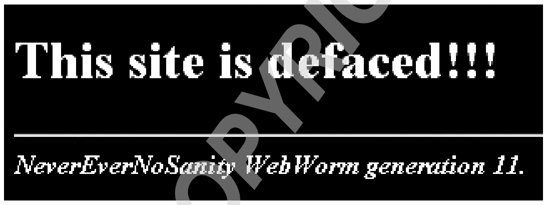
CHAPTER
1

That Horrible Sinking Feeling

Insight into web application security and why you should care about it

I remember it quite clearly. I woke up, stumbled to the coffeemaker to start a brew, went back to my computer to look for updates on the phpBB message board to chat with some friends, and was panicked by what I saw: My home page had been replaced by a message from the ''SantyWorm'' that looked something like Figure 1-1.


This site is defaced!!!

NeverEverNoSanity WebWorm generation 11.

Figure 1-1 Imagine if your website were replaced with this.

My heart began to race, and I worried about what might have happened and how I might fix it. I poked around the administrator pages of the site, but every way that I tried to fix it was met with the ''hax0rs lab'' message mocking me. Then, defeated, I slumped over in my chair, hung my head, and exhaled deeply. All I wanted was a forum to talk with my friends. I'd never considered that I would need to update that software from time to time. I was naïve.

Avoiding That Sinking Feeling

If you've had that experience, you know it's not a good one. The best-case scenario is the one that I was in—I had a recent backup of both the files and the database. I used a web-server-level password to lock out access from everyone but me, deleted everything, restored the backup, upgraded my site to the latest version of phpBB, and then let visitors back into the site. The worst-case scenario—well that's hard to imagine.

What is the worst-case scenario if your site gets attacked and the security is broken? Perhaps the usernames, passwords, and emails get stolen from the site, which could then ultimately allow the attacker to log in to your bank and take your money. Perhaps your site becomes a spam relay or a download source for malware, infecting thousands of computers. Or perhaps your site guards valuable proprietary information about your company, which the attacker can copy without your knowledge. As Kevin Mitnick wrote in his book *The Art of Deception* (Wiley Publishing, 2003), ''When you steal money or goods, somebody will notice it's gone. When you steal information, most of the time no one will notice because the information is still in their possession.''

My goal with this book is to reach out to people who are naïve about how to keep a Drupal site secure. Perhaps you're not as inexperienced as I was—why did I think that I wouldn't need to update the software!—but there is a lot of information you will need to know to keep your Drupal site secure. To some extent you can simply follow the security updates closely, and that's all you need to know. Then you would rely on the other users of Drupal to make sure the software is secure. But . . . should you trust them?

It's Up to You

Sadly, the reality is that you cannot simply rely on other Drupal users to keep the code safe. A surprising number of websites are configured insecurely. A similarly surprising number of contributed or custom modules and themes contain logical or programmatic vulnerabilities. You must pay attention if you are going to keep your site safe.

When you have finished reading this book, you will know what steps you should take to protect a basic Drupal site, how to review a module to find weaknesses and how to fix them, and what extra steps you can take to protect your site if you need additional protection.

What Is Web Application Security?

I don't want to get totally philosophical on you, but I do spend some time with some deep thinkers up in Boulder. There are several aspects that most people include in the concept of *website security*. Generally, a site is secure if it is safe from danger or loss. For this book I'll define site security as follows: A site is secure if private data is kept private, the site cannot be forced offline or into a degraded mode by a remote visitor, the site resources are used only for their intended purposes, and the site content can be edited only by appropriate users.

Keeping your site secure by that definition should be simple, and yet there are dozens of methods to violate a part of the rule of security, and hundreds of examples of vulnerabilities within the Drupal project have been revealed over the last few years. So what can we do?

Security Is a Balance

You may already be feeling overwhelmed. To be perfectly safe requires so much work—how can anyone do it? The fact is that a typical site shouldn't implement every security recommendation in this book. Running a site is always a balance between what is practical, reasonable, and necessary.

Most security best practices have trade-offs from somewhere else. Sure, it would make your site instantly safer to use an SSL certificate for every visitor to every page, but that adds additional load on the server and additional cost to you. Or if you use a self-signed certificate, it adds additional work for your site visitors in order for it to work.

As the site administrator you must understand potential security weaknesses, your users, the priorities for your site, and your budget, and you must balance them all. Hopefully you already know your budget and the priorities for your site. Your users will probably let you know if a new security process annoys them too much. It's my job to explain the weaknesses and solutions so you can decide whether to implement them. On the other hand, many of the recommendations are absolutes. There simply is no reason to leave an SQL injection vulnerability in your site.

Common Ways Drupal Gets Cracked

This section is a review of some of the most common vulnerabilities found in Drupal.

The Drupal API provides protection against most of these common security vulnerabilities, but in order for that protection to work, themers and module developers must actually use that API. Unfortunately it is often the case that new developers to Drupal are unaware of how to properly use the API.

