Universal PDF XSS V1.1

Weak web browser add-ons and plug-ins are threatening web applications and workstation security!

Q2/2007 Cyrill Brunschwiler, Compass Security AG

An advanced technical study about mitigation approaches and a definitive solution to prevent web applications from Universal PDF XSS exploits.

Introduction

The Universal PDF XSS vulnerability was first introduced at the 23rd Chaos Communication Congress[1] and was originally discovered by Stefani Di Paola[2]. The vulnerability is neither a common web application weakness nor is it a client web browser problem. The vulnerability targets a widely used browser plug-in, which enables users to read Adobe PDF documents: The Adobe Reader plug-in.

"Which versions are affected?"

Good question. Basically, Acrobat Reader and Acrobat versions 7.0.8 and earlier on Windows and Linux in any browser (see Advisory [3]). However, there are many configurations and combinations which seem to be safe.

"Why should I care then? This is definitely an Adobe issue. However, what is XSS[4] anyway?"

XSS is the abbreviation for a commonly known web application attack: Cross-Site scripting. The technique is based on the web browser scripting language JavaScript which allows changing web page contents dynamically and thus to improve a web site's usability. Unfortunately, JavaScript may be used to spoof faked login forms or to steal session credentials as well. Having valid session credentials enables an attacker to impersonate customer accounts. However, attackers need a weakness in the web application itself or may abuse the newly discovered Adobe Reader plug-in issue to successfully execute a malicious JavaScript code fragment in the victims' web browser.

Furthermore, proof of concepts described in several blogs[5] point out, that the weakness even allows attackers accessing and transmitting data from a vulnerable client computer's file system. Since this specific issue can be fixed by updating to the latest Acrobat version only, it is left to the users responsibility and therefore, the following text focuses on web application protection.

"Oh, XSS! I remember we recently got a web application firewall installed which mitigates this type of attack. "

Some entry server products actually can partly mitigate Universal PDF XSS attacks if they are configured to do so, but very few are. The mechanism referred to is also known as URL encryption or URL protection whereas the entry server replaces all links in HTML pages with an encrypted representation. Since the key for encryption is bound to the client's session, every link is unique and hard to guess. Nonetheless, there are possibilities where this feature, which was formerly designed to mitigate Cross-Site Request Forgery CSRF[6], does not always prevent from Universal PDF XSS.

"Okay, tell me how the request looks like. Our engineers will update the web application firewall patterns."

Unfortunately, the attack vector is never sent to the server but let me explain that in detail.

The Attack Vector and Its Finesse

The simplest way to exploit the Adobe PDF Plug-In vulnerability is to craft a special link such as:

```
http://any.whe.re/file.pdf#a=javascript:al
ert(document.cookie)
```

The browser will send the following request to the server:

```
GET http://any.whe.re/file.pdf HTTP/1.0
```

You might have recognized that the so-called anchor is missing within the browser request:

```
#a=javascript:alert(document.cookie)
```

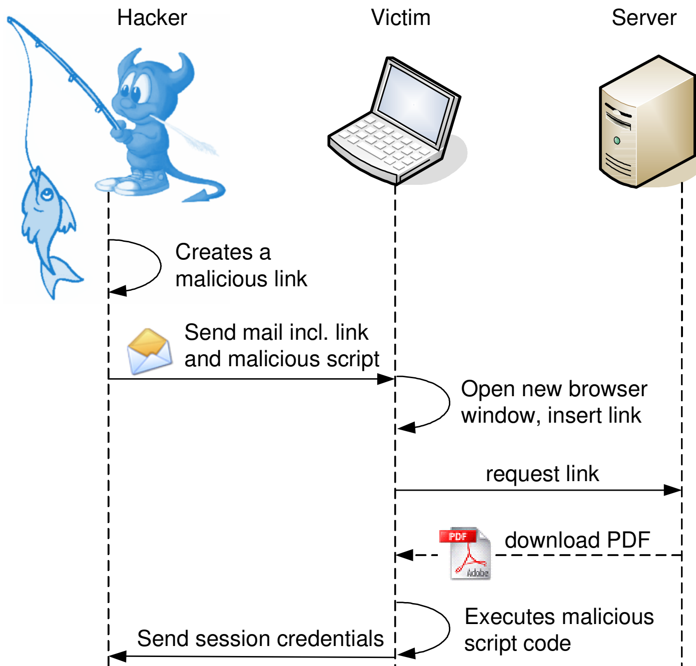
That is why application firewalls cannot detect this attack! Later on, the browser will call the anchor by the time the document (file.pdf) has fully returned from the server and loaded in the current browser window.

The Attack Sequence

As you know now, how the attack vector looks like, let us have a closer look at the attack sequence. The diagram below assumes that the victim has to click on an arbitrary link which causes the web browser to request the server for the PDF document.

If the victim has already authenticated, the server will deliver the requested PDF immediately. Otherwise the user will be asked for his or her credentials first.

Afterwards, the malicious JavaScript code from the #anchor gets unfiltered passed through the Adobe Reader Plug-In into the web browser, which executes it and sends the victim's session credentials to the hacker. The script might also show another login screen or ask for further credentials… Several roads lead to Rome (not to say all).



BSD Daemon Copyright 1988 by Marshall Kirk McKusick.
All Rights Reserved.

"This does not apply to us. Our servers use strong encryption and require strong 2-factor authentication."

To clearify it right from the beginning. Neither SSL encryption nor Challenge-Response or One-Time-Password mechanisms are designed to protect against such an attack.

"Hmm, I mean is there a way to mitigate the attack? I saw several proposals float around the net."

Web research revealed several potential solutions. Some more and some less secure. Even the OWASP[7] solution lacks mitigation in multi-tier architecture environments.
The different proposals are discussed and rated using smiley's ☺ for positive value and bombs 💣 for negative value.

Option 1: Referrer Header Check

Usability: 💣💣
Security: 💣💣💣

It is proposed to verify whether a certain PDF document was requested from a web page within the same virtual domain by checking the Referrer HTTP request header before delivering the document.

This proposal has several drawbacks. First of all, clients cannot access PDF documents coming from 3rd party web sites (e.g. Search Engine). Furthermore, the referrer can be spoofed (e.g. using Flash).

Option 2: Content-Disposition Header

Usability: 💣
Security: 💣💣

It is proposed to insert a Content-Disposition Header in every HTTP response which returns a PDF file.

```
Content-Disposition: attachment; a.pdf
```

This measure causes most browsers to display the Open/Save as… dialog rather then loading the PDF in the embedded Acrobat Plug-In. Opening the document from the dialog does not launch the Universal PDF XSS attack. Internet Explorer MIME-Sniffing feature still causes the Plug-in to start if the Content-Type Header is just changed to application/octet-stream.

Option 3: OWASP Filter Servlet [8]

Usability: ☺
Security: 💣

It is proposed to setup a redirect mechanism using a token, which is appended to the origin URL and allows verifying that the link was not designed by an attacker. The token itself is based on the client IP address and the current time.

Unfortunately, the client IP address in entry server and proxy environments is always the same and therefore an attack during the ten seconds timeout is still possible.

"Hold on. I was asking for a definitive solution."

As you may have recognized, it is pretty difficult to propose a feasible solution. Currently it looks like that the only safe way to protect from the Universal PDF XSS attack, is to create a secure random token which is bound to the client's session ID and attached to every requested PDF document link.

The code to the following sequence diagram is appended to this document and is considered beta status.

The Solution Explained

If a client somehow clicks on a manipulated link, his or her web browser will automatically request the file.pdf for the first time. At that stage, the server has no chance to detect if a malicious anchor was part of the link.

Therefore, the server creates a new random token and if necessary a new session.

Afterwards, the random token is saved to the current session. To overwrite a potentially malicious anchor, the server redirects to a link which has #X appended. Moreover, a token gets appended to the redirect URL to be able to recognize whether the requested link was already sanitized.

Item	Purpose
#X	Overwrite a malicious anchor
random	Remember if the link was cleaned
session	Check that not the hacker asked for a random token

The second request to the server will contain the random token and the former session. In the sample filter code, session handling is based on Cookies but the code could easily be rewritten to allow URL rewriting which would support clients that reject Cookies as well.

However, in the second request the server has to verify whether the random token was created for the current session to make sure a hacker did not acquire valid tokens and is trying to phish [10] clients with them.

When the server finally delivers the requested PDF document, the originally requested malicious link is sanitized in a way which cannot harm vulnerable Adobe Reader Plug-Ins anymore and thus poses no threat to the web application.

Hacker
Client
Server
Creates a malicious link
http://any.whe.re/file.pdf#a=javascript:alert('XSS')
Open link in browser
Request http://any.whe.re/file.pdf
Create session
Create random
session.add(random)
Redirect http://any.whe.re/file.pdf?t=random#X + set session
#a=javascript:alert('XSS') => #X
Request http://any.whe.re/file.pdf?t=random (session id)
Check if random belongs to the current session
Sanitized link does not harm vulnerable Acrobat Plug-Ins anymore

BSD Daemon Copyright 1988 by Marshall Kirk McKusick. All Rights Reserved.

Developer Notes

The proposed solution is thought for web entry server infrastructures. The filter has to be installed and configured in the presentation tier behind the entry component. Furthermore, it is assumed that the entry server or login application handles the sessions correctly (e.g. session fixation prevention) and requests users to authenticate for the targeted PDF documents.

There is a remaining threat in case of the module is used to protect PDF resources which do not require users to authenticate for. Basically, the module does not protect web sites which do not need any authentication. To improve the filter for such purposes one should restrict session handling to cookies. Otherwise the module can be bypassed by conducting a session fixation attack.

The Java Servlet snippet below shows how to ensure that the application only accepts sessions which were delivered as cookies.

```
if (req.isRequestedSessionIdFromCookie())
{
   //session delivered in cookie
}
```

Keep in mind that this mechanism is based on cookies and a client has to support this feature to be able to access the PDFs. Web crawlers generally do not support cookies and will fail to update the index with your PDFs content because the module will refuse access.

Browser Issues

It has been reported that some browsers do not properly handle the redirect URL and cause the entire filter to fail. This results in forbidden access every time the client tries to request a PDF resource. Sites which preferred to add the OWASP PDF-XSS filter might have recognized the same issue as well.

At least Internet Explorer 6 SP1 was identified to cause the problem in particular setups.

In this case, the Universal PDF XSS filter does not properly work because of the browser wrongly assumes that the anchor of the redirected URL is part of the query itself.

```
Sample filter response:

HTTP/1.0 302 Moved Temporarily
Location: /file.pdf?t=RANDOM_TOKEN#X
...
```

```
Sample browser request:

GET /file.pdf?t=RANDOM_TOKEN%23X HTTP/1.0
Cookie: JSESSIONID=RANDOM_SESSION
...
```

Unfortunately, the anchor gets URL encoded (# => %23) and becomes part of the token value, which the filter recognises as such. Of course, the filter will never find an appropriate token in the current user's session.

```
Assumed token value: RANDOM_TOKEN%23X
```

Therefore, a little fix has to be applied which recognises that the Token contains an anchor and cuts the remaining string.

```
if (token != null)
{
   int anchor = token.indexOf('#');

   if (anchor >= 0)
   {
      token = token.substring(0, anchor);
   }
}
```

The sample filter in the appendix already got this fix applied.

Last but not least, the pragmatic way

Site administrators might want to exploit the Universal PDF XSS vulnerability by themselves just to identify vulnerable Adobe plug-ins and forwarding the bogus clients to an update web page.

About the Author

During his studies for Dipl. Inf. Ing. FH, which he completed in December 2004, Cyrill Brunschwiler concentrated on computer security, as well as on application and Internet technologies. As a parttime employee he supported the Compass Security team, alongside his studies with software engineering and the development of our course and e-learning environment for applications- and network-security. He joined Compass Security AG as a full time security analyst in January 2005.

cyrill.brunschwiler@csnc.ch
http://www.csnc.ch/

About Compass Security

The Job of a security specialist is like searching in the fog. The more opaque the environment the harder it is to find traces and establish methods and tactics. A good compass can help to determine the direction and choose a path that will securely lead to the destination.

Compass Security Network Computing AG is an incorporated company based in Rapperswil (Lake Zurich) Switzerland that specialises in security assessments and forensic investigations. We carry out penetration tests and security reviews for our clients, enabling them to assess the security of their IT systems against hacking attacks, as well as advising on suitable measures to improve their defences.

Compass Security has considerable experience in national and international projects. Close collaboration with the technical enable universities of Lucerne and Rapperswil Compass to carry out applied research so that our security specialists are always up-to-date.

GLÄRNISCHSTR. 7
POSTFACH 1671
CH-8640 RAPPERSWIL

Tel. +41 55-214 41 60
Fax +41 55-214 41 61
info@csnc.ch www.csnc.ch

References

[1] 23rd Chaos Communications Congress
http://events.ccc.de/congress/2006/Home

[2] Adobe Acrobat Reader Plug-in - Multiple Vulnerabilities
http://www.wisec.it/vulns.php?page=9

[3] Cross-site scripting vulnerability in versions 7.0.8 and earlier of Adobe Reader and Acrobat (CVE-2007-0045)
http://www.adobe.com/support/security/advisories/apsa07-01.html

[4] Cross-site scripting explained
http://www.spidynamics.com/whitepapers/SPIcross-sitescripting.pdf

[5] Discussion,remediation trials and PoC on how to exploit local PDF documents (blogs of pdp and RSnake)

http://ha.ckers.org/blog/20070103/pdf-xss-can-compromise-your-machine/

http://sla.ckers.org/forum/read.php?2,4785

http://www.gnucitizen.org/blog/universal-pdf-xss-after-party/

[6] Cross-site request forgery explained
http://www.tux.org/~peterw/csrf.txt

[7] Open Web Application Security Project
http://www.owasp.org/

[8] OWASP PDF XSS Filter Servlet
http://www.owasp.org/index.php/PDF_Attack_Filter_for_Java_EE

[9] Phishing: In computing, phishing is a criminal activity using social engineering techniques. Phishers attempt to fraudulently acquire sensitive information, such as passwords and credit card details, by masquerading as a trustworthy person or business in an electronic communication. (en.wikipedia.org)

[10] Sample Filter Source (please see the following pages)

Credits

Thanks to the Compass Security Team Members who spent their coffee break brainstorming and cross checking.

