Website security

What could hacked companies have done to minimise the data stolen?

Back on 20 April, Sony confirmed that its cloud service, the PlayStation Network, which provides a community for PS3 and PSP players and social gaming using those systems was suffering a 'network outage'. As the outage dragged on into another week, Sony finally admitted that PSN had been infiltrated by an outside hacking group and that the hack had resulted in customer information being downloaded. As soon as Sony discovered the hack, it took down PSN and Qirocity, its music-in-the-cloud service.

No details about how the hack happened were published then, or since. In his opening statement to the Subcommittee on Commerce, Manufacturing and Trade of the US House of Representatives Committee on Energy and Commerce on 2 June, Tim Schaaff, President of Sony Network Entertainment International, said: "By the evening of Saturday, April 23rd, we were able to confirm that intruders had used very sophisticated and aggressive techniques to obtain unauthorised access to servers and hide their presence from system admins."

He didn't expand on this rather hand-wavy explanation any further.

Sony did, however, release information about what had been stolen. It included names, addresses, email addresses, birthdates, PSN handles, but perhaps more importantly, logins (including passwords) and possibly credit card information. The breach exposed details about 77 million users of PSN, and a later hack against Sony Online Entertainment exposed details of another 25 million accounts.

Although Sony implicated the Anonymous hacking group or someone close to them for the hack, it is unclear whether they were involved or not. Not that it really matters, because a new group had formed and thrust itself into the limelight: Lulz Security, also known as LulzSec.

This new group had a declared mission to highlight bad network security by exposing code and account information from high-profile targets, and to advertise the break-in by posting jokey messages on Twitter and the hacked website (sometimes opting for impressive ASCII art showing various ships flying under a Lulz banner). The group has been incredibly prolific: ▸

first coming to light for hacking PBS and Fox News to highlight Bradley Manning's treatment at the hands of Wikileaks and the US government, then moving on to Sony. At the time of writing, the group had announced that it was disbanding, with police knocking on the doors of several suspected members.

Show your working

LulzSec sometimes provided information about how it performed the hack. This is invaluable as a teaching aid for developers who are responsible for the outward-facing computer systems of the companies they work for, if only they would use it.

By far the simplest attack LulzSec used was the SQL injection attack. This particular attack was used against Sony Pictures to compromise account information for over one million users. How does it work?

Let's give you the lay the land. The HackMe company has a website that, like many corporate websites these days, is a database-driven Content Management System or CMS. The content of the web pages – the text for articles, blog posts, and so on – is held in a database. When a user navigates to a particular page, the CMS will build the complete HTML for the page at run-time from a set of templates that define what a page should look like and its content. The page, if you like, is dynamic, not static. Some CMSes are open source or commercial (examples include WordPress and Moveable Type), while others are written in-house.

HackMe decided to write its own CMS, and took the approach that pages are to be numbered. People who come to its site to look at a particular page will navigate to a URL that looks like this: 'http://hackme.com/?page=42'.

The part after the question mark in the URL is known as a query. Generally the query will consist of a set of key=value pairs separated by ampersand characters. The query is extracted by the web server software as a set of parameters (known as the query string) that it then passes onto the web application that's rendering the pages. Simply put, the URL is split into 'http://hackme.com/' and 'page=42', so here we have one parameter whose key is 'page' and whose value is 42.

The CMS was written to take the query string, separate out the key=value pairs and then use them to identify the page needed and to build it and return it. So far, so good.

The developer in charge of writing that part of the code cut corners and decided to extract the page number by just taking everything after the equals sign, assigning it to a variable (here called 'pagenumber') and then constructing a SQL statement of the form:

```
SELECT 'content' FROM 'pages'
WHERE 'id' = '' + pagenumber + '';
```

We're assuming here that the database table is called 'pages' and the column that's storing the text for the page is named 'content'. The primary key for this table is 'column id', so this SQL statement says, in essence: "Get me the content field for the page whose id is 'pagenumber' from the 'pages' table."

If the URL had been constructed normally, the data returned would indeed be the contents of the content column for a particular page record. However, we can start to mess around with the input and hope that the developer that wrote this code did, in fact, cut corners. So, suppose, instead, we navigate to this URL: http://hackme.com/?page=42 UNION SELECT table_name FROM information_schema.tables --

The UNION operator combines two individual SQL statements and returns the results from both as a single result set. The double hyphen at the end is the token to start a SQL comment, and its effect is to hide anything else the developer might be adding to the SQL statement he's really trying to build. The effect of this would be to inject an extra SQL statement into the CMS developer's constructed SQL statement. The two statements would be executed instead of just the original one. And, what is worse, the results of the second would be tacked on to the first and thereby rendered in the page. We, the hacker collective, now know the names of the tables in the database, and can continue crafting more and more SQL statements to get at the actual data in those tables.

SQL injection

This is the basis of a SQL injection attack. We 'inject' some extra SQL code into some input directed at the CMS, and hope that the developer who wrote it didn't check the input to be valid, in range, of the correct

Short URLs

This is another quick example of sanitising data. If you use Twitter, you'll be familiar with URL shorteners like **http://bit.ly**, **http://awe.sm**, and so on. These are just simple database applications that match up a short URL to the longer one and can generate short URLs for new links. If we were to write our own URL shortener that would accept URLs like 'http://hack.me/hj8Gt6' and redirect them to the real page, it would behoove us to check that the short URL 'looks' like one of ours, isn't too long (say six characters maximum), only uses alphanumeric characters, and so on.

Only then, once we've sanitised the data from the outside world, would we access the database of short URLs to find out the corresponding long URL and perform the redirect. ■

Spotlight on… Cross-site scripting

Another vulnerability that's used more to hack the user of a website than the web server is XSS, or cross-site scripting. Again, it's a problem involving the sanitisation of data

XSS occurs when an unsuspecting user sends harmful data (usually JavaScript code) to a vulnerable web application (more often than not, the site was written in PHP). The data sent is usually a hyperlink containing the malicious code, and the user clicked on this link from Twitter or Facebook or similar. These days the link could even be a short URL so the user doesn't actually see the real link at all (besides which the embedded code will be encoded in some form). After the web application processes the URL, it renders a page for the user that will contain the malicious code in the same way it would if it were valid content from the website.

One example could be that the code issues a request to redirect the user to the hacker's site. The code could read the user's cookies for the vulnerable site (that's the context it's executing under) and then construct the URL to pass on the cookies. The hacker's site then merely issues a 404 or even replicates the vulnerable site so that the user is then encouraged to login, for example. ■

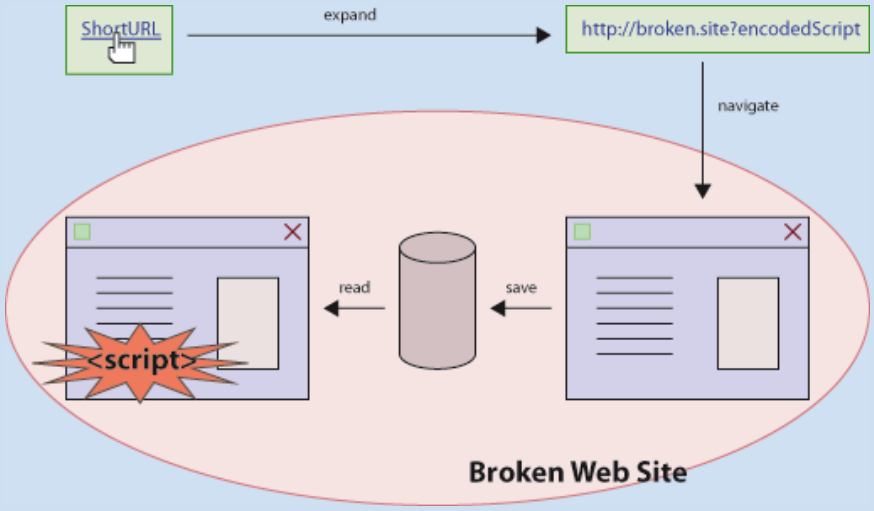


Figure 1: Simple XSS data flow from short URL to compromised web page.

type, and so on. In our example with the HackMe company, we struck lucky.

The input doesn't have to be via a URL, as in our example. Other vectors for attack include typing specially constructed SQL into input fields on the site (take for example, login details: what would happen if I typed my name as 'HAXOR UNION somesqlcode --' for example?), or opening up the JavaScript for the site – all browsers now come with a developer mode – and using it to probe AJAX entry points to the site.

The upshot of this discussion of SQL injection attacks is that a web application like a CMS can't trust any data it gets from a browser. All data has to be sanitised, either by validating the input to be of the correct type, in range, and so on, or by converting special characters in the input into an encoded form (or by removing them entirely). If our mythical developer at HackMe had validated the value of the page number to be (a) numeric, and possibly (b) between one and the total number of pages on the site, we wouldn't have been successful with our SQL injection.

As we've seen with the LulzSec attacks, once you have access to the database on a vulnerable web site, you can pretty much do anything you want. One favourite trick is to alter the main page of the site (since its text is held in the database for the CMS, it's pretty easy to find the right record and modify it) to advertise your successful attack.

What LulzSec was particularly good at exposing was the fact that if a site is vulnerable to a SQL injection attack, it's likely that the developers stored all kinds of information in the database that shouldn't be there. The first big no-no from a security viewpoint is storing passwords in plaintext. This flies against everything we have learned about website security in the past 15 years. Let's see why. First of all, it's a fact of life that people reuse their passwords across several different websites. All you have to do is to read about the after-effects of a data breach: people find their Facebook page defaced because they used the same password for everything they did online and some hacker found it.

