning of XML based PDF forms. The objects are:

1. XFA DOM Object.
2. XPath Object

XFA allows multiple form fields with the same name and multiple copies of the same form. This model is very useful in designing number of forms used for web application testing by generating PDF with fuzzed inputs. Our attack also utilizes this XFA model in order to produce attacks based on JavaScript protocol handler.

[8] The Protocol Handler Anatomy:

It is one of the best practices to look into working stature of pluggable protocol handlers that are used in a browser system. Most of the vulnerabilities encompassing browser environment originate from this. By looking into the pluggable handler, one can see how the application works when a dispatcher call is made directly to any other software installed on the system. Usually that software can be open in the browser context and the content is considered as html. One should carefully check the parameters passed. A general behavior works as structured below:



A1 , A2, A3 → Applications
PPh(1), PPh(2), PPh(3) → Pluggable Protocol Handlers

The parameters include inputs such as strings, numbers etc due to which after a specific limit buffer overflows occur. So it considered to be as a security test. It's like disseminating a custom URL that is provide with protocol or designed manually. The designed custom URL is used after installation of an application. The PDF document uses certain protocol handlers for functional purposes. The browser understands below mentioned handlers for different protocols efficiently. For Examples:

a) mailto:
b) file:
c) ftp:
d) telnet:
e) view-source:
f) chrome:
g) gopher:
h) http:
i) https:
j) JavaScript:
k) news:
l) res:

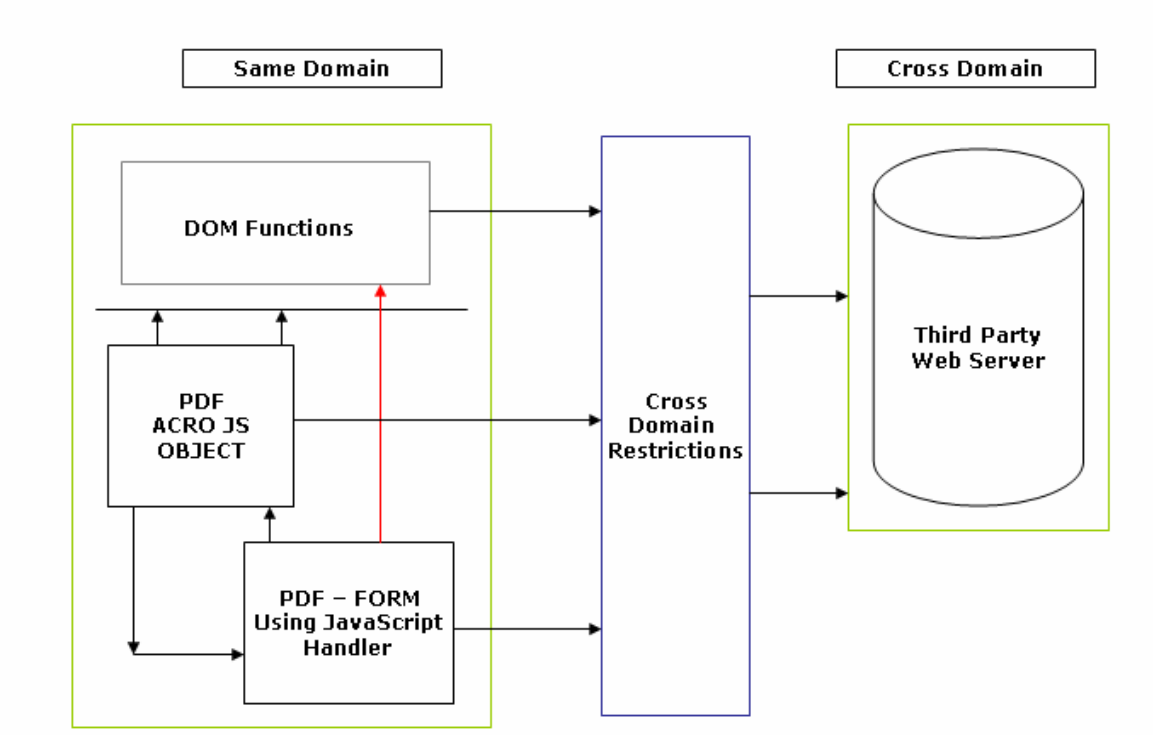
The protocol handlers make a call to the respective software or object to which it is registered for. As our attack revolve around JavaScript mainly, the JS protocol handler executes all JavaScript related functions and calls. If a PDF is designed to execute certain number of JS calls, it works fine in the context of browser when a PDF is viewed in it.

[9] The Acro JS / Browser JavaScript – Attack Model

There are a lot of differences between the Acro JS and standard JavaScript used. Adobe has implemented JS with different specifications when there is a requirement in PDF. The controls that are designed in PDF work efficiently with Acro JS. For Example: you require an app object in order to trigger certain JS calls in PDF itself. We are laying stress on the standalone PDF's here. But these objects work fine even when a PDF is opened in the browser. This is because the PDF Reader software which is opened in the browser context to display that file and the objects are executed too. The security mechanism comes to play as a part of software restriction which is nothing to do with browsers. If the objects are not designed appropriately with Acro JS it will not work. There are certain points that are mentioned below:

9.1 The Acro JS objects can not access HTML Objects directly.
9.2 There is no inheritance characteristic defined for Acro JS objects and are individualistic in functioning.
9.3 Acro JS is extracted from ECMA Script.
9.4 Objects in Acro JS are entirely different from JavaScript (HTML) Objects.

But the JavaScript work interchangeably. The major difference lies in the Document Object Model. Adobe introduces this behavior for more robustness and dynamic content. Acro JS does not support DOM. It means the document object can not be accessed with Acro JS. On the contrary JavaScript in browsers require DOM for effective rendering of web applications basically dynamic HTML content. One thing is sure one can not extract DOM based information by using Acro JS objects. That's why it is hard to attack web application with Acro JS when the PDF is opened in browser. Let's look at the JavaScript access check model with respect to same and cross domain.



Adobe PDF – Acro JS and JavaScript Handler Access Check Model

The use of JavaScript handler provides a functionality of executing calls with objects like window, document etc. The security mechanism does not allow the execution with standalone PDF's but it can be bypassed to some extent when rendered in browser. Using the DOM, the attacker gain access to the PDF document (and Acrobat) in a structured and object oriented fashion. A document object model basically sets out what objects exists in the document and the features that can be used. This behavior shown in red can be exploited as per the need of the penetration tester to perform the tests in the number of web applications.

[10] Understanding the HTTP Form Submission in Acrobat Reader / Designer

One can design malicious PDF in different ways. It depends on the requirement of an attack vector. The basic point is to understand the element of attack while performing penetration testing. The designing of forms require a specific XML submit button code for executing this control in the browser context. When a HTTP submit button is placed under mentioned point should be taken care of:

10.1 The HTTP submits form requires a destination URL. Adobe has implemented a check on it for cross domain access.

10.2 JavaScript protocol handler should be used inline in the URL box as a destination point.

10.3 It uses HTTP POST request to return their data from the URL.

10.4 Direct script injection does not work.

10.5 Information is mainly restricted to the execution of DOM calls on same domain.

10.6 The URL should be pointed to HTTP: protocol handler but we use JavaScript instead of it.

10.7 URL Encoding mechanism can be used directly.

10.8 It can be signed appropriately with a certificate so that data in the form buttons should not be tampered once submitted to the URL.

Note: Adobe points it to be as a functionality of protocol handler to execute scripts in the same domain. But this functionality has been subverted for web penetration testing purpose. It has been tested on number of browsers and attack has been implemented on real world scenarios. The outcome is always positive, the DOM calls executed successfully.

A very simple example code:

```
<subform h="10.5in" w="8in">
      <field h="11.9052mm" name="Hit_Hard" w="146.0496mm" x="31.75mm" y="73.025mm">
        <?templateDesigner isHttpSubmitObject true?>
        <ui><button/> </ui>
        <font typeface="Myriad Pro"/>
        <caption><value> <text>Definite Gift for You on your Birthday - My Dear</text>
          </value>
 <para hAlign="center" marginLeft="0pt" marginRight="0pt" spaceAbove="0pt" spaceBelow="0pt"
textIndent="0pt" vAlign="middle"/> <font baselineShift="0pt" size="8pt" typeface="Verdana"
weight="bold"/> </caption>
        <border hand="right"> <fill> <color value="212, 208, 200"/>
          </fill> <?templateDesigner StyleID apbx1?></border> <bind match="none"/>
<event activity="click">
<submit format="formdata" target="javascript:alert(&amp;quot;PWNED - Silent HTTP Form Repurposing
Attacks - Execution of DOM Call Successful&amp;quot;);" textEncoding="UTF-8"/>
</event> </field>
      <draw h="56.398mm" name="StaticText1" w="190.5mm" x="9.525mm" y="12.7mm">
        <ui> <textEdit/> </ui> <value>
          <exData contentType="text/html" maxLength="0">
<body xmlns=http://www.w3.org/1999/xhtml xmlns:xfa="http://www.xfa.org/schema/xfa-data/1.0/"
xfa:APIVersion="2.7.0.0"><p style="text-align:center;font-family:'Times New Roman';font-
size:18pt;letter-spacing:0in"><span style="xfa-spacerun:yes">

</span>PDF Silent HTTP Form Repurposing Attacks<span style="xfa-spacerun:yes"> </span>
</p><p style="text-align:center;font-family:'Times New Roman';font-size:18pt;letter-
spacing:0in"><span style="xfa-spacerun:yes"> </span></p>
```

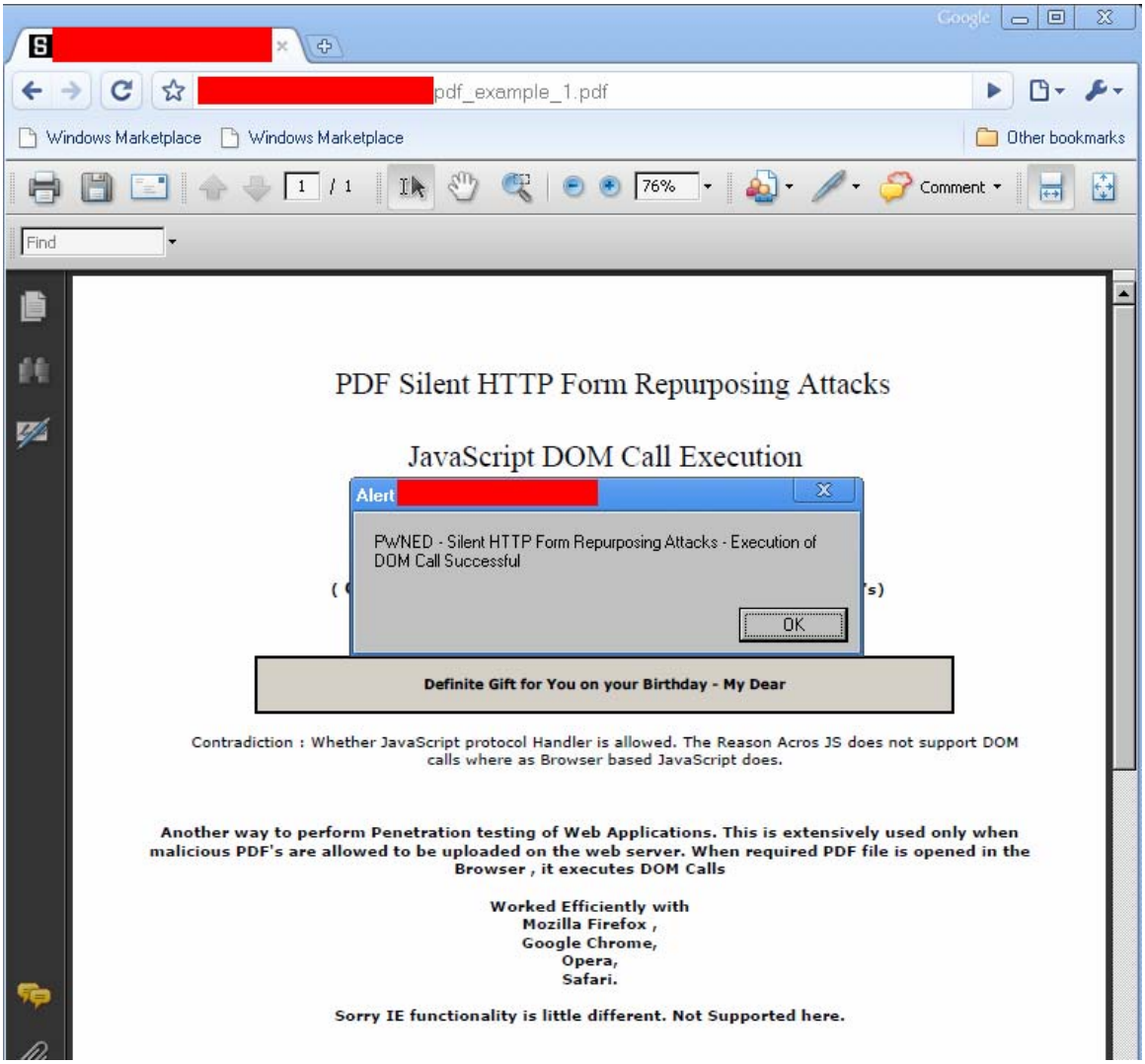
```
<p style="text-align:center;font-family:'Times New Roman';font-size:18pt;letter-spacing:0in">JavaScript
DOM Call Execution</p><p style="letter-spacing:0in"><span style="xfa-
spacerun:yes"> </span></p><p style="letter-spacing:0in">
<span style="xfa-spacerun:yes"> </span></p><p style="letter-spacing:0in">
<span style="xfa-spacerun:yes"> </span></p><p style="letter-spacing:0in">
<span style="xfa-spacerun:yes"> </span></p><p style="letter-spacing:0in">
<span style="xfa-spacerun:yes">

</span>Attack Vector - Pen Testing Web Application : PDF Uploading</p><p style="text-
align:center;font-family:Verdana;font-size:9pt;font-weight:bold;letter-spacing:0in">
<span style="xfa-spacerun:yes"> </span>(<span style="xfa-spacerun:yes"> </span>
```

---

[11] Brower State Check

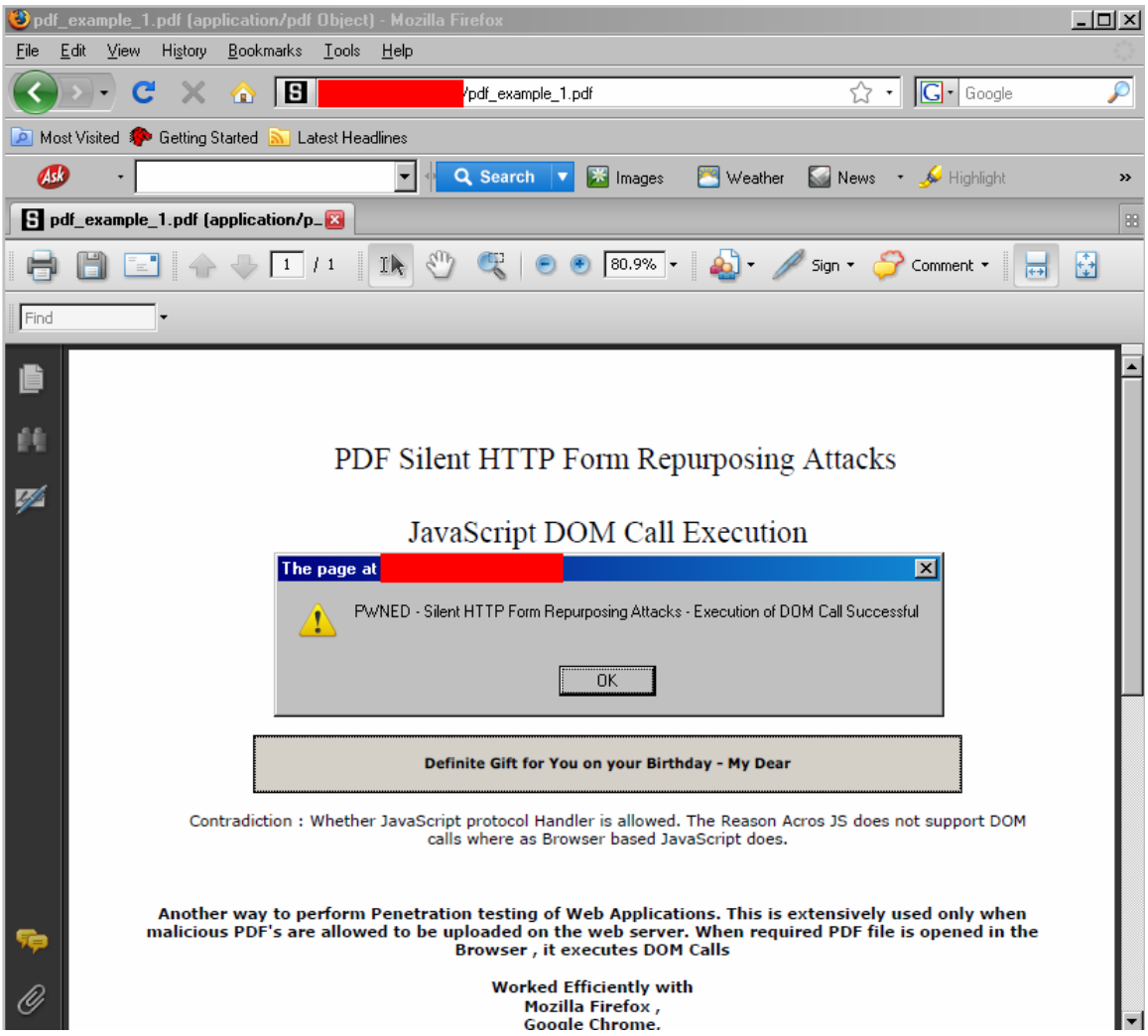
The document should be uploaded on the web application and must be viewed in the browser to retrieve more information. Let's analyze this behavior with different browsers:

[11.1] Google Chrome:



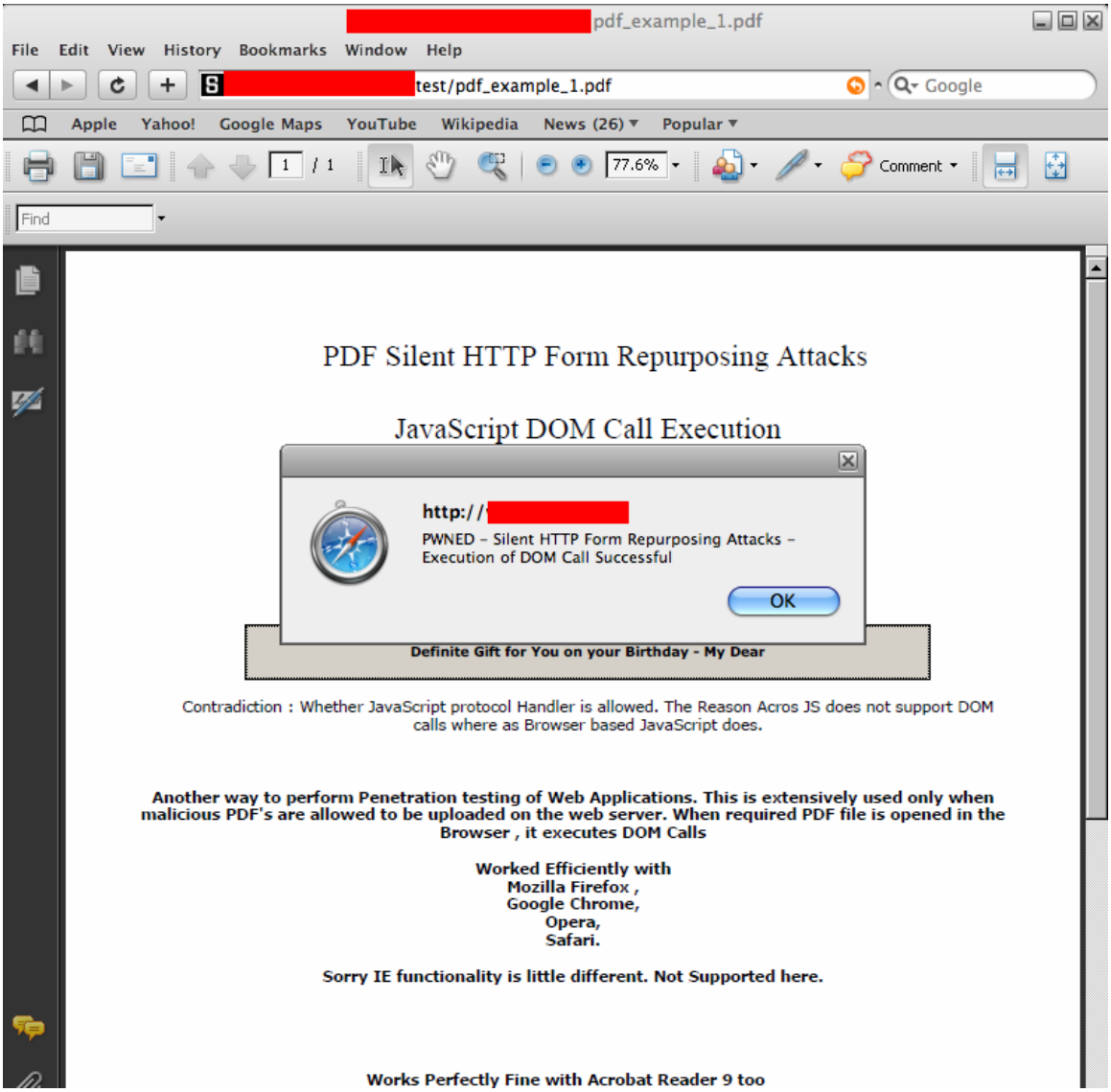
Result: Positive

[11.2] Mozilla Firefox



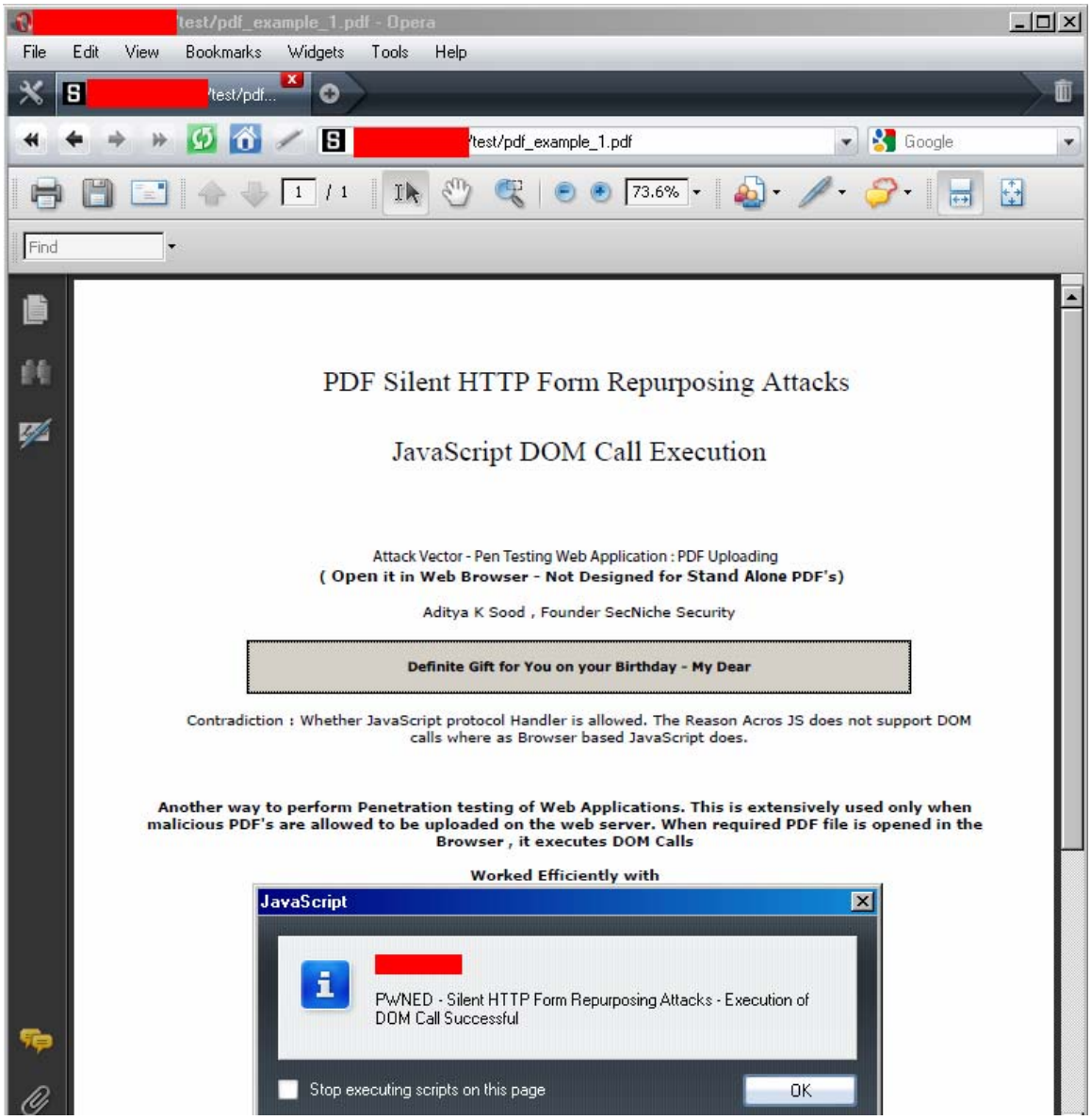
Result: Positive

[11.3] Safari



Result: Positive

[11.4] Opera



Result: Positive

This shows most of the browser work efficiently with this type of attack vector.

Please see the Appendix for live case study.

[12] Recommendations

There are certain recommendations that need to be implemented in order to prevent this type of behavior which are mentioned below:

1. Never allow the PDF file to be opened in the browser itself.

2. In designing custom application with different user roles appropriate check should be applied on the type of file being uploaded. The content-disposition parameter should be applied in a right manner because it explains the behavior of content whether it should be used inline or an attachment. The RFC 2183 clearly explains about the security problems. It is advisable the content should be treated as attached and not inline in the body part. Avoid inline opening of PDF documents.

   ```
   Content-disposition: attachment; filename=gift.pdf
   Response.AddHeader "content-disposition","attachment; filename= gift.pdf"
   ```

With the new security features the adobe has prevented the Cross Context Scripting. This is because certain attacks won't work on standalone PDF documents. You will not be able to access local file system.

3. The filters should be applied appropriately so that content is not rendered as dynamic. Use of flash to display the PDF document is a good solution to prevent these type of attacks. Numbers of online viewers follow this approach of viewing PDF document as SWF file by appropriately converting it.

The above presented solutions can limit this attack to some extent. The PDF should be treated as standalone document and should not be allowed in browsers for viewing**.**

[13] Conclusion:

This attack vector explains the manipulation of JavaScript model in PDF to test the web applications. There is a contradiction about the JavaScript protocol handler inside PDF as Acro JS and Browser based JavaScript is different in their specifications. If it is considered as a part of functionality then this attack vector can be used any time for conduction web penetration testing by uploading malicious PDF files. It works efficiently on all versions of Adobe Reader if this behavior is tested. It reveals all information that can be extracted by DOM functional calls. It is successfully tested on number of platforms and noticed same output all the time. Well , attack vector depends on the benchmark through which simulated functions can be used in differential way.

[14] References:

1. http://eusecwest.com/esw08/esw08-sood.ppt
2. http://www.adobe.com/devnet/acrobat/javascript.html
3. http://www.ietf.org/rfc/rfc2183.txt

Appendix

Case Study: http://www.pdfmenot.com

"PDFMeNot– an online service that lets you read PDF files inside the browser without Adobe Reader."

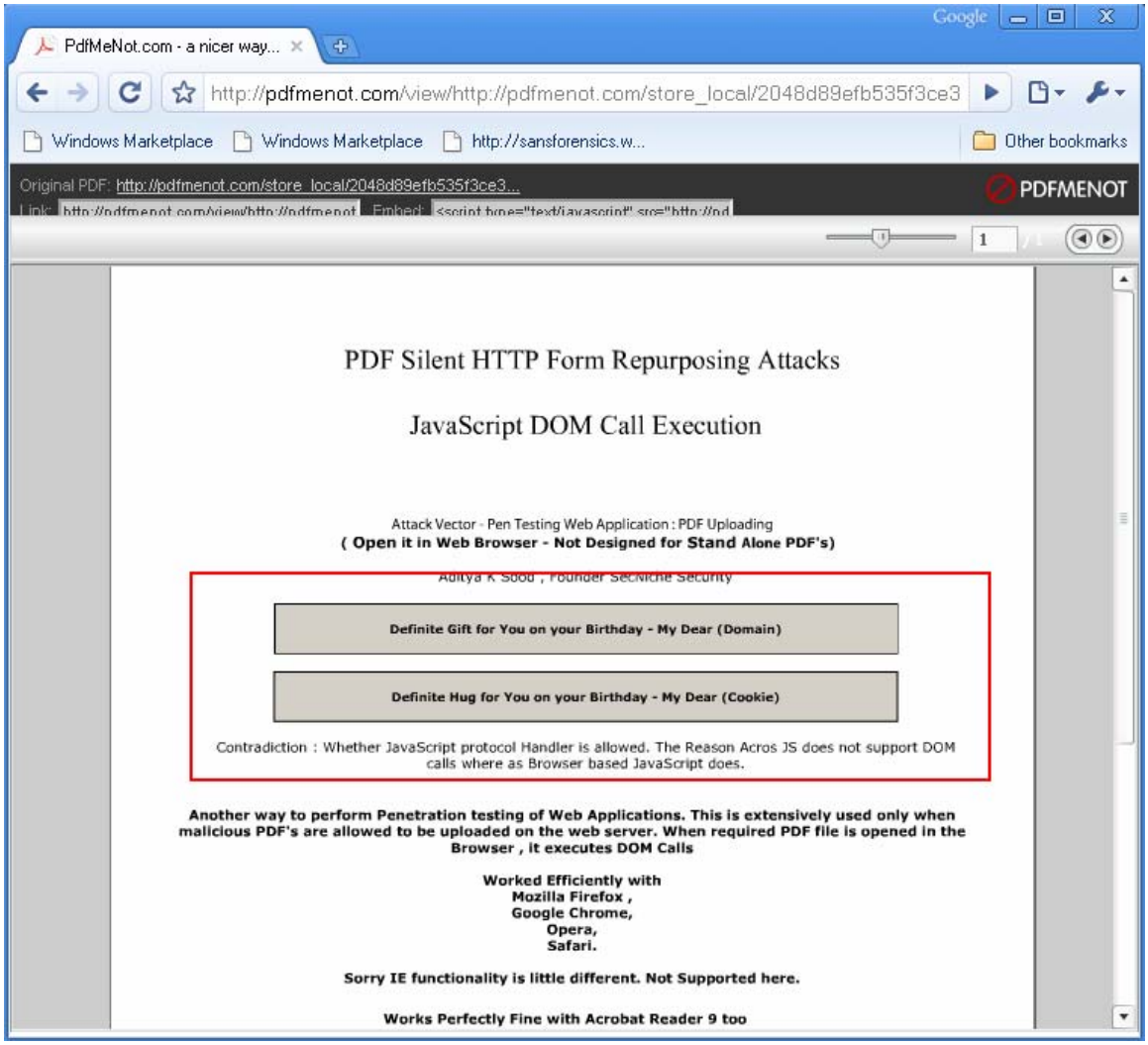
This test proves the below mentioned issues:

1. The use of appropriate conversion and filter results in prevention of these types of attacks.
2. Providing a direct link to open the PDF in browser with an interface with Adobe Reader favors this attack.

Let's analyze the cases:

Step 1: The attack file is uploaded

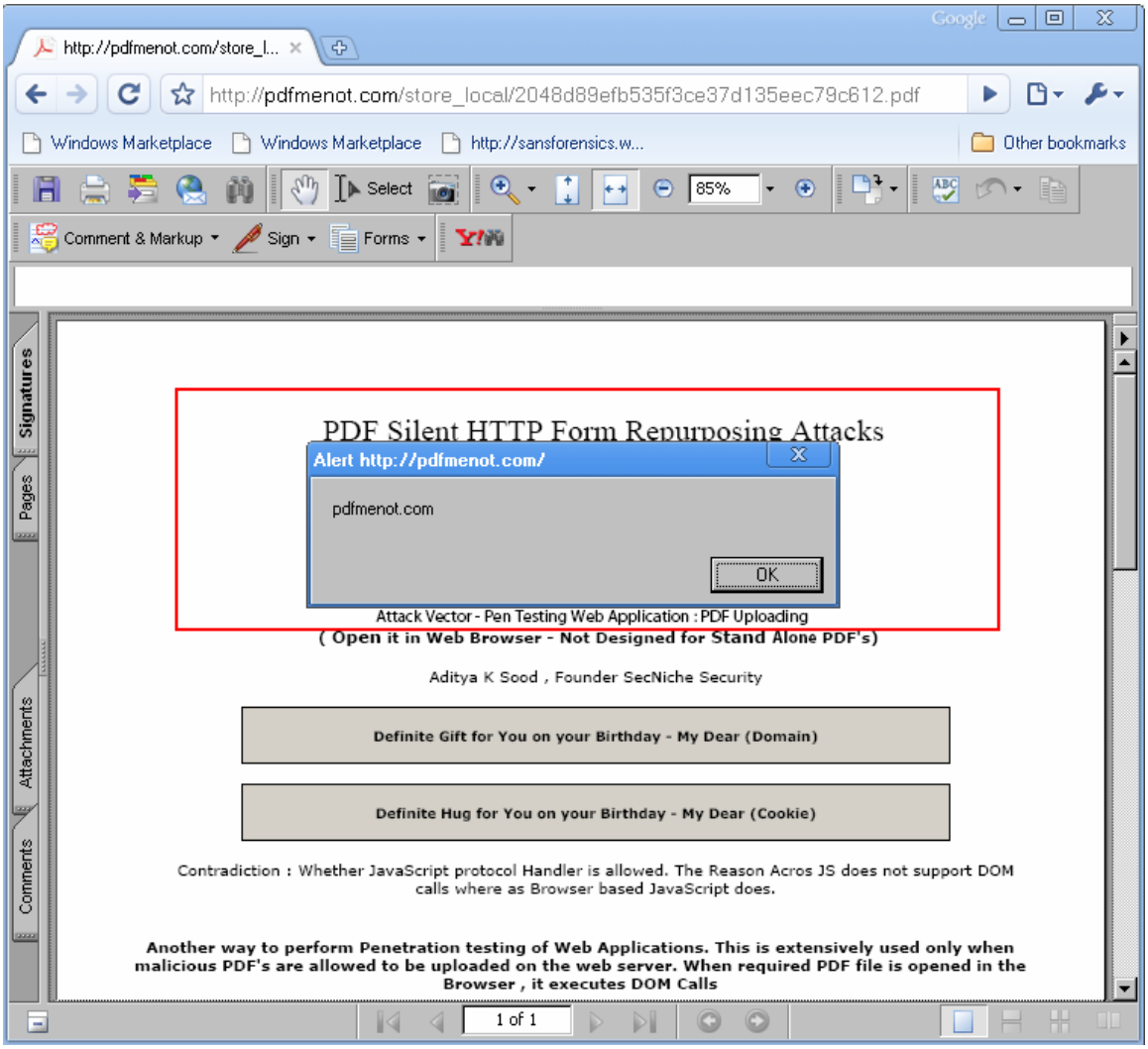
Step 2: The file is converted to view online without acrobat reader. As a result of it no interaction is possible with the attack fields present on the PDF. This actually serves our first issue as discussed above.



The attack does not work in the case provided above. But on the contrary the direct link to PDF is provided as:

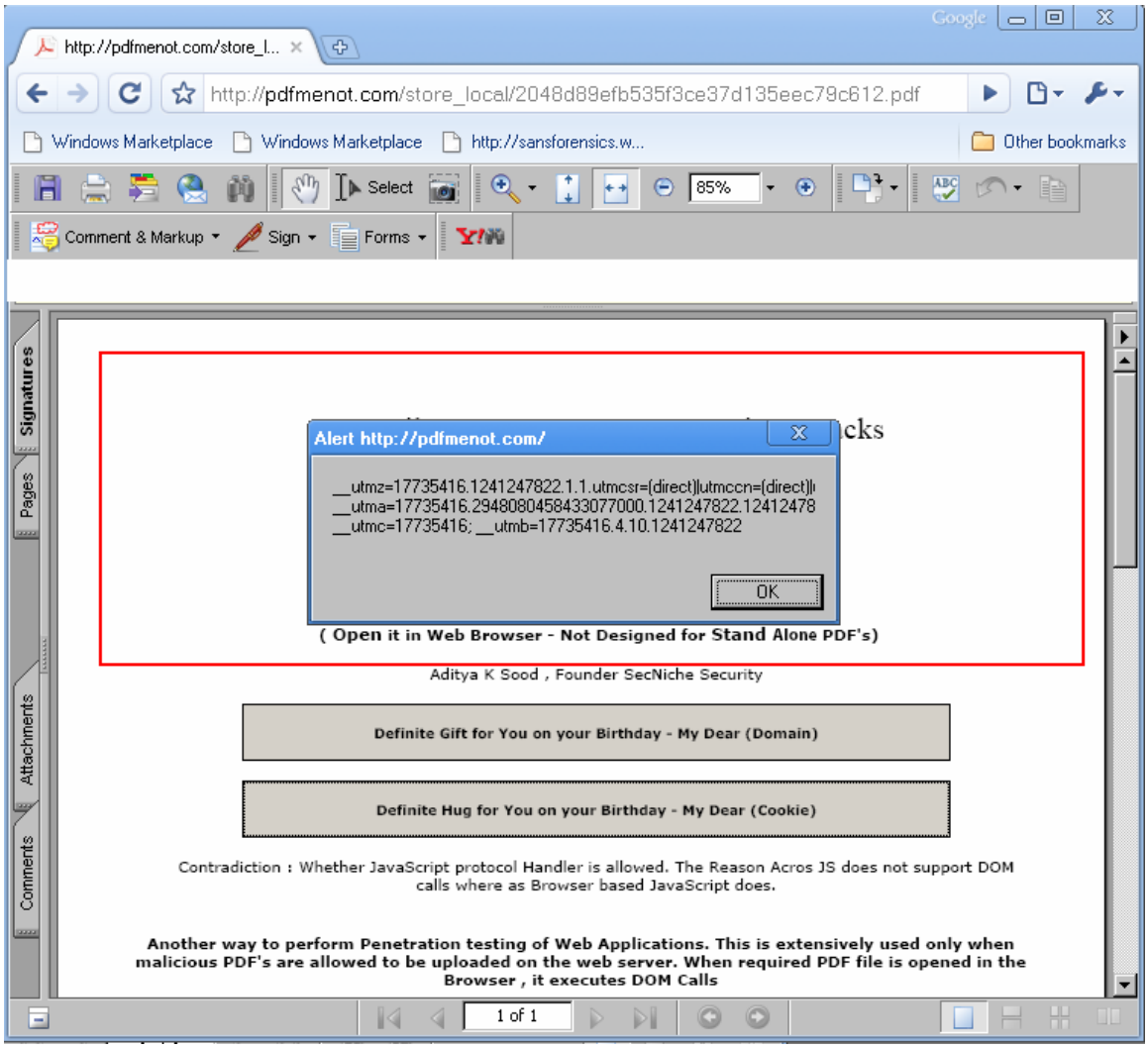
When a link to PDF file is executed it results in opening of PDF file in browser in the context of that domain. We are able to produce to different variants which are presented below.

1. Domain Check - Successful



It is easy to extract the domain by executing “document. domain” DOM function through JavaScript

2. Cookie Extraction - Successful



This explains the overall behavior of the nature of these types of attacks.