Process Innovations For Security Vulnerability Prevention In Open Source Web Applications

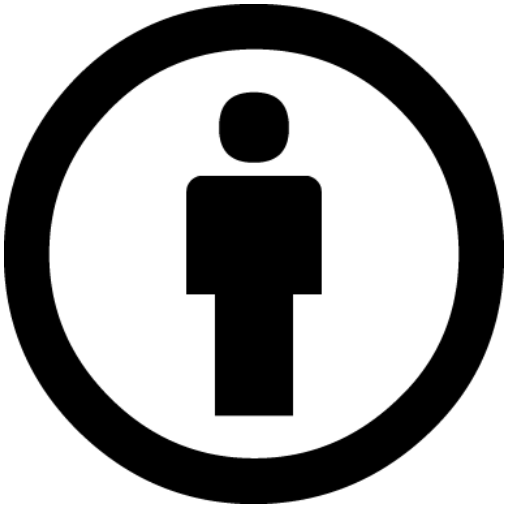
Diploma Thesis

Florian Thiel

April 30, 2009

Department of Mathematics and Computer Science
Institute for Computer Science
Software Engineering Working Group

Responsible University Professor: Prof. Dr. Lutz Prechelt
Supervisor: Dipl.-Medieninf. Martin Gruhn



Affirmation Of Independent Work

I hereby declare that I wrote this thesis myself without sources other than those indicated herein. All parts taken from published and unpublished scripts are indicated as such.

Berlin, April 30, 2009

Acknowledgments

First of all, I would like to thank Professor Lutz Prechelt who suggested and supported the topic of this thesis and enabled me to spend months of interesting research in the fascinating world of security research. I also owe a lot to Martin Gruhn, my supervisor, who provided lots of feedback and was especially helpful to flesh out the argumentation and create a good narrative.

This thesis would not have been possible without the support of my parents, who enabled me to do what I love in the first place, provided feedback and tons of emotional (and nutritional) support throughout the time of writing.

I am also very grateful to all of my friends who provided additional feedback, did proofreading, joined me in the library, had countless cups of coffee with me or took my mind off of work for some time. You know who you are!

I dedicate this thesis to my past and current flat-mates of WG Graf Lotte who are the best people you could wish to live with. Thanks for the great time we had so far!

Abstract

SQL Injection Attack (SQLIA) and Cross-Site Scripting (XSS) are abounding vulnerabilities in web applications. This work describes the technical background of both vulnerabilities and develops an annotation-based practice for the prevention of these vulnerabilities in open source web applications. Findings from the introduction of the practice into real-life applications are given and refinement recommendations for further research are provided. In addition, this thesis analyses nine open source web application projects. The analysis identifies social and technological factors which influence the applicability of the developed practice and the ability of the projects to innovate in general.

Contents

Chapter 1

Introduction

> Mr. McKitrick, after careful consideration I have come to the conclusion that your defense system sucks.
>
> General Beringer
> *WarGames*

Web applications are on the rise. Lightweight, ubiquitous alternatives to classical desktop applications become part of the work-flow. So-called social networks establish completely new forms of applications that are intrinsically based on communication. All these new applications run inside the web browser. The web browser becomes the execution environment, taking over the importance that was once reserved for the *operating system*.

> *Social Networks*
> Social networks are applications where the relationship between users plays a central role. All content is generated by users and the web application only provides the framework, The large proportion of user input makes these kinds of applications a prime target for attacks.

The web application paradigm not only changed the execution environment, it brought new technologies, languages, interaction models, and, most importantly, public access to applications[1]. Web applications make extensive use of client-side scripting languages like JavaScript and interactions rely on client/server communication. The web browser becomes the universal client to all web applications. Not one of these ingredients is new, but their combination exposes applications and their users to new threats.

1.1 New Old Threats

Public exposure of private data, data manipulation or erasure were serious threats in the age of desktop applications. But as data was stored locally at

[1] or at least their entry pages

private computers or in a company's data center which was not directly connected to the outside world, the cost of attacks was significant, and computers and companies had to be attacked individually.

The cost/benefit ratio turned in favor of the attacker. Data stored in web applications has a common access point (the web application interface) and a single attack can affect many users. Furthermore, with ubiquitous networking, identity data is much more valuable. Individuals as well as large companies carry out substantial financial transactions through web applications such as web shops or online banking portals. Rich personal profiles are now stored online and immensely valuable for advertising purposes. Impersonation is also a serious additional threat when more and more of personal life happens in the Internet. E-Government will take that even further.

The Open Web Application Security Project (OWASP), a project that collects and publishes information concerning web application security (see also A.5 on page 127), published a list of the top 10 web application vulnerabilities in 2007 [Wik09c]. XSS and SQLIA placed first and second, respectively, based on a vulnerability trend analysis by MITRE [CM07a]. The analysis also found that the total number of web application vulnerabilities surpassed the number of buffer overflow vulnerabilities, the previous most common vulnerability in a world of languages lacking automatic memory management. A proliferation of amateur software (due to easy access to programming environments that enable quick deployment of applications, even by inexperienced developers) and the ease of exploitation of web applications are the primary reasons for this trend, according to the authors [CM07b].

Security professional Jeremiah Grossman did an informal survey on money spent for security in professional software development. Application security came last, after network and host security [Gro09a]. Analysts from the Gartner group showed convincing evidence on why this is the case, notwithstanding developers knowing better: Network and host security measures provide quick one-stop solutions with measurable impact while application security needs costly and time-consuming training of developers, process changes etc.

It does not matter whether you look at professional or amateur web application development, application security and the respective development processes play a major role in the overall security of a deployed web application.

1.2 Open Source Software

Open source development describes a software development model where source code is licensed to be open to the public, freely modifiable and redistributable, as long as the derived software uses the same license. Many popular web application projects are developed as open source software.

The Open Source Initiative (OSI) is the authoritative source of the definition for open source software and certifies licenses for compatibility with their criteria. Their definition of open source includes additional criteria such as non-discrimination against technology and uses of the software or other software [Coa06], but the interesting property for the discussion in this thesis is the

accessibility of the source code and the transparency of development processes.

Unlike most closed source software[2], open source developers communicate via public mailing lists, use public web sites or wikis for documentation and coordination. Issue trackers and mailing lists are also mostly public. These properties make open source web applications a great target for research — not only on the development results (the source code and the application) — but also on the development process itself. This thesis therefore focuses on open source applications.

1.2.1 Security And Open Source Development

There has been much discussion about whether open source development provides better application security as the closed source model. On the one hand, the visibility of the source code allows more people to look at the source code and find vulnerabilities, on the other hand attackers don't have to rely on black-box testing for identifying flaws [Cow03]. Much of the discussion about security of open source software is based on anecdotal evidence and *common-sense* arguments ("with closed source software, customers are at the mercy of the software provider"; "in open source software development, nobody forces developers to do code reviews, so nobody does them", cp. [Oba02]) and there is not much hard evidence which model has the better security track record. In this thesis I will ignore this discussion and limit myself to open source projects purely due to their research-friendly properties.

1.3 On Security

This thesis is about *application security*. Application security, as opposed to network or host/system security deals with the security properties of an application itself, not of the surrounding elements (operating system, network, firewalls, etc.). A comprehensive approach to security would use security features of all the components making up the web application's environment (web server, database, operating system) but the discussion here is limited to issues directly related to the application. This does not imply that all vulnerability defenses should be in the application itself. Grossman makes a point saying that a Web Application Firewall (WAF) may be the more cost-efficient choice for businesses in some cases [Gro09a].

1.4 Goals Of This Thesis

The overarching goal of this thesis is to develop a process innovation to prevent security vulnerabilities in open source web applications. Part of my work consisted of discovering and formulating specific research questions and strategies. Dealing with the two most common vulnerabilities according to OWASP's top ten was a given to limit the scope of the research.

[2] throughout this thesis, I will use this term for all software that is not open source

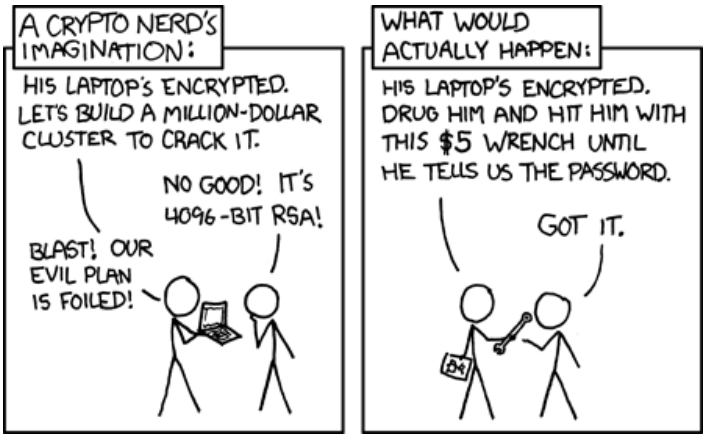


Figure 1.1: There are always multiple ways to attack a system
by xckd, http://xkcd.com/538/, used with permission

I started with research into the technical background of two vulnerabilities — called SQL Injection Attack (SQLIA) and Cross-Site Scripting (XSS) — and also web application security in general. The results can be found in chapter 2. After looking at the source code of the first web application project (WordPress, see 4.2.1 on page 55), I decided to design and evaluate an innovation that targets improving the architecture of an application. The main reasons for this direction of research was that it turned out that best practices and technology to prevent SQLIA and XSS are readily available but not used everywhere (my first look at WordPress also indicated as much).

The innovation developed (see chapter 3) was presented to two web application projects (see case studies, chapter 4 on page 53). It became clear that technological and process-related properties for security in the web applications under discussion were not yet sufficiently understood and needed more background research. That's why the scope of this thesis was extended to identify properties (called *concepts* in this thesis, see chapters 2 and 4 for details) of open source web application projects that influence the introduction of process innovations. Seven more web application projects were analyzed to gain insight about these *concepts*.

1.4.1 Research Questions

Three questions were devised after enough up-front research to clarify that the development process is the major factor in web security vulnerability prevention. They align with the goals stated above. I will come back to these questions in the respective parts of the thesis.

Q1 How do SQL Injection Attacks and Cross-Site Scripting work and how can they be prevented?

Q2 Is the innovation that is developed in this thesis useful to prevent the vulnerabilities under discussion?

Q3 Which properties of the projects determine the usefulness of the innovation?

1.4.2 Core Tasks – Structure Of The Thesis

The chapter structure of the thesis corresponds to the core tasks of the research.

1. **Threats and technical background** — In chapter 2 I explain the vulnerabilities under discussion in this thesis in depth. The discussion includes threats posed by the vulnerabilities, state of the art attacks and insight into the root nature of the vulnerabilities.

2. **Ideas for improvement** — The insight gained in chapter 2 led to ideas for vulnerability prevention techniques. I present the most promising idea in chapter 3. The chapter also contains the details on research methodology and candidate selection for the case studies in chapter 4.

3. **Case analysis** — Chapter 4 contains the actual analysis of the real-world web application projects. The documentation includes the validation of the process-based improvement idea presented in chapter 3 and the analysis of vulnerability prevention concepts of the source code and the development process. These concepts relate to those identified in chapter 2 and to concepts discovered while conducting the studies, also covered here.

4. **Conclusion and further research** — If you are only interested in the results of this thesis, skip to chapter 5. The chapter presents a summary of the results from the previous chapters, identifies questions left unanswered and newly opened questions. The chapter also presents qualifications for the validity and relevance of the research conducted herein.

The chapters are all loosely coupled, with the exception of chapter 4, concerning the case studies. That chapter heavily relies on the ideas introduced in chapter 3 which should be read before. Chapter 2 can be skipped if one is very familiar with SQLIA and XSS. Chapter 5 — containing the conclusion — can be read independently for a quick glance at the results but interesting facts will be lost when skipping all other chapters.

1.5 Classification Of This Thesis

From a research methodological standpoint, this thesis is situated in the field of qualitative case study research. Case studies were conducted for this thesis, leveraging documentation analysis (source code and project web sites), archival records (mailing lists and source code repositories), interviews (questionnaires with feedback) and participant-observation (project interaction and innovation introduction attempts)[3].

From a research topic standpoint, this thesis touches application security, code analysis, software architecture, software development processes and open source software development. The specific research topic is software vulnerability prevention in open source software. Since architectural improvements of

[3] terms after Yin [Yin08]

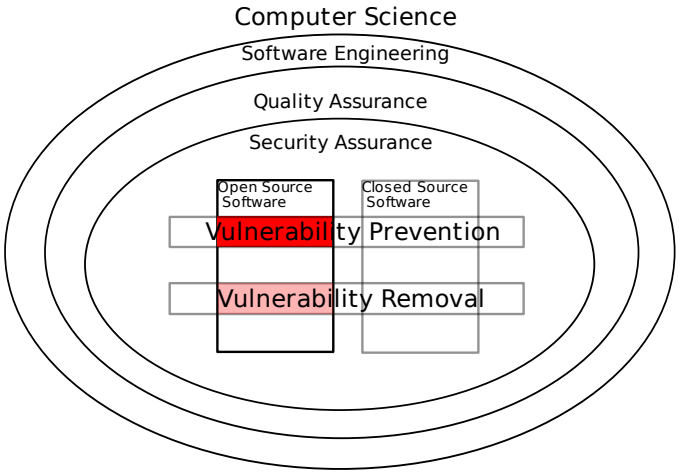


Figure 1.2: Classification of this thesis' topic in the scientific context

existing applications, which are part of this thesis, can also remove vulnerabilities, to a lesser degree this thesis also handles vulnerability removal. See figure 1.2 for a visualization for this thesis' research focus.

1.6 A Note On Sources

This thesis uses online sources as well as traditional reviewed ones. Section A.4 on page 125 in the appendix contains information about the reasons to include these sources and explanations of their use.

1.7 Definition Of Fundamental Terminology

1.7.1 Concepts

The *concept* is a fundamental term throughout this thesis. *Concept* is used literally to refer to the concept behind a *phenomenon*. It serves to tie phenomena in the projects under discussion to the same roots, the same *concepts*. Chapter 2 will identify technical concepts that are related to best practices and anti-patterns. These concepts were identified during the research on the security vulnerabilities considered on this study.

Chapter 4 introduces more concepts. These concepts were not known upfront but discovered during the innovation introductions or the project analysis. Discovered concepts are explained at the end of chapter 4 in section 4.5.

Definitions of technical concepts in chapter 2 are marked at the page margin with the name given to the respective concept. Occurrences of concepts in actual web application projects are marked at the page margin with the name of the concept and the page number of the page where the concept was originally defined.

These concepts play an important role in the analysis of the applicability of process innovations to open source web applications in chapters 4 and 5.

1.7.2 Security Term Definitions

Security has many aspects, not all of which can be separated intuitively. The following definitions are used throughout this thesis. Note that these definitions are not universal and you should not rely on them being well-defined (or being shared) in all literature.

Security — Security can mean a lot of different things in different contexts. Since it's most appropriate to the topic at hand, I will use the definition for *information security* found in Wikipedia[Wik09k]: Security means keeping information *confidential*, *integer*, and *available*. As you will see, all the *weaknesses* under discussion in this thesis can be classified according to these three threats.

Error — An error is an (intended or unintended) event during system construction that leads to a *defect*.

Defect — A defect is a structural deficiency (in relation to the requirements) in a system, caused by an *error*. Every defect is caused by an *error*.

Weakness/Vulnerability — A weakness or vulnerability is a *security*-related *defect*. I use both terms interchangeably in this thesis.

Attack — An attack is an attempt (by an *attacker*) to use a *vulnerability* to affect system behavior.

Exploit — An exploit is a successful (from the attacker's perspective) *attack* on a *vulnerability*. An exploit is also used for a piece of software that conducts a specific attack.

Risk — *Risk* is the product of the probability of an attack and the resulting loss, often quantified monetarily. *Risk* serves as a useful metric for the amount of protection against a certain threat should be implemented.

1.8 Secure Web Applications

Making secure software is challenging and there is a lot of literature [RvW03] [HL02] about how to architect or write software that is secure enough for its intended applications[4].

Securing web applications does not involve any fundamentally new techniques. Maliciously crafted user input has been the cause of buffer overflows, a major problem in languages without automatic memory management [RvW03,

[4]Security is not binary. There is no totally secure system. Protection against a certain threat is always associated with costs and a tradeoff has to be made between the costs of implementing the security measures and the risk (see also [Sch06])

p. 101]. For traditional networked applications like remote shells, buffer overflows (and therefore user input) were already important, but it took targeted attacks and inside knowledge to gather important data.

With web applications and worldwide access the attack surface increased. Application providers cannot control who enters data into their applications and basically have no control over the clients that send data. Furthermore, attack scenarios became much simpler. Unification of technology (there are only a couple of different web browsers or databases to attack) and unification of interaction structures (page logins work similarly everywhere) lead to an explosion of attack opportunities.

Remember that the fundamentals of application security did not change when web applications appeared. Things got more complicated since there are many more new ways of interacting with web application users and many new technologies involved. However, traditional security principles did not become obsolete through the introduction of web applications.

Chapter 2

The Weaknesses - Background, Threats, Prevention

> If you think technology can solve your security problems, then you don't understand the problems and you don't understand the technology.
>
> Bruce Schneier

This chapter covers the technical background of the two vulnerabilities discussed in this thesis, SQLIA and XSS. It discusses the fundamental problems posed by untrusted input in the world of web application and then presents the two vulnerabilities and their mitigations in detail. This chapter provides answers to the first research question, Q1: How do SQLIA and XSS work and how they can be prevented.

2.1 What Is User Input?

In web applications, all data not directly under the application's control is considered untrusted. It is critical that all untrusted input is sanitized appropriately. What is appropriate often depends on the context. The vulnerabilities under discussion in this thesis are a consequence of putting untrusted data where it could be harmful without prior sanitation.

2.1.1 Sources

There are different sources of untrusted input which have to considered when building web applications:

GET/POST Parameters — GET and POST are HyperText Transfer Protocol (HTTP) methods that take parameters to submit data to the web server[1].

[1] PUT requests are also part of HTTP but browsers usually do not support it; in principle, PUT should be handled similarly to GET and POST

This is the most direct form of getting data into the web application. GET/POST requests can also be submitted in an asynchronous manner using XMLHttpRequest, also known as Asynchronous JavaScript and XML (AJAX), a technology that allows network communication without a complete page reload in the browser.

Cookies Cookies are set by the web server and saved on the user's browser but are also transmitted back by the user's browser on each request and could have been manipulated in-between. Therefore, the data in cookies has to be treated as untrusted.