I've certainly seen advice along the following lines: create three passwords for yourself – a password for high security sites, like your bank, shopping site, or anywhere you're putting money down for goods or services; a medium security password for 'social' sites like Facebook or Twitter or Gmail; and a password for every other site where you have to register in order to use the services. This kind of advice is fine as far as it goes, but it's not fine-grained enough and still suffers from the problem of hackers finding your password from a site with awful security and then having free rein in pretending to be the online you elsewhere.

Hashed passwords

Second, it's a well-accepted practice to hash passwords with a cryptographic hash algorithm. The best developers not only hash the passwords, but salt them too (that is, add a large, cryptographic random block to the password before hashing). Although the 'plain old hash' can be vulnerable to attacks with rainbow tables (see below), the salted version cannot. That way, should the hacker manage to get into the site's registration database and see the salted hashed passwords (which, of course, look just like regular hashed passwords) there's still no way to reverse engineer the passwords.

How can you tell if a site uses hashed passwords? Since hashing is a one-way algorithm, there's no simple way to recover any password and therefore, if you forget your password, the site will send you a new one (that you are supposed to change immediately) or will send you a temporary link to the site (usually containing some long GUID-like string so you can't guess it) to let you change it. If the site sends you your original password, be very, very cautious, and use a unique password just for that site.

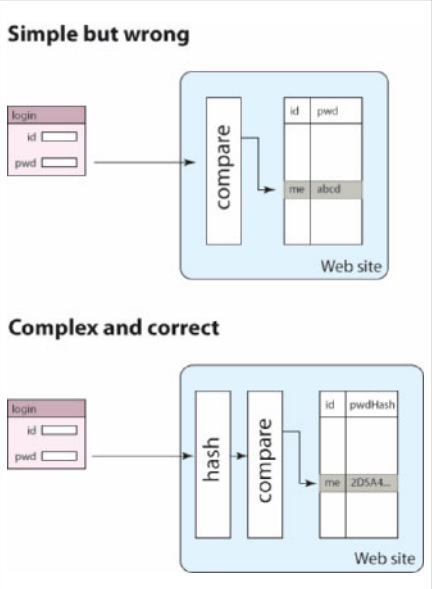
Multiple databases

The second no-no is storing everything about a user in one database. If you think about it, in using many sites there are two modes of operation: Firstly there's your regular use of the site (posting your status on Facebook, searching for something to buy on Amazon, that kind of thing), and then there's the super secret stuff like modifying your profile (including your password) or changing your mode of payment or credit card details, and so on. Sites should try and separate out these concerns into at least two databases, the super secret database behind extra security, the day-to-day database containing the salted hashed passwords (so users can login) on another machine. That way, if the hacker gets through the first set of defences all they get are user ids, salted hashed passwords, general usage information. The really interesting stuff like your credit card details, your birthdate, your mother's maiden name, are elsewhere.

Unfortunately, as we've seen, Sony didn't do all that it could have done and laid itself open to not only being hacked, but having a lot of users' personal information downloaded as well. The main problem for Sony was that all of the personal data for its users was downloaded, so how could it verify its users once it had fixed the holes? At one point it tried asking for the user's date of birth, but of course that was in the hackers' hands as well.

Although hacking websites is probably going to continue apace, I hope that this discussion has brought some light on how sites should be protecting themselves, and how you can help protect yourself. PCP

Julian M Bucknall has worked for companies ranging from TurboPower to Microsoft and is now CTO for Developer Express.
feedback@pcplus.co.uk



▲ **Figure 2: Simple password verification (but broken) vs more complex version (but secure).**

Rainbow tables

Rainbow tables are a technique that uses large, pre-computed tables to help you crack hashed passwords. They use a class of functions called reduce functions that calculate a contender password from the hash. These reduce functions are used alongside the hash functions, applied in a chain: hash followed by reduce followed by hash. You end up with a candidate password and a final hash, but that chain covers all the intermediary passwords that were also hashed. Given enough reduce functions (thousands of them) and enough time, you can create a large table of initial passwords and final hashes.

To crack a password, take its hashed value and check if that hash in your table. If it is, reproduce the chain until you get to the point where you can read off the password. If not, reduce the given hash with that final reduce function, hash the results, and check that new hash to be in the table. Continue this cycle until you find a match of the computed hash and an entry in the table (and therefore the password from regenerating the chain), or run out of reduce functions. ■

Password managers

My advice is to have a separate password for every site you visit and to not even try to remember your passwords for them all. There are more and more 'speciality' shopping sites, for example, and you have no real way of evaluating their security (although I would try the trick of 'forgetting' your password and seeing how they respond – if they send you your password back, be very careful).

I certainly can't remember several hundred passwords, so I use a password manager. There are several good ones out there, including some open source ones, and they all have the same basic functionality: storing passwords, generating secure passwords, and usually automatically completing website login screens.

Like it or not, websites are going to get hacked, so it's important that we take steps to protect ourselves the best we can. ■