Security in Web Applications and the Implementation of a Ticket Handling System

Tomas Forsman

January 27, 2014
Master's Thesis in Computing Science, 30 credits
Supervisor at CS-UmU: Jan-Erik Moström
Examiner: Fredrik Georgsson

Umeå University
Department of Computing Science
SE-901 87 UMEÅ
SWEDEN

Abstract

Today the Internet is filled with various web applications. One category of things that can cause a lot of problems are security holes. Some of them are due to programming mistakes, some due to inexperience, or in other ways failure to protect the system against harmful input.

Part one of this thesis will look into some common problem areas in web application security and how to make those areas less problematic. There will be a summary of those problem areas and also some more detailed explanations. These areas include SQL injections and Cross-Site Scripting which, by prominent security companies, are deemed to be the most problematic areas on the web right now regarding security.

Part two is the implementation of a ticket handling system for computer support at Department of Computing Science, Umeå University. Such a system is responsible for receiving requests from employees and students, and managing them in a way that is easy to overview and handle. Having helpful supporting systems will, in turn, make it easier to provide good support to the employees and students. Knowledge from part one of this thesis is used to make the implementation in part two a secure application.

Contents

List of Figures

Chapter 1

Introduction

The number of web applications on the Internet is growing rapidly, but unfortunately many of them have security holes. These holes can range all the way from minor issues to disasters for the host system and its owners. Systems are attacked for various reasons, including sabotage, information theft, or for monetary gains such as installing spam bots which then are sold to spammers. Regardless of why the attacks are done, it is important to make sure that the attacks will not succeed.

In the first part of this thesis, I will look into various security problems in web applications and how to avoid them. The second part is the implementation of a ticket handling system for use at the Department of Computing Science, Umeå University.

Chapter 2

Problem Description - Web Security

2.1 Problem Statement

The word “security” comes from the Latin word “Se-Cura” that means “without care” or “carefree”, that you do not have to care about something. One way of thinking about it is that you do not have to care about constant threats in one way or the other, because various security mechanisms should protect you, your system, and your data from unauthorized access, modification, or harm.

That is, a secure computer system should resist intrusion attempts and protect itself from being accessed by unauthorized people. It should take care to handle data in a way that will not lead to unwanted results, for example stop working or modifying the wrong data. It is quite common for programmers to assume that all input data will be correct, which in reality is not the case.

On the Internet, there are lots of web applications which handle data and requests from both users that has been authenticated (logged in) and those that has not. A user that has not been authenticated should not be able to submit a request or any data to the application that gives them elevated rights beyond what was allowed by the systems owner. The application should not crash either, rendering it unavailable for the legitimate users.

2.2 Consequences

What could happen if security is ignored in a software project? If a security hole is found, there will be different consequences for the different users/organizations involved. The customers using the actual software will probably be the ones affected the most in the short term, with a need to clean up and try to restore their services. Quite probably, those customers have customers of their own being affected by these problems. In the long run, an organization that provides, or uses, buggy and insecure software will probably lose customers.

Regardless of what kind of damage is caused by the security issue, it is something that you would prefer to avoid. It will cost both time and money to analyze, repair, and fix — time and money better spent on something else.

Consequences of a Denial of Service

A Denial of Service is an attack method to prevent regular service from working, for example by sending more requests than the system can handle. This kind of attack is hard to protect yourself from, and the immediate consequence is that your legitimate users will be unable to use the service. If the service is critical for the organization or even the society, the cost of a disruption like this can be high.

Consequences of Data Loss

If an attack of some kind results in data loss, either by triggering a bug that will drop data or through malicious direct intent to delete data (see SQL injection later on), work to restore lost data, if possible, will take time and money. Unless proper and timely backups are being made, the data loss can be permanent.

Consequences of Data Theft

Some attacks are performed with the intent to steal information, which could include lists of credit card numbers or other security sensitive information of the affected site. In 2011, the Sony PlayStation Network suffered an intrusion where first reports claimed 2.2 million credit card numbers might have been stolen. This would have, in total, caused a massive amount of problems and huge costs for everyone affected, but it seems like the theft may not have been completely successful — no confirmed evidence of credit card fraud due to this has been presented [1]. Sony has been fined £250,000 due to their inability to protect customer data.

Consequences of Data Injections

Another big problem area is when someone exploits a security hole to add hidden material to a web service. This hidden material could be HTML content trying to spread malware, adding victim machines to spam botnets. Lists with infected machines are then sold to spammers for profit.

If a web site has been modified by intruders to deliver malware to visitors, many people will probably treat the web site owners as hostile, giving them a bad reputation. Good reputation takes a long time to build, but can be ruined very quickly. There is a high chance that the web site will end up on blacklists that warns for malicious sites, even if the actual intent of the site is not malicious.

Consequences of Stolen Accounts

Depending on how the web site is run and how the break-in is performed, the account it is running under can be hijacked by the intruders. New programs can be uploaded to the site, possibly using it as a stepping stone for new attacks against others. Spam is a big business nowadays, so chances are high that a spam bot will be installed or the account will be used in other ways to send large amounts of junk mail.

Chapter 3

Problem Description - Ticket Handling System

3.1 Problem Statement

At the Department of Computing Science at Umeå University (hereafter called CS), there is a group of people working as computer support. Managing the flow of work requests to be done can be tricky and some kind of system to keep track of these requests is necessary. Currently there is a system in use, but it does not easily give a clear overview of current activities. With input from the computer support group, a new system will be installed or constructed to aid their work and make work request management easier for the users at CS.

The current system relays incoming requests to the members of the support group. By modifying the e-mails slightly, the communication regarding a certain ticket is ensured to be grouped by a unique ticket ID. All communication is relayed to both the customer and the members of the support group. Each member of the support group can take ownership of a ticket by simply replying to the email, or it can be assigned to someone else if needed. A ticket can be marked as solved, which removes it from the list of open tickets to be processed.

3.2 Goal

The goal is to have a new ticket handling system which should provide more functionality than the old system without any obvious security flaws. It should aid the support personnel in their daily work and also give submitters a way of keeping track of their own submissions. If an existing system meeting the requirements can not be found, a new system should be created while taking advantage of knowledge gained in part one regarding web application security.

3.3 Basic Requirements

Requirements deemed necessary at the time included:

- Preferrably be free of charge.
- Open Source, if local modifications are needed.
- Most day-to-day work should be doable over e-mail.
- Web interface for additional overview and more control over details of tickets if necessary.
- The system should retain all information regarding tickets, even after several updates, being closed etc.
- Methods available to search for data in old tickets using various criteria.

This list is not complete, and unfortunately such a list was not formally produced and written at the time.

3.4 Related Work

There are many issue tracking and ticket handling systems out there, but they are not tailored to the work flow of our organization. Some of them are so complex that they are hard to use because they support too much, whereas others are too simple to be a good help for us.

To be fair, much has changed in other systems since this thesis started about eight years ago. Some of the more prominent candidates for similar work are:

- Must, the software we currently use, written primarily by Magnus Jonsson as a replacement for an old piece of software called Rust [2].
- RT, Request Tracker [3] - a system for bug tracking, help desk ticketing, and more.
- BugZilla [4] - software primarily designed for helping in software development, managing issues/bugs etc.
- Jira [5] - another software primarily for project and issue tracking in software development.
- OTRS [6] - Open-source Ticket Request System.

These were not all available when this thesis started, and those that did exist were a lot less mature than they are now. An in-depth analysis was not done and saved, as the report would be written "soon" and it could be done at that time instead. Unfortunately, that is not how it all turned out — a decision was made to create a whole new replacement, so design and coding started.

Chapter 4

Security in Web Applications

4.1 Introduction

There are various definitions of the term “security problem”. For instance, it can be that someone will gain access to information they should not have access to, or gain the possibility of modifying information they only should be able to read. Attacks that will cause a system to be unavailable to its legitimate users, also called Denial of Service (DoS) or Distributed DoS, can be considered a security issue.

This thesis will mostly look into the area of information flow. DoS attacks are often harder to protect yourself from in other ways than buying “enough hardware” to handle all situations or getting help from the network provider to block the attacks.

The *Common Weakness Enumeration* (CWE) [7] each year compiles a list of the “Top 25 Most Dangerous Programming Errors” [8]. Input has been taken from security experts, *“The SysAdmin, Audit, Network, Security Institute”* (SANS) [9], *“The MITRE Corporation”* [10], and others to form this list. Many of these entries apply to web applications, and some of them will be explained below among others.

Another community/organization of interest is *Open Web Application Security Project* (OWASP) [11] which is trying to make application security visible. They publish a “Top 10 Web Application Security Risks” list [12] every few years.

4.2 Why Do We Get These Problems?

There are many varying reasons as to why web applications can be so problematic. Below is a list of some of the reasons.

Web servers are good targets

Web servers and their web applications has become a de facto standard method for providing an application or service to both customers and non-customers. The separation of authorized vs anonymous access is often only done by some part of the request, leading to a mix of authorized and anonymous requests to the same service. In many other types of systems, the distinction between authorized and anonymous users are more strictly separated, making it easier to enforce the security. A system that is out of your reach, and you are unable to talk to, is hard to break into. In the mixed web environment, communication channels are open for both customers and attacks.

Hand crafting assembly code in an attempt to crack a system is for most people a lot harder than modifying a web browser request. On the other hand, attacking a web application probably requires an active communication line to the victim site — leaving an audit trail to follow. A commonly used method of hiding that trail is to setup an anonymizing proxy on another system, using The Onion Router (TOR), or similar.

A web application is a blend of too many technologies

A normal web application usually contains HTTP, HTML, CSS, and JavaScript on the client side, while SQL and various other programming languages is what make things happen on the server side. Making sure that the data being handled is non-destructive for all layers it will pass through can be tricky. Just because some piece of data can be “just some text” for one layer (for example SQL), it could be fatal code in another (when being put to into the HTML mix to the client), resulting in a sad victim.

Low level of entry to start creating web applications

With the development tools we have today and the Open Source movement, it is quite easy to get started in writing web applications. Install a pre-built web server&scripting stack and you are ready to cut’n’paste code from the plethora of examples out there in minutes. Security holes are probably not the first thing people think about when trying to learn new technologies.

Rush to innovate

As the world works today, if you have an idea that might give you a good business - you better act on it fast. If you do not, someone else will probably beat you to it. This means that people will not put that extra time into making things really secure, but settle for “secure enough” or even “Why would anyone try to harm us?”.

Lack of education

The education of today does not focus on writing things in a secure manner, but the focus is on the basic algorithms needed to solve the “actual problems” instead while mostly assuming that all inputs are following the expected schema.

4.3 Explanations

To make further reading easier, this section will explain some of the acronyms and terms used later on.

DoS Denial of Service, some form of attack causing an overload of the target system which then will be too busy to service legitimate requests. An example of this is sending thousands of requests per second to a system which is only capable of handling fifty per second. The system will then be too busy with the illegitimate requests to take care of its legitimate customers. Another way of getting to the same result is by sending requests that for one reason or another can use abnormal amounts of resources (CPU, memory, bandwidth). If a small request can result in large amounts of resulting bandwidth or large amounts of processing needed, that can be exploited.

DDoS Distributed Denial of Service, when using multiple systems to perform a DoS on a target system. Much harder to stop, since with a DoS you can for instance just block the sending computer - either at the target system or at some point in the network path between the attacker and the target. If there are 10,000 or 100,000 attackers, this can be problematic.

XSS Cross-Site Scripting, allowing a malicious user to send data to a web site which other users download and use or run. Examples include embedding JavaScript that creates ad pop ups or tells a user’s web browser to perform actions it would normally not have done. This can also be used to cover up evidence of the fact that the system is already compromised.

SQL Structured Query Language, an ISO/ANSI standardized language used in relational database engines to manage database content.

SQL Injection Entering database request code in a place where the system expects the name of the user, e-mail address or similar. By crafting it in certain ways, the system can be persuaded to bypass access control, to modify, or destroy data on the attacked system.

Exploit A program or defined method to attack a vulnerable system or program.

CGI Common Gateway Interface, a standard for running programs as web applications. Defines how parameters are passed, how input/output is handled etc.

PHP Programming language commonly used in web systems. Easy to learn, which (unfortunately?) helps inexperienced programmers to create web applications.

CAPTCHA Completely Automated Public Turing test to tell Computers and Humans Apart. A challenge-response test trying to make sure that there is a human, not a computer/script that is trying to do things. Common methods are; try to answer natural language questions, read some distorted text or say which picture contains cats vs dogs. In essense, give a problem that is hard for computer programs to solve but supposedly trivial for humans (many of the distorted text CAPTCHAs are quite hard to read).

4.4 Problem Areas

4.4.1 Programming Languages

There are a few programming languages that are more problematic than others regarding security in web programming, either by lack of protection from potential problems, or by design or implementation flaws.

C

The programming language C can be used to create high performance applications. Unfortunately, this "performance advantage" usually comes with a cost in the form of what safety net (or rather lack thereof) you get from the compiler/interpreter. Lately, some compilers (GCC and Clang) have started providing some protection [13, 14, 15, 16], but mostly you get very little help. For instance, the string handling in C is not very friendly or protective against errors. *Computer Emergency Response Team* (CERT)[1] has a website [17] and released a book on the topic of Secure Coding in C/C++ [18].

A common problem is to allocate a string of static size and then receive data of unknown size from the user without bounds checking.

Example:

```
char username[1024];
int admin = 0;
strcpy(username, getenv("HTTP_USER"));
```

What happens if the user name that comes from the environment is longer than 1024 chars? In many cases, it will start overwriting variables that comes next — like the `admin` flag in this example.

[1]An organization that works against attacks on networked systems, operated by Carnegie Mellon University.

Format string exploits are more common in non-web applications, but exists there too. The functions `printf()` and `syslog`, for example, accept format strings that explain how output from the rest of the parameters should be formatted.

Example: First, a correct example which will output the variable `username` as a string and the variable `uid` as a decimal integer:

```
printf("Username: %s (uid %d)", username, uid);
```

If the idea is just to print out the contents of a string without any additional formatting, people can be tempted to use the following:

```
printf(getenv("HTTP_USER"));
```

This will work fine until someone enters a correct format string into the `HTTP_USER` variable, for instance `"%s"`. The function `printf` will then try to fetch the next parameter off the stack, which will cause security problems such as overwriting memory or just crash. The same problems applies to `syslog()`, a function to write something to Unix operating system log files. The use of this function is less common in CGI/web programs, but they most probably do exist.

PHP

PHP [19] is often blamed for poor security, both in the programming runtime itself and the design of the language surrounding frameworks. It is very easy to get started writing web applications with PHP, which attracts inexperienced programmers that "just want to try something" without thinking much about security. A recent article called "PHP: a fractal of bad design" [20] is worth a read.

Searching for "PHP" at *Common Vulnerabilities and Exposures* (CVE) [21] currently gives a list of about 5000 different security holes (2013-12-02). Doing a Google search for "PHP bad security" currently gives about 130 million hits (2013-12-02). While not all of them are immediately about PHP itself being bad, it does give a hint.

On top of this, there have been a few language settings that have made it easy to accomplish insecure code.

allow_url_fopen A setting that when enabled (default) will cause file I/O functions to accept URLs in addition to local files. For example with the HTML form variable `user_image` containing `http://stric.se/stric.jpg`:

```
$img = $_REQUEST['user_image'];
$handle = fopen($img, "r");
```

This means getting the variable *user_image* from the client and opening the file it points at. Using `fopen` like this may not the best idea, because it allows the client to ask the server to contact remote servers, but should not cause immediate harm to the server itself. It allows for easy methods to retrieve external data, such as remote images. Unfortunately, PHP used the same method when retrieving code to be run (i.e. when using the calls `include`, `include_once`, `require` and `require_once`).

The following code has been seen in a few places:

```
include("header.php");
include($_REQUEST['page']);
include("footer.php");
```

This part of code can then called with something like `index.php?page=news.php` to show a common header, the content `news.php` and a common footer, instead of using copy&paste to put the header and footer into every page. Unfortunately, specifying `index.php?page=http://evil.website/evildoing.php` will cause the page to download malicious code and run it. This is probably not what the original author intended.

Recent PHP versions (v5.2.0) has added `allow_url_include` which affects `include()` etc and is disabled by default, so the default settings are now more secure.

register_globals A setting that is disabled by default since v4.2.0 (some old applications might still depend on it being enabled), but caused all HTML form variables to be imported into the normal PHP variable namespace.

Example:

```
if (... some check ...) {
    $admin = true;
}
if ($admin) { ... }
```

Unless `$admin` is pre-set to `false`, all you need to do in this example is to append `?admin=true` to the URL to bypass the access checks. Without this option enabled, the user-sent admin variable will only end up in `$_REQUEST['admin']` and not in the main variable namespace, `$admin`.

Since PHP does not require that you predeclare variables to be used, people tend not to do that — and in turn get affected by this.

This setting has been deprecated in PHP v5.3.0 and removed in v5.4.0.

4.4.2 Data Submission

When a web application allows the user to submit information to be processed and displayed to others, it is important to take care when handling the data. The inherent problem in the web world, as explained earlier, is that the information flow is a mix of human readable text and programming code to be executed by various parts of the entire chain of processing units.

To avoid problems, special care needs to be taken in every step to ensure that a piece of information that is harmless to the current step will not cause harm to the next (or the one after that).

Some of the methods described below are easy to understand how they can be a problem and possibly how to protect yourself from them. Other methods are much harder to grasp and might require knowledge about how a web browser keeps different "security contexts" for different parts of a web page and communication, which is not trivial.

SQL Injection

An SQL injection is a method of crafting special input sent to a database server with the intent of making it execute code, unintended by the author of the system.

A web application is quite often connected to some form of database for data storage, for instance an SQL database. Special care must be taken to avoid problems when handing user-specified data over to the database server. The CWE Top 25 list [8] mentioned earlier has listed SQL injections as a very problematic area.

To understand the problem, here comes a short introduction to SQL. There are four basic statements used in SQL during regular program flow;

SELECT Retrieving data from the database. Example:

```
SELECT * FROM users WHERE username='Anna';
```

will retrieve data from the `users` table that matches the user *Anna*.

UPDATE Modifying existing data in the database. Example:

```
UPDATE users SET admin=1 WHERE username='Anna';
```

will set the column `admin` to 1 on user *Anna*.

INSERT Adding new information to the database. Example:

```
INSERT INTO person (username, name, admin) \
            VALUES ('Anna', 'Anna Olsson', 0);
```

DELETE Removing information from the database. Example:

```
DELETE FROM person WHERE lastlogin > 365;
```

will remove all persons whose `lastlogin` is over 365 (days for example).

There are more commands, and more complex ways of doing things if you want, but for simplicity — let us stay with these.

The basic problem comes from lack of validation of user-specified data. For instance if the user is supposed to enter a user name, there are five basic methods of handling that data.

1. Take the data without any validation — big security hole.

   For example with the following pseudo code:

   ```
   $result = querydb("SELECT * FROM users WHERE username='$user'
                 AND password='$password'");
   ```

   ...which is supposed to retrieve user information from the `users` table where the user name matches `$user` and password matches `$password`. If it does match, that probably means that the end user specified a correct user name and password, and the user should be authenticated as `$user`.

   Unless properly taken care of, one could specify "*' OR 1=1 OR password='*" as password, giving something along the lines of

   ```
   SELECT * FROM users WHERE username='admin' AND
         password='' OR 1=1 OR password=''
   ```

   The end result here is that the SQL server will retrieve data from the `users` table where a user has the user name `admin` and no password *or* if the number `1` equals `1` (always true). Given the usage plans above, the user will now be authenticated as `admin`, since the verification query succeeded and returned some result.

   The attacker can also do other fancy things, like "*'; DROP TABLE users;*" as exemplified in the comic XKCD - Exploits of a Mom [22]. One can only wonder how many systems would break on such a name (if it would get accepted as an official name, which may be doubtful).

2. Filter out characters or strings known to cause problems — possible security hole. Maybe you did not know of all the special characters used, a new version of the database program introduced a new control character or changed software uses different control characters.

   Let us say that the web developer knows that "`<script>`" is bad and should be stripped out. The attacker knows this and sends "`<scr<script>ipt>`" instead. If the filter system is not careful, it will then strip "known bad strings" from "`<scr`*`<script>`*`ipt>`" resulting in "`<script>`".

3. Escaping characters known to cause problems — possible security hole. The concept is to tell the parser that the next character should not be interpreted as a control character, but as a regular character. This can cause trouble for the same reasons as the previous method.

Example: Turning...

```
'; DROP DATABASE users; SELECT '
```

...into ...

```
\'; DROP DATABASE users; SELECT \'
```

...so that the database program will not interpret the ' characters as start/end of string but just a regular character.

4. Filter out anything except characters known to not cause problems, which is more secure than the previous methods but also (due to laziness/ignorance) the most invasive method. This can be used when you have a restricted subset of valid characters (for instance entering a date), but should probably not be used when validating names for example. The risk of "foreign" names to be filtered out or mangled is quite high, known from personal experience — changing my last name from Ögren to Forsman was partly done to avoid having to deal with issues like this.

5. Separate structure (SQL) and data. Most database engines can keep queries and data separately, resulting in safe handling of data without having to worry about SQL injections. One has to keep in mind that the data will probably be used or presented in some other step, which could have its own set of issues if the data is presented unvalidated. This includes having JavaScript code in your "name" which is then presented on a web page, causing Cross-Site Scripting (see below).

Another way of avoiding problems is to restrict access to the minimal needed. For instance, a search application does not usually need write access to the database tables containing user account information. Restricting the search application to use an SQL View or database account that only has read access to those tables will make SQL injections less useful. This is not a replacement of techniques described above, but rather to be used in addition to them.

Cross-Site Scripting, XSS

Cross-Site Scripting is a method of embedding code on a web page that will turn the victims web browser into a tool used for malicious purposes.

The term Cross-Site Scripting originates from when attackers used to load a page from site *A* into a HTML frame on malicious site *B* and then use JavaScript coming from *B*, to which the user was connected, to manipulate data on *A* according to *B*'s will. Later on, the term still was used but for slightly different attacks. The latter way is often by embedding JavaScript into a page which uses the victims web browser to modify or steal data (such as cookies, which then can be used as a key to access a web site or similar). The script is run in the context of the legitimate site, so the script has access to information related to that site such as cookies, the Document Object Model (DOM) and HTTP authentications.

One attack method with this method is to post some JavaScript aimed at someone logged in as administrator. By modifying the DOM the script could then send off a web request in the background to add a new administrator account. One of these attacks aimed at sites using WordPress [23] puts some more JavaScript into the “realname” part of the new administrator account to hide its existence from the real administrators when looking at the user list. It also modified the page using DOM and JavaScript to change the counter of how many user accounts of each type (regular, administrator etc) you currently have, all to avoid being detected.

If a site has properly protected itself against SQL injections by separating code and data in SQL calls, then it might gladly present the raw HTML/JavaScript in the output to the web client. Just because it is safe for SQL does not mean it is safe for the entire ecosystem.

Example: A malicious user goes to a web forum to register a new account and enters the following (not so common) name:

```
<script>alert('Your computer is under attack! Visit
http://www.superduperantivirus.com for protection!');</script>
```

The malicious user then goes on to post something on the forum, which will make the “name” of this user appear on the page. When the next unsuspecting victim comes along to read about jet engine powered potato peelers (could be the next big thing) on the forum and reads the post by Mr. `<script>alert('Your`..., the computer all of a sudden tells him that his computer is not feeling well and so the victim goes on to install some “protection”.

Cross-Site Request Forgery, XSRF

Cross-Site Request Forgery is the method of using malicious code to send data requests to a web site previously contacted by the victims web browser, piggybacking on previously established communication.

Let us say user X logs into some regular site A and authenticates to do something there (read mail, forum site, . . .). Site A might then send some cookies to X which is stored in the web browser. Later on, X goes to browse some other site which for one reason or another includes HTML code to send data to site A and some JavaScript code to automatically send it away. The web browser will then attach the authentication cookie/information X received from A.

This will most often only enable one-way communication, but it can be enough to send a “change my password to blah” or similar requests.

One way to solve this is to require two-factor password change of some form.

Example: Code from `cwe.mitre.org`:

```
<SCRIPT>
function SendAttack () {
    form.email = "attacker@example.com";
    // send to profile.php
    form.submit();
}
</SCRIPT>

<BODY onload="javascript:SendAttack();">

<form action="http://victim.example.com/profile.php"
          id="form" method="post">
    <input type="hidden" name="firstname" value="Funny">
    <input type="hidden" name="lastname" value="Joke">
    <br/>
    <input type="hidden" name="email">
</form>
```

If a malicious site has code like this, then the victim client will not see anything weird happening and the victim server will see a legitimate call that sends valid cookies for login sessions or similar.

This is not easily blocked, but one method is to generate a nonce (a cryptographic nonsense word/number used only once) when the user logs in and pass that to the client which then has to send it to the server for every request. Since the example code above has no access to a legitimate server response, containing this nonce, it cannot pass it along as verification. If a page is vulnerable to Cross-Site Scripting, then this token/nonce approach can be defeated. By randomizing the name of the hidden token/nonce and possibly generating a new nonce for every page, the application is much harder to exploit.

Some other methods of blocking this is to use a CAPTCHA, or ask the user to enter their password again when doing sensitive calls to the application.

Cross-Site Script Inclusion, XSSI

Cross-Site Script Inclusion is a way to use method/information overloading onto otherwise valid communication to intercept potentially sensitive data.

One example of this type of attack is when a site is sending JavaScript Object Notation (JSON) code as a reply to a request, which is then supposed to be filtered through some code in the client web browser before being presented to the client.

Let us say a site has a page that sends a request to their server asking for some information and returns it in JSON code format, which then the client page evaluates to produce the final result on the client screen.

With XSSI, an attacker can make their own page that sends off the same request to the good site asking for the information. The client browser will attach any login information needed, as it appears to be a valid call to the site. Instead of using the site provided callbacks to render the data on the client screen, the attacker has their own version that intercepts and captures that sensitive data. The end result is that the malicious site can run their own code in the security context of the good site, using the victim login information but sending the captured data home to Evil Inc.

Hidden form variables

To preserve state information throughout a visit to a website, there are mainly two methods being used. One method is to store the state information on the server and just leave a small identifier on the client system so you can make sure that a client uses the correct data stored on the server. That part is described in the Sessions part below.

Another method is to just store the information in "hidden"[2] form variables in the HTML code. For instance, you can store `username="hugo"` as a variable. The main problem with this method is that the user (with just a little bit of extra knowledge or tools) can change that user name from *hugo* to for instance *administrator* or such.

Using hidden form variables can be useful for non-security data, such as that the user picked blue background in the last step.

Example HTML code:

```
<form action="admin.cgi" method="PUT">
    <input type="hidden" name="preauthenticated" value="0">
    <input type="submit" value="Administrate this app">
</form>
```

Although this example is very easy to break, there are lots of places where similar (but not as obvious) problems appear.

Cookies

Cookies are a method of storing small pieces of information in the web client that can be accessed by a server with the rights to do so. This is mostly used to just store an identifier so the server knows which client it is talking to.

The cookie consists of various predefined attributes and application specified attributes. The predefined attributes are expiration date, domain name where the cookie is valid, and a web path where it is valid.

[2]They are only hidden from the regular rendered view of the web page, but in plain view if you look at the source code.

As with hidden form variables, the user can modify the contents of a cookie, and its contents should not be trusted at any time. Using it just to store a unique identifier (large enough to prevent guessing collisions) and then store actual relevant information on the server is a preferred method.

Microsoft created a flag for cookies called *HTTPOnly*, which is an additional information the server sends to the client saying that this cookie is not to be accessible by scripts. As of 2011, about 99% of web browsers out there supported this flag according to Google Browser Security Handbook [24] and OWASP [11].

Cookies cannot contain viruses or other harmful data to the client, but they can be used for tracking a client across multiple sites which is a privacy concern. In 2002 the European Union adopted a law [25] that, amongst other things, made it prohibited to store cookies on a client computer unless the end-user had accepted them. Sweden adopted this law in 2003. In June 2012 the *Article 29 Data Protection Working Party* released an updated “Opinion” [26] on how to interpret the previous law, making “necessary” (the text is a bit vague) cookies and cookies not meant for tracking exempt from the need for end-user acceptance. The law has not changed, but this update seems to be the opinion of the Data Protection Working Party, and may show a possible future for this law. In January 2013, the Information Commissioner’s Office (the organisation responsible for policing the UK cookie law) announced that they will no longer ask for permission, only inform the customer that they will be using cookies on their own web site.

Sessions

When using so called Session variables, you need some way to keep track of which client belongs to which set of data. Two common methods are to store a unique key at the client side, either by putting it in a web browser cookie, by putting it as a hidden form variable, or including it in the URL (`http://site.example.com/url/;jsessionid=...`). The last section mentioned that hidden form variables should not be used for anything security related, but if all you store is a key that is unique and large enough to avoid brute force attacks, it is mostly safe.

If form variables or the session id is included in the URL, the “key” can slip out from the client computer when you send it off to a friend (“Hey, have a look at this site”). You surely do not want to mistake that other client for being the same as the first one, so you need to for instance store from which IP address[3] that key is valid. This is not used all the time, which can lead to either information leak or malfunctioning depending on what information is stored in those session variables. Unfortunately, Carrier Grade NAT (an ISP putting multiple customers behind the same IP address) makes this less useful.

You probably want to keep track on how long the key should be valid as well. If a certain key is not used for a week and then it is presented again, chances are that it is not the original client. Even if it is, requesting a client to login again after being idle for a week is probably not too much of an inconvenience.

[3]Internet Protocol address, a more or less unique number for each computer connected to Internet.

4.4.3 Shared Systems

Quite often, different web applications are hosted on the same web server but still being run as the same operating system user. This can lead to information leak or manipulation of data for other web applications. The main problem here is, for instance mod_perl[4], mod_php[5] and such, which are embedded in the web server (for performance and statefulness). As the code for *webapp1* is run as the same operating system user as *webapp2*, all *webapp1* need to do is to dig into the directory where *webapp2* is located to find various data.

There are many ways of solving this:

- Running a separate web server instance (as different users, maybe on different computers) for each web application. Since you normally want web applications to run on port 80[6], you need to put some clever system in front of this to separate out requests to the correct instance. One example is to use Apache HTTP Server [27] in Reverse Proxy mode. See Figure 4.1.
- Having different IP addresses for each one, but still on the same machine. See Figure 4.2.
- Running each web application on their own machine, physical or virtual. See Figure 4.3.
- Using something like Perchild MPM[7], Metux MPM, Peruser MPM or MPM-ITK for the Apache HTTP Server, which can keep track of different virtual hosts and run them under different operating system users. Unfortunately, development of several of these seem to have stopped. See Figure 4.4. A more common trend nowadays is just to setup a new virtual machine for each project.
- Not using the embedded modules for running code, but spawning separate processes for each request. This could be used with something like FastCGI [28] which is running outside the web server software and just getting requests sent there when something needs to be run. See Figure 4.5.

The Apache Project [29] has a page listing some possible solutions for how to perform Privilege Separation [30].

[4]Apache HTTP Server module for executing Perl code without starting the interpreter each time.
[5]Same as above, but PHP.
[6]Assigned by IANA, Internet Assigned Numbers Authority.
[7]Apache Multi-Processing Modules, the server code responsible for managing client connections.

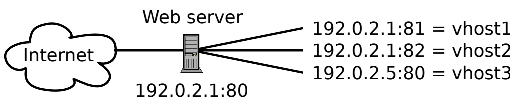


Figure 4.1: Multiple ports on a single machine with for example a HTTP splitter

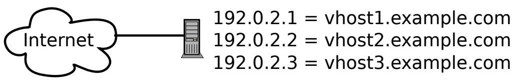

Internet
192.0.2.1 = vhost1.example.com
192.0.2.2 = vhost2.example.com
192.0.2.3 = vhost3.example.com

Figure 4.2: Multiple IP addresses on a single machine

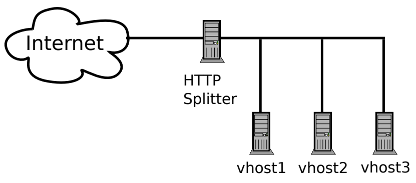


Figure 4.3: Separate web server machines with a HTTP splitter in front of them

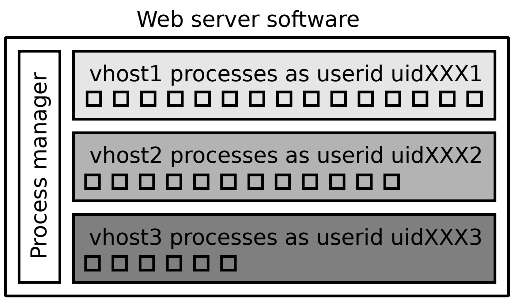

Web server software
Process manager
vhost1 processes as userid uidXXX1
vhost2 processes as userid uidXXX2
vhost3 processes as userid uidXXX3

Figure 4.4: Web server keeping pools of processes running under different userids

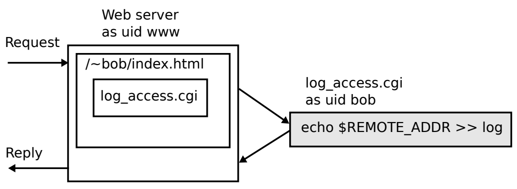


Figure 4.5: CGI applications running as different userid

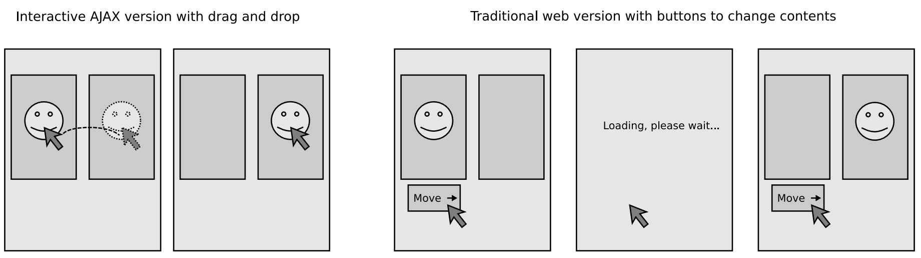

Interactive AJAX version with drag and drop
Traditional web version with buttons to change contents
Move
Loading, please wait...
Move

Figure 4.6: AJAX vs traditional method of moving an object

Virtual Machines

Since hosting multiple mutually distrustful customers in the same operating system can be problematic, people have started to turn towards isolation through virtualization for a solution. Recent papers [31] have shown that even two separate virtual machines hosted on the same physical host can leak information, such as cryptographic keys, due to properties of CPU caches and content-dependent code paths.

This shows that hosting a security sensitive application on a Cloud broker can be insecure due to properties of the actual executing unit, in addition to the fact that you are handing over your secrets to someone else.

4.4.4 AJAX

The “Web 2.0” concept involves technologies such as AJAX[8], see Figure 4.6. Unfortunately, as with many other new technologies, people are using it recklessly without much thought to security.

The basic concept of AJAX is that when the user interacts with some object on the web page, the web browser sends a small request to the web server asking for something to be done or for some additional information. This information is then inserted live into the page and the information on the users screen changes. The traditional “Web 1.0” method involves reloading the entire page, causing slowdowns and a flow more like a “slide show” than something you really interact with.

Just like the problem with hidden form variables, many AJAX applications can be abused by modifying what is being sent to the server, unless proper precautions has been taken to stop such abuse.

Some web browsers restricts the communication via `XmlHttpRequest` to just allow communication with the server where the code came from. This is done to avoid some types of security problems.

In AJAX, more code is being run in the client, which lacks control, instead of at the server where you can control and validate everything as needed.

[8]Asynchronous JavaScript And XML.

4.4.5 Faulty Assumptions

“People have no reason to break into our systems”

Does not matter, someone will try anyway.

“We have a firewall to protect us”

If you allow logins from outside, then your firewall is probably not going to save you.

“Everyone follows the standards”

This assumption can cause a lot of problems such as information leak. An example of this will be shown below.

Misuse of HTTP status codes

The HyperText Transfer Protocol (HTTP), the protocol used to request and receive web pages on the Internet, uses numerical replies to provide a machine parsable reply. Each numerical reply corresponds to a specific status and is is well-defined on how to react on each of them. If status codes are misused, potential security problems can arise which will be explained in the example below.

Here is a list of common status codes:

200 OK, content found, here is your data.

301 Moved Permanently, the requested data has moved and the new location should be used in the future.

302 Moved Temporarily, the requested data can be temporarily found at a new location.

403 Permission denied, current authentication does not authorize access to the requested data.

404 Content not found, it does not exist.

Unfortunately, some software, like the InfoGlue [32] system used at Umeå University, did not follow this protocol correctly up until recently. When a requested resource was not found, it should have given the response *404 Not found* which both machines can understand and humans can interpret. Instead, it redirected via *301/302* to another page (giving a *200 OK*) that says in human readable (but not machine parsable) form that the requested page could not be found. Apart from being annoying for the visitor (URL field in the browser becomes changed to the error page), this means that if a computer software requests a page/file that does not exist, it will appear as if it *does* exist. This problem has been fixed in InfoGlue after being pointed out as bad behavior with possible security implications.

Google Webmaster Tools (GWT) [33] is a system to check statistics, configure some aspects on how fast Google will crawl a certain web site and so on. In order to avoid that any random person on the Internet can get statistics on a certain web site, GWT uses a method of authenticating that you are actually the rightful owner of that web site. Before giving out any information to you, you are to add a specific file to your web server under a very specific name, so that when GWT checks, your web server will say "Yep, here is the data". Fortunately, GWT does not rely on only verifying whether you have added the file or not, but also verifying the content. If GWT would have only checked if the web server says "Yep, here is the data", then the old behavior of InfoGlue would have provided anyone access to "managing your site" at GWT. Allowing anyone to viewing full statistics could reveal at least the presence of otherwise hidden material, in other words — an information leak.

There is a high probability that somewhere on the Internet, there is some other tool similar to GWT, which checks if the requested file exists - but does not check its content, since normally you do not have files named e59ff97941044f85df5297e1c302d260.txt or something similar.

4.5 Modifying Runtime Environments

Even if the best solution is to secure all the components of a system, there can be times when modifying a component is not possible but you still want to add some layer(s) of security to your system, just in case.

4.5.1 Suhosin

Suhosin[9] [34] is a protection system for PHP. It is a two-part system that can be used separately or in combination. Suhosin is OS independent.

The first part is a patch[10] to the PHP core source, adding protection against some buffer overflows and format string vulnerabilities.

The second part is a PHP extension module that implements a long list of protection features, such as blocking known GET/POST/COOKIE variables that can cause problems, protect against session hijacking etc.

To make it easier to start using, Suhosin can be run in a "simulation" mode that will only warn you when something would have been stopped if it had been fully enabled. This allows you to check whether your applications would have been stopped by Suhosin rules even when running under normal conditions.

[9] South Korean word roughly meaning "Guardian angel".

[10] Modification to the source code, altering the default behavior.

4.6 Operating System Level Protection

Instead of (or in addition to) having built-in protection in the runtime environment, such as the PHP / Perl / Ruby / Python interpreter, you can make the operating system enforce some protection nets for the applications.

4.6.1 Access Control

One common model for dealing with access rights in an operating system is the Discretionary Access Control where the access policy to an object is determined by the owner of the object. You can as an example modify the file permissions on a file in your home directory to either allow everyone on the system to do whatever with it, or you can say you want to keep it private to yourself.

Another model is the Mandatory Access Control (MAC) where the access policy is determined by the system (and by that extent, the system administrator) instead of the owner of the data. Rules dictate which subjects (people and their processes) can do what operations (read, write, create, delete, . . .) on which objects (files, network connections, memory, . . .).

4.6.2 AppArmor

To quote the homepage of AppArmor [35]; "*AppArmor is an effective and easy-to-use Linux application security system. AppArmor proactively protects the operating system and applications from external or internal threats, even zero-day attacks, by enforcing good behavior and preventing even unknown application flaws from being exploited.*". AppArmor is a Linux specific MAC system which is installed by default on Ubuntu among others.

The framework allows the system administrator to define rules on what an application can and can not do regarding executing other applications and making file operations. The original design is for managing individual applications, but web applications are often run "inside" the web server (like Apache HTTP Server) through software like mod_perl, mod_php and other mod_*someotherlanguage*.

The security rules for the web server (reading files, writing to web server logs), and say a web mail system (connecting to an IMAP server, reading/writing state files, writing to web mail logs) are quite different. If you would apply the web server security profile to the web application or allowed both to do whatever the other should be able to, the result would not be satisfactory. AppArmor can switch security profile when starting the web application (with a separate profile for each application), and switch back when done using the mod_change_hat software for Apache HTTP Server for example. AppArmor allows the system administrator to define what each application (and in the web eco system, even "sub-application") should be able to perform and if the application tries to do something forbidden, it will both be stopped from doing that and the issue will be reported for further investigation.

4.6.3 SELinux

Security-Enhanced Linux (SELinux) [36] is another Linux MAC system, developed by US National Security Agency (NSA). It uses strict rules and also labels data with appropriate security levels, which is checked to see that the current user is allowed to access that particular data. SELinux has “higher potential” than AppArmor, but is also a more complex system, giving a higher probability of misconfiguration that will cause security issues. Red Hat Enterprise Linux has SELinux enabled by default since version 4 released in 2005.

4.6.4 Grsecurity

Grsecurity [37] is a patch to the Linux kernel that offers a plethora of protection layers and methods to make sure that no unintended information leak, modifications or code executions happens. It also provides protections against some race conditions and provides auditing that will log and alert when something prohibited is attempted. Address Space Layout Randomization (ASLR) is a technique to make it harder to predict memory addresses and thus make some attacks much harder. ASLR came from a sub-project in Grsecurity, called PaX, and is now part as a security measure in for example OpenBSD, Linux, Solaris, Microsoft Windows, Apple Mac OS X, Apple iOS and Android.

4.6.5 Solaris Trusted Extensions

The Sun/Oracle operating system Solaris includes Solaris Trusted Extensions [38] which has been along for a long time, at least previously commonly used by US Military. It is similar to SELinux but fully integrated in the desktop environment, clearly labeling which application is running under which privilege, and managing the flow of information to avoid information leaks and other problems.

As an example, an application running with low security labels is not allowed to paste data from the clipboard copied by an application running with high security labels. The filesystem is also aware of what label different data belongs to, even if Zones virtualization technology is used.

4.7 Security Programming in Practice

WebGoat [39] is a deliberately insecure web application maintained by OWASP [11] designed to teach web application security lessons. As of October 2013, it contains over 30 different lessons including weak session cookies, SQL injections, and more.

Chapter 5

Ticket Handling System

5.1 Introduction

At the Department of Computing Science at Umeå University (hereafter called CS), a few people have the role of computer/tech support. A part of their daily routine is to receive requests for doing some kind of work, like requests for help with installing some software, create a Subversion repository, refill paper in printers, and similar tasks. Some of the requests need quick attention while others are more of a heads-up for something that will happen later on. At any given time, there can be quite a lot of ongoing tasks to be done now or later.

Normally in organizations like this, some form of ticket handling system is most often used to aid the support staff in keeping track of the information flow regarding each of the tasks.

Part two of this Masters Thesis is to create such an application, specifically tailored for the needs and use-cases of the support group at CS.

5.2 Definitions

Ticket A group of information regarding the data involved in a specific case, such as all the e-mail correspondence and metadata like when it is supposed to be done. In both previous and the new system, a ticket identifier in the form of `support#20120102.3` is used to indicate that it belongs to the queue "support", started on 2012-01-02 and was the 4th (zero based counting) ticket of the day.

Queue A specific stream/group of tickets normally grouped together due to some common criteria, for example having the main queue for the department and also a separate queue for a sub-organization with a slightly different constellation of support staff.

Owner The person currently designated as the one responsible for solving a certain ticket. By default, the first support staff that replies to a ticket becomes the owner. If for example multiple staffers send a similar reply to a new ticket just when it arrives, the first reply will claim ownership and the second will be rejected because (s)he is not the owner. The reply can be forced to be sent anyway if deemed necessary.

5.3 Previous System

The system to be replaced, Must, is written in Perl 5 with only a few external modules. It only accepts input via e-mail and stores all such in traditional Mbox[1] format with additional metadata storage as simple key/value files. It is a really simple system, which has proven to be robust against changes through system upgrades.

As an e-mail comes into the system, only a few select parts of it is parsed. The most important fields being checked for is the sender, recipient and subject line — as the subject line can contain information such as ticket number and also administrative commands. The mail body is copied verbatim to the relevant recipients, without the need to parse it at all.

There is a command line interface which can read data/metadata files directly for showing information. For commands that modifies anything, it will send an e-mail with commands in the subject field to the main system for further processing.

As a bonus with this system, all history of all changes to a ticket is preserved and can be played back if needed — for example when importing to a new system.

5.4 Design Background and History

Since this system is to be run at the Department of Computing Science, whose main computing infrastructure is Unix/Linux based, it seemed like a good idea to base it on that.

We wanted to have both an e-mail interface just like before, but also a web interface for viewing and managing tickets. Due to the "hit'n'run" design of CGI based web applications, it would be slow to re-parse all the metadata at each page view, so the obvious choice for data storage then became an SQL database with pre-parsed information. Unfortunately this meant that we could no longer just store the message body verbatim and pass it along to whoever was to take part of it. If the information is supposed to be sent out via e-mail, then sending the mail verbatim is fine, but for web clients it needs to be parsed, sanitized and then displayed.

Parsing an e-mail in full is not a trivial task, as it can be a big tree of sub-parts with various content and content-types. A normal method is for example to have a `multipart/alternative` with two sub-parts, `text/plain` and `text/html`. The `text/plain`

[1]http://en.wikipedia.org/wiki/Mbox

part is probably suitable for showing in a command line interface, as that is supposed to be a pure text representation of the same content as the `text/html` one. Unfortunately, that is often not the case — but instead it can contain a single line saying “Your e-mail client does not support HTML”. Adding to the list of problems is also the fact that there are many applications out there generating e-mails that only “almost” adheres to the standards, which makes parsing even harder.

Having a seemingly prepared chunk of content in HTML format would at first sight be very suitable for showing in a web application, but then also vulnerable to many of the attacks explained in earlier chapters. Some form of sanitization need to be performed before sending it to the web browser, both to prevent XSS and such, but also preventing non-malicious style sheets from the mail content to affect the web interface of the ticket handling system. HTML is not good at having HTML sub-content that is to be sandboxed within a small box, unless you start using frames or iframes, of which both have their share of usability problems.

At first, I started designing a web application that read data from the SQL server and presented it to the user. Modifications were written directly back to the database, using PHP DataObjects[2]. This result was pretty easy to manage and there was no easy way of breaking it, but then I started working on receiving data via e-mail as well which meant parsing the non-trivial e-mail structures and choosing what to present and how. All of a sudden, the project grew quite a bit and things like audit records about what happened to a case started to get complex, much more than the previous method of just storing all incoming mail. Presenting the changes over time also proved to be a hard task to get right, since there is a lot of information that could change inbetween two messages in the same ticket.

Due to availability of helping frameworks, the e-mail handling code was written in Perl. Having requirements of both PHP and Perl might seem bad, but each of them actually has advantages over the other for the different use cases — at least in simplicity of getting up and going, without spending a few years of writing code. One problem with this was that I now had two different systems (written in two different languages) having partly overlapping functionality, like updating the database with changes.

Sad to say, I had fallen for the old problem of starting to code before the design was complete. Trying to rescue the situation and get a systematic audit log for current and future purposes, I modified the web application to change its behavior into mimicing the old system — all changes are to be sent as control e-mails to itself. The base control messages were very much the same as the old system, which had been proven to be easy but useful enough. One possibly confusing thing about this method is that changes in the web interface does not happen immediately. If one closes a ticket and then is sent back to the list of open tickets, the should-be-closed one is still there, only to be gone if you reload a second later. A positive thing about this was that control logic for updating the database was moved to one place (e-mail parser) instead of both in the e-mail system and the web system, reducing code duplication (in different programming languages).

After having spent way more time than this thesis should cover, I had a system which pretty much did the same thing as the old one but had way more system requirements

[2]An object-oriented approach to SQL in PHP that I had used for other projects before.

but without doing that much more. This is not a good conclusion to end up at, but nevertheless it is what happened.

5.5 System Description

5.5.1 System Requirements

- Unix-like operating system
- Web Server (for example Apache HTTP Server)
- SMTP Server (for example Postfix)
- PHP 5 for the web interface
    - PEAR DB_DataObject
    - HTML Purifier
- Perl 5.8+ for e-mail gateway and command line interface
    - DBI
    - DBD::Pg
    - MIME::Parser
- PostgreSQL RDBMS

5.5.2 System Parts

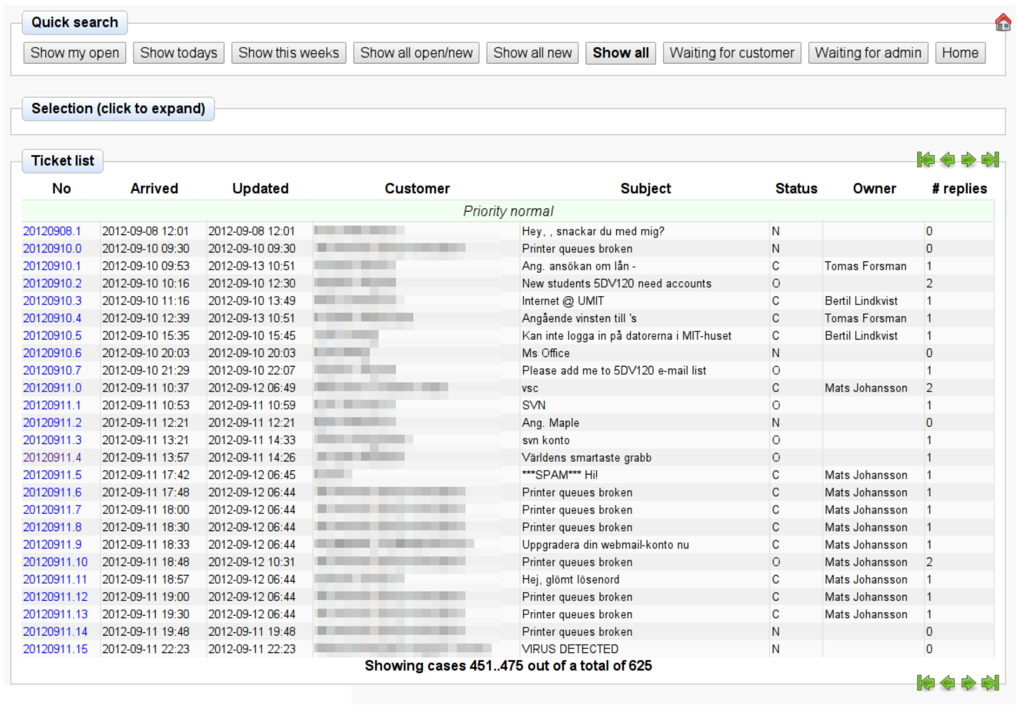
The ticket handling system consists of four parts; the database storing all the data, a command line client for trivial management tasks, a web application for managing tickets, and an e-mail gateway for receiving both new tickets and also for ticket management via e-mail.

Database

The database is the central data storage for the entire system. All other components communicate with the database for information retrieval, and the e-mail gateway does updates. The database is separated into several SQL tables for structured data and metadata storage.

Command Line Client

This client can be used for trivial management tasks like listing open tickets, changing owner or closing tickets without a reply. It does not allow adding or modifying any data content.



No	Arrived	Updated	Customer	Subject	Status	Owner	# replies
Priority normal							
20120908.1	2012-09-08 12:01	2012-09-08 12:01	[redacted]	Hey, , snackar du med mig?	N		0
20120910.0	2012-09-10 09:30	2012-09-10 09:30	[redacted]	Printer queues broken	N		0
20120910.1	2012-09-10 09:53	2012-09-13 10:51	[redacted]	Ang. ansökan om lån -	C	Tomas Forsman	1
20120910.2	2012-09-10 10:16	2012-09-10 12:30	[redacted]	New students 5DV120 need accounts	O		2
20120910.3	2012-09-10 11:16	2012-09-10 13:49	[redacted]	Internet @ UMIT	C	Bertil Lindkvist	1
20120910.4	2012-09-10 12:39	2012-09-13 10:51	[redacted]	Angående vinsten till 's	C	Tomas Forsman	1
20120910.5	2012-09-10 15:35	2012-09-10 15:45	[redacted]	Kan inte logga in på datorerna i MIT-huset	C	Bertil Lindkvist	1
20120910.6	2012-09-10 20:03	2012-09-10 20:03	[redacted]	Ms Office	N		0
20120910.7	2012-09-10 21:29	2012-09-10 22:07	[redacted]	Please add me to 5DV120 e-mail list	O		1
20120911.0	2012-09-11 10:37	2012-09-12 06:49	[redacted]	vsc	C	Mats Johansson	2
20120911.1	2012-09-11 10:53	2012-09-11 10:59	[redacted]	SVN	O		1
20120911.2	2012-09-11 12:21	2012-09-11 12:21	[redacted]	Ang. Maple	N		0
20120911.3	2012-09-11 13:21	2012-09-11 14:33	[redacted]	svn konto	O		1
20120911.4	2012-09-11 13:57	2012-09-11 14:26	[redacted]	Världens smartaste grabb	O		1
20120911.5	2012-09-11 17:42	2012-09-12 06:45	[redacted]	***SPAM*** Hi!	C	Mats Johansson	1
20120911.6	2012-09-11 17:48	2012-09-12 06:44	[redacted]	Printer queues broken	C	Mats Johansson	1
20120911.7	2012-09-11 18:00	2012-09-12 06:44	[redacted]	Printer queues broken	C	Mats Johansson	1
20120911.8	2012-09-11 18:30	2012-09-12 06:44	[redacted]	Printer queues broken	C	Mats Johansson	1
20120911.9	2012-09-11 18:33	2012-09-12 06:44	[redacted]	Uppgradera din webmail-konto nu	C	Mats Johansson	1
20120911.10	2012-09-11 18:48	2012-09-12 10:31	[redacted]	Printer queues broken	O	Mats Johansson	2
20120911.11	2012-09-11 18:57	2012-09-12 06:44	[redacted]	Hej, glömt lösenord	C	Mats Johansson	1
20120911.12	2012-09-11 19:00	2012-09-12 06:44	[redacted]	Printer queues broken	C	Mats Johansson	1
20120911.13	2012-09-11 19:30	2012-09-12 06:44	[redacted]	Printer queues broken	C	Mats Johansson	1
20120911.14	2012-09-11 19:48	2012-09-11 19:48	[redacted]	Printer queues broken	N		0
20120911.15	2012-09-11 22:23	2012-09-11 22:23	[redacted]	VIRUS DETECTED	N		0

Showing cases 451..475 out of a total of 625

Figure 5.1: List of tickets in the web application

Web application

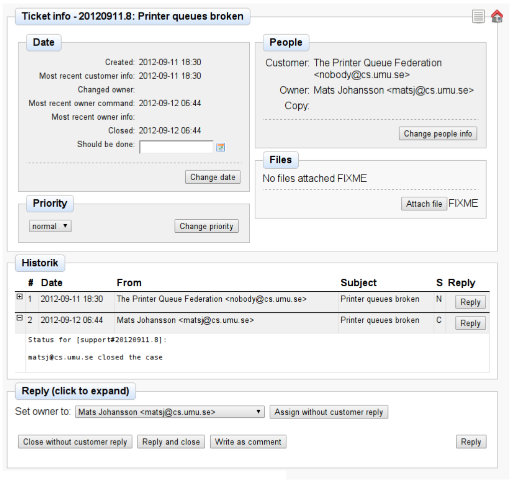
The web application consists of a few different parts; authentication, summary screen, ticket list view and ticket view.

Authentication Authentication is performed with logins using regular CS accounts.

Summary Screen The summary screen shows a summary of how many tickets you are involved with, in one way or another. For non-support persons, it shows which ones you have submitted yourself. For support staff, the summary contains information for the different queues that they are flagged as active in.

In the summary screen there are various shortcuts for quick searches of the various queues, like showing all tickets assigned to "me" etc.

Ticket List This is where one can search for either something specific or some meta-groupings of tickets, like "open tickets", "tickets that should be done this week" etc. See Figure 5.1.



Ticket info - 20120911.8: Printer queues broken

Date

Created:	2012-09-11 18:30
Most recent customer info:	2012-09-11 18:30
Changed owner:	
Most recent owner command:	2012-09-12 06:44
Most recent owner info:	
Closed:	2012-09-12 06:44
Should be done:	

People

Customer:	The Printer Queue Federation <nobody@cs.umu.se>
Owner:	Mats Johansson <matsj@cs.umu.se>
Copy:	

Files

No files attached FIXME

Attach file FIXME

Priority

normal

Historik

#	Date	From	Subject	S	Reply
1	2012-09-11 18:30	The Printer Queue Federation <nobody@cs.umu.se>	Printer queues broken	N	Reply
2	2012-09-12 06:44	Mats Johansson <matsj@cs.umu.se>	Printer queues broken	C	Reply

```
Status for [support#20120911.8]:

matsj@cs.umu.se closed the case
```

Reply (click to expand)

Set owner to: Mats Johansson <matsj@cs.umu.se>

Figure 5.2: Information about a specific ticket in the web application

Ticket View In the ticket view, all information regarding a specific ticket is shown. Metadata like the date of last update from the customer, from the support staff and due date are shown and in some cases available for modification. See Figure 5.2.

The entire conversation history regarding the current ticket is shown in a list and new replies can be added. Controls for changing the owner of the ticket are available and also controls for changing the state, like open or closed.

Some JavaScript code is active to give bonus functionality, but all basic functions are working even without JavaScript enabled.

E-mail gateway

The e-mail gateway receives e-mail from either customers, support staff, or CLI/web UI, and acts upon the contents of them. If the e-mail has the necessary metadata to be tied into a previously known support ticket (having `[support#20120102.3]` or similar in subject), then the new e-mail is parsed and attached to the old ticket, updating necessary metadata depending on the contents and new e-mails are sent out to people involved in the ticket. If the e-mail does not contain metadata tying it to a previously known ticket, it is treated as a new ticket. Depending on the recipient address, it will be put into different queues.

Support staff can also use the e-mail gateway for administrative commands, like replying to one of the customer mails and add the command ”solve” to mark the ticket as solved/closed, removing it from the view of the currently active tickets. Closed tickets are still available in the system, but does not need to be shown in the daily workflow.

No real security measure is put in place to authenticate the sender for administrative commands, such as GPG or password. Sender addresses in e-mail can be forged, just like on a post card, but in this case the potential security threat from forged administrative commands are deemed not to be problematic.

5.6 Database

The SQL database consists of a couple of different tables, all joined together in various ways. Not all traditional database designs have been applied to the full extent, such as "no information should be duplicated, but referred instead". One example of that is the name ↔ e-mail mapping, because it is not guaranteed to be constant (people do change names). In most other cases there are hard references with unique id numbers used as mapping, ensuring database consistency.

List of primary tables used in the system:

events is a list of "events" that is a new state of a ticket, containing most of the information about a case. Updates to an old case use the last event as base for the new event.

cases is a list of cases, but only with references to events describing the current state of the case and a text identifier of the case.

queues is a list of queues with their descriptions, whether they are public and last ticket number.

adminmap is a list of who is administrator over which queue. Only administrators are allowed to look at all the tickets in a queue.

statetypes is a list of different states (new, open, closed, spam, ...) with some surrounding information on whether to treat each and one of them as active or not etc.

Size calculations on the database regarding future growth has been done, data from the last 10 years in the old system is about 300MB, which should not be considered a problem.

5.7 Data Flow

5.7.1 Data Input

Ticket creation can be done in two ways; either by sending an e-mail to the system or by visiting the Create New Ticket (CNT) page. The CNT page gathers the input sent from the client via a web form and sends an e-mail to the e-mail gateway for processing. Either way, it is the e-mail gateway that receives the new ticket and fills all the relevant database tables with information regarding this ticket. At the same time, mails will be sent out to all the administrators for the targeted queue, after modifying the `From:` and `To:` fields of the e-mail to point at the e-mail gateway. This is done to assure that all communication goes through the system, both to preserve a clear history of what has happened in the ticket and as notification to other support personnel that it is being handled. A notification is sent to the original submitter with a welcome message explaining that their mail has been received and how to proceed if further information is needed.

To make it clear which ticket and queue each mail belongs to, the `Subject:` header is modified to include a ticket ID.

Example:

```
Subject: [support#20120102.3] Help! My network cable is on fire!

It is burning!
```

This means that it was sent to the "support" queue at 2012-01-02 and was ticket number 4 so far that day (zero-based counting). This ticket ID is kept in the subject line for all incoming and outgoing mails as a method of keeping all communication intact.

5.7.2 Updates

When one of the administrators receive the first mail sent out by the system, the easiest method to communicate with all relevant parties is just to send a regular mail reply (which will go to the e-mail gateway). The e-mail gateway then sends copies of this mail to the mail addresses associated with this specific ticket, normally the administrators of the queue and the submitter. Additional people can be added to the ticket, and thus they will also receive future updates.

The ticket is kept open as long as it has not yet been closed. A very common case is that only one reply from the administrators is needed to solve the ticket. To do that, one can add a flag within the `[]`'s indicating that the ticket should be marked as solved.

Example:

```
To: support@cs.umu.se
Subject: [support#20120102.3:S] Re: Help! My network cable is on fire!

Fire department has been alerted and is on the way.
```

Here, the `:S` will tell the system that the administrator would like to mark the ticket as solved, while still sending this mail to the submitter for information. This will remove the ticket from the list of active tickets, as the fire department will take care of the cable. Not seldom, the original submitter will send a reply after that with a token of gratitude or notification of acknowledgement, and will thus open the ticket again.

If the e-mail is actually junk mail or notification to the submitter is for other reasons unwanted/unnecessary, it can be marked as closed with `:C` which will not send any notification to the submitter — only a notice to the administrators that it now is closed (and thus removed from the list of active tickets).

5.8 Technical Comparison

In this comparison table, some technical feature differences between the old and the new system will be shown. *Old* refers to the system previously used, Must, and *New* refers to the newly developed one which lacks a proper name. The different methods, e-mail, web, and Command Line Interface (CLI) are listed for different use cases.

Feature	Old	New
Show tickets via e-mail	Partially, e-mail summary each week	No (Weekly summary should not be hard to implement)
Show tickets via web	Partially (A limited version based on the CLI has been created)	Yes
Show tickets via CLI	Yes	Partially
Accept new tickets via e-mail	Yes	Yes
Accept new tickets via web	No	Yes
Accept new tickets via CLI	No	No
Accept updates via e-mail	Yes	Yes
Accept updates via web	No	Yes
Accept updates via CLI	No	No
Management via e-mail	Yes	Yes
Management via web	No	Yes
Management via CLI	Yes	Yes
Data storage	Mbox	SQL
Metadata storage	Plain text	SQL
Search method	Read all data every time	Pre-parsed and indexed through SQL
Filtering possibilities	Trivial	Many
Programming language(s)	Perl, (HTML, CSS)	Perl, PHP, SQL, JavaScript, (HTML, CSS)
System requirements	Few	Many
Maturity	10+ years of use	Never used

As can be seen in the table above, the web interface is the more prominent feature of the new system, along with its filtering possibilities to find specific tickets. The old system can, given a ticket id, open a text based mail client (`mutt` or `pine`) on a mailbox

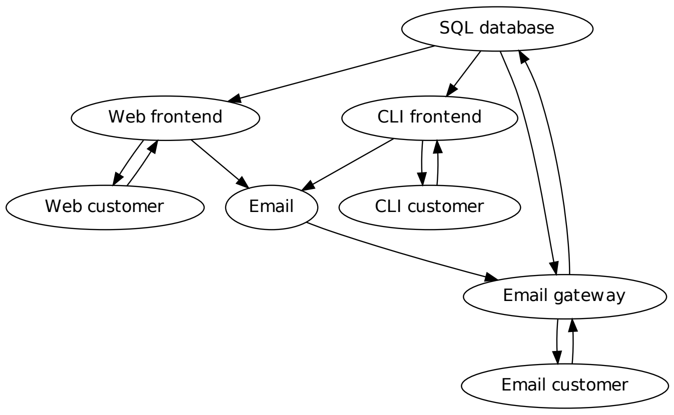


Figure 5.3: Overview of data flow in the new system

containing all e-mail correspondence for that ticket. Due to the storage model of the new one, this function has not been implemented (yet).

A visual description of the data flow in the new system is shown in Figure 5.3. The old system, Must, looks similar but does not talk to a dedicated SQL database, but reads and writes data and metadata files itself. As can be seen in the figure, only the e-mail gateway writes data to the SQL server, the other frontends perform all changes through e-mail commands to the e-mail gateway.

Chapter 6

Results and Conclusions

6.1 Part One - Web Security

Some of the work in this section comes from information gathered during daily work, then refined and hopefully presented to the reader in a form that can be comprehended even if you are not deeply involved in these kind of areas of Computer Science.

If you turn back the clock about 10 or 15 years, people did not care much for security in web applications because there was no one trying to exploit it. But now, awareness of the security issues has begun to grow.

Unfortunately, there are many applications that has been developed for many years and started without a secure design. They can either be patched “as you go” or be rewritten from scratch. The former method has the disadvantage of maybe breaking things and it might be hard to fix all problems due to the design. Rewriting applications from scratch with security in mind will probably give the most secure end result, but getting there will take a lot of work and time. Telling users that the next version will be delayed for 3 years because you are rewriting everything might not be appreciated and you could end up with a new product that breaks old work flows and contains brand new security holes instead.

As many web applications are written in some interpreted language such as PHP, Python, Perl, Ruby etc, it is quite important that the language interpreter itself does not contain security holes. Even the most securely coded application can be broken into if the framework it uses has holes. The reverse also applies, that even with a very secure interpreter and framework, you can make mistakes that leaves gaping holes.

Every day, new and inexperienced programmers write code that are exposed to the Internet. If one of those programs happen to get popular and its use start to spread, there will be people out there that write exploits for those programs. Throughout history, there has been many applications that are popular and people like, but the foundation of the programs are like Swiss cheese. Patches to fix security issues are released all the time, but new holes are found because often the holes are fixed by removing the symptoms and not fixing the cause.

It is often easier (at least slightly) to write code if you do not have to care about every attack vector that might affect it. People then start out the easy way to get something working as fast as possible, instead of doing more grunt work without any "visual results". As time passes by, some might think "I will have to go back and fix this later", but pressure from users to implement feature X might change the priorities to do something that you can actually show instead of "I did 4 weeks of serious coding and now everything looks and works exactly like before". Unfortunately, this can (and probably will) come back and bite you, with upset users as the end result and costing various organizations a lot of money in cleanup costs.

The basic problem is probably that we humans want to have things cheap and want it now. Postponing something and giving it a higher cost because of something that might not affect me now is not part of the short term plans. Unfortunately, every now and then you need to think about longer term plans as well.

6.2 Part Two - Ticket Handling System

At first glance, this seemed to be a not-too-hard thing to write and it should easily fit within the expected time frame of this masters thesis. As time passed, both while actively working on this and not, it became clear that this is quite a big project. Since the purpose of this thesis was to make something useful, I kept on working on it. After spending way more than the expected 30 credits, an insight started to grow in my mind. The new system was more complex, primarily in system requirements, than the old (Must) but did not do *that* much more useful work.

The old system has been working for quite some years now, through many systems upgrades without requiring large amounts of work to adapt to the new environment. The new system with its additional requirements would add a burden and fragileness in that respect for no real gain.

With just a little bit of work, a trivial web interface for the old system was created, mostly for getting an overview of the ticket queues and for an easier way of cleaning up spam. Parts of this was made much easier due to work done on the "replacement", so it was not all wasted. Some more work will be done on this, but not in the same scale as initially planned.

Parsing an e-mail sounds like an easy task, even when using frameworks that do most of the hard work for you. Unfortunately, e-mail can still be a complex thing that requires a whole lot of work to Get It Right^{TM}. A single e-mail can be a tree of various content components, including another complete embedded e-mail with its own tree of content, that should be handled in one way or another. Once that possibly complex e-mail has been parsed and its components stored somewhere, it will need to be displayed. What part to display and how? If the e-mail is HTML based, it needs to be sanitized before published before sending it off to the client, to avoid JavaScript attacks and whatnot. Putting hard restrictions on what people can send in (like simple text+image, nothing more) is not an option, as it will only cause problems when people are in distress.

The next time someone (like yourself) says "This should be easy enough", run! :)

Chapter 7

Acknowledgments

I would like to thank my wife for putting up with me during this time and also my supervisor for accepting how long it has taken to complete this. To the computer support people; sorry that this didn't turn out the way it was intended.

References

[1] Howard Stinger. A letter from howard stinger. http://blog.us.playstation.com/2011/05/05/a-letter-from-howard-stringer/.

[2] Craig Ruefenacht. Rust: Managing problem reports and to-do lists. In *Proceedings of the 10th USENIX conference on System administration*, LISA '96, pages 81–90, Berkeley, CA, USA, 1996. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=1029824.1029840.

[3] Best Practical Solutions LLC. Request tracker. http://www.bestpractical.com/rt/.

[4] Mozilla Project and individual contributions. Bugzilla. http://www.bugzilla.org.

[5] Atlassian. Jira. https://www.atlassian.com/software/jira.

[6] OTRS Group. Open-source ticket request system / open technology real services. https://www.otrs.com/?lang=en.

[7] SANS, MITRE, et al. CWE: Common Weakness Enumeration. http://cwe.mitre.org.

[8] CWE. Top 25 Most Dangerous Programming Errors. http://cwe.mitre.org/top25/index.html.

[9] SANS Institute. SANS Information, Network, Computer Security Training, Research, Resources. http://www.sans.org.

[10] The MITRE Corporation. MITRE–Applying Systems Engineering and Advanced Technology to Critical National Problems. http://www.mitre.org.

[11] OWASP: Open Web Application Security Project. https://www.owasp.org.

[12] OWASP. Top 10 Web Application Security Risks. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project.

[13] Initiated by Google. AddressSanitizer: a fast memory error detector. https://code.google.com/p/address-sanitizer/.

[14] Initiated by Google. MemorySanitizer. https://code.google.com/p/memory-sanitizer/.

[15] Initiated by Google. ThreadSanitizer, a data race detector. https://code.google.com/p/thread-sanitizer/.

[16] Kostya Serebryany. Poster from 2012 LLVM developers meeting: Sanitize your C++. http://llvm.org/devmtg/2012-11/Serebryany-ASAN-TSAN-Poster.pdf. Google.

[17] CERT. Secure coding. http://www.cert.org/secure-coding/.

[18] Robert C. Seacord. *Secure Coding in C and C++*. Addison Wesley Professional, 2005.

[19] The PHP group. PHP: Hypertext Processor. http://www.php.net.

[20] Alex Munroe. PHP: a fractal of bad design. http://me.veekun.com/blog/2012/04/09/php-a-fractal-of-bad-design/.

[21] SANS, MITRE, et al. CVE: Common Vulnerabilities and Exposures. http://cve.mitre.org.

[22] Randall Munroe. XKCD: Exploits of a Mom. http://xkcd.com/327/.

[23] Matt Mullenweg and Mike Little et al. WordPress: Blog Tool and Publishing Platform. http://wordpress.org.

[24] Michal Zalewski / Google. Google Browser Security Handbook. https://code.google.com/p/browsersec/wiki/Part1.

[25] European Union. Directive 2002/58/ec of the european parliament and of the council of 12 july 2002 concerning the processing of personal data and the protection of privacy in the electronic communications sector (directive on privacy and electronic communications). http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=CELEX:32002L0058:EN:NOT.

[26] Article 29 Data Protection Working Party. Opinion 04/2012 on cookie consent exemption. http://ec.europa.eu/justice/data-protection/article-29/documentation/opinion-recommendation/files/2012/wp194_en.pdf.

[27] Apache Software Foundation. Apache HTTP Server. http://httpd.apache.org.

[28] Open Market. FastCGI: A High-Performance Web Server Interface. http://www.fastcgi.com.

[29] Apache Software Foundation. Apache Software Foundation. http://www.apache.org.

[30] Apache Software Foundation. Apache Privilege Separation. http://wiki.apache.org/httpd/PrivilegeSeparation.

[31] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 305–316, New York, NY, USA, 2012. ACM. URL: http://doi.acm.org/10.1145/2382196.2382230.

[32] InfoGlue Community. InfoGlue. `http://infoglue.org`.

[33] Google. Google Webmaster Tools.
`https://www.google.com/webmasters/tools/`.

[34] Stefan Esser et al. Suhosin - Hardened PHP Project.
`http://www.hardened-php.net/suhosin/`.

[35] Novell. AppArmor Application Security for Linux.
`http://wiki.apparmor.net/index.php/Main_Page`.

[36] NSA. Security-Enhanced Linux.
`http://www.nsa.gov/research/selinux/index.shtml`.

[37] Brad Spengler. grsecurity. `http://grsecurity.net`.

[38] Sun/Oracle Inc. Solaris Trusted Extensions.
`http://docs.oracle.com/cd/E23824_01/html/821-1482/index.html`.

[39] OWASP. Owasp - webgoat.
`https://www.owasp.org/index.php/OWASP_WebGoat_Project`.

[40] Neil Daswani, Christoph Kern, and Anita Kesavan. *Foundations of Security: What Every Programmer Needs to Know.* apress, 2007.

All URLs were verified at 2013-12-02.