HTTP Headers — HTTP headers like `Referer`[2][sic!] are generated by the user's browser, too. Browsers usually don't provide a user interface to manipulate their values but they can be modified by browser plug-ins or while on the wire.

The Database — The database (or other means used for persistent storage) should not be considered a safe source. Other applications could be using the same database or data could have been written to the database without being sanitized (intentionally or unintentionally). This can also occur if sanitation behavior changes between versions of the same application.

SOAP Web Service Requests — SOAP [Wik09n] web service requests are semantically similar to GET/POST requests. They are not handled explicitly in this thesis but the same principles that should be used for data coming from GET/POST requests apply to web service requests.

In this thesis, user input refers to any of these sources. The exact submission formats and constraints may be different between the sources, but the general principle of untrusted input must be applied to data from any of these sources.

Other local sources of untrusted data like environment variables that could be controlled by local users on the web server's system are not inside the scope of this thesis.

2.2 SQL Injection

Web applications usually use relational databases[3] to store persistent data. This includes data concerned with the application domain (e.g. address datasets for a phone book application) as well as data concerned with the technical domain (like user accounts, passwords or authorization settings). The interface to the database is usually the Structured Query Language (SQL), a standardized language for querying and modifying data in relational databases. Although the

[2] contains the Uniform Resource Locator (URL) of the page that led the browser to the current page

[3] the injection problem is in no way limited to relational databases; the main cause, the dynamic creation of parameterized queries also exists for XML-based querying [Tea08] and basically for all cases of dynamic code generation that is dependent on data not under the system's control; all web applications discussed in this thesis use a relational database

SQL standard undergoes regular revisions and database vendors add proprietary extensions to their database products, the issues with SQLIA discussed here are fundamental to SQL and the usage patterns in web applications and not specific to any implementation.

2.2.1 Relational Databases

Relational databases consist of tables, which contain the actual data in rows. A table has a predefined set of columns, each of which have a specific type. Some databases have an abundance of types available[4] but most real use in applications is limited to 4 basic types: numerics, strings, dates and byte streams. Unfortunately, mapping database types to the types of a web application language is not trivial because there no 1:1 relation between the types. Listing 2.1 shows an example of an SQL table with types, listing 2.2 shows an example of actual data in table rows.

```
+---------------+---------------+------+-----+---------+
| Field         | Type          | Null | Key | Default |
+---------------+---------------+------+-----+---------+
| user_id       | int(11)       | NO   | PRI | NULL    |
| name          | varchar(255)  | YES  |     | NULL    |
| description   | text          | YES  |     | NULL    |
| login         | varchar(50)   | NO   |     | NULL    |
| psw           | varchar(50)   | NO   |     | NULL    |
| status        | varchar(5)    | NO   |     | NULL    |
| rights        | text          | NO   |     | NULL    |
| login_tries   | tinyint(4)    | YES  |     | 0       |
| createuser    | varchar(255)  | NO   |     | NULL    |
| updateuser    | varchar(255)  | NO   |     | NULL    |
| createdate    | int(11)       | NO   |     | NULL    |
| updatedate    | int(11)       | NO   |     | NULL    |
| lasttrydate   | int(11)       | YES  |     | 0       |
| session_id    | varchar(255)  | YES  |     | NULL    |
| cookiekey     | varchar(255)  | YES  |     | NULL    |
| revision      | int(11)       | YES  |     | NULL    |
+---------------+---------------+------+-----+---------+
```

Listing 2.1: structure of Redaxo4's rex_user table

```
mysql> select * from Users;
+------+-----------+----------+------+----------+
| uid  | firstname | lastname | age  | password |
+------+-----------+----------+------+----------+
| joe  | Joe       | Bloggs   |   47 | secret   |
| mary | Mary      | Smith    |   29 | password |
+------+-----------+----------+------+----------+
```

Listing 2.2: an example of a table's rows (MySQL)

[4] the popular open source database PostgreSQL even has types for geometry or computer network addressing [Gro09b]

A SQL statement consists of several *clauses* which define the action the statement executes. Clauses can select database columns or rows that are affected by the query or join tables together using relational algebra [Wik09m]. This introduction only includes the parts of SQL relevant to the discussion here. The complete SQL ISO/IEC standard (ISO/IEC 9075) is very detailed, consists of multiple parts and is not available free of charge[5].

```
SELECT firstname,lastname FROM Users WHERE uid='joe';
```

Listing 2.3: Simple SQL query to fetch user names from a Users table

```
SELECT Users.firstname, Users.lastname, Department.name
 FROM Users, Departments
 WHERE Users.deptid = Department.id;
```

Listing 2.4: A simple SQL join on two tables

Listing 2.3 gives an example of a very simple SQL SELECT query. The relevant part of the SQL for discussion in this thesis is the so-called Data Manipulation Language (DML), a subset of SQL containing only INSERT INTO (adding a new dataset), SELECT (querying existing data), UPDATE (updating existing datasets) and DELETE (deleting datasets) statements. Other statements (for creating or altering tables, creating indexes for faster access or managing databases) are equally vulnerable to SQLIA but usually not used with user input (since they are mostly relevant at deployment time) and their exploitation provides no further insight relevant to this thesis.

The example in listing 2.3 demonstrates column and table selectors and predicates. The result of the query is a list of the firstname and lastname columns of the Users table. The WHERE evaluates to true when the uid column is equal to joe, essentially limiting the rows returned by SELECT.

The query in listing 2.4 demonstrates how to join together data from multiple tables (database modeling uses *normalization* [Cod70] to eliminate redundancies, so information is usually distributed amongst multiple tables). Since column names might not be unique between tables, they are given in fully qualified notation. The predicate to WHERE now only evaluates to true when the value in the User's deptid column matches the id column's value in the Department table.

The DELETE clause (listing 2.5) only takes a database name and an optional predicate and removes all the rows where the predicate evaluates to true. Else, all rows are removed.

```
DELETE FROM Users WHERE uid='joe';
```

Listing 2.5: A simple DELETE statement with predicate

[5]International Electrotechnical Commission (IEC) web store: `http://webstore.iec.ch/webstore/webstore.nsf/artnum/041727`

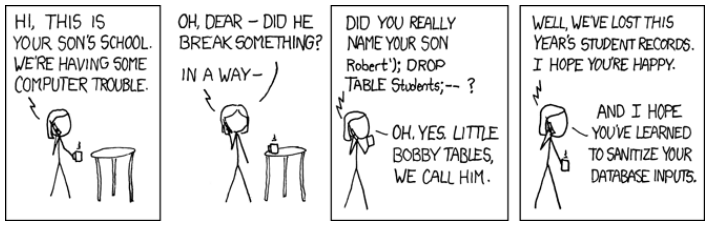


Figure 2.1: Simple SQL Injection by xckd, http://xkcd.com/327/, used with permission

The UPDATE clause (listing 2.6) takes the table to operate on, the columns to modify, the values to set and an optional predicate (again, without a predicate, the values are set for all rows).

```
UPDATE Users SET age=47 WHERE uid='joe';
```

Listing 2.6: Simple example for an UPDATE statement

Adding new rows is done using the INSERT INTO clause (listing 2.7) which also takes a table name, then a list of the columns to set and at last the values to set (in the same order as the columns).

```
INSERT INTO Users(uid,age,password) VALUES('joe',47,'foo');
```

Listing 2.7: Simple example for an INSERT INTO statement

All these examples are *static* queries. SQL injection can only happen when parts of the query can be manipulated by an attacker at runtime. In a realistic application, the uid joe will not be hard coded but come from external data, probably from values under the user's control (see section 2.1 on page 9). There is no special mark-up in SQL for query arguments (like arguments to a WHERE clause). This is not a problem of the language itself[6], but a problem of usage patterns in applications: SQL commands are often constructed using string concatenation in the calling application, making it impossible to separate (unsafe) user input and the fixed part of the statement (see examples in 2.2.6 on page 22).

2.2.2 A Simple Example

Since there is no differentiation between "argument" data and control data, passing crafted data can not only change the "arguments"[7] of the query but may lead to a completely different query. Consider the seminal[8] comic in figure 2.1:

```
uid, password = (<user input>)
```

[6] you could say the same thing about all languages that allow runtime evaluation of code

[7] they are not arguments from the standpoint of SQL; they are arguments because of the way the query is created

[8] by web standards

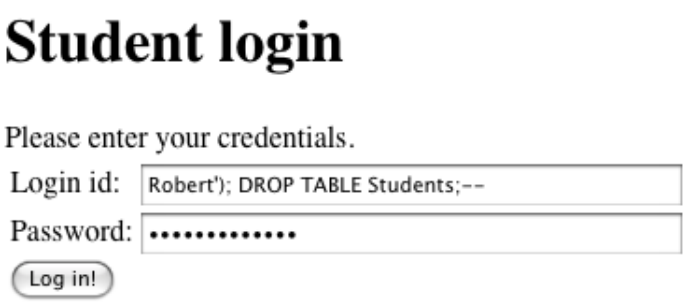


Figure 2.2: Example login screen

```
query = ''SELECT fullname FROM Students
  WHERE (uid='%s' AND password='%s');'' % (uid,password)
```

Listing 2.8: SQL Example with external data

Listing 2.8 on the preceding page shows what the code of the school's software might look like (the example is written in Python; see section A.3.1 on page 124 in the appendix for information on the use of Python in the examples in this thesis). The query should return the full name of the student if a user id with the given password exists and no results at all if `uid` and `password` don't match[9]. Assume the `uid` and `password` strings come from a web page form like the one shown in illustration 2.2.

If these values make their way into the `uid` and `password` variables without appropriate filtering or sanitation, the code shown in listing 2.8 is vulnerable to SQL injection. The resulting SQL query, given the input shown in illustration 2.2 would look like listing 2.9.

```
SELECT fullname FROM Students WHERE (uid='Robert');
DROP TABLE Students;-- '); AND password='somepassword';
```

Listing 2.9: SQL injection example

The carefully crafted parentheses and quotation marks terminate the first query and add another one, deleting the table with the student's credentials from the database. The attacker uses hyphens (-) to mark the remainder of the query string as a comment. This eliminates the rest of the query[10] and prevents further clauses not under the control of the attacker from interfering with his injection. Note that in this example, the attacker does not even need a valid student account since he was able to eliminate the password check. The attacker was able to have the database treat his input as control data because the query construction was not done carefully enough.

[9] This is a very poor design practice as it assumes unencrypted passwords are stored in the database. Nevertheless, the problems with this query would not disappear when encryption or password-checking outside the database would be used.

[10] technically, the rest of the line

2.2.3 Threats

The attack seen in section 2.2.2 targets data integrity. By deleting the `Students` table, which appears to be used for authentication purposes in the school's system, it is also a denial of service attack. With the table missing, no other students are able to log in, and – depending on the design of the school's system – it might stop working altogether.

Data manipulation and denial of service are popular examples but there are more threats due to SQL injections:

Accountability If an attacker is able to bypass the password check required for system login by exploiting an SQLIA vulnerability, the attacker can impersonate other users of the system. The other user cannot be held accountable for actions taken "in his name" and forensic analysis might be much more difficult.

```
``SELECT firstname, lastname, permissions
  FROM Users WHERE login='%s' AND
     password='%s';'' % (login, password)
```

Listing 2.10: SQL injection vulnerability, login circumvention

```
login = ``johnsmith''
password = ``1' OR 1=1;--''
SELECT firstname, lastname, permissions
  FROM Users WHERE login='johnsmith'
  AND password='1' OR 1=1;--'
```

Listing 2.11: SQL injection vulnerability, login circumvention example

The `OR 1=1` part makes the password condition a tautology, so the SQL statement always returns the user data for the user with the given `login`.

Confidentiality Confidentiality is violated when an attacker is able to modify a query in a way so that it includes data not normally visible with the attacker's privileges. Confidentiality evasion is commonly done using an `UNION SELECT`[11] attack [HVO06], an SQL statement which combines the result of multiple `SELECT` statements.

```
``SELECT firstname, lastname FROM Users
   WHERE department=%s '' % deptid
```

Listing 2.12: SQL injection vulnerability, access control circumvention

```
deptid = ``5 UNION SELECT Accounts.firstname, Accounts.
    identifier FROM Accounts;--''
SELECT firstname, lastname FROM Users
```

[11] `http://www.w3schools.com/sql/sql_union.asp`

```
WHERE department=5 UNION SELECT Accounts.firstname,
    Accounts.identifier FROM Accounts;--
```

Listing 2.13: SQL injection vulnerability, access control circumvention example

The maliciously constructed statement now contains not only the data from the `Users` table but also (probably sensitive) financial data from the `Accounts` table. The data from the Accounts table is appended to the original list of names, exposing the account identifiers.

Use as an attack vector Since some database engines support operating system interaction commands, an attacker might be able to inject SQL code to run system commands with the database's privileges. Microsoft's MS SQL Server provides support for shell command execution and even access to linked SQL servers (possibly putting servers that are normally protected from the outside world by a firewall at risk). Using SQL servers as an attack vector requires knowledge of the specific server in use since it depends on proprietary extension to the SQL standard. Security researcher Victor Chapela presented some examples, including uploading files from an SQL server to the attacker's machine through a reverse connection and network exploration at an OWASP conference in 2005 [Cha05].

2.2.4 SQL Injection Vulnerability Preventions

Injection Flaws

The main issue, allowing SQL injections in the first place, is the missing differentiation between data and control statements in the query construction [GE08] when using simple string concatenation. For string concatenation to be safe, the unsafe data to be used in the string has to be sanitized first, making it impossible for user input to be interpreted as control data.

```
``SELECT name FROM Users WHERE departmentid=%s AND
  lastname='%s' LIMIT %s'' % (deptid, lastname, limit)
```

Listing 2.14: Simple SELECT query with numeric and string parameter in selector

Let's look at a simple example that includes 3 values that get inserted into the query string (2.14). deptid and limit should be numeric, lastname is a string.

Before any validation or filtering takes place, data has to be canonicalized (see section 2.1 on page 9). Otherwise, all methods operate on data in an unknown format[12] and malicious data may pass undetected.

[12]note that data does not inherently carry information on how it should be interpreted; guessing the format by the presence of certain characters gives attackers the opportunity to decide how their input should be interpreted and thus increases the attack surface

Escaping

To separate *string* data and control statements, strings have to be placed in quotation marks. To prevent the quotation marks contained in the string itself from terminating the string early (and thus returning to control mode), they have to be *escaped*, usually using backslashes ('\'). Backslashes themselves also have to be escaped by prepending another backslash. escaping

Quoting and escaping can only be used for strings, not for numbers, table names, LIMITs or ordering directions (DESC/ASC). All the latter ones have to be validated correctly before insertion.

Data Validation

Validation against SQL injections is relatively straight-forward for non-string data. Numbers have to be made a proper numeric type (by casting, depending on the language), arguments to LIMIT have to be positive integers; column names have to refer to existing columns and the only valid directional arguments for row ordering (ORDER BY) are ASC and DESC. Having said that, doing correct validation requires knowing the type of the data one is dealing with ("is this an argument to WHERE or to LIMIT?"). Applying validations for the wrong type can have disastrous results. Unfortunately, automatic validation requires an explicit type modeling so that code (maybe a framework) can establish which validation to apply. validation

These rather simple rules make non-string types safe to use if they are consistently applied (this is a big *if*!). For strings, some pitfalls exist.

Multi-Byte Characters

Putting backslashes into strings sounds simple, and it usually is as long as one is dealing with character encodings that only use one byte per character, e.g. ASCII [Wik09g]. Multi-byte encodings use more than one byte of memory to represent a single character. They appear in character encoding schemes like the ones used by Unicode [Wik09o]. If the code that does the escaping is does not detect how the data is encoded, escaping may fail to prevent code injections. If there are valid characters in the encoding where the second (or any following) byte is a quotation or backslash character when interpreted as a single byte, code injection is possible.

Consider the following example, where the attacker is able to inject the string X' OR 1=1/* after the login= part.

```
SELECT * FROM Users WHERE login='X' OR 1=1/*' AND password='
    secret'
SELECT * FROM Users WHERE login='X\' OR 1=1/*' AND password='
    secret'
```

Listing 2.15: Example of defeating escaping using multi-byte characters

If the code doing the escaping and the database are aware of the use of multi-byte characters, the login column will be matched against the injected string,

which – most certainly – will not return any results. But if the web application is not aware of multi-byte characters but the database is, the escaping code will insert a \ in-between the two bytes of the multi-byte character, leading to X\'. If X\ is a valid multi-byte encoding of a character (which is true for the Asian encodings like GBK or SJIS [Han07]), the byte that appeared as a backslash becomes part of the multi-byte character and the single quote goes unescaped, terminating the string. The attacker successfully injects a tautology (which makes the WHERE clause return all rows of the table) and removes the password checking condition.[13].

For escaping to work properly, it is therefore important that the whole system (web application, database and all other components involved) is aware of the character encoding used. This issue is similar to canonicalization: a component always has to be aware of the format of the data it handles.

Early Escaping/Late Modification

early escaping

Escaping must be applied as late as possible. Modifying escaped data can undo the escaping or enable attacks impossible without escaping [GE08]. Consider the example in listing 2.16 that truncates the length of user names:

```
originalusername = "joeuser'"
escapedName = escape(username) # username == "joeuser\'"
injection = escapedName[:8]    # injection == "joeuser\"
"SELECT * FROM Users WHERE login='%s' AND password='%s'" %(
    injection, password)
```

Listing 2.16: Early escaping leads to vulnerability

Injecting an original username of joeuser' and a password of OR 1=1 -- leads to the following, incorrectly quoted SQL statement:

```
SELECT * FROM Users WHERE login='joeuser\' AND password=' OR
    1=1 --'
```

Listing 2.17: Truncated variable breaks SQL escaping

The trailing backspace in the username escapes the ending quote and so the login selector terminates at the (originally) opening quote of the password selector. The tautology selects all the rows and the opening comment renders the closing quote of the password selector meaningless.

This example should make clear why it is good practice to delay escaping until the actual construction of the final query string. There is another reason that concerns trace-ability: If escaping happens where the data gets passed to another execution environment (e.g. the database), it is immediately obvious to any reader of the code that escaping is taken care of and *how* escaping is being done. If escaping happened somewhere else, it would have to be codified and enforced throughout the project (e.g. by using a layered architecture, see the

[13]This example was adapted from a talk by PHP security expert Stefan Esser [GE08]

ultimate framework in section 2.6 on page 42) so that developers can rely on it (cp. barricade concept in [McC04, pp. 203]). Otherwise, a code reviewer would have to trace the code back to the place where escaping actually happened, which is cumbersome and error-prone.