Thanks to Michael Brandmaier, Ivan Ristic, pdp and Amit Klein who all pointed out remaining risks and issues.

```java
/**
 * Software published by the Open Web Application Security Project (http://www.owasp.org)
 * This software is in the public domain with no warranty.
 *
 * @author     Jeff Williams <a href="http://www.aspectsecurity.com">Aspect Security</a>
 * @created    January 4, 2007
 *
 * The Compass Security modified filter is partly based on its OWASP forerunner.
 *
 * @author     Cyrill Brunschwiler <a href="http://www.csnc.ch/">Compass Security AG</a>
 * @created    May 4, 2007
 */

import java.io.IOException;
import java.security.SecureRandom;

import javax.servlet.*;
import javax.servlet.http.*;

public class UniversalPDFXSSFilter implements Filter {

        private static final String TOKEN_NAME_URL = "t";
        private static final String TOKEN_NAME_SESSION = "TOKEN";
        private static final int TOKEN_BYTES = 20;

        private static sun.misc.BASE64Encoder s_encoder = new sun.misc.BASE64Encoder();
        private static SecureRandom s_random = new SecureRandom();

        public void doFilter(ServletRequest request, ServletResponse response,
                        FilterChain chain) throws IOException, ServletException {

                HttpServletRequest req = (HttpServletRequest) request;
                HttpServletResponse res = (HttpServletResponse) response;
                HttpSession session = req.getSession(true);

                // Get token from url
                String tokenUrl = req.getParameter(TOKEN_NAME_URL);

                // -- fix IE6 SP1 issue --------------------------
                if (tokenUrl != null) {
                        int anchor = tokenUrl.indexOf('#');

                        if (anchor >= 0) {
                                tokenUrl = tokenUrl.substring(0, anchor);
                        }
                }
                // ------------------------------------------------

                // Get token from session
                Object objToken = session.getAttribute(TOKEN_NAME_SESSION);
                String tokenSession = null;

                if (objToken != null) {
                        tokenSession = (String) objToken;
                }

                // On your marks...
                try {
                        // If the token in the url matches the previously saved session
                        // token and neither of them matches null then its fine to serve
                        // the document.
                        if (tokenSession != null && tokenSession.equals(tokenUrl)) {
                                // We're safe. Serve the document.
                                chain.doFilter(req, res);
                                return;
                        }
// next page ...
```

```
// ... pervious page


                else if (tokenUrl == null && tokenSession == null) {
                        // There's neither a token in the URL nor in the session.
                        // Let's create a new token for the session and redirect
                        // to the appropriate URL containing the token and #fake.
                        String token = createToken(session);
                        String url = createUrl(req, res, token);

                        // Save the token to the current session
                        session.setAttribute(TOKEN_NAME_SESSION, token);



                        // Redirect to the safe url
                        res.sendRedirect(url);
                        return;
                }
                else if (tokenUrl == null && tokenSession != null) {
                        // The URL does not contain token but the session already
                        // contains one. Looks like the user requested a document
                        // earlier. Let's recycle the token.
                        String url = createUrl(req, res, tokenSession);

                        // Redirect to the safe url
                        res.sendRedirect(url);
                        return;
                }
                else {
                        // This is a pretty interesting case. Either the client rejects
                        // cookies or it got hooked up. Might be worth a log record.
                        // Some might prefer beeing bit more kindly to their customers
                        res.sendError(HttpServletResponse.SC_FORBIDDEN);
                        return;
                }
        }
        catch (Exception e) {
                throw new ServletException(e);
        }
}

// @return the redirect URL having a token and #X appended
private String createUrl(HttpServletRequest req, HttpServletResponse res,
                String token) {

        // get the absolute document path => base
        String url = req.getRequestURI();

        // if exists, append the query to the base url using a ?
        String querystring = req.getQueryString();

        if (querystring != null) {
                url += "?" + querystring + "&" + TOKEN_NAME_URL + "=" + token + "#X";
        }
        else {
                url += "?" + TOKEN_NAME_URL + "=" + token + "#X";
        }

        return url;
}



// next page ...
```

```
// ... pervious page



        // @return the newly created token
        private String createToken(HttpSession session) {

                // fill a new array with random bytes
                byte[] rand = new byte[TOKEN_BYTES];
                s_random.nextBytes(rand);

                // encode the random string in base64 and replace unusable characters
                // base64 chars: ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
                return s_encoder.encode(rand).replace('+', 'X').replace('/','Y').replace('=','Z');
        }

        public void init(FilterConfig filterConfig) throws ServletException {
        }

        public void destroy() {
        }
}
```