Vulnerabilities within the code of a site are the biggest category of weaknesses. However, as you'll see in Chapter 2, they are only one kind of potential weakness in your site.

This chapter introduces the Vulnerable module. Drupal's functionality can be extended with the use of modules. Modules are a common source of security weaknesses on sites. You can download the Vulnerable module from `http://crackingdrupal.com/content/drupal-vulnerable-module`.

NOTE **This URL is formatted with the full http:// on the front of it because you are expected to actually visit it. Either `example.com` or the short-hand notation for a URL that shows just the information after the Drupal root is used throughout the rest of the book for URLs that are important less for their content than how the data is used in the URL. For example, the URL for the login page in an example can be expressed either as `http://example.com/user` or simply `/user`.**

The purpose of the Vulnerable module is to provide easy-to-understand examples of the different vulnerabilities covered in this book and how to fix them. These examples are fake, but the vulnerabilities they represent are real, and you only have to look at past security announcements to see real-world examples of the flaws. This module is useful as an example for the book and for your own study, but it should never be installed on a real site.

NOTE **The entire set of vulnerabilities attackers use is enormous and growing all the time. Covering all of them would be a waste of your time. Instead, this book covers just the most common and most important vulnerabilities so that you can focus on what really matters.**

Authentication, Authorization, and Sessions

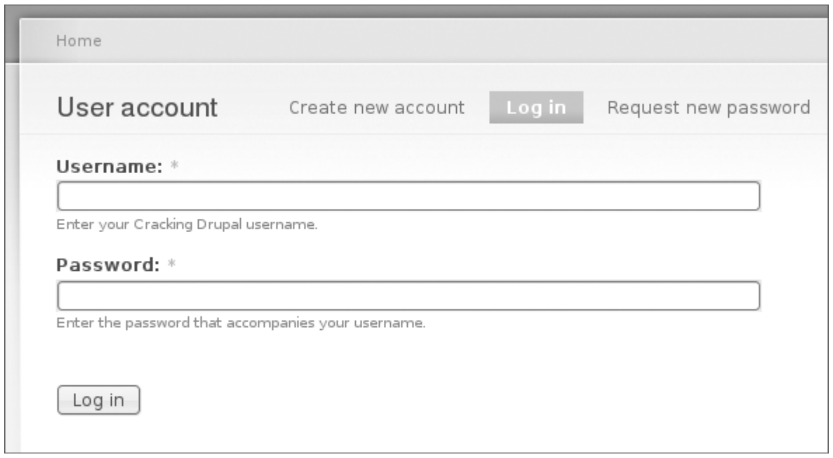
The three interrelated concepts of *authentication*, *authorization*, and *sessions* govern users and permissions. Together, they form a key part of a site's attack surface, because vulnerability here allows the attacker to pretend to be another user on the site or do something that's not allowed. In a system like Drupal, where the administration interface is merged with the regular interface, this area is even more critical. Finding a weakness here may allow an attacker to assume the role of an administrative user or view private content.

NOTE **The *attack surface* of a site is like a map of the ways to crack into the site. Certain parts of the attack surface are more likely to yield valuable results.**

Authentication: Prove Your Identity

When you go to a bank and withdraw money from your account, the bank has security processes to make sure that you are really the person who has the permission to take this action. If you use an ATM, your ATM card and PIN act as proof of your identity. If you go to an agent of the bank, your driver's license or passport may be your proof. Similarly, different websites use various mechanisms to prove your identity.

By default Drupal uses the common username and password combination to authenticate users (see Figure 1-2). Numerous other contributed modules can be used to enable alternate authentication mechanisms.



Username: *

Enter your Cracking Drupal username.

Password: *

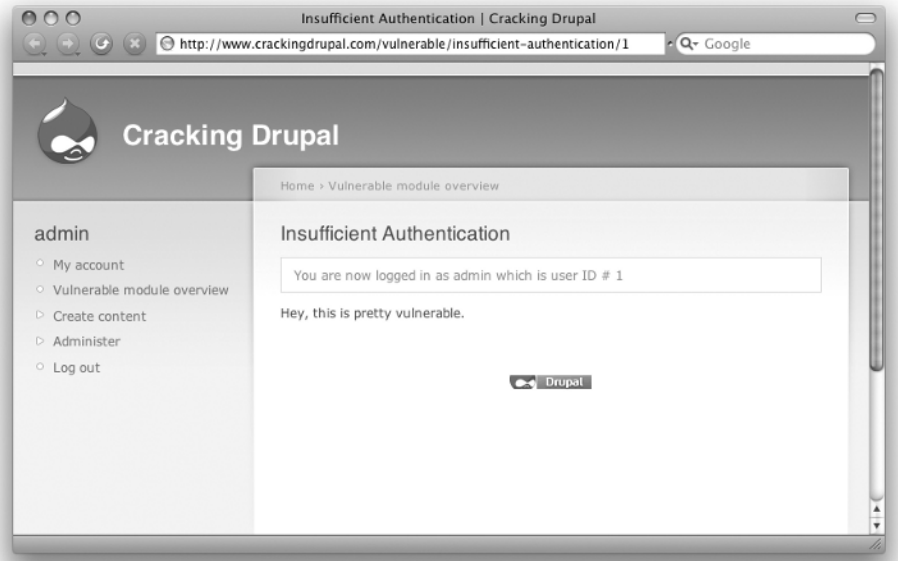
Enter the password that accompanies your username.