Prepared Statements

prepared statements

Prepared statements were designed as a mechanism to improve the performance of database queries. Queries which are repeated many times using different parameters (e.g. different values in WHERE selectors) can be optimized and pre-compiled by the database. Prepared statements are usually part of the database Application Programming Interface (API) as they are closely tied to the database implementation.

Prepared queries became the weapon of choice for SQL injection vulnerability prevention as they effectively restore the separation between control statements and parameters. Prepared statements are always static, as pre-compilation only works for a fixed statement structure. The variable parts of the query are marked with place-holders, which are later bound to a specific value.

```
PreparedStatement updateSales = con.prepareStatement(
        "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ? ")
            ;
updateSales.setInt(1, 75);
updateSales.setString(2, "Colombian");
updateSales.executeUpdate():
```

Listing 2.18: Java JDBC style prepared statement

```
$sql = 'SELECT name, colour, calories FROM fruit
  WHERE calories < :calories AND colour = :colour';
$sth = $dbh->prepare($sql, array(PDO::ATTR_CURSOR => PDO::
    CURSOR_FWDONLY));
$sth->execute(array(':calories' => 150, ':colour' => 'red'));
```

Listing 2.19: PHP PDO prepared statement, named parameters

Listings 2.18 (for Java) and 2.19 (for PHP) show two examples of prepared statements. Note that the Java example uses types when binding the placeholders to actual values while the PHP version does not. When binding variable values, the developer should make sure that they conform to the correct type (e.g. by casting) in PHP. Failing to ensure correct typing does not directly affect SQL injection as the prepared statement handler always correctly escapes and quotes values for their specific type.

Limitations Of Prepared Statements

Prepared statements *can* solve the problem of character encoding mismatch between application and database (see section 2.2.4 on page 17). They are

usually part of a database abstraction library, which contains specific drivers for the various database backends the library supports. It is up to the driver to detect encodings and translate accordingly. Unfortunately, some database backend drivers (or for particular databases, some versions) do not support prepared statements at all and in this case the abstraction library falls back to emulation, without native access to database internals. So, in the end, users of database abstraction libraries still have to be aware of how prepared statements are implemented, in regard to the specific database in use. Switching databases without re-evaluating the use of prepared statements can make an application insecure.

Furthermore, as prepared statements were not designed as a tool to prevent SQLIA but to increase performance, they lack some essential properties to solve the SQLIA problem in its entirety:

1. Prepared statements only work for numbers and strings, not for table or column names, LIMITs, ordering (ORDER BY) or queries using IN. Additional validation still has to be used for these types.

   Furthermore, not all prepared statement implementations support all query types [Fis09]. If unsupported query types are needed, the developer is back to square one and has to manually implement strong escaping.

2. Prepared statement abstraction support is not available in older versions of of some languages. PHP only has an abstraction library to use database-side prepared statements since version 5.1 (released in 2005) [Fis09, Gro09e]. This prevents some projects from using the native support, sometimes implementing their own prepared statements (see the WordPress episode in section 4.2.1 on page 57).

   The Java database abstraction interface JDBC gained support for prepared statements in 2001 [Hei01]. Database abstraction implementations with support for prepared statements also exist for modern scripting languages like Ruby or Python and is part of most contemporary web development framework.

3. Prepared statements only work on static queries. If the number of parameters or the query structure depends on the parameters, prepared statements cannot be used. If there is just a limited set of different queries that can occur, it is possible to explicitly code all possible queries. The laws of combinatorics quickly make this approach cumbersome but being able to avoid dynamic queries is a major gain in the battle against SQLIA [FLS08].

4. Prepared statements are usually not type-safe (Java's JDBC is an exception, as are *Safe Query Objects* (see section 2.2.8 on page 26)). They can safely escape data into the place-holders but they can't guarantee that the id is actually a number. This has to be taken care of by validation or manual type-checking or casting.

Despite some shortcomings, prepared statements are the single most effective measure against SQLIA. If real prepared statement support is available

in the language libraries and the database and the use of dynamic queries can be avoided (or at least minimized) most SQL injection vulnerabilities can be prevented simply through the use of prepared statements.

Object-Relational Mapping / Model-Driven Design

For object-based languages, *Object-Relational Mapping (ORM)* techniques can make data access to a relational database mostly transparent to the application. ORM provides rules on how to persist domain objects into relational databases and how to re-retrieve them. The exact mapping is either done by convention (the ActiveRecord implementation of Ruby's Ruby on Rails web framework automatically maps class names to a table of the same name, with the class attributes as columns) or by explicit configuration (Hibernate, a popular Java ORM tool, can use eXtensible Markup Language (XML) configuration files[14] or Java annotations[15]). ORM

An ORM can take care of most of the queries which would have to be written manually otherwise. Since ORMs usually use prepared statements internally and can do some automatic validation, many opportunities for introducing vulnerabilities are eliminated. ORMs rely on an explicit data model because they have to derive queries and dependencies from the object graph.

In some cases, ORMs do not eliminate the need to write raw SQL queries entirely, but usually support an abstract query language that is simpler than plain SQL, portable between databases and not prone to injection attacks. This is possible because this internal language is much simpler than plain SQL and the ORM knows the semantics of the query, restoring the differentiation between control and data for the query construction.

ORM tools are available for all popular web application languages, but they are not ubiquitously used. Only one of the applications I discuss in the case studies uses an ORM. I explore the reasons for the limited use in chapter 4 on page 53.

In summary ORM is a very effective way to prevent SQLIA, because it minimizes hand-written SQL, takes care of sane escaping automatically and also provides some validation. For most ORMs, the advantages of prepared statements also apply because they are used internally.

The only scenario where ORMs cannot be easily used are applications without an explicit data model.

2.2.5 Bad Practice SQL Injection Mitigations

Web application projects use various, often home-grown, methods for SQL injection prevention (see the discussion of WordPress in section 4.2.1 on page 55). Some of them are widely known to be ineffective or even dangerous but still

[14] `http://www.hibernate.org/hib_docs/v3/reference/en/html/tutorial-firstapp.html`

[15] `http://www.hibernate.org/hib_docs/annotations/reference/en/html/entity.html#entity-mapping`

in wide use. I discuss some popular mitigations that do not keep up to the promise.

False Friends - Ineffective Mitigation Functions

inferior methods

PHP[16] contains a couple of functions that were designed as security devices but are severely flawed. Better alternatives are available, but old habits seem to die hard.

addslashes — addslashes() [Gro09c] escapes strings with backslashes for database queries. It is not aware of character encoding and other intricacies of the database used. Security researcher Chris Shifflet wrote a blog post on how to exploit an SQL query protected by addslashes() using multi-byte characters[Shi06]. Also see section 2.2.4 for details on the encoding problem. mysql_real_escape is available in PHP and does character set-aware escaping for MySQL. Note that addslashes() does not work for escaping URLs due to the differing encoding rules.

magic_quotes_gpc — magic_quotes_gpc() [Gro09d] is a configuration option for PHP that enables automatic escaping using addslashes() for all operations that use data from GET or POST requests and cookies. Since it uses addslashes() to do the escaping, the encoding problem mentioned above also applies to magic_quotes_gpc(). In fact, switching it on makes it impossible to prevent incorrect escaping of data acquired from GET, POST or cookies. According to Bernardo Damele, security researcher and author of the sqlmap injection automation tool [Dam], the magic_quotes_gpc() feature can even be used as an attack vector for SQLIA [Dam09].

magic_quotes_gpc() has been deprecated as of PHP 5.3 and will disappear in PHP 6 [Gro09d].

The methods mentioned above are still in popular use, especially in projects that still want to support PHP 4. In PHP 4, the data access abstraction PDO or the mysqli extension cannot be used, preventing the use of secure alternatives to the above methods.

2.2.6 Not That Easy – Real-World SQL Injection Walk-Through

The examples given in 2.2.3 on page 15 are idealized, naïve versions of SQL injection vulnerabilities which seldomly appear in real-world web applications. Exploiting a vulnerability usually involves a lot of up-front research and trial-and-error, and depends on the overall structure and the intricacies of the system. Usually, attackers don't have access to the source code, either, so they have to fall back to black-box testing strategies. Since this thesis deals with open source web applications and aims to provide insight into how to help these

[16] other languages may also have such "false friends" but the ones in PHP seem to be the best known

applications become more secure, I use all available means to gain knowledge about the project. This includes source code.

As an introduction, I discuss a very simple, dynamically generated SQL query used by Mambo, a PHP web content management system, and line out some ways that could be tried to exploit this query[17]. An in-depth discussion of Mambo, including the innovation introduction, can be found in the case studies chapter (section 4.2.2 on page 67).

```
1485 $query = "SELECT name FROM #__sections WHERE id='$section'";
1486 $database->setQuery( $query );
1487 $section_name = $database->loadResult();
```

Listing 2.20: Mambo: `administrator/components/com_categories/admin.categories.php`, (revision 125)

The example in listing 2.20, taken from a method called showCategories in the Mambo project code-base[18], shows a common usage pattern of SQL in that project. The query is constructed as a string, this string is then passed on to a small "database abstraction" layer (setQuery()), the query is executed on the database and the result is returned (loadResult()).

The variable $section is used to dynamically construct the query, making it a potential vector for an injection attack. As it turns out, the variable even is an *argument* to the method containing these three lines of code. The method does not have any kind of control on the contents of the variable.

Calling setQuery() on $database sets the query to be executed next. Since the method only has one parameter, which is a string, the setQuery() method can not have any semantic knowledge about the query. Best practice dictates that developers should use a prepared statement which keeps control statements and arguments separate.

The loadResult() method calls another method on the "abstraction layer" that actually executes the query. Mambo uses the mysql_query[19] PHP method, which calls the actual PHP database driver for MySQL. The mysql_query method only supports the execution of a *single* query. This prevents any kind of SQL injection attacks that involve piggy-backing attacks [HVO06], where an attacker is able to modify the query string in a way that results in it containing two statements. An example is shown in 2.21.

```
SELECT name FROM Users where uid='joe'; DROP TABLE Users;
```

Listing 2.21: Multiple SQL queries

Ensuring that only single queries are possible is a case of defensive design and a known good practice.

[17] I don't believe this query is exploitable under realistic conditions (I would have notified the developers), the discussion is merely a walk-through of the data flow in a web application using SQL.

[18] see appendix A.1.2 on page 114 for information about how source code was acquired and how to acess the exact file versions referenced in this thesis

[19] `http://de.php.net/mysql_query`

Since there is no way to manipulate stored data through SQL with a SELECT query[20], attacks targeting the integrity of the database cannot be mounted using this query.

The result of the database query is returned to the caller by the loadResult() method. This method only returns the first column of the first row of the returned dataset[21]. This also limits the applicability of injection attacks targeted on gaining privileged information. Even if an attacker is able to inject SQL code into the query, the application code will only allow him to access the first column of the first row. If, in addition, he is able to add columns to the query, he would not be able to reorder columns, effectively rendering the attacks useless.

How It Got Here – Input Validation

Having covered output filtering, this section describes how input validation and filtering happens in the Mambo example. Note that the result of input processing defines the state of all data inside the application (see section 2.1 on page 9 for a more detailed description).

The $section variable got passed as an argument to the method showCategories() and gets passed on to the database without any manipulation or validation. This seems to be safe here[22] but it is impossible to conclude safety just from this piece of code. Someone doing code review would have to have knowledge about where sanitation happens in the system and then conclude if all the data entering the showCategories() has undergone sanitation for output to the database. This is very hard to decide with certainty unless the application has a defined and enforced component structure with interfaces. It becomes clear in the next paragraph that $section has been filtered before but this is not explicitly codified and can only be verified by looking at the data flow manually.

The $section variable is called from `administrator/components/com_categories/admin.categories.php` upon initialization of that file. The variable comes from an HTTP GET or POST request (see 2.1.1 on page 9). In Mambo, a sanitation method called mosGetParam() is used on data supplied through cookies or GET/POST requests. This method is the generic input filtering method of Mambo. Developers have to explicitly call this function to retrieve parameters from requests. Otherwise, no input validation/sanitation takes place.

By default, mosGetParam() removes leading or trailing whitespace, encodes Hypertext Markup Language (HTML) (see 2.3.3 on page 32), and converts strings with a numeric value to proper integer values. This is (as indicated by the in-line documentation for the method) done for security reasons and contains a mixed bag of countermeasures:

1. Removing whitespace does not have any clear security implications. This seems to be done for formatting reasons only.

[20] the only useful queries would involve a UNION SELECT or a subselect, neither of which can manipulate data

[21] the dataset only has one column since the SQL query string only selected the `name` column; since the WHERE clause contains the section id which is presumably unique, the query should also only return one row

[22] which only means that I can not see a way in which it could be exploitable

2. Encoding HTML prevents the input from containing HTML or JavaScript-code and is generally used to prevent XSS vulnerabilities. In Mambo, this happens upon data entering the system. This is an instance of the early escaping concept.

3. Conversion to proper numeric types is a defensive practice against SQL injections, ensuring partial type safety. mosGetParam() takes an argument that defines the default value that is returned when the requested value is not present. It also determines if the returned value has to be a number. If the default value is numeric, the requested value is converted to a number, too. This is an effective prevention for injection attacks on numeric fields (which are not delimited by parentheses).

The Mambo example looks safe but highlights some architectural weaknesses that make code review – especially for security – particularly hard. We will see more of these issues in the case study in chapter 4.

2.2.7 Advanced SQL Injection

The cases discussed until now are plain SQL injections to manipulate queries. For the sake of completeness, the use of SQL injections for information gathering and the technique of blind SQL injections (which is related) are explained. The possibility of blind SQL injections should be an even stronger motivator to implement strong injection prevention measures.

Information Gathering

Victor Chapela of the OWASP explains how to use crafted SQL queries to gather information about the software in use, its versions, configuration and the structure of queries [Cha05]. Knowing the type of database — for instance — lets the attacker focus on attacks feasible with that specific database. This is often useful for advanced features like calling external programs or establishing network connections (see also section 2.2.3 on page 15).

Detection of database type and version is possible through knowledge of behavior specific to database types or versions. Databases adhering to the SQL standard support different extraneous features whose presence can be checked by provoking errors that are caused by missing features. Query responses also vary slightly between databases but still stay inside the limits given by the SQL standard. These differences can be probed for and there is no obvious way to prevent this. Bernardo Damele created an automatic scanner called *sqlmap* that uses these signatures to identify databases and configuration [Dam09].

Blind SQL Injection

Maor and Shulmann [MS08] offer insight into techniques to find SQL injection vulnerabilities in the absence of error messages and database signatures. The strategies they apply use structured exploration of the context where injectable data would be included and moving forward from there to get quoting right.

They propose a way to figure out the number of columns and their respective types without any feedback from the database. Knowledge about the table structure is important for all targeted real-world attacks.

2.2.8 More SQL Injection Prevention Techniques

There are some advanced SQL injection prevention techniques which can provide a very high level of protection but require a lot of effort or restructuring of the application. I have not seen any of these in the projects I looked at in chapter 4 on page 53. I present them here anyway to further show that very strong prevention methods exist and that missing technological support is not the reason for the abundance of SQL injection vulnerabilities.

stored procedure

Stored Procedures *Stored Procedures* are predefined queries, stored in the database. They look like function definitions but use the native SQL dialect of the database. The procedures can be parameterized and executed, similar to prepared statements. Type checking for the functions is available in most implementations, so stored procedures can provide a high level of protection. Nevertheless stored procedures can only be used together with correct escaping techniques since malicious input may modify the procedure and provide an injection vector.

```
Create procedure user_login @username varchar(20), @passwd
    varchar(20) As
  Declare @sqlstring varchar(250)
  Set @sqlstring = 'Select 1 from users
   Where username = ' + @username + ' and passwd = ' +
      @passwd
  exec(@sqlstring)
Go
```

Listing 2.22: Stored procedure definition in MS SQL

The example shown in listing 2.22[23] shows a stored procedure definition in MS SQL Server. username and password are vulnerable to injection if escaping is not taken care of outside the procedure [ABC+08].

Stored procedures also share the problem of dynamically generated queries with prepared statements. If it is necessary to support dynamic queries, all bets are off and all precautions that apply to using simple string concatenation to construct queries have to be done manually (see section 2.2.4 on page 19).

Stored procedures are usually not portable between database backends making the database layer harder to maintain if support for different databases is required. Stored procedures can be useful if they cover all database operations and are not modifiable at runtime. If they are combined with sensible escaping they provide a high level of security.

typed queries

Strongly typed queries Cook and Rai [CR05] describe a method called *Safe*

[23] taken from the OWASP Testing Guide [ABC+08]

Query Objects which is a kind of "advanced prepared statement" for Java that also bridges the type gap between the application's type system and the database's. Since SQL queries are evaluated at runtime, the application cannot know which type the database expects for each place-holder. *Safe Query Objects* move all the type-checking to the Java language, effectively preventing all attacks enabled by type mismatch.

The *Safe Query Objects* method is very powerful but requires fundamental changes to the way queries are constructed, is limited to Java and only provides real advantage if the query structure is static.

In Java, the prepared statement implementation of JDBC (Java's database abstraction library) also use Java's type system when inserting user data into the query, providing part of the value of *Safe Query Objects*[24]

2.2.9 SQL Injection Vulnerability Detection Techniques

These techniques are alternatives to code review for detecting SQL vulnerabilities. They are helpful for benchmarking but should only be used to augment safe development practices[25].

Black-Box Checking — Black-box checking puts input that could be useful for an injection attack into all publicly accessible form fields, cookies, etc. It is very useful as a quick check for vulnerabilities as it can be automated and tools are readily available. OWASP hosts the SQLiX project [Wik08] which provides automated testing tools for SQL injection vulnerabilities.

Taint-Based Flow Analysis — As one of the root problems of SQLIA is missing validation of user input, tracing the flow of user input through the system is a reasonable approach to prevent unsanitized input from reaching the database. Taint-based approaches mark (*taint*) data that originates in sources external to the application and only remove the taint marker when the data is sanitized. This happens either implicitly when the string is modified or explicitly when the developer *untaints* the data.

Taint analysis has a performance penalty and does not guarantee that the right sanitations are applied[26], but it is a powerful method to find missing sanitation or validation. Perl introduced a taint mode long ago when web applications were still simple CGI scripts [chr04]. The Ruby language core also has support for tainting [TFH04, 397].

2.2.10 Summary

Although the field of SQL injection and injection prevention is rich with tools and techniques of varying quality, three essential and very effective methods

[24] this does not make JDBC understand the query, it just catches type errors because the developer is forced to explicitly give a type when inserting data into the query

[25] as the famous saying, attributed to Edsger W. Dijkstra, goes: "Testing shows the presence, not the absence of bugs"

[26] it only guarantees that someone decided "it's safe now" before

stand out: input validation plus prepared statements or ORM provide very effective injection prevention, if the overall system configuration is sound (permissions, character encodings, etc.).

All three of these prevention methods are much easier to implement when all data types that occur in the system are well defined. That means that allowed characters and other constraints for every type are either centrally documented and enforced by developers or even enforced by a framework that automatically does validation and filtering.

Furthermore, the consistency of the use of the prevention methods benefits from defined layers for input validation and database abstraction, possibly using ORM. With a defined and uniform location, it is always clear which measures data has already passed through and thus which sanitations have been applied.

This uniform location can be codified in developer documentation and be baked into the architecture of the application, forcing the use of the security features. The section on the *ultimate framework* (page 42) shows how such a framework-based approach could look like.

The technical means to counter SQL injection vulnerabilities are available but architecture plays an important role to be able to consistently use them. The case studies in chapter 4 shows how open source web application projects actually protect themselves from SQL injections.

2.3 Cross-Site Scripting

Modern web browsers do not merely display static information, but act as an execution environment for *web applications*, complete with event handling and one or more client-side languages. Today's applications use user-supplied data to generate the web pages. Besides HTML code for the page's structure, pages include code for the client-side languages, which a user's web browser will happily execute. Data delivered to a web browser not only control the page's layout, but also its behavior.

Since modern web browsers are supposed to run the *applications* delivered and under the control of a web server, trust between web server and the user's browser is crucial. Client-side languages therefore usually run in a sand-boxed environment and only have access to functions related to the site the user is visiting in the web browser. The *same-origin* policy [Zal09a] dictates that browsers only execute server-supplied code for sites originating from the same server. Unfortunately, in the presence of injection vulnerabilities, browsers can be tricked into executing code that is not under the control of the site that delivered it but under the control of an attacker.

In this section, I discuss the technical background, threats and mitigation practices for client-side script injections known as XSS. I also take a look at popular but ineffective mitigation practices.

The discussion in this thesis is limited to JavaScript-based XSS. XSS affects all client-side languages that execute server-provided code, but the principles for script injection prevention will be similar.

2.3.1 Threats

Cookie Theft

The single most popular threat posed by XSS is *cookie theft*. The HTTP protocol, which is used to transfer web pages, is state-less. To be able to handle transactions that require multiple page requests (such as shopping carts in a web shop), the requests have to be connected to a single user identity. This is often done using cookies. Cookies are small strings which are saved to the user's browser and then transmitted back on all requests to the site that provided the cookie. Setting a cookie that identifies the user after the user has logged in to a site provides the site with the user's identity on all subsequent requests.

If an attacker is able to get a hold of such an identifier stored in a cookie, he is able to impersonate the person that the identifier was generated for. This enables the attacker to read the victim's emails and contacts in a web-based mail application, use the victim's instant messaging service, post to the victim's weblog or transfer money from the victim's bank account, using an online banking site[27].

[27] online banking applications usually require another authentication factor for money transfers; at least for German banks, being logged in usually does not allow money transfers without further authentication using e.g. transaction numbers

2.3.2 Other XSS Threats

Since XSS enables (in principle) arbitrary rewriting of web pages, the only limiting factor to XSS attacks is the sand-box that contains the script (see section 2.3.5 on the browser security model). Popular XSS attacks include web site defacement[28] or phishing[29].

The popular MySpace[30] social network became prey to a famous XSS attack that turned out to be the first XSS worm. A user called *Samy* planted JavaScript code in his public MySpace profile that replicated into the profile of everyone who visited *Samy*'s MySpace page. Furthermore, the code added *Samy* and other random MySpace members as the victim's friends. The code replicated very quickly and initiated so many friend requests that MySpace could not handle the load and had to be taken offline for a short time [Moo05]. A dissection of the worm's internals and the reasons why MySpace was susceptible to the XSS attack is available from Google's Evan Martin [Mar08].

Lam et al. identified threats posed by the possibility to misuse unexpecting user's web browsers to conduct denial-of-service attacks driven by JavaScript code [LAAA06]. Finding injection vulnerabilities in popular web sites could open up vast numbers of participating "attacking" browsers.

A recent addition to the XSS threat zoo is *clickjacking*. An attacker can modify a web page so that it invisibly includes parts of another page. A user clicking an innocuous-looking button may in fact interact with a different button on another page and (considering he is logged in to that other page) initiate an action on the other page on the attacker's behalf. This could involve bidding for an item in an ebay[31] auction. The issue was about to be made public at the OWASP NYC conference in 2008, but Grossman and Hansen (who had discovered the issue) canceled the talk after they discovered a proof of concept on how to use the attack to hijack microphones and web cameras for use as a surveillance device. The publication was delayed to give Adobe, maker of the popular Flash[32] plug-in for web browsers, time to update the plug-in, as it played a critical role in the proof of concept attack [Han08]. Flash provides an additional execution environment for server-provided code running inside web browsers, which is also able to access computer hardware.

```
<html xmlns="http://www.w3.org/1999/xhtml"
      version="XHTML 1.2" xml:lang="en">
  <head>
    <title>Demo Page</title>
  </head>
  <body>
    <h1>This is a demo page</h1>
```

[28] modifying the layout or contents of a web page in order to humiliate the owner of the page or to disturb his business

[29] tricking users to enter credentials (e.g. for banking applications) by creating a web site that poses as the *real* page but logs the user's credentials; XSS makes it possible to modify the original page, enabling attacks where server certificates are still valid

[30] `http://www.myspace.com`

[31] `http://www.ebay.com`

[32] `http://www.adobe.com/products/flashplayer/`

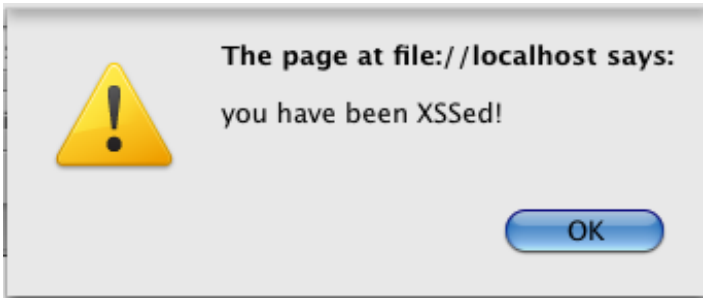


Figure 2.3: Simple browser pop-up created by JavaScript

```
  </body>
</html>
```

Listing 2.23: Simple XHTML page

2.3.3 Introducing JavaScript

HTML[33] is the most popular mark-up language for the web. It describes the structure of a web page and its elements. Every element is written as a *tag* (e.g. `<table>`) with optional arguments (e.g. `<img src=''file.jpg''/>`).

The source code of a very basic HTML page looks like the example given in listing 2.23 on the preceding page. The parts inside the `<head>` tag contain meta information about the page, the parts inside the `<body>` tags contain the actual (visible) part of the page.

All modern browsers have built-in support for JavaScript[34], a dynamic, weakly typed, prototype-based language with first-class functions. It is the most popular language for client-side scripting of web pages. Uses include the animation of objects, drop-down menus or graphics, support for drag-and-drop of objects on web pages and support for partial reloading of web pages through XMLHttpRequest, creating a more desktop-like user experience. It should be obvious that having an attacker control the behavior of a web application can be dangerous.

Types Of JavaScript

JavaScript is executed by the browser's JavaScript engine if it is encountered in a web page. If an attacker is able to inject code into a web page he usually tries to inject JavaScript, because having control over the behavior of the web page is much more valuable than just manipulating layout (which would be possible by just injecting HTML). The pop-up window shown in illustration 2.3 was created using the JavaScript code from listing 2.24 on the following page.

[33] I use HTML and its XML counterpart Extensible Hypertext Markup Language (XHTML) interchangeably in this thesis, except where the differences are important for the discussion

[34] I use JavaScript throughout this thesis to refer to any languages compatible to ECMA-262 ECMAScript [Wik09j]. The existing differences between the dialects are not relevant to the fundamental issues with XSS discussed in this thesis.

```
<html>
  <body>
    <form>
      <script>alert('you have been XSSed!')</script>
      <h1>Student login</h1>
      Enter your login name:
      <input type="text" size="40">
      <input type="submit" value="Log in!">
    </form>
  </body>
</html>
```

Listing 2.24: Trivial example of JavaScript in a web page

Encoding For JavaScript

escaping

Escaping is used to strip syntax-significant (in terms of HTML) characters of their special meaning. In HTML, *character encodings* [DYI+05] are used, which look like this: `<` or `&#60;`. Both examples encode a less-than ('<') sign. The first one uses a symbolic name, the second one the Unicode enumeration. Both prevent the character from being parsed as a mark-up delimiter.

What is interpreted as JavaScript depends on the context in which it appears. Therefore, different injection preventions have to be chosen, depending on the context. As not all contexts have reliable prevention patterns, the OWASP XSS prevention cheat sheet [Wik09e] suggests not using dynamic values in some contexts. These are

1. script tags: `<script>$input</script>`
2. an HTML comment: `<!-- $input -->`
3. an HTML attribute name: `<p $input="foo">`
4. an HTML element name: `<$input foo="3">`

The other contexts each need different encoding strategies but can be safely used.

Encoding for HTML element contents If the user input is to be displayed inside an HTML element (e.g. inside a `<p>` paragraph), OWASP's XSS prevention cheat sheet suggests the characters in table 2.1 on the next page be HTML encoded in order to prevent attackers from being able to switch in or out of a script execution context: This case is straightforward. A code example that would need this type of escaping looks like listing 2.25, where `name` is the variable that needs escaping:

```
print "<div>" + name + "</div>"
```

Listing 2.25: Escaping needed for HTML element

character	encoding
&	&amp;
<	&lt;
>	&gt;
"	&quot;
’	&#x27;
/	&#x2F;

Table 2.1: Character encoding for XSS prevention in HTML element contents

Escaping for HTML tag attributes In `<font color=’’#00ff00’’>`, `color` is an attribute to the `<font>` tag. If the contents of a tag attribute is dynamically generated from user input, care has to be taken that breaking out of the parentheses is not possible.

```
<form type="POST">
...
<button color=%s>
</form>
```

Listing 2.26: tag attribute injection example

The `color` attribute value of the button comes from a string possibly under an attacker's control. Since it is not quoted, an exploit would be possible if the attacker manages to inject `#00ff00 onload=very_evil_method()`. The resulting HTML would then look like 2.27.

```
<form type="POST">
...
<button color=#00ff00 onload=very_evil_method()>
</form>
```

Listing 2.27: tag attribute injection exploit

In well-formed XHTML, all attribute values have to be quoted, but some browsers interpret unquoted attributes anyway. If HTML attributes are not quoted, JavaScript injection is very difficult to prevent. It is therefore good practice that attributes are always quoted. Quoted attributes can only be broken out of using the corresponding quote (either ’ or ‘‘).

OWASP's cheat sheet suggests a very strict white-listing approach to also protect attributes that are not quoted. Complete protection can only be achieved if all characters, besides alphanumerics with American Standard Code for Information Interchange (ASCII) values below 256, are HTML encoded. Alpha-numeric characters never have any syntactically delimiting function in HTML and are therefore safe.

Note that the injection in listing 2.27 does not rely on quotes. With little modification, the injected string could further reduce the number of non alpha-numerical characters, avoiding even more black-list filtering approaches.

The use of the `onload` event handler attribute causes the browser to execute the attribute value as JavaScript when the page is loaded. No user interaction is necessary.

The OWASP cheat sheet lists three more different injection contexts (as JavaScript data, in Cascading Style Sheet (CSS) properties and in URL attributes). OWASP suggests to encode all non-alphanumeric characters, as for HTML attributes in general.

For the different contexts, only certain parts can be generated dynamically while retaining injection protection. For URLs, the access protocol (e.g. `http://...`) should always be fixed, as `javascript:` is also a valid protocol, but executes the rest of the attribute as JavaScript. For insertion into a JavaScript context, only literals (e.g. variable values) may be inserted and have to be correctly escaped. Otherwise, it is possible for the attacker to disrupt the control flow. The examples from the OWASP page[35] in listing 2.28 illustrate the correct use of dynamic data in HTML pages.

```
<script>alert('...ESCAPE UNTRUSTED DATA BEFORE PUTTING
      HERE...')</script>

<span style=property : ...ESCAPE UNTRUSTED DATA BEFORE PUTTING
      HERE...;>text</style>

<a href=http://...ESCAPE UNTRUSTED DATA BEFORE PUTTING
      HERE...>link</a >
```

Listing 2.28: JavaScript data, CSS property in attributes and URL in attributes

The basic problem of all these encoding approaches is that it completely prevents the use of user-provided HTML. Web site providers might want to allow users to use mark-up to structure contents or use images. Web logs, auctioning platforms and web forums often allow custom mark-up. There are three ways to allow mark-up without compromising robustness against XSS vulnerabilities:

1. Use a limited non-HTML mark-up language. Textile[36] and Markdown[37] are two simple mark-up languages that provide a subset of HTML's functionality. Scripting is not included. The mark-up is translated into proper HTML by the web server upon delivery to the web browser. The reduced mark-up languages themselves do not use characters that have special meaning in HTML (like `<,>` or `&`) and thus all characters can be HTML encoded, while still retaining mark-up capabilities for user input.

   Security of this method relies on the security of the transformation process to HTML, but as the transformation code can be used as a component, updating the component suffices to stay secure if vulnerabilities are found.

[35] `http://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet#Untrusted_Data`

[36] `http://www.textism.com/tools/textile/`

[37] `http://daringfireball.net/projects/markdown/`

These mark-up languages still depend on proper validation and filtering of input values, since e.g. HTML links are usually supported by the mark-up. The use of the JavaScript protocol in an URL would still introduce a CSS vulnerability (also see section 2.3.3 on page 32).

2. Use a parsing filter like AntiSAMY [Wik09b]. AntiSAMY is a project hosted at OWASP which provides sophisticated filtering for HTML. It uses a policy definition file to determine which tags and attributes are allowed in a piece of HTML and strips the rest. It internally builds and validates the HTML tree to overcome injections using broken HTML (which some browsers try to repair). AntiSAMY is available for Java and .NET.

   AntiSAMY provides a high level of security against XSS attacks but requires profiles that define what HTML elements and attributes are allowed in which input.

   HTMLPurifier[38] is an alternative to AntiSAMY for PHP. It relies exclusively on an audited white-list and also enforces well-formed HTML. It does not support a sophisticated rule model like AntiSAMY but requires less configuration.

3. Use a context-aware HTML templating language. A recent addition to the pool of effective XSS mitigations is Google's Automatic Context-Aware Escaping, called Auto-Escape [Bou09]. It is an HTML template language which supports insertion of variable contents. It releases the template programmer from applying the right encoding depending on the context (see section 2.3.3, *Encoding for JavaScript*), thereby enabling fully automated XSS prevention.

2.3.4 Types Of XSS

XSS attacks are classified into three categories, depending on how they can be exploited [HVO06, Wik09h]:

DOM-based (type 0) local attacks that appear within a page that itself generates HTML via JavaScript without any server interaction; I do not discuss this case in this thesis because it is rare; most web applications do round-trip communication to the web server

non-persistent/reflected (type 1) occurs when malicious input is directly used to generate output to a page which then contains the attack code; the attack is usually targeted to a single user

persistent (type 2) the attack code is not directly returned to a user's web browser but written to a persistent store (e.g. a database); the actual attack happens when the data is used to generate a page; this kind of attack has much higher reach and is harder to detect (since the time of the exploitation and the time of the actual attack differ)

[38] `http://htmlpurifier.org`

Persistent (type 2) exploits are one of the main reasons why sanitation of HTML has to take place when the web page is generated. Relying only on input validation/filtering is not enough, since it is often not clear what to validate/filter for (cp. section 2.2.6 on page 24 on user input). Even if current prevention mechanisms do not allow malicious HTML to be stored in the database, older versions of the software might have been vulnerable and the database may already contain data unsafe for output without correct escaping[39].

2.3.5 Browser Security Concepts

Since JavaScript is a powerful language and running network-provided code on user's machines is inherently risky, web browsers provide different mechanisms to limit the permissions code running in the browser has. First of all, all JavaScript code is sand-boxed inside the browser, so there is no way to access the outside system. Access inside the browser is restricted to some browser APIs and the Document Object Model (DOM). The DOM is a tree-structured representation of a web page including all of its properties. HTML elements can be accessed and modified programmatically, cookies can be read and windows can be resized and navigated (e.g. to a new URL).

Access inside the browser is also restricted. Browsers can display different pages at the same time, belonging to different sessions. For example, scripts running on a yellow pages web site must not interfere with an online banking site. Without any access controls, the pages could manipulate each other's content and functionality at will and retrieve private information.

Access control between web pages is provided by the *same-origin policy*. For DOM access, the access control mechanism works like this: Every page (called a *document*) has a `domain` property. Another document can access the complete DOM of that page if the protocol, hostname and port number of both domain properties match. This rule models a trust relationship between all the contents hosted under a single hostname. If two pages want to collaborate but are not hosted under the same hostname, both can reset the domain property to a *right-hand, fully-qualified fragment* [Zal09a] of their current domain. This means that `hosta.example.com` and `hostb.example.com` can both set their domain properties to `example.com` but not to `somethingelse.com`[40].

Michal Zalewski explains some serious shortcomings of the same-origin policy for the DOM in [Zal09a]. According to Zalewski, the basic problem is that the same-origin policy provides no security framework for real cross-site interaction. Cross-site interaction is desirable for sites collaborating and re-using other site's services, but the current browser security features do not provide a secure way to configure trusted collaboration among sites.

The same-origin policy is only partly relevant to the discussion of XSS in this thesis, as collaborating sites should still be treated as sources of potentially malicious data and proper validation and sanitation is needed.

[39] for cleaning databases of XSS code see section 2.3.6 on page 38

[40] Setting the domain to a top-level domain is also not allowed to prevent too permissive settings but lists of top-level domains are not always up-to-date in web browsers

Many current attacks inject data directly into a single page and do not rely on the interaction of different sites. In this case, the same-origin policy does not prevent the code from being run. It appears to come from the same server (which it in fact does as it is included in a legitimate page) as the rest of the page and is thus safe by definition.

For the same-origin policy for cookies (which semantically differs from the DOM policy, see [Zal09a]) the same circumvention holds. Cookies for a page are accessible to code injected to the web page.

2.3.6 Additional XSS Mitigations

Defense-in-depth approaches for XSS are very important because of the abundance of XSS vulnerabilities. Programmers and web browser users can both reduce the threat posed by XSS significantly by the measures explained here. Note that these techniques don't solve the problem of XSS so it is still the application developer's (and architect's) responsibility to prevent XSS vulnerabilities in the first place.

HttpOnly Cookies

Cookies are normally accessible to JavaScript code through the DOM. Injected code could read the cookie and navigate the browser to a site of the attacker's choice, submitting the cookie's value as a parameter (or, if possible, even use an XMLHttpRequest, so the submission is invisible to the user). Most browsers nowadays support a flag on cookies called *HttpOnly* which causes cookies to be inaccessible to scripts through the DOM. Unfortunately, requests through XMLHttpRequest receive the full response from the web server which includes the cookies to set. Michal Zalewski found [Zal09b] that some browser implementations don't hide the *HttpOnly* cookies from these types of requests so they are still accessible to scripts by parsing the web server response. This gap seems to get closed by current browser generations but it cannot be relied on that *HttpOnly* cookies are not accessible to scripts.

Client-Side Script Blockers

NoScript[41] is an extension to the popular Firefox[42] browser and prevents script execution on web pages by default. This provides perfect protection from XSS but renders many pages unusable. Users can selectively activate scripting for pages they trust.

Additionally, NoScript provides an Anti-XSS filter that sanitizes parameters before they are sent to a web server. It sanitizes parameters getting submitted from trusted sites to untrusted sites and can thus prevent many users from being attacked in the presence of XSS vulnerabilities in a trusted web page.

NoScript requires manual installation by the user and will thus probably not cover all installations of Firefox anytime soon. Additionally, Firefox is still a

[41] `http://noscript.net/`

[42] `http://www.mozilla.com/en-US/firefox/firefox.html`

minority amongst the web browsers in use. Microsoft's Internet Explorer 8 will include XSS filtering facilities in the standard distribution, bringing client-side filtering to a more general audience.

As a caveat, client-side script blockers can only protect against type 0 and type 1 XSS attacks (see section 2.3.4 on page 35 on XSS attack types). Type 2 (persistent) attack code is already present at the web page visited and does not pass through the client-side filter. Nevertheless, browser-based filtering is a good choice in a defense-in-depth strategy.

Static Analysis Of Persistent Storage

Buying into the defense-in-depth argument that someone might have already circumvented a web site's protective measures, the *Scrubbr* tool [Sul09] provides after-the-fact sanitation. *Scrubbr* scans a database's contents to find content that does not conform to a policy file. The tool uses the same policy engine as AntiSAMY (see section 2 on page 35). The web site provider still has to provide a policy file that corresponds to his needs which can take a lot of effort if the business data is diverse and uses complex formats.

2.3.7 Summary

XSS vulnerabilities are much harder to prevent than SQL injection vulnerabilities since the attack surface is larger. Virtually all user input can carry XSS which eventually becomes included in a web page. For SQLIA, it suffices to protect data going into the database.

In addition, sanitation for XSS is much harder since there are different contexts that require different sanitations and — to make things worse — browsers are very liberal when it comes to treating strings as valid HTML. For SQLIA, only using standard SQL features results in very predictable behavior for virtually all databases.

Nevertheless, workable preventions for XSS vulnerabilities are available. The OWASP cheat sheet provides very secure white-listing methods, numerous libraries for XSS sanitation are available, even "automatic" solutions (like Google's AutoEscape). The only remaining difficulty is to decide where to put the sanitations. In an Model-View-Controller (MVC) architecture, the responsibility clearly falls to the *view*. Without a clear architectural separation, explicit codification (as in policy documentation) is necessary to ensure that all data is consistently sanitized.

Even though the correct sanitation of XSS is harder than sanitation for SQLIA, in the end it also boils down to having clear responsibilities for sanitation. I find it surprising that the seminal book about XSS, *XSS Attacks: Cross Site Scripting Exploits And Defenses*[Han08], co-authored by expert practicioners like Hansen and Grossman, limits itself to the technical complications of XSS, without mentioning application architecture as a way to effectively put the presented mitigation methods to effective use.

2.4 Input Or Output Filtering

The Common Weakness Enumeration (CWE), a comprehensive classification of security weaknesses (see section A.6 on page 128 in the appendix), lists *Failure to Sanitize Data into a Different Plane (aka 'Injection')* [DVT09a] as the root cause of both SQL[43] injection and XSS[44].

The root cause in fact is missing sanitation of data that gets passed to a different execution context where data safe for local use may be dangerous (the web application doesn't mind SQL queries with an SQL injection inside). From a conceptual standpoint, the vulnerabilities could be tackled by only using output filtering when changing planes.

Nevertheless, input and output filtering both have advantages and disadvantages and combining them makes preventing vulnerabilities easier. Consistent input filtering touches all incoming data, providing a single point all data passes. At that time, it is not always clear what happens to user input. Input filtering cannot always tell if the data gets written to the database or if it will appear on a web page. Therefore, input filtering should ensure that data conforms to the expectation of the web application itself (probably dictated by the business rules), not the format needed for output to the database or a web page. This requires clear type definitions for all data the application will encounter (see section 2.6 on page 42).

Output filtering can be aware of the context the data is actually used in, if there are multiple possibilities. Therefore, specific filtering can take place here. Also, output filtering is only used on the data that actually needs filtering, which has performance benefits for complex filtering needs.

Independent of the type of filtering, the use of input and output filtering has to be consistent. It has to be clear throughout the web application in which state data is. Multiple sanitations can have adverse effects and even remove the sanitation effect completely (also see section 2.2.4 on page 18 on early escaping).

I created a description of a best practice web application architecture (see section 2.6 on page 42) after looking at the first web application project (WordPress, see section 4.2.1 in the case study chapter). It shows what is possible when applications leverage data modeling in a web application framework.

2.5 General Security Practices

This section contains some general practices of secure software development which are not specific to the vulnerabilities under discussion in this thesis but cover sane software development practices in general. I discuss them because I found some of these best practices missing in the projects discussed in the case studies in chapter 4. These concepts can help with reasoning about the kind of process innovations that are helpful in actually preventing security vulnerabilities in open source web applications.

[43] *Failure to Preserve SQL Query Structure (aka 'SQL Injection')* [DAVT08]
[44] *Failure to Preserve Web Page Structure (aka 'Cross-site Scripting')* [DVT09b]

2.5.1 Don't Build It Yourself - Code Reuse

code reuse

Developing software is error-prone, especially for security purposes. Input and output sanitation are especially tricky because of many exceptions and implementation details [Wil09].

Ready-to-use components for many security-related aspects are available from web application frameworks or as stand-alone components (see the discussions in the respective sections: Scrubbr (2.3.6), ORMs (2.2.4) or AntiSAMY (2.3.3)). OWASP takes this a step further, collecting best practice security components into a single library, the OWASP Enterprise Security API (ESAPI). The ESAPI covers input and output validation, logging and intrusion detection and is available for various programming languages [OWA09].

The use of available components (in general, but especially for security purposes) constitutes a best practice since it saves effort and reduces defects in the software [BBM96] [RvW03, p. 49].

2.5.2 Defensive Design

defensive design

Defensive design (or defensive programming) [McC04, pp. 187] expects things to go wrong. It uses secure defaults in programming and expects and handles exceptional cases in a secure way (related to *graceful degradation* [RvW03, p. 43]) .

The *principle of least privilege* [RvW03, p.40] is also part of defensive design. It states that a component should always only have the privileges it absolutely needs. The principle is related to software development but is also very important for the configuration and deployment of software.

2.5.3 Defense In Depth

defense in depth

Defense in depth (or layered security) [McC04, pp. 203] [RvW03, p.48] acknowledges that security measures may fail. Implementing multiple layered measures reduces the risk of a complete security breach. Defense in depth is not "adding a little of this, and a little of that" but requires a comprehensive concept for the defense measures to be effective together. Defense in depth is especially valuable when attack patterns are fuzzy and not completely guaranteed to be detected by a single measure or in high risk environments. Defense in depth can also be useful for cost-savings. Security practitioner Grossman counts Web Application Firewall (WAF)s[45] as a cheap measure to prevent known attacks on applications without having to fix the application code [Gro09a].

2.5.4 White-listing

white-listing

White-listing is the defensive design approach to filtering and validation. White-listing uses a definition of *known good* input and filters out or fails on the rest [McC04, p. 188]. Its "evil twin", blacklisting (2.5.5) is still in wide use (see chapter 4), despite of known shortcomings.

[45] which only filter user input for known attack signatures but do not fix the application defects

2.5.5 Blacklisting

Blacklisting uses a definition of "known bad" data and lets the rest pass. It is known as a bad practice as it is easy to forget input patterns that can be put to malicious use. Also, attack techniques evolve and attackers find new ways to use seemingly innocent input for exploits. White-listing is more resilient to new attacks. Blacklisting should only be used to supplement white-listing for adding known exceptions.

blacklisting

2.6 Building The Ultimate Framework

After having done the research about SQLIA, XSS it was clear that sufficiently advanced technology to prevent both vulnerabilities was available (for details see sections 2.2 and 2.3).

I thought about how a best practice web application framework could automatically apply the techniques needed to prevent SQLIA and XSS.

Having database access, HTML output filtering and other sanitation methods in one single place respectively (a layer) clearly makes these methods much more maintainable. It also helps making the responsibilities for sanitation clear and defines which state user input is in after having passed through a certain layer (cp. the discussion of input and output sanitation in section 2.4 on page 39).

However, developers would still be responsible to apply the right sanitations or validations in the right places.

Explicit data modeling can help. Existing web application frameworks like CakePHP, Django or sophisticated web application stacks like Java EE or Spring use an MVC-like architecture and explicitly declare business objects with their attributes. Having a centralized declaration of data types and constraints allows the framework to make appropriate decisions about validation and filtering.

Basically out of frustration with WordPress not using any framework and data modeling (see section 4.2.1 in the case study chapter for details), I spent some days thinking about a solution that would automate all input and output filtering in the application framework, freeing the developers from caring about these issues once and for all.

The idea depended on having very *rich* type definitions. Web applications often deal with types supported by the database (which are basically numbers and strings, see section 2.2.1 at the beginning of this thesis). Most non-numeric types are treated as strings, without any further semantics attached. If developers could use types like `username` that have stricter constraints (like a specific length or allowed characters), the framework could take over all validation and sanitation. ORMs could be used for database access, preventing SQL injections (given proper input validation enforced by the framework, this would be safe) and the problem of selectively allowing HTML as user input could be solved: a rich type could contain a definition of allowed HTML, e.g. an AntiSAMY policy file (see section 2.3.3 on page 32) which could automatically sanitize user input into HTML.

This approach would have required a large standard library of different commonly used rich types because sanitation needs could be vastly different for different data formats. I figured that unless support for the most commonly needed types in web applications was built-in, developers would just rely on simple types like strings, losing the semantics provided by rich types.

The type definition prototype I came up with looked like listing 2.29. The code is a fictional example of a model in the Django [Fou09] web application framework (which I had worked with in an earlier version) for a special text field that may contain HTML heading tags but no other tags. The special text field would inherit from normal Django `Field`s and add custom validation and

transformation to HTML.

```
class MyTextField(models.Field):
    # may only contain <H1>
    htmltransformation = AntiSamy(H1Profile) # to HTML
    validator = HtmlValidator(tagsAllowed=(h1))
```

Listing 2.29: Rich data model example

I discovered that Django actually used much of what I had in mind. Django is a good example for a web application framework because it demonstrates many best practices and stays pragmatic enough to not overburden developers (with highly theoretical ideas like mine)[46].

2.6.1 Data Modeling

In Django, a data model looks like listing 2.30[47].

```
class Donor (models.Model):
    choices = ( ('A', 'add'), ('D', 'delete') )
    operation = models.CharField(max_length=1, choices=choices
        , default='A')
    approved = models.BooleanField(default=False)
    ceo = models.ForeignKey(Ceo)
    first_name = models.CharField(max_length=63)
    last_name = models.CharField(max_length=63)
    donor_address = models.ForeignKey(Address)
    ctime = models.DateTimeField(auto_now_add=True)
    mtime = models.DateTimeField(auto_now=True)
    objects = DonorManager()
```

Listing 2.30: Django data model example

Listing 2.30 shows a Donor class with several attributes operation, approved, etc. The attribute types in this example are tied to types available in SQL, but Django also supports types not directly available in SQL. Types are modeled as classes and can implement their own validation code. Thanks to the object-oriented nature of types in Django, types can be based on already available similar types. The framework uses this type and data model for client-side validation of web forms generated through its forms API, for sanitation and validation on the server side and for communication with the database. The concept of having semantically rich "real" types for the data dealt with in the application unifies a lot of validation and filtering.

The database layer is completely abstracted using a custom ORM, though it is possible to use raw SQL. For specific requests not covered by general

[46] there are other frameworks with similar features; I chose Django because I was partly familiar with it

[47] excerpt of `http://github.com/thejefflarson/ceo_campaign_contributions/blob/9e65f2d4377d4ed188671d603e9085946cb50a87/finance/models.py`

ORM functionality, models can be extended with the necessary queries. Django supports prepared statements.

Django also already contains a library of commonly used data types which are not covered by SQL, like IP addresses or files.

There is one big difference between my idea of the ultimate framework and Django: Django does not allow HTML user input by default. All input is entity-encoded by default and mark-up is provided through Markdown, a non-HTML mark-up language (previously discussed in section 2.3.3 on page 32). This obsoletes the need of many data types that would only differ in the HTML tag that would be allowed in the input. This reduces the number of different types needed to a manageable level. Django's diverse built-in types further reduce the need to introduce new types.

Django is a prime example on how having an explicit data model and a framework to support it helps with most of the issues involved in SQLIA and XSS. Django has had XSS vulnerabilities in the core code[48] and it cannot prevent all types of XSS attacks automatically (since the dependence of scripting on context makes automatic solutions impossible) but it would prevent most of what has been found in the open source web applications discussed in chapter 4 on page 53.

Django provides a solution that has most of the benefits of my rich type idea but is much simpler: Using a non-HTML mark-up library for all user input allows the HTML encoding of all user input while allowing the use of mark-up. This approach does not require specific types for different HTML transformation rules and as such does not overwhelm the developer with the need of many different types just for the sake of correct HTML encoding.

In the presence of inherently safe mark-up languages, my complicated approach to XSS did not seem to be worth it. Being able to use the rich formatting and layout options provided by HTML is seldom necessary for user input since it usually represents a single element of a web page, not the page itself. The ultimate framework remained a thought experiment which proved to be too complicated in a world with nearly-ultimate frameworks readily available.

The essential lesson to take away from this story is that an ultimate framework would use data modeling and strong compartmentalization of sanitation responsibilities.

[48] `http://www.djangoproject.com/weblog/2008/may/14/security/`

Chapter 3

The Process Improvement Idea

> An abstraction is one thing that represents several real things equally well.
>
> Edsger W. Dijkstra

As laid out in chapter 2, prevention of SQLIA and XSS is not technically hard when using well-known best practices. Furthermore, clear software architecture that isolates interaction with external systems (the user's browser, a database, etc.) provides clear separation of concerns, a prerequisite for consistent use of the vulnerability preventions throughout the system (see section 2.6 on page 42 on the idea of an ultimate framework).

Having looked at WordPress as the first web application project (see section 4.2.1) and finding no clear architectural separation of concerns, I decided to concentrate on a process innovation targeting architecture.

3.1 Why Annotations?

There are other ways to try to prevent security vulnerabilities, like integrating automatic static analysis checking tools into the development process or using black-box penetration testing on the application. I focus on the architecture level because it increases overall application security. Static checking or black-box testing can only find *defects*, single instances of vulnerabilities. These methods do not provide much information about the principle behind these vulnerabilities, which would allow developers to prevent most (or all) related defects (see the episode in section 4.2.1 about WordPress). Still other methods for vulnerability prevention, such as mandatory developer training or a security awareness campaign, might not fit too well into an open source application project with volunteer developers. Architectural improvements seemed to have the most long-term benefits.

Open source projects are, due to the openness of the development process, developed evolutionary. It is difficult to make fundamental changes[1]. The

[1] fundamental disagreements often lead to forks, multiple projects continuing in different

process innovation therefore has to support incremental development. Moving towards an ultimate framework (cp. section 2.6) is hard for projects that do not already have strong data modeling, as it often changes the architecture of the application completely.

So, to provide web applications with an incremental path to architecture, I chose to use source code annotations that would mark the places that needed change. For reasons why I consider annotations especially well suited for an evolutionary approach, see section 3.2 with details on the annotations. During my research, I asked several web application projects if they liked the idea of the annotations and if they were willing to accept a patch from me that would provide annotations for one specific change (either related to SQLIA or XSS). All the details can be found in chapter 4.

I found some recurring fundamental concepts that obviously had an influence on the capabilities of the projects to accept my process innovations. Therefore, I added some additional projects to the analysis where I did not try an innovation introduction but only looked at the source code and some process properties to see if these concepts discovered were even more widespread (cp. section 4.3 on page 77).

3.2 Context Provided - The Annotations

3.2.1 Reviews

Code reviews are a popular means for improving software quality, especially for security [Lai98] [RvW03, p. 141] [FH06]. Their value is mainly determined by the number of anomalies or defects they find and by the probability that the issues get corrected. Reviews are not only useful to find individual defects (that can often be done more efficiently with automated testing) but also to point out architectural issues or smells[2].

In this thesis I discuss annotations meant for the refactoring[3][4] towards isolation of functionality relevant for prevention of SQLIA and XSS vulnerabilities. In-code annotations are a good companion to code reviews since they capture the insight gained in the review directly in the code. The basic process I designed and then validated with open source web application projects consists of two parts:

1. a detailed description of the current problem, a goal definition ("how does the code look when we're there?") and a description of the refactoring process, including what the associated annotation looks like and what the different variants (if any) of the annotation mean; this description could

directions

[2]this term, signifying a deeper problem in the code than a simple defect, became popular after the release of the now-seminal book *Refactoring* by Fowler et al. [FBB+99]

[3]refactoring, according to Fowler [FBB+99], is improving the structure of code without changing its functionality

[4]in a strict sense, this is not refactoring: it is supposed to change the functionality to consistently use filtering and validation, which may not be the case before

be captured by the external issue tracker, the project Wiki or some other public documentation facility; this is called the *issue document*;

2. a patch (or multiple patches), containing the annotations; this is called the *annotation set*

The detailed description accompanying the annotations is really important as the annotations — having mnemonic but short identifiers as names — cannot themselves explain the whole refactoring process. The description is also the base on which communication with the project members takes place. Therefore, the goals and the way to get there have to be presented convincingly. It helps to have spoken to project members before, to make sure that the direction of the refactoring coincides with the direction the project is heading (see examples in the case studies, chapter 4).

3.2.2 Benefits Over Issue Tracking Software

Many software projects use an issue tracker (or bug tracker) to manage defects, larger issues and tasks of the project. These tools do not work well with issues related to multiple (possibly lots of) places in the code (see the anecdote about the WordPress bug tracker in section 4.2.1 on page 61).

For WordPress, my first try to use the annotation method in a real world project, there were around 450 places in the code that I suggested to be changed. Filing 450 single tickets in the issue tracker would require lots of effort to set up and then to mark as done. Using a single ticket and referencing (by line number, e.g.) the offending places would require constant updating of the ticket while the code is incrementally cleaned up.

Such a task is incompatible with traditional issue trackers alone for the following reasons:

1. Filing a single task is too coarse-grained. It would take months to complete and does not provide any milestones to use for orientation or motivation.

2. There's no easy and maintainable way to connect the task in the tracker to the code that needs changing. Either filing multiple tasks or having one task description with a list of places is cumbersome.

3. In order to be able to see the progress of the task, one would again need to file it as multiple issues or constantly updated the references in a single issue.

Putting annotations directly into the code solves these problems:

1. Removing an annotation because the concerned issue has been fixed is a measurable achievement. A developer can fix any number of issues in one go and always have a closed package of work (annotations may not depend on another annotation in the same *annotation set*). The time span between the achievement (fixing some code) and the reward (getting to remove an annotation) is much shorter with annotations. Having a

single issue in the tracker can take months to fix all the related issues. The direct feedback in the code should be a positive motivational factor for fixing the issues indicated by the annotations.

2. The annotation is where the offending code segment is. This provides developers with an immediate connection to the context for the issue at hand. Because the annotations have a clearly defined structure, it is easy to find them, even with a simple text editor that supports string searches.

3. Writing a small script that counts remaining annotations, even specific types of annotations (see also section 3.2.3), is trivial. Progress feedback could also easily be added to a standard Integrated Development Environment (IDE).

One can think of more advantages that annotations might have in comparison to traditional external issue tracking. Having the annotations directly in the code raises awareness of the presence of the issues encoded into the annotation. Issues are no longer hidden away in an external context (maybe a developer has to use a completely different application to view issues in the issue tracker). Working with the code also means working with the annotations. The desire to get rid of the annotations might be larger than with external solutions. Substantiating these assumptions is not possible with the data collected in this thesis and would require further research.

One disadvantage which is often attributed to code annotations is the tendency for the program code and the annotations to disconnect over time (when code is changed) or to create additional maintenance work to keep the source code and the annotations synchronized. This does not apply to the annotations presented here since annotations are removed from the source code after the issue they annotated is resolved.

In conjunction with security issues, annotations also show one actual disadvantage: They can only be used to outline structural weaknesses and not to mark concrete vulnerabilities. Since the annotations will be committed to the public source code repository of the respective project, attackers could monitor the repository for annotations that mark a vulnerability. This would save them a lot of time searching for vulnerabilities by themselves. Using annotations in the way this thesis suggests is mostly harmless. The presence of annotations could be taken as a hint that an application has structural deficiencies, but a potential attacker would still have to go through the source code, understand the issue the annotation is about and then find vulnerable spots.

3.2.3 How Does It Look Like? - Structure Of The Annotations

To be portable across programming languages and projects, the annotations have to be simple and not use features specific to a language or development culture. An example annotation looks like listing 3.1:

```
// @RawSQLUse, trivial_implementation, SELECT
```

Listing 3.1: An annotation example for the case of raw SQL use

Annotations are placed inside comments, using the standard comment construct of the respective language. Most languages have simple mark-up for one line or multi-line comments. `//`, for example, works in languages like C, Java, or PHP, while Python or shell scripts use `#`. Placing the annotations in comments prevents them from interfering with the language-specific parser. Prepending `@` differentiates the annotations from normal comments and makes them stand out visually. Java, PHP's documentation system and Python also prepend `@` to annotations[5], so the notation will be familiar to many developers.

Annotations can consist of up to 3 parts. Originally, annotations only had an identifier, the so-called *issueName*. The *issueName* is unique to an *annotation set* and can be used to refer to it. It can also be used as a search keyword to find all instances in an *annotation set*. The *issueName* is designed to be mnemonic to be easy to remember, and also to be used as a name when communicating about an annotation[6]. I describe the development of the two other annotation parts using examples from the WordPress episode, in which I developed annotations for consistent use of SQL sanitation (details can be found in section 4.2.1). The two additional parts developed for SQL annotations are also useful for annotations targeting XSS prevention (see section 4.2.2 on page 74 in the Mambo episode) but may not be generally applicable for all projects or vulnerabilities.

While annotating the first project (SQL in WordPress, see section 4.2.1 on page 55), I decided to include a second element that classified the effort required to resolve a particular annotation instance. Since the annotation method produced lots of annotation instances, I found it useful to give developers an easy way to find the low hanging fruit. The *effortIndicator* gives a rough estimate of the effort required to resolve the respective annotation instance. The values are specific to the *annotation set*, since different refactorings can require vastly different levels of effort. The example in listing 3.1, taken from the WordPress project, uses an *effortIndicator* value of `trivial_implementation` to signify that resolving this particular annotation instance requires implementing a method, but that similar methods, which could be used as an example, already existed. What the respective *effortIndicator*s mean is specified in the *issue document*. Detailed discussion of the values used for each of the annotations can be found in the respective episodes in chapter 4.

I later introduced a third part I deemed helpful: *requiresFeatures* is a very lightweight way to express dependency on certain features. This can best be explained with an example: For the abstraction of raw SQL into functions that produce SQL in WordPress, several functions would have been needed to construct parts of an SQL statement, e.g. a WHERE clause. If a certain annotation mentioned WHERE in *requiresFeatures*, a developer could easily find all the annotations he could resolve after implementing the function needed for WHERE. Developers do not have to revisit all annotation instances after implementing one feature but can easily search for just the ones that depend on the feature

[5] which have language-specific semantics

[6] see Evan's "Domain-Driven Design" [Eva03] for the importance of a common vocabulary in software projects

implemented. Note that in order to keep the methodology lightweight, only the *issueName* is mandatory. The *effortIndicator* and *requiresFeatures* can be used if the annotator considers them helpful.

As you have seen, the annotations are a low-tech approach to encourage refactoring. The annotations are designed for human consumption. Tools can be used to make dealing with the annotations more efficient by providing quick access through search, but there is no intrinsic need for tools. Using plain strings with a clearly distinguishable format (through the use of `@` and an unique *issueName*) only requires very basic tool support to use the annotations to their full potential[7].

Prerequisites

Obviously, the annotations cannot be used for all projects. I assume the following as given for any project the annotations can be used with.

- The project community values quality, especially system security. If this would not be the case, no changes to the development process targeting security would be accepted by the community.
- The project community is generally aware of SQLIA and XSS vulnerabilities. Teaching the community about the vulnerabilities first would take too much effort.
- The application does not yet have most of the properties of an *Ultimate Framework* (section 2.6) as the annotations could not provide major benefits in this case.

For the candidate selection for the annotation introduction I used these prerequisites in the following way: I would not consider a project for the annotations if it did not have any protection against SQLIA or XSS yet (this is easy to determine by looking at the web page of the project or skimming source code) or if it already uses a comprehensive explicit data model (which can be determined by the same means).

3.2.4 Trying Out The Annotations - Innovation Introduction

After designing the annotation technique I validated the method with open source web application projects (see the case studies for WordPress (section 4.2.1) and Mambo (section 4.2.2)). Since I was not able to do an in-depth long term study to observe the number of vulnerabilities over time, I chose acceptance (meaning the community agrees to have the annotations added to the official source code repository) of the annotations as the goal for the annotation introduction. One can expect the community of a popular web application project to be proficient enough in development practices to tell if the project benefits from a practice or not. Assessment of the effect the annotations have in the project and if security really benefits from the annotation method requires further research.

[7] namely string search which is available in virtually all text-editing software

Getting a project to just adopt a till-then unknown practice as complex as the annotations (the practice is not an especially complex practice but it is not immediately obvious and needs time to get used to) and then run with it is illusory. Therefore, I wrote the *issue document* and made an *annotation set* for the project and planned to let the developers do the implementation. If that would work out, I would encourage them to use the full-blown process on another issue. See chapter 4 for details.

3.2.5 Data Collection

I attempted 3 introductions of the annotations idea into web application projects. An introduction attempt is called an *episode* and consists of the following phases:

1. I suggest the annotation idea with a concrete plan for architectural refactoring to the respective project community, including the first *annotation set*.

2. I discuss the suggestion with the community and possibly integrate helpful suggestions.

3. The community includes the annotations into the source code repository.

4. The development community works with the annotations.

All interaction data produced during this interaction is used to provide answers to research questions Q2 (suitability of annotation approach) and Q3 (properties of projects influencing usefulness of annotations). Note that phase 4 was not reached in any of the episodes.

3.3 Code, Process, Fixes - Project Analysis Approach

The concepts discovered in the innovation introduction episodes encouraged me to look at a bunch of other open source web applications to see if the concepts were specific to the two projects I had chosen.

3.3.1 Data Collection

I extended the case study to more web applications where I would not try to do an innovation introduction but just look at the project to see if similar concepts to those discovered with WordPress and Mambo would emerge. I used two sources of data for each project:

1. The source code: I looked for recurring concepts (the significance of concepts is discussed in the section about fundamental terms on page 7), best practices or patterns that relate to security concepts in the technical discussion in chapter 2.

2. Public documentation and interviews: Web sites, Wikis and other forms of public documentation tell about process-related concepts which are present in the respective projects. I augmented public information with informal interviews to tap into more implicit knowledge that is not codified publicly.

I extracted concepts that identified important properties of the project related to the ability to accept architectural changes as suggested by the annotation method.

The project analysis makes use of some ideas of *Grounded Theory* according to Charmaz [Cha06]. The findings emerged from looking at raw data and distilling — at first very concrete, then more abstract — concepts that recur in different projects. I found this technique, which is closely related to what Charmaz calls *initial coding* and *focused coding*, very useful. I did not collect the amount of *rich data* Charmaz deems necessary for Grounded Theory, so the analysis does not live up to the standards of real Grounded Theory research. Nevertheless, the ideas of conceptualization and grounding theory on the data helped me to concentrate on *what is really there* and avoid speculation.

Project Questionnaire

To augment and ground the concepts I found in the analysis of the source code and online documentation, I designed a short questionnaire. The questionnaire focused on technical mitigation concepts used by the projects and on process properties which relate to discovered concepts. The latter includes the explicit codification of sanitation behavior and APIs.

The questionnaire can be found in the appendix in section A.4 on page 122 in an email sent to the Joomla! development mailing list. The same email was sent to all projects I looked at in this thesis but some did not respond (including WordPress and Mambo). The discussion of the answers to the questionnaire are included in the respective project discussions in chapter 4.

Chapter 4

The Cases

> Architecture is easy: you just stare at the paper until droplets of blood appear on your forehead.
>
> Anonymous

This chapter describes the case studies conducted for this thesis. I discuss the selection of the candidates chosen for the case studies and document the case studies in depth, including the extraction of concepts observed in each case (cp. *fundamental terms* on page 7).

The chapter closes with a discussion of the concepts observed in the various projects. Note that none of the concepts attributed to the various projects do imply the presence of vulnerabilities in any way. They only represent findings about the projects I deem relevant for application quality, especially in respect to the vulnerabilities presented in this thesis.

4.1 Candidate Selection

Initially, I simply looked at open source web application projects I already knew and which had a history of either SQLIA or XSS vulnerabilities. This way I started off with WordPress as my first candidate.

The other projects included in the analysis present well-known names like TYPO3, Joomla!, Drupal and phpBB.

Some projects were added to the candidates because they were forks of projects already in the analysis (or were founded in response to such a project) and had a different approach as the original (habari and Zikula). One project, Mambo, was added to the list of candidates following a suggestion in the WordPress mailing list.

I included one more project after receiving some suggestions from a friend — riotfamily — because it is a web publishing platform written in Java and because it shows many of the properties attributed to the *Ultimate Framework*.

One may wonder why all these applications are more or less in the web publishing or Content Management System (CMS) domain and are all written in PHP. Web publishing systems are very exposed to user input which makes them a natural target for the kind of research done in this thesis. Applications that

have a closed user group (like an internal Customer Relationship Management (CRM) or Enterprise Resource Planning (ERP) system) are often not directly exposed to the Internet, making them unavailable for remote testing by security researchers, resulting in less advisories and less awareness.

For web publishing systems, PHP seems to be the language of choice. The projects chosen for the analysis here are the most well-known applications. I did a search on *freshmeat.net*[1] in order to include other projects in the research but I did not find anything with a substantial amount of popularity[2].

4.1.1 Scope

Most of the applications analyzed here support *plug-ins* which extend the functionality of the core application or modify the behavior of the application itself. Plug-ins (known under various names in the different projects) are developed completely independently from the application core, in most cases by different developers. My research is limited to the application *core*. Plug-ins have vulnerabilities of their own and are treated as independent projects. Plug-ins are only relevant to the research in this thesis as they are dependent on APIs provided by the application cores (cp. the missing interface concept on page 95).

4.1.2 Notation

Cases contain references on the page margins. These are used to mark occurrences of concepts observed in the respective case as a help for the reader. Concepts referenced either come from technicalities in chapter 2 or, if they were first discovered in the case study, are explained in section 4.5 on page 93, later in this chapter.

4.2 Innovation Introduction

This section describes the two projects where I attempted to introduce the process innovation presented in section 3.2 on page 46. An in-depth code analysis is provided for each of the projects and the course of the discussion about the innovation introduction is depicted, including an analysis of noteworthy concepts discovered while interacting with the projects.

[1] `http://freshmeat.net`

[2] as indicated by the developer numbers on *sourceforge.net*, freshmeat's project repository

4.2.1 WordPress

My journey into the world of web application security began with WordPress.

WordPress is a popular web-logging platform licensed under the General Public License (GPL) and written in PHP. It appeared as a fork of the b2\cafelog software in 2003 and is its official successor. WordPress development is led by Matt Mullenweg, its original developer, and Ryan Boren. The two team leaders and 6 *contributing developers* have commit access to the source code repository, although many code contributions come from the community. Mullenweg also heads the company automattic[3] which provides commercial services like hosting or plug-in development for WordPress.

WordPress considers itself a web publishing *platform* and supports so-called plug-ins. Plug-ins are available[4] for caching support, aggregating contents from other web pages, changing the way commenting on blog articles works and many other things. Plug-ins can significantly alter the way WordPress works and are developed separately from the WordPress core.

Code Analysis

WordPress is a PHP web application. As there is no application server present, the interaction model is request based. WordPress is called and initialized for every request. There is no internal persistence, data has to be written out to the database at the end of each request.

WordPress only supports MySQL databases for persistent storage. There is no dependency on external PHP libraries and no support for HTML templating languages. WordPress requires PHP 4.3 or newer and MySQL 4.0 or higher [Wor09b][5].

Code Structure The WordPress source code consists of 4 major parts, separated into directories[6]:

`wp-admin` contains code relevant to the administration interface

`wp-content` contains themes (which determine the layout and design of the pages generated by WordPress) and plug-ins

`wp-includes` contains code used throughout the system (for things like database access or support for RSS).

the base directory which contains the entry points to the application for web browsers (`index.php`), feed readers (`wp-feed.php`) and for Remote

[3] `http://www.automattic.com`

[4] `http://wordpress.org/extend/plugins/`

[5] this is relevant for the compatibility with external libraries

[6] for access to the WordPress source code repository, see section A.1.2 on page 114 in the appendix

Procedure Call (RPC)s (`xmlrpc.php`), the login page and configuration files.

The flow through a request in WordPress is similar for all request types. WordPress reads configuration files (which contain the access configuration for the database), then checks what kind of request came in, connects to the database to fetch the necessary data and then calls the respective method to display the data.

The interesting parts for this discussion are input handling, database access and the way HTML is displayed.

Input Handling

In WordPress, all data coming in through GET/POST parameters or cookies is automatically quoted and escaped for database output (see listing 4.1). If the execution environment is configured to use automatic quoting (get_magic_quotes_gpc(), cp. section 2.2.5 on page 22), the data is un-escaped beforehand. There is no type based input validation in the input layer.

```
500 // If already slashed, strip.
501 if ( get_magic_quotes_gpc() ) {
502         $_GET    = stripslashes_deep($_GET    );
503         $_POST   = stripslashes_deep($_POST   );
504         $_COOKIE = stripslashes_deep($_COOKIE);
505 }
506
507 // Escape with wpdb.
508 $_GET    = add_magic_quotes($_GET    );
509 $_POST   = add_magic_quotes($_POST   );
510 $_COOKIE = add_magic_quotes($_COOKIE);
511 $_SERVER = add_magic_quotes($_SERVER);
```

Listing 4.1: WordPress: `wp-settings.php`, (revision 10443)

Early Escaping →p. 18

A look into the add_magic_quotes() method reveals a case of *Early Escaping*. The method calls escape() on the database object. One would expect by the name that this method does escaping relevant to the safe insertion into a database. But it is not clear at this point if the data will actually make its way to the database. Data is escaped to be safe for insertion into a database regardless of destination. Combined with unclear responsibilities for escaping (concept *Inconsistent Use Of Sanitation*), this can lead to vulnerabilities.

Use Of Inferior Method →p. 22

Furthermore, the escape() method actually only calls addslashes() on its argument and we got to know this method as a weak security mitigation. The method seemed to call mysql_real_escape() which would be safe against character set evasions (see description in chapter 2, section 2.2.5) but it is commented out with the comment: *causing problems*[7].

[7] WordPress `wp-includes/wp-db.php`:430 (rev. 9953)

Data Access Abstraction

WordPress only supports MySQL, so there is no need for a database abstraction layer. All database requests, however, are done through methods in `wp-db.php`[8]. I refer to this file as the *database layer* in the following.

Currently, there are two different ways SQL queries are passed to the *database layer*. In the example in listing 4.2, the calling code is responsible for all sanitation and escaping. The `query()` method directly executes queries on the database. Since the database layer only gets the full SQL string, it does not have any information about the semantics of the individual parts (cp. section 2.2.4 on page 17). The database layer cannot do any sensible sanitation before execution. As it appears, this way of doing SQL queries is mostly used in simple cases, where there are only one or two variables in the query. Injections are prevented in this case by type-casting variables (for integers)[9] and by calling escape methods for string types. This is usually done only a few lines away from the actual query, but it sometimes requires some looking around to see where variables come from and how they were treated.

```
22 $deleted_spam = $wpdb->query( "DELETE FROM $wpdb->comments
       WHERE
23  comment_approved = 'spam' AND '$delete_time' >
       comment_date_gmt" );
```

Listing 4.2: WordPress: `wp-admin/edit-comments.php`, (revision 10438)

Yet, most queries use a custom implementation of *prepared statements*. WordPress has a `prepare()` function which uses `printf()` (similar to the C `printf()`) to insert variables into a string. Listing 4.3 is an example that uses this method. There is no obvious reason to have different methods for database access and this can create problems with the consistency of data treatment. Therefore, I promote this to a concept called *Non-Uniform Database Access*.

Non-Uniform DB Access →p. 95

```
681 $wpdb->query( $wpdb->prepare("UPDATE {$wpdb->posts} SET
682  post_parent = %d WHERE ID = %d", $local_parent_id,
       $local_child_id));
```

Listing 4.3: WordPress: `wp-admin/import/wordpress.php`, (revision 10339)

Using `printf()` has the advantage of being able to have typed variable insertions. The `%d`s in the excerpt guarantee that the inserted value is inserted as a digit.

The `prepare()` method takes a format string and an array of values to be inserted into the format string as arguments. It simply removes single quotes and double quotes around the format string placeholders and finally single-quotes them. All the extra arguments (the values inserted into the format

[8] WordPress `wp-includes/wp-db.php` (rev. 9953)

[9] type-casting to integers in PHP removes the non-integer part of the value and always returns a proper integer, thereby preventing injections for numeric columns

string) are escaped using the escape() function[10] and then the final string is constructed using vsprintf ()[11].

WordPress developed their own version of *prepared statements* (cp. 2.2.4 on page 19) which does not provide the performance benefits of a native library for prepared statements by the database provider. The possibility of SQLIA using character set inconsistencies (cp. section 2.2.4 on page 17 in chapter 2) is also present, since the implementation of prepare() is separate from the database.

There is a PHP library for the use of MySQL databases, called `mysqli` (for MySQL *improved*). It supports real *prepared statements* and is native to PHP. WordPress cannot use this library because it is only officially part of PHP since PHP 5.3[12] and the WordPress community insists on supporting PHP versions 4.3 and up. They mainly justify this with lacking support for PHP 5 with some web hosters [Wor09b]. Time and time again, the debate about migrating to PHP 5 comes up on the WordPress developer mailing list (e.g. the thread emerging from a new developer's request in [wp:24350]). The recurring discussions prompted Matt Mullenweg, the project lead, to write a blog post [Mul07] clarifying his support for PHP 4, citing low adoption rates and the lack of convincing new features in PHP 5 to make the change worthwhile.

The arguments provided for sticking with PHP 4 exchanged on the mailing list also boil down to missing or not-trivial-to-setup support for PHP 5 by web hosters. The convenience of users of WordPress seems to be the major concern in discussions (see [wp:24353]).

PHP 5 is incompatible to PHP 4 in some minor areas [Gro09d]. It provides improvements for object-oriented programming, database access, performance and XML handling [Fel04]. Developers seem to prefer writing for PHP 5, but respect the decision by Mullenweg ([wp:24535]).

The presence of advocacy projects for PHP 5, which list projects and hosters that already support PHP 5 (like *GoPHP5* [DGD+08]) indicates that there are indeed issues in upgrading web hosters to PHP 5. Many old applications rely on PHP 4 so the hoster cannot upgrade; applications don't update to PHP 5 because the hoster does not provide PHP 5 support; it is apparently not trivial to support both PHP 4 and PHP 5 at the same time[13].

Nevertheless, after 4 years of PHP 5's availability and the discontinuation of PHP 4 [Gro08], the move to PHP 5 seems to be necessary, especially because it is required by functionality very useful for a project like WordPress. The `mysqli` library, which would add real prepared statements to WordPress, or *HTMLPurifier* (cp. section on HTML escaping on page 32) would be very helpful to WordPress as both provide the state of the art for SQLIA and XSS mitigation, respectively. I call the concept of not using a desirable technology because of version dependencies a Legacy Constraint. The concepts *Use Of Inferior Method* and *Structural Conservatism* also apply here.

legacy constraint →p. 97
Use Of Inferior Method →p. 22
structural conservatism →p. 97

[10] again with addslashes see section 4.2.1 on page 56
[11] which is functionally equivalent to sprintf () but takes an array of arguments
[12] `http://de.php.net/manual/en/mysqli.mysqlnd.php`
[13] it needs howtos and some trickery as in `http://www.howtoforge.com/apache2_with_php5_and_php4`

HTML Output Encoding

Output to HTML happens in so-called *templates*. The WordPress template language uses PHP functions that act as template tags and generate HTML. So-called *themes* control the layout of the pages by calling template functions. Themes can be replaced by the administrator of the WordPress installation and provide a great deal of freedom concerning layout.

```
13 <h2><?php the_title(); ?></h2>
```

Listing 4.4: WordPress: `wp-content/themes/default/page.php`, (revision 8999)

Listing 4.4 shows how the title of a blog post is rendered in WordPress's default theme. WordPress does not use plain variables (besides very few exeptions) in the theme files. All the dynamic data is provided through function calls.

In WordPress, it is the responsibility of template methods to do sanitation and escaping for HTML. This is reasonable as people designing the themes might not possess the technical skills necessary to deal with escaping issues. However, this behavior would require explicit codification of the details of the encoding mechanism. Without codification it is difficult to know if and how data has been sanitized by just looking at the theme code. In addition, since the template tags cannot know where the produced output will be inserted, the use of generic template tags makes it impossible to deal with differing escaping needs (cp. section 2.3.3 on page 32).

Most of the template tags use WordPress's built-in filtering functionality. WordPress supports named filter chains that can be applied to data. Filters are simple functions that do things like escaping, re-formatting or tag balancing. Filters exist for encoding quotes into HTML entities, removing HTML tags, obfuscating email addresses as an anti-spam measure and many other things. Excerpt 4.5 shows an excerpt of the default filter setup.

```
50 // Display URL
51 $filters = array('user_url', 'link_url', 'link_image', '
       link_rss', 'comment_url');
52 foreach ( $filters as $filter ) {
53     add_filter($filter, 'strip_tags');
54     add_filter($filter, 'trim');
55     add_filter($filter, 'clean_url');
56     add_filter($filter, 'wp_filter_kses');
57 }
```

Listing 4.5: WordPress: `wp-includes/default-filters.php`, (revision 10442)

Yet, not all the template tag methods work as consistently as expected. the_title_attribute ()[14] returns a blog post title with HTML stripped, the_title ()[15]

[14] WordPress `wp-includes/post-templates.php`:74 (rev. 10339)
[15] WordPress `wp-includes/post-templates.php`:45 (rev. 10339)

returns the title without stripping it. If an attacker succeeds to get JavaScript code into the database field for blog post titles, WordPress is vulnerable to XSS attacks[16].

WordPress is not directly vulnerable in this case since WordPress entity-encodes post data before writing it to the database[17]. Nevertheless, this is not defense-in-depth. As highlighted by the existence of persistent XSS removal tools and the argumentation in section 2.3.6 on *legacy* XSS contained in the database, relying only on input filtering is ill-advised. Moreover, the documentation of the_title() does not state that the return values of the template tag need additional sanitation[18]. For the_title_attribute(), the documentation notes that it duplicates the functionality of the former in a safe way, but not vice versa. This must obviously lead to confusion among developers. Probably such knowledge is passed on verbally and seasoned developers know it. For people not that experienced in WordPress development (and maybe sometimes other, too) this creates an ambiguous interface to output sanitation and may lead to vulnerabilities. WordPress has documentation on data validation [Wor09a], but the document fails to explain who is responsible for validation and where that should happen. In the big picture, this is *Inconsistent Use Of Sanitation*.

Reliance On Input Filtering
→p. 95

Inconsistent Use Of Sanitation
→p. 95

An even larger issue is connected to the development of plug-ins. Plug-ins use the same functionality as the WordPress core. There is no defined interface[19] for the plug-ins. They can use the same raw database querying functionality the WordPress core uses, passing literal SQL to the database. There are two problems related to this. Firstly, plug-ins can mess with all the data in the WordPress database and bugs in plug-ins affect the whole system. Secondly, the lack of a defined interface exposes internal changes of functionality to all plug-ins. This is sometimes cited on the mailing list as a reason against changes to functionality in the WordPress core ([wp:24358]) and prevents evolution of the application. Both problems have the common reason of the lack of an interface to the functionality provided by the WordPress core. This lack is expressed in two different ways: The first one is missing protection from third party plug-ins by lack of isolation. The second one is the other way round, missing protection for third party plug-ins against spillage of internal changes. I call the base concept *Missing Interface*. I found further examples of the implications of that concept during the innovation introduction (see the discussion in section 4.2.1 later in this chapter).

Missing Interface
→p. 95

Handling Of Past Vulnerabilities

Jeremias Reith found an XSS vulnerability in WordPress which relied on the undifferentiated escaping of all input data for database use (explained earlier in section 4.2.1) [Rei09]. The `host` header of the HTTP protocol (which is used by WordPress to determine which instance of WordPress to access if multiple

[16] I tried it with the_title() and the_author_url(), it worked in both cases.

[17] [`WordPress`: (rev.])wp-includes/post.php135610400

[18] `http://codex.wordpress.org/Template_Tags/the_title`

[19] besides the Plug-in API, which only provides hooks for plug-ins to modify the behavior of WordPress but does not offer any contracts

WordPress installations share the same IP address) counts as user input as it is set by the user's web browser (or by a malicious user himself). The header passes the blanket sanitation all input goes through in WordPress but the sanitation is concerned with safety for insertion into databases. It is not useful in a context for URLs (cp. section 2.3.3 about JavaScript contexts) where the result is interpreted by a web browser.

The fix[20] led to the use of clean_url() on the incriminating data, which does safe sanitation for URLs but the default behavior of escaping the host header for database use did not change. Having a comprehensive data model would have enabled the WordPress community to include the `host` header as an URL data type and it would automatically be sanitized accordingly in all occurrences. The actual fix only prevented this specific vulnerability, ignoring a long-term solution.

The Proposal

I perceive SQLIA as easier to fix than XSS as there is only one target sanitation context for data (the database). This is why I chose to deal with annotations for architectural improvement of the database layer in my first innovation introduction.

As explained above, database interaction in WordPress always happens through a single "layer"[21], but this layer is very thin. It basically provides a method to set the SQL statement to execute, the custom implementation of *prepared statements*, some helpers for data sanitation and various methods to access the result of the query. There are also abstractions of INSERT[22] and UPDATE that only take the table, columns and values to set and construct a static query in `wp-db.php`. The abstraction for INSERT, insert(), is replicated in the excerpt 4.6 as an example.

```
661 function insert($table, $data) {
662     $data = add_magic_quotes($data);
663     $fields = array_keys($data);
664     return $this->query("INSERT INTO $table (`" . implode('`,`
        ',$fields) . "`) VALUES ('".implode("','",$data)."')");
665 }
```

Listing 4.6: WordPress: `wp-includes/wp-db.php`, (revision 9935)

At the time I wrote the annotations (January 2009), there were two tickets in WordPress' issue tracker related to these methods: The first one was a suggestion to add type information for the values passed to the methods,[23] which would allow type-specific sanitation. The second one suggested to make remaining raw INSERT and UPDATE statements throughout the code use the

[20] `http://core.trac.wordpress.org/changeset?new=9754%40branches%2F2.6%2Fwp-includes%2Ffeed.php&old=8336%40branches%2F2.6%2Fwp-includes%2Ffeed.php`

[21] WordPress `wp-includes/wp-db.php` (rev. 9935)

[22] WordPress `wp-includes/wp-db.php`:661 (rev. 9935)

[23] `http://core.trac.wordpress.org/ticket/7171`

abstraction methods instead[24]. The second ticket was already 8 months old at that time.

These tickets seemed to be a good fit for the annotation approach. My goal was not only to help move the INSERT and UPDATE statements to the abstraction layer but also to abstract as many other statements as possible. The ultimate goal would have been to only use the abstraction methods to communicate with the database and get rid of raw SQL outside the database "layer" altogether. This seemed a bit to ambitious for a first contribution to an open source project, so I did not state this goal in the *issue document*. As the annotation process supports increments, further abstractions could easily be added later.

I labeled the annotation `RawSQLUse` and started annotating (using simple string searching for `wpdb->query`). Looking at the use of raw SQL, I was able to make out three distinct effort levels required to abstract one particular SQL statement. These became the *effortIndicator*s as introduced in the annotation presentation in chapter 3. They are defined as follows:

`method_exists` An abstraction method already exists, it only has to be used. There were 85 cases like these. These were the INSERT and UPDATE statements mentioned earlier. Listing 4.7 shows an example.

`trivial_implementation` A similar abstraction to the one needed already exists for another type of SQL query. It requires very little effort to introduce the new abstraction. I found 172 queries that fit into this category. Listing 4.8 is an example of this type of annotation.

`simple_code` Abstracting queries of this type would require an abstraction method for an SQL clause that is not yet supported in any other method in the "abstraction layer" but the feature does not require any complex programming logic. This applies to support for SQL functions like SUM[25]. I put 111 queries into this category. Listing 4.9 illustrates this kind of annotation.

`algorithmic` The query is structurally complex or in some other way unique. Introducing a generalized abstraction requires sophisticated construction of SQL clauses[26]. The remaining 74 queries required such sophisticated handling. Listing A.5, showing an example, can be found in the appendix on page 123 since the query building takes numerous lines.

```
680 // @RawSQLUse, method_exists
681 $wpdb->query( $wpdb->prepare("UPDATE {$wpdb->posts}
682   SET post_parent = %d WHERE ID = %d", $local_parent_id,
        $local_child_id));
```

Listing 4.7: WordPress: `wp-admin/import/wordpress.php`, (revision 10339 (locally modified))

[24] `http://core.trac.wordpress.org/ticket/6836`

[25] used on a column and replaces the list of values in that column by the sum of its values

[26] an example would be to support WHERE clauses with parentheses, AND, OR, and NOT

```
688 // @RawSQLUse, trivial_implementation
689 $post_ids = (array) $wpdb->get_col( $wpdb->prepare("SELECT
        post_id
690   FROM $wpdb->postmeta WHERE meta_key = 'blogger_blog'
691   AND meta_value = %s", $host) );
```

Listing 4.8: WordPress: `wp-admin/import/blogger.php`, (revision 10339 (locally modified))

```
1021 // @RawSQLUse, simple_code
1022 return $wpdb->get_var( $wpdb->prepare("SELECT COUNT(*)
1023   FROM $wpdb->term_taxonomy WHERE taxonomy = %s $where",
         $taxonomy) );
```

Listing 4.9: WordPress: `wp-includes/taxonomy.php`, (revision 10428 (locally modified))

As stated above, finding the uses of raw SQL was straight-forward using string search and led to reliable results. The classification for the *effortIndicator* is also reliable because of its coarse-grainedness. Since the database abstraction resided in only one file, it was trivial to find out if an abstraction already existed or if needed helper functions would be provided. All the cases that did not fit into the three easily distinguishable classes (`method_exists`, `trivial_implementation` and `simple_code`) was attributed to `algorithmic`. This catch-all class obsoletes the need for exact knowledge of the intricacies of the project and the database abstraction layer, knowledge someone new to the project (like me) would not have. Therefore, I am convinced that the classification is valid.

Suggesting Changes

I wrote an explanation on what I was trying to do and where I thought WordPress should go, including the patch I created. To get a better chance of being heard (and taking advice from the *little innovator's guide* (contained in [Oez]) I first wrote to the WordPress core developers[27]. I wanted to make sure they would not object to my suggestion and immediately shoot it down in the discussion on the mailing list. Therefore, I sent an identical mail to each of the core developers, explaining what I was was planning for WordPress (see one mail in A.1 on page 117 in the appendix).

I got answers from three developers. The first one (Alex King) told me that he was not against the proposal but that he was also not the person that needed to be convinced[28][29]. The other two were very positive: Ryan Boren forwarded the mail to WordPress' closed security mailing list, told me about the two tickets related to my plan[30] and asked me to follow up[31]. Mark Jacquith

[27] whom I found at `http://codex.wordpress.org/Copyright_Holders`
[28] [wp:wp_alexking.txt]
[29] whatever that meant
[30] on which my suggestions were actually based on
[31] [wp:wp_ryanboren.txt]

considered the annotations the way to go for `insert` and `update` and expressed support[32].

I felt encouraged and addressed the WordPress community at large ([wp:25220]).

Discussion

The first bunch of responses was full of misunderstandings: One developer assumed that I wanted to move *all* SQL to the database layer to enable support for different database back-ends [wp:25240]. Someone else questioned my notion of having raw SQL access throughout all parts of the code was a bad thing [wp:25233]. I clarified my point of centralizing database access [wp:25245].

The discussion that followed (with some more clarifications from my side from time to time) brought the following interesting observations:

1. Plug-in developers were considered as not as skilled, especially concerning security issues, and were known to mostly write quick and dirty code. ([wp:25254], [wp:25386])

2. A few developers expressed that using raw SQL was not a bad thing. Someone argued SQL was the dominant data access language and that nobody had found a reason to move away from it [wp:25384]. It even went that far that one developer claimed that I thought WordPress developers were "idiots" because they could not write safe SQL statements [wp:25258].

3. One developer argued strongly against the use of abstractions known from frameworks like Zend or CakePHP on the basis that it "holds your hand" [wp:25258] and that developers would have to learn a new language [wp:25384]. Another one argued that abstractions were never as well thought-out than the original language [wp:25365].

   Some developers also insisted that developers would program their way around abstractions because they would miss powerful features or it would be more complicated if they used the abstractions [wp:25385], [wp:25384].

4. Several developers stated that my suggestions would not improve the security of WordPress, because it already was on a reasonable level. ([wp:25248], [wp:25384], [wp:25386])

5. The community argued strongly against my entire suggestion as they felt it made things more complicated. ([wp:25370], [wp:25382], [wp:25381], [wp:25365])

6. There was a short side-tracked discussion, initiated by two list members who provided arguments for the large benefits to be had using database abstractions offered by frameworks (CakePHP and Zend in their case). The discussion went on for a while and even brought up arguments for abstraction in general ("not everyone's [code] is [perfect]" [wp:25361]) but

[32] [wp:wp_markjacquith.txt]

it ran dry without any conclusions. ([wp:25326], [wp:25255], [wp:25362], [wp:25366])

Observation 1 ("unskilled developers") supports the problem explained in the concept *Missing Interface*. Plug-in developers are seen as not skilled for secure programming. Not having a safe interface for them puts WordPress as a whole at risk. Furthermore, developers argued against changing the "interface" for plug-ins because it would force the plug-in developers to update their plug-ins. [wp:25385]. This is also related to the concept of *Missing Interface* because the reverse dependency of WordPress on the support for plug-ins hinders the advancement of WordPress development.

Missing Interface →p. 95

Observation 2 ("raw SQL is not bad") highlights the basic problem of SQL in web applications. Although there is nothing intrinsically wrong with SQL as a data access language, including user input into the construction of statements is dangerous and should be done in a controlled fashion. Even if WordPress core developers are really skilled, errors are still possible and can be better contained if construction of SQL statements only happens in a limited number of places. I do not promote this observation to a concept as there is too little evidence as to what these arguments were grounded on.

I call the concept behind observation 3 ("abstractions hold your hand") *Fear Of Loss Of Power*. This might sound a bit drastic, but WordPress developers seem to be unwilling to adopt methods that would limit access to WordPress internals. This might fit into the WordPress development model because there is no explicit system design which codifies which functionality WordPress provides to third parties and where the boundaries are. Without a common boundary definition, there is no way to communicate what is supposed to be accessible from outside of the core and what is not. The issue with Fear Of Loss Of Power is that it prevents the introduction of an application core that is only accessible through an API and thus manifests tight coupling between components of the application.

Fear Of Loss Of Power →p. 97

Observation 4 ("does not improve security") is based on the notion that appropriate methods to prevent SQLIA and XSS are already available to developers and are in actual use inside WordPress. This is true, the custom `prepare()` statement, type-casting and the use of filters for data goes a long way to prevent vulnerabilities. The problem, as also was one of the starting points for trying to introduce process innovations, is that these methods are not applied consistently (see also the discussion on WordPress source code earlier in this chapter). This is also exemplified by a suggestion to just file an issue in the tracker for missing sanitation [wp:25384] in a certain spot. This does not solve the fundamental problem that sanitation can be easily forgotten. The WordPress project does not have a comprehensive vision and approach for security in the project. There is a good document on how to do data validation on the WordPress web site [Wor09a], but it concentrates on the technical methods available for SQLIA and XSS prevention in WordPress and only says that data validation has to take place both on input and output. It fails to mention where input and output sanitation should actually happen. This surely leads to *Inconsistent Use Of Sanitation*.

Inconsistent Use Of Sanitation →p. 95

Missing Data Modeling →p. 97

The reasons behind observation 5 ("more complicated") might be the *Missing Data Modeling* and not enough exposure to abstraction methods by the developers. Some developers on the mailing list made a case for the simplicity of SQL abstractions but they seem to be in the minority. Abstraction is more difficult when there is no unified data model and so this seems to be the main reason for developers to consider abstraction more difficult.

Observation 6 ("frameworks can provide benefits") can be attributed to the *Structural Conservatism* concept. Some developers see the benefits of frameworks but the majority wants to keep the old structure.

Aftermath

Since I felt that my annotation suggestions did not have a clear enough vision for where it would lead WordPress, I followed up a couple of times, also giving examples of what the code would look like [wp:25374], which were met with some appreciation [wp:25379], but to no avail.

My last message concerning the annotations to the WordPress mailing list, providing a version of the annotation patch with only the `method_exists` annotations and requests for feedback ([wp:25394]), remained unanswered.

It is not exactly clear to me why the innovation introduction failed. I clearly did not understand the WordPress development community well enough in the beginning. I did not expect the attempt to move SQL to a specific layer being questioned. It was also not clear (at least in the beginning) that WordPress was stuck with PHP 4 and this prevented the use of many useful libraries like HTML Purifier, real prepared statements and PDO.

From the concepts observed in WordPress, the following were most obstructive to the annotation introduction: *Missing Data Modeling*, *Legacy Constraint* and *Missing Interface*. Because of the missing modularization and thus missing abstractions in the then-current code, the community was wary about abstractions. Legacy constraints prevented the introduction of libraries that could have helped. The missing interface tied WordPress to an even stronger *implicit* interface (the whole core available to outside callers) and made changes very difficult. For conclusions and an assessment of the observed concepts, see section 4.5 at the end of this chapter.

In the final days of the discussion on the WordPress mailing list I received a personal message telling me that I might not get far with my approach and inviting me to work on Mambo, another GPL web application that was looking for help (see mail in section A.2 on page 119). I chose Mambo as the second project for an annotation introduction.

4.2.2 Mambo CMS

Mambo is the second project I approached with the annotation innovation developed in chapter 3. I did not originally include the project in the list of projects to look at, but was contacted by a project contributor who followed the discussion on the annotation introduction for WordPress and asked me if I wanted to help the Mambo project[33]. I accepted the invitation, especially because it sounded welcoming and showed a genuine interest.

Mambo is a popular CMS, written in PHP and released under the GPL. It features a history of being started as a commercial product, then made open source, being the target of legal threats, getting rescued by the original owners and later losing some of its core development team to the Joomla! project (which is discussed in section 4.3.1). Mambo was founded in 2000, being developed as open source software since 2004 [Wik09l].

Code Analysis

As of the time of writing, Mambo is at version 4.6.5. Mambo includes several third-party components in its sources, mostly small libraries concerned with input filtering (PHP Input Filter[34]), Really Simple Syndication (RSS) support (Magpie[35]) or caching (Cache Lite[36]).

Mambo supports plug-ins which are developed independently and add various functionality to the system. In Mambo, these plug-ins are called *mambots*.

The source code tree[37] differentiates between *components* (the main content containers in Mambo), *modules* (small structural elements that don't take user input, e.g. menus) and the *mambots* (plug-ins) that change the behavior of Mambo. The *administrator* subdirectory contains the *backend* of Mambo where an administrator can manage the Mambo site and its contents. The content pages visible to normal visitors to a Mambo site are called the *frontend*.

The Mambo source code is compatible with PHP from version 4.3 upwards and makes heavy use of classes to encapsulate the different content elements supported by Mambo.

Input Handling

The method mosGetParam()[38][39] does input handling in Mambo. The in-file documentation says it should be used to retrieve all parameters from GET/-POST and contents of cookies. The method supports specifying a default value

[33] [mambo:mos-lynne.txt]
[34] `http://freshmeat.net/projects/inputfilter/`
[35] `http://magpierss.sourceforge.net/`
[36] `http://pear.php.net/package/Cache_Lite`
[37] see the source code referencing section in the appendix, A.1.2 on page 114
[38] the "mos" part comes from *Mambo Open Source*, a name the project once carried
[39] Mambo `index.php`:78 (rev. 1739)

that is used if the value requested is not present and a bit-mask parameter that determines which sanitations should be used on the parameter. Trimming of whitespace, the removal of HTML (via strip_tags()) and the conversion of strings containing numeric values to proper integers are supported. By default, all three modifications are enabled.

While removing HTML from user input is a decent measure for values that should not (according to some business logic) contain HTML and trimming strings is useful for later comparisons, the purpose of the numeric conversion is dubious. Values are only converted if they actually contain a number, as determined by PHP built-in is_numeric()[40]. The conversion is only applied if the value was a numeric string beforehand anyway. It does not protect against injections because injection strings would not be modified (since they would not only contain a number).

Input parameters used widely (such as `task` or `section`) are extracted using mosGetParam() directly in `index.php` at the beginning of a Mambo run[41]. This is not a case of *Early Escaping* because the method is only applied to parameters that contain control values that determine what page is rendered. These parameters can only take a few well-defined values and may never contain HTML.

Since mosGetParam() uses PHP's built-in strip_tags() which removes entire HTML tags including the contents but does not gracefully handle broken HTML, the parameters expected to contain strings are additionally handled with htmlspecialchars(), another PHP built-in which does entity-encoding for characters with special meaning in HTML. Numeric values are cast to a proper integer format, removing injection opportunities.

Besides the generic input handling in `index.php`, every part of Mambo that uses input parameters is responsible for their appropriate sanitation. The distribution of sanitation responsibility between `index.php` and the actual "users" of the data does not seem to be clear: In the contents administration module, the `cid` parameter is retrieved through mosGetParam() which already had been retrieved in `index.php`[42].

This is just an example (there are probably more instances) and does not have any adverse effects, but it shows the difficulty of maintaining consistent input handling without a framework taking care of this or clearly codified rules.

Inconsistent Use Of Sanitation
→p. 95

Data Access Abstraction

Mambo only supports the MySQL database as a backend. The database is modeled as a class[43], as are tables[44] and rows[45].

[40] `http://de2.php.net/is_numeric`
[41] Mambo `index.php`:216 (rev. 1739)
[42] Mambo `administrator/components/com_content/admin.content.php`:22 (rev. 1488)
[43] Mambo `includes/database.php`:17 (rev. 1709)
[44] Mambo `includes/database.php`:806 (rev. 1709)
[45] Mambo `includes/database.php`:687 (rev. 1709)

Object Mapping Data objects correspond to a database row and are loaded and stored through their respective class, making the design similar to the *Active Record* pattern as described by Fowler [Fow02]. The code for object retrieval, creation and deletion is thus very generic and does not need any custom SQL. This part of Mambo corresponds roughly to the *model* layer in the MVC pattern[46]. Listing 4.10 shows an example of object retrieval in Mambo. Empty objects are constructed and later filled using their unique id. The database tables to consult are specified in the object's classes.

```
349 $menu = new mosMenu( $database );
350 $menu->load( $Itemid );
351 $pagetitle = $menu->name;
```

Listing 4.10: Mambo: `components/com_content/content.php`, (revision 1730)

Sanitation for the database happens in the save() functions of the respective objects, which also writes the objects back to the database. The content component[47] provides an example: The file `content.class.php` contains the class mosContent, representing a content entry (like a page) in Mambo. This class specifies a check() function that is used to validate and filter the data before writing to the database[48].

The check() method is not called by the generic save() function in mosDBTable, the base class of all data objects[49], but has to be called explicitly. Internally, the check() method for the content component (and for many others) uses PHP Input Filter, a filter against XSS[50]. Unfortunately, the filter uses a blacklist to filter out unsafe HTML, a known weak practice. Additionally, the parsing code is complex and may have non-trivial defects[51]. As the filter is used on output to the database, this also constitutes *Early Escaping* as it prevents XSS code from entering the database, which is unnecessary. For output towards HTML handling code has to be aware of whether the data has been processed with an anti-XSS filter before.

Use Of Inferior Method →p. 22
Early Escaping →p. 18

Other Database Access Only loading and saving of objects are abstracted generically into data objects. Business logic functionality (e.g. voting in a poll components) is encapsulated into a corresponding file with helper functions for all of Mambo's components[52]. For the content component, the file is called `content.php` and is located in the same directory as the class file. Functionality in these files makes heavy use of raw SQL, passing verbatim queries to the

[46] `http://c2.com/cgi/wiki?ModelViewController`, in the MVC interpretations where the model is only a data container

[47] in the Mambo `components/com_content` (rev. 1753) directory

[48] the default behavior is inherited from mosDBTable and mosDBAbstractRow contained in Mambo `includes/database.php` (rev. 1709)

[49] Mambo `includes/database.php`:974 (rev. 1709)

[50] Mambo `components/com_content/content.class.php`:221 (rev. 1699)

[51] Mambo `includes/phpInputFilter/class.inputfilter.php` (rev. 1)

[52] depending on your school of thought, this is either part of the *model* or the *control* layer in MVC

database layer. Often, the queries are quite complex[53]. Escaping of values for SQL never happens directly where the query string is concatenated, but usually a few lines earlier. As always with missing codification of who is responsible for sanitation, one cannot be sure if sanitation has really taken place. Arguments to functions are included into SQL strings without having been sanitized, relying on callers having made sure that data has been appropriately sanitized for SQL. This is an instance of *Inconsistent Use Of Sanitation* because there is no convincing documentation that clarifies where sanitation has to happen and would therefore render this implicit inter-function trust sensible. Note further that Mambo does not use any form of *prepared statement*.

Inconsistent Use Of Sanitation →p. 95

All occurrences of sanitation for SQL use getEscaped(), which is a method in the database class. It uses either mysql_escape_string() or mysql_real_escape_string()[54], depending on the version of MySQL available.

At last, a quirk of unknown origin: Mambo uses quotes for all variable insertions into SQL statements although strictly speaking this is incorrect for numeric types. This does not seem to cause any adverse effects and is accepted by MySQL.

HTML Output Encoding

User input is rendered by Mambo components. Components can contain all types of content Mambo supports. A component usually takes the main part of the screen estate, *modules* are arranged around the component contents according to a template file. Template files reside in the `templates` subdirectory, including images and CSS styles belonging to the respective theme. The actual template files contain an HTML page layout and calls to Mambo functionality. The template includes modules and the main body (filled by the *component*s) by calling Mambo functions which in turn include the appropriate parts.

Components each have a file that ends in `.html.php` which renders the respective component contents as HTML. These files consist of functions which mainly produce HTML with some PHP code interleaved. The PHP parts include variables and return values of functions. None of these values are encoded here. These files can be seen as the *view* layer in a MVC architecture.

The params variable is used for communication between the *controller* methods and the *view* methods. They are filled by the controller methods and used in the views to display the actual content (like titles or the main body text of a page). The data for the params values usually comes directly from the database. No intermediate filtering is done. As seen above, input filtering takes place for data put into the database, so data inside the database is seen as trusted[55]. This is a case of *Reliance On Input Filtering*. The Mambo developer manual contains a section on *Writing For Security*, which mandates to apply

Reliance On Input Filtering →p. 95

[53] see Mambo `components/com_content/content.php` (rev. 1730) for examples

[54] the version that handles character encodings correctly, see the discussion of PHP built-ins in chapter 2, section 2.2.5 on page 22

[55] One of the developers also stated that Mambo does all sanitation on input in [mamboIRC:2009-02-05]

htmlspecialchars ()[56] to strings before output, but that does not seem to apply for data coming from the database.

Innovation Introduction

Targeting Mambo for the second innovation introduction was a result of me being contacted by Lynne Pope, a member of the Mambo development team, as explained at the end of the WordPress episode in section 4.2.1 on page 66. After looking at the source code, I concluded that annotations to help with consistent sanitation of SQL and to HTML output could be useful for the project.

Mambo has a developer mailing list, but very little communication happens there. The Internet Relay Chat (IRC) channel *#mambo-cms* on the *Freenode* network[57] is mainly used for discussion amongst developers. Therefore, the interaction data I collected mostly consists of chat transcripts. The chat transcripts are split up by date and thus the references have date granularity. See section A.1.4 in the appendix for instructions on how to access the transcripts.

Episode On Annotations for SQL abstraction Like in the WordPress episode I started with annotations to reach a higher level of SQL abstractions. Mambo already had a higher level of abstraction than WordPress because they used an object-oriented approach and all the hand-written SQL was specific to a business object and not interweaved in web page output. Nevertheless, there was no use of prepared statements and third party extensions were allowed to send arbitrary SQL queries to the database, a case of *Missing Interface*.

Missing Interface →p. 95

On my first visit to Mambo's IRC channel I asked the developers if they considered the use of raw SQL a problem. Although they did not say that clearly, they obviously had not considered this and used SQL in many places [58]. In the second conversation I learned that the team considered the code of Mambo as quite old and was thinking about moving to CakePHP, a PHP framework[59]. This would implicate requiring PHP 5 for running the new version of Mambo, as opposed to Mambo 4.6, which also works with PHP 4. I also learned that the team working on Mambo was really small and had a chance to explain my annotation approach. As the chat channel was usually quite empty I only had the chance to talk to one developer, but he wanted me to go forward with the annotations for SQL and first concentrate on the *backend* code (the administration and content editing part of Mambo) because it had less exposure to reviews. I stated my plans to annotate structurally simple uses of SQL first, so that they could be moved to an abstraction layer and sanitation could be made consistent there.

Technology Improvement Planned →p. 98

Since the developers told me Mambo's backend would be the most worthy part to look at, I limited the annotations to that[60].

[56] a PHP built-in for encoding of HTML
[57] `http://freenode.net/`
[58] [mamboIRC:2009-02-04]
[59] [mamboIRC:2009-02-05]
[60] which is contained in the `administrator` subdirectory

The issue I was annotating was similar to the WordPress episode and so I chose the same *issueName*: `RawSQLUse`. I wanted to introduce an even simpler version of the annotation than the one I had used for WordPress, so I decided to use only one *effortIndicator*. The Mambo source-code did not contain cases where an abstraction was already available but not consistently used (as it had been the case in WordPress, see section 4.2.1). As I had announced in the chat session, I annotated code spots where a simple abstraction would suffice to get rid of many raw SQL uses and have them replaced by function calls to the abstraction layer. The *effortIndicator* I used was `trivial_implementation`, as in the WordPress episode. Since it was easy to find *all* cases of raw SQL use by simple string search, I marked all occurrences at once, even the ones that would not be trivial to change. These latter ones got an *effortIndicator* of `unclassified`, so it was easy to come back to them later. There was a total of 219 annotations classified as `trivial_implementation` and 107 `unclassified` ones[61]. Listing 4.11 shows an example of a `RawSQLUse` annotation with a `trivial_implementation` *effortIndicator*. Listing 4.12 shows an example of an annotation with an `unclassified` *effortIndicator*. The latter example is `unclassified` because it contains a LEFT JOIN, column wildcards, ORDER BY and LIMIT statements.

```
125 // @RawSQLUse, trivial_implementation, SELECