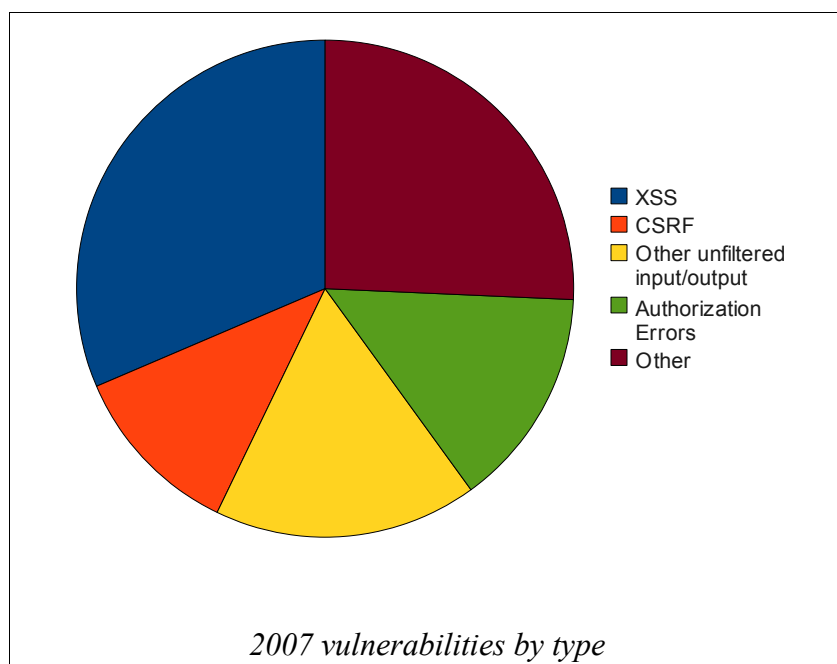


Recommended Core Security Improvements

By Jesse Crawford for the Drupal Security Team
12/15/2007

In 2007 there were a total of 33 security vulnerabilities identified handled the Drupal Security Team, 4 in the Drupal core and 29 in contributed modules. Of these, 19 were a result of input that was not sanitized properly, resulting in vulnerability to cross site scripting and, in some cases, arbitrary code execution and SQL insertion. Other errors included unprotected utilities and sample files, programs that did not adequately protect secure items, and design flaws resulting in back door authorizations. The Security Team worked with developers to resolve these security vulnerabilities, providing updated versions and patches.



Input Sanitation

Over half of these vulnerabilities were due to improperly sanitized and validated input. One of the main culprits here was not using the Forms API. My first recommendation is that contributed project developers be strongly encouraged to use the API, and that core modules should use the API whenever possible. Because the API includes security validation, use of it should greatly reduce the occurrence of vulnerabilities to cross-site scripting and SQL injection.

Another important point is education – although it is already being done, a further effort should be made to show developers the risks associated with improperly sanitized input and how to easily prevent it by using the forms API. Developers also seem unsure of when and what to use when it comes to input sanitation – Although the documentation at <http://drupal.org/writing-secure-code> seems extensive, Developers continue to make input handling handling errors. It may be helpful to include more examples for XSS protection, and further information on the kinds of risks XSS includes.

Just as important as sanitation of input is sanitation of output. It is important that developers use provided functions to ensure that data output to the user is free of unauthorized scripts and potentially harmful code. Developers should be sure to use the appropriate filter function (often just check_plain)

to verify that output is clean.

Security-Mindedness

The brunt of the other vulnerabilities were simply cases of people not developing in a security-minded fashion. A prominent example of this is SA-2007-009, a Highly Critical vulnerability caused simply by a module that included unprotected example scripts capable of modifying the file system. This risk is easy to mitigate – the easiest method is simply to remove the example scripts and provide them as a separate download for those that would like to see them. Alternatively, they could have included prominent warnings to the user that the scripts should be protected; or the scripts could be modified to use Drupal authentication. However, these scripts slipped through.

The best solution to this is education. Developers should be reminded (perhaps repeatedly) that security is a primary component of a successful project, not an afterthought. Developing with security in mind will reduce almost all vulnerabilities, and make it easier to resolve those that slip through. To this end, it may be helpful to release further documentation for developers covering security principals.

Although this has not been as much of a problem for the core, it still bears importance to core developers. SA-2007-025 is a vulnerability very similar to SA-2007-09 in the Drupal core. It was found that the Drupal installation scripts, which remained after installation, would run when Drupal could not connect to its database, allowing any anonymous user to configure Drupal to use a different database. This risk is also easy to patch: simply instruct the user to remove the installer once installation is complete, or have it automatically deleted once installation is verified. However, this error passed unnoticed and became a Highly Critical security vulnerability.

It is important that developers also focus on catching the most obvious vulnerabilities – sometimes developers get too caught up in guarding against small, complex vulnerabilities, and they don't catch simpler and sometimes very obvious risks.

Authorization API

A good number of vulnerabilities were caused by modules that did not always check the users authorization to access a resource. It seems that there may be a need for an easier to use API to centralize security vulnerability, specifically, a new API for user authorization management. Such an API should include the ability for modules to expand the API; for example, the ability for modules to specify a script returning true or false that the API should always execute as part of determining if a user is authorized for a resource. That way modules can have their own authorization systems that other modules can easily honor. Implementation of such a system would greatly reduce the number of authorization bypass vulnerabilities.

In conclusion: Remind developers that security is a primary component of good software, develop APIs that make it easy to develop secure applications, and continue to make security a key part of core and third party development.