Figure 1-2 The login form.

Weaknesses in Authentication

There are several potential weaknesses related to authentication. The two biggest are that users may choose a weak password and that on most sites passwords are sent in plain text over communication methods that can be intercepted—notably, unencrypted HTTP over unencrypted WiFi. Weak passwords are vulnerable to a *dictionary* or *brute force* attack in which a script attempts to log in to a site using common passwords and eventually uses every possible combination of characters until it successfully logs in.

A less-common but still important concept is that of *insufficient authentication* (Figure 1-3). Authentication is insufficient if, for the kinds of transactions to be carried out, the proof of identity of the user is not strong enough to provide sufficient certainty for the site. The sample Vulnerable module has a feature that allows anyone to log in as any user simply by providing the user ID of whatever user she wishes to be. Especially in Drupal where user IDs are sequential integers and where the user ID 1 is all-powerful, this is probably a bad idea outside of an extremely controlled environment (such as a development computer that is never connected to a network). But it could be that the default username/password combination that Drupal uses is insufficient if your site is a financial website or contains valuable secret information. In that case you may want to use a third-party identity verification system based on a stronger authentication mechanism, such as an RSA SecurID token, sometimes referred to as an *RSA key fob*.


Insufficient Authentication | Cracking Drupal
http://www.crackingdrupal.com/vulnerable/insufficient-authentication/1
Google
Cracking Drupal
Home › Vulnerable module overview
admin
My account
Vulnerable module overview
Create content
Administer
Log out
Drupal

Insufficient Authentication

You are now logged in as admin which is user ID # 1

Hey, this is pretty vulnerable.

Figure 1-3 Insufficient authentication from the Vulnerable module lets an attacker become user 1, or 3, or 30, without any proof.

CAUTION In the example Vulnerable module, there is a dubious feature that lets any user impersonate any other user on the site simply by specifying the user ID number in the URL at `vulnerable/insufficient-authentication/1`. Specifying the 1 is especially dangerous because user 1 on a Drupal site is a special user who has been granted all roles. This may be handy on a development site but is obviously dangerous for any other site. Figure 1-3 shows an account right after someone used this feature to become user 1 on this site.

It is up to each site to determine an appropriate level of authentication for its users. Often username and password are enough. However, as the example Vulnerable module shows, it is possible for a contributed module to create a situation that bypasses the normal login process and allows an attacker to gain access of another user.

Authorization: Permissions and Access

One thing that makes Drupal a great system to use is its rich system of roles and permissions. *Permissions* control actions that can be taken. *Roles* are groups of permissions that can be granted to users. A site can have an arbitrary number of roles, a role can have an arbitrary set of permissions, and a user can have an arbitrary number of roles. When a user has two roles, his or her total set of permissions is the union of the permissions for those two roles. Two special roles—anonymous and authenticated—are required on every site and define the permissions granted to any user based on whether the user is logged in or not.

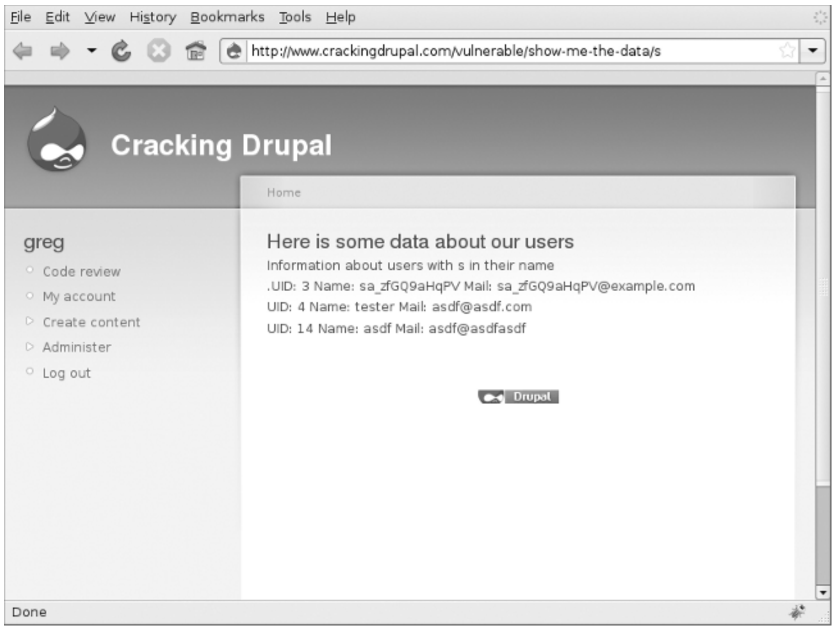
In addition, Drupal has a system of specific object access, which allows third-party modules to define grants related to node and taxonomy objects. This allows a site to have private and public nodes depending on the taxonomy term applied to a node. This access system is covered in more detail in Chapter 7.

Going back to the bank example, once you have established your identity by an authentication means, you then may be limited in the actions you can carry out—that you are authorized to do—based on your permissions or on the level of authentication. For example, your ATM card and PIN are relatively easy to steal, so users who use this authentication mechanism are able to withdraw only a finite amount of money from the bank. On the other hand, if you go to an agent of the bank and present your passport and driver's license and then request to withdraw a much larger sum of money, the agent is likely to let you do so. You may be required to have a specific level of permission on the account to be able to withdraw all the money in the account or to close the account.

Weaknesses in authorization occur when a user is permitted to see data or perform an action that should not be allowed. For example, a module may show information that should be private, such as the email address shown in Figure 1-4, or allow a user to delete or modify content she should not be able to change.

The Vulnerable module contains an example that, even when used properly, bypasses these two types of authorization. It is available to all visitors of the site and shows user email address information for any users of the site based on characters found in their username. The style of the query bypasses several layers of what would normally be proper user authorization checks:

- The list shows all users regardless of whether their accounts are active, though Drupal normally doesn't show profiles for inactive users.
- Email addresses should be shown only to users with the ''administer users'' permission.
- Only users with ''access user profiles'' permissions should be able to see this data.



http://www.crackingdrupal.com/vulnerable/show-me-the-data/s

Here is some data about our users

```
Information about users with s in their name
.UID: 3 Name: sa_zfGQ9aHqPV Mail: sa_zfGQ9aHqPV@example.com
UID: 4 Name: tester Mail: asdf@asdf.com
UID: 14 Name: asdf Mail: asdf@asdfasdf
```

Figure 1-4 Authorization bypass reveals users' email addresses.

This simple example shows how a module developer who wanted to share information could easily create a situation where data is easily available to site attackers. Later you will see how an attacker could combine this page with SQL injection to get virtually any data from a site.

Session Management and Weaknesses

The Internet is based on HTTP protocol, which provides no system itself for keeping track of users. When a user requests a page, the web server sends it back and the interaction is complete. When a user requests the next page, it may be handled by a different web server process or even a completely different physical server.

Imagine if, in the bank example, you first proved your identity to one agent of the bank and then when you wanted to make the withdrawal, a different agent at the bank helped you. To keep track of who you are, the bank might issue you a unique number. When you make a request to do something, you also provide your number. The agent compares that number to a list the bank keeps, and then the bank can be sure of your identity. This is basically how session ID numbers work for web applications.

Web application developers typically store the session ID in a cookie. During every subsequent request to the web server, the user's browser sends this cookie to identify the user.

This process presents several opportunities for weaknesses. Because the session identifier is stored on the client computer, an attacker can send any session ID value with his requests. If he sends the session ID of a different user, he can impersonate that user. If the session IDs are easily predictable (for example, if they are just the user ID of the user or if they were based on the user ID and the time that the user logged onto the site), then an attacker can easily guess the session ID of a user to gain that user's permission. Fortunately, Drupal core handles the majority of session management for Drupal and does a good job of following industry best practices for session management.

However, if a normal site user is accessing a website over an unencrypted connection such as a shared WiFi network, then an attacker could monitor the traffic on the network, determine the session ID of the user, and then use it in his own requests to pretend to be the other user. Possible solutions to this problem include educating your users and using HTTPS for all authenticated sessions.

A more common problem in Drupal is code similar to that shown here:

```
global $user;
$original_user = $user;
$user = user_load(array('uid' => 1));
my_module_code_to_do_stuff();
$user = $original_user;
```

This code allows a module to temporarily become another user, perform some action as that user, and then switch back to the original user. If there is a redirect or fatal error that stops the normal flow of code execution before the user object has been set back to the original user, the user session has been changed to a different user. Because this pattern is normally done to temporarily give the user more permissions than normal, it is an opportunity for privilege escalation.

Command Execution: SQL Injection and Friends

Command execution generally includes operating system commands and SQL injection. However, in general, this is a potential issue for all systems that your site interacts with, such as XMLRPC, REST, and SOAP. The basic problem is that data from the user (the content of your blog post) is mixed with control information (the query to insert that content into the database) and the combined string is executed against the database. This book focuses on SQL injection more than other types of command injection because it is the most common command-injection issue found in Drupal. However, the same concepts apply to interactions with any system.

TIP ***SQL*** **stands for** ***Structured Query Language*** **and is the name of the particular language used to interact with databases. SQL is meant to be the same for all databases, but in practice it varies widely from one database to another.**

There are several common models for safely handling user data:

Rejecting known bad input: Using blacklists to filter input is the process of refusing to accept data that contains items that are in a list of inappropriate characters. This is not particularly useful because it relies on the programmer to write code to handle an exhaustive list of bad inputs. That is a difficult task in the first place and impossible to do once you consider that new technologies with new vulnerabilities are constantly being invented.

Accepting known good input: Using a whitelist to determine safe input is safer than rejecting known bad because a list of safe input should stay safe into the future.

Both rejecting known bad and accepting known good are extremely limited in their usefulness to store anything more than simple text without any special characters. Drupal deals with rich data sets from clients such as HTML, which makes these two strategies unsuitable. These methods are not used in Drupal and therefore are not discussed in the rest of the chapter. Some other options include:

Sanitizing data before it is stored works well in a simple system but fails when the input is later used in a variety of contexts; rules to sanitize the data for use in one context may not protect another context. For example, sanitizing text to prevent XSS when you display in the context of a browser will not protect a site from SQL injection when the data is used in the context of a database query. The extremely flexible nature of Drupal requires that you use data in different contexts, so this architecture does not work for Drupal.

Safe data handling provides protection by using a means of interaction that separates the user data from the control statements. An example of this is using a parameterized query that contains no dynamic SQL. Parameterized queries were designed at a basic level to provide protection for mixing user data and command data. Safe data handling is useful where it is supported, but not all systems support it.

Boundary validation is the process of accepting all user input and then filtering it upon output depending on the nature of the boundary. Drupal relies primarily on the boundary validation pattern (see Figure 1-5).

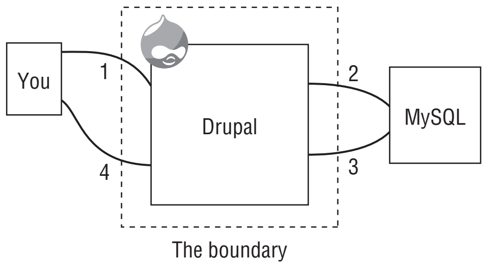


Figure 1-5 Boundary validation.

In this diagram you can see the flow of a typical page-request cycle for creating a new blog entry on a site. The data flows are labeled 1 through 4 and described as follows:

1. The user has posted the form to the web server, which hands the data to Drupal. Drupal first makes semantic checks on the form data to ensure that the user hasn't tampered with the drop-downs, check boxes, and radio buttons in the form to, for example, create a blog post with a taxonomy term that is not allowed.
2. Drupal executes queries against the database to insert the user's blog entry for storage. At this phase Drupal is sending data beyond its boundary, so it must filter it to make sure that any characters inside the user data that may alter the impact of the SQL statements are ''escaped.'' The escaping is done in a context-sensitive manner. Since this is a database, the filtering is appropriate to SQL.

TIP **When interacting with other systems, certain characters have special meanings. In SQL, the single quote is used to separate string data from the rest of the statement. If a user has the last name O'Henry, then the single quote in the name could be misinterpreted. To handle these situations, SQL provides the slash escape character to allow the insertion of the single quote character into the database.**

3. This is where Drupal retrieves data from the database. In general there are no concerns here, except that the system must remember which fields in the database are generated by the system (for example, sequential ID columns) and which are user-provided values that must be filtered.
4. The retrieved data is shown to the user. Because the data from step 3 includes some data from users, the data is filtered prior to being sent to the user's browser. Much like step 2, this filtering should be done in a context-sensitive manner that will work specifically for HTML data being sent via HTTP and rendered in the context of a browser.

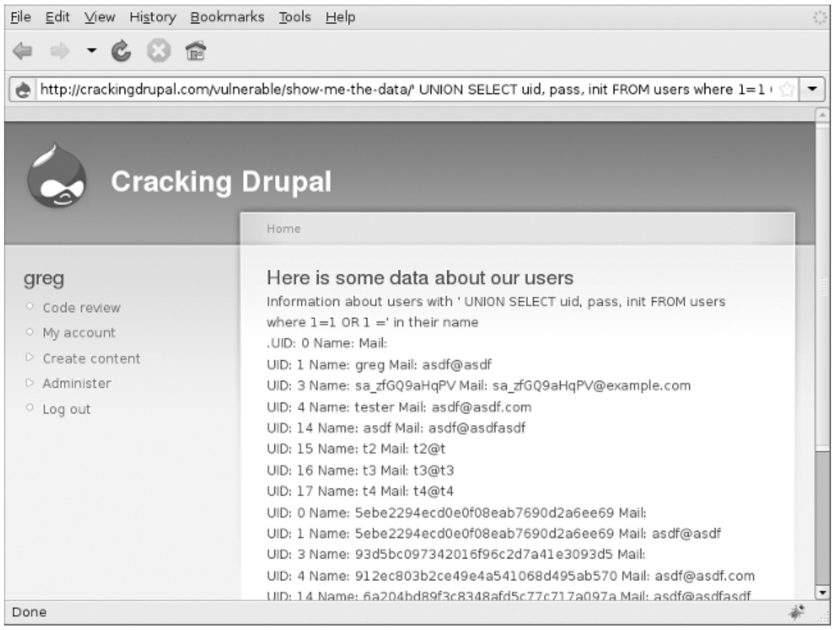
These strategies for validating user data are used for different reasons in different areas. For example, Drupal rejects known bad data such as special characters in usernames because they are inappropriate for usernames. However, even after rejecting inappropriate characters, the query to insert that username into the database and the functions—which prepare the username for display to a browser—still perform boundary validation to filter the username in a way that is useful in that context.

SQL Injection

The Vulnerable module provides several examples of SQL injection. A simple example is available at the URL `vulnerable/show-me-the-data/' UNION SELECT uid, pass, init FROM users where 1=1 OR 1 ='`

Using the SQL `UNION` keyword, you can append data from a totally separate query into this page. In this example, you get the user ID, the MD5 (Message-Digest algorithm 5) hashed version of the password, and the email that was used when the account was created (stored in the `init field`). You can see the result of this modification in Figure 1-6, where in addition to the normal results you also see sensitive data like the hashed version of the password and email address. With the hashed password and email addresses of a user, an attacker can prey on the fact that most users use a limited number of passwords and try to use that password and email combination on commonly used websites.

TIP **Instead of just storing your password, Drupal stores a unique string that is derived from your password using a function. This is a one-way function, which means that you can take a password, send it through the function, and get the calculated hash value, but you cannot take a hash, reverse it through the function, and get the password. That said, the MD5 function used by many systems, including Drupal, is becoming increasingly unsafe given modern computer-processing capabilities. Therefore, you should still protect the MD5 hash of the password as if it were the password itself. In Drupal 7, the MD5 hash has been replaced with a more secure hash.**



```
http://crackingdrupal.com/vulnerable/show-me-the-data/' UNION SELECT uid, pass, init FROM users where 1=1
```

Here is some data about our users

```
Information about users with ' UNION SELECT uid, pass, init FROM users
where 1=1 OR 1 =' in their name
.UID: 0 Name: Mail:
UID: 1 Name: greg Mail: asdf@asdf
UID: 3 Name: sa_zfGQ9aHqPV Mail: sa_zfGQ9aHqPV@example.com
UID: 4 Name: tester Mail: asdf@asdf.com
UID: 14 Name: asdf Mail: asdf@asdfasdf
UID: 15 Name: t2 Mail: t2@t
UID: 16 Name: t3 Mail: t3@t3
UID: 17 Name: t4 Mail: t4@t4
UID: 0 Name: 5ebe2294ecd0e0f08eab7690d2a6ee69 Mail:
UID: 1 Name: 5ebe2294ecd0e0f08eab7690d2a6ee69 Mail: asdf@asdf
UID: 3 Name: 93d5bc097342016f96c2d7a41e3093d5 Mail:
UID: 4 Name: 912ec803b2ce49e4a541068d495ab570 Mail: asdf@asdf.com
UID: 14 Name: 6a204bd89f3c8348afd5c77c717a097a Mail: asdf@asdfasdf
```

Figure 1-6 SQL injection is being used to show any data an attacker might want.

In this example, the `UNION` query could be used to get information about what other databases are on this server, the tables they contain, and the data in those tables. If you have an e-commerce site, donations database, or any private information such as email addresses or secret plans for world domination, an attacker would be able to use a hole like this to see that information.

Arbitrary File Upload

Another related problem is arbitrary file upload, which often leads to code execution. Drupal has many features and modules that allow users to upload a file. Within core alone, there are the Upload module, user avatars, the logo, and the favicon upload tool. Among contributed modules, there are dozens of ways to upload files: image, imagefield, filefield, embedded media field, video, and audio. Vulnerabilities in the code or configuration of any of these features could allow an attacker to upload an arbitrary file that contains PHP code, JavaScript, or another kind of code that can compromise the security of your site.

Cross-Site Scripting

The basic purpose of Drupal is to take data from users, store it, and display it back to other users. This can cause a problem when an attacker finds a way to add code of some sort into the site so that it executes when other users look at it. JavaScript is the most common vehicle for these attacks, but any language that is executable by the browser can be used. This code has the ability to take actions impersonating the user, and if the code runs on your Drupal site, it has access to your full session and can do anything that you as a user are able to do, like delete content or change your password.

Cross-site scripting (XSS) attacks can be reflected, stored, or DOM based:

- Reflected XSS is any situation where user-supplied data from a page request is immediately displayed back to the user.
- Stored XSS is common in systems like Drupal, which store user-supplied data into a database.
- XSS attacks on the DOM directly alter the code—again, typically JavaScript—rather than trying to inject code into the page itself.

The Vulnerable module has several examples of reflected and stored XSS based on injecting JavaScript into the page. On the ''vulnerable/show-me-the-data'' page it is possible to use the tag `<IMG SRC=javascript:alert('XSS')>` as the last part of the URL and have the Opera browser execute the JavaScript. Figure 1-7 shows the results of this attack.

Generating a JavaScript message window in a page is an easy way to determine if the page is vulnerable—if you see the message, the page is vulnerable. There are many more ways to execute more complex XSS, though they often depend on different parsing rules or vulnerabilities of the browser.

Cross-site scripting is another area where the concept of context-appropriate boundary validation is used. Drupal provides a system of HTML filters to remove malicious code from HTML before it is sent to the browser. Of course, it's up to the coder to actually use those HTML filters.

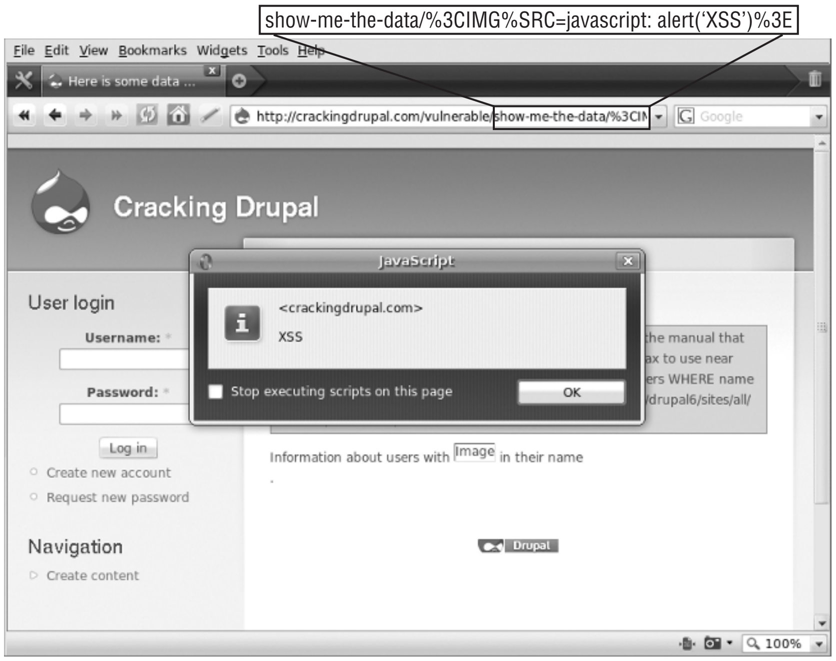

show-me-the-data/%3CIMG%SRC=javascript: alert('XSS')%3E
File Edit View Bookmarks Widgets Tools Help
Here is some data ...
http://crackingdrupal.com/vulnerable/show-me-the-data/%3CIM
Google
Cracking Drupal
JavaScript
<crackingdrupal.com>
XSS
Stop executing scripts on this page
OK
User login
Username: *
Password: *
Log in
Create new account
Request new password
Navigation
Create content
Information about users with Image in their name
Drupal
100%

Figure 1-7 A browser alert showing us that this page is vulnerable to reflected XSS.

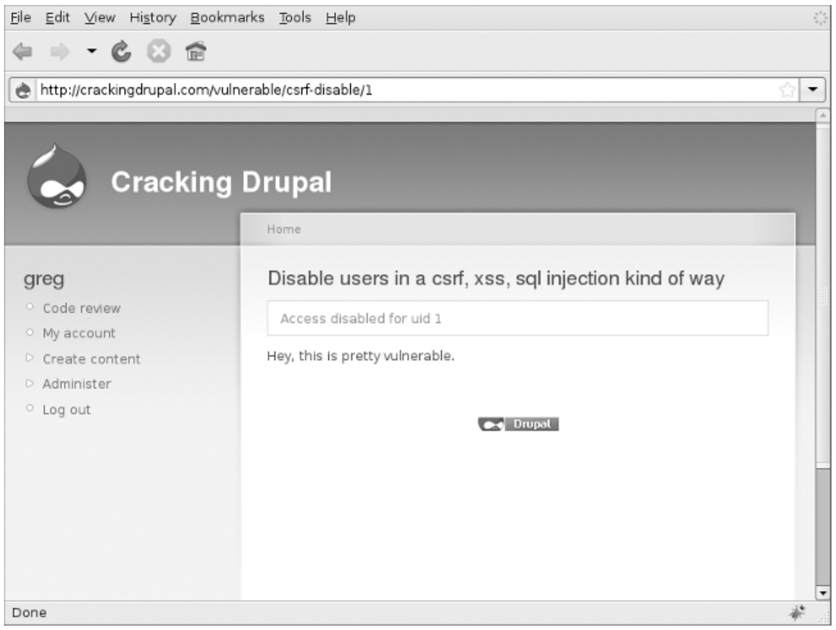
Cross-Site Request Forgery

The nature of a cross-site request forgery (CSRF) is that an attacker can make ''you'' do something without your knowledge. This is similar to stealing your session but limited to specific actions on a site. There are two basic types of CSRF: those based on `GET` requests and those based on `POST` requests.

TIP The HTTP specification defines several types of server requests, among them `GET` and `POST` requests. A `GET` request is probably the most common; it happens every time you click a link or type an address into your browser. A `POST` is generally what happens when you submit a form to a site.

Drupal core provides protection against a `POST` CSRF using a token system. When a form is built using Drupal's Form API (FAPI), a token is added to the form based on the session ID and a private key from the site. When the form is submitted, the Form API confirms the presence and validity of the token. This requires that a `POST` to the site be based on a current session and makes it more difficult for an attacker to develop a generic attack on forms in Drupal.

The more common problem in Drupal comes from modules that take action based on a GET request. The Vulnerable module provides a feature that disables user accounts based on the URL. This feature is demonstrated in Figure 1-8.


File Edit View History Bookmarks Tools Help
http://crackingdrupal.com/vulnerable/csrf-disable/1
Cracking Drupal
Home
greg
Code review
My account
Create content
Administer
Log out
Drupal
Done

Disable users in a csrf, xss, sql injection kind of way

Access disabled for uid 1

Hey, this is pretty vulnerable.

Figure 1-8 Requesting this URL disables any user of the site.

This simple code can be exploited in a variety of ways, such as tricking a user who has the permission to access the page into clicking on a URL like `http://example.com/vulnerable/csrf-disable/1` or, even easier, getting the user to look at a page with an ''image'' embedded into it with the source pointed at that URL: `<img src=''http://example.com/vulnerable/csrf-disable/1''>`.

CSRF is increasingly not a problem for Drupal because the few remaining modules that take actions like this are fixed to use a form of some sort. However, it is often tempting when building a rich AJAX feature to slip back into creating a CSRF vulnerability via GET requests. The security team is working on an API to make this much easier for module developers, but that API is not yet available. There are still methods that can be used to provide security for links. The system is based on the same token system used to protect Drupal forms. However, because this practice of taking action in response to GET requests is not as common or standard as the form system, there is no way to provide this protection automatically or easily.

The Big Scary World

Are you feeling overwhelmed yet? There are many ways for your site to become insecure, and this chapter focused on the vulnerabilities in code. In the next chapter you'll learn about some of the problems outside Drupal, and the list of potential problems gets even larger.

At this point, you should have a good understanding of some of the issues involved in writing secure code. You should understand authentication, authorization, sessions, and the relationships among them. Often the results of a weakness in this area are the same—an attacker pretending to be someone else or seeing something he shouldn't—but the nature of vulnerabilities is different. You should understand code execution, the most common type of code execution in Drupal—SQL injection—and the role that boundary validation plays in protecting against code execution. You should understand cross-site scripting, where boundary validation is also important. Finally, you should know how to recognize a cross-site request forgery, where an attacker can trick you into modifying your own site without you even knowing it.

The Most Common Vulnerabilities

Looking back at all security announcements that have been posted on drupal.org since 2005, you can see which are the most common types of vulnerabilities; the vulnerabilities by type for Drupal core that have been contributed since they were reported publicly are shown in Table 1-1. Cross-site scripting is the single most common issue. The ratio of problems is relatively consistent between core and contributed modules.

This table shows us that over time the most common problem has been cross-site scripting, which is also a very dangerous problem. Recent changes to Drupal core will help to reduce this problem somewhat, but it is still one of the biggest areas that need attention.

Comparing core versus contributed modules, it's clear that contributed modules are a source of a lot more occurrences—more than two times as many—although when you look at vulnerabilities per line of code, core has had more announced vulnerabilities than contributed modules. Of course, this analysis covers only the issues that were reported to the Drupal security team. There are many more issues that haven't been found yet or that a maintainer silently fixed.

Table 1-1 Announced vulnerabilities by type for Drupal core and contributed code

VULNERABILITY	OCCURRENCES	OCCURRENCES AS A PERCENT OF THE TOTAL
XSS	55	44
Access bypass	17	14
CSRF	12	10
SQL injection	12	10
Code execution	10	8
Clarifications and announcements	4	3
Session fixation	3	2
Privilege escalation	2	4
Arbitrary file upload	2	4
Mail header injection	2	4
CAPTCHA bypass	2	4
HTTP response splitting	2	4
File overwrite	1	2
Logging sensitive data	1	2
Session impersonation	1	2

Summary

In this chapter, you learned about many kinds of vulnerabilities, but within Drupal and this book it's clear that the most important areas to focus on are XSS, access bypass, CSRF, and SQL injection. These four types of vulnerabilities are the focus of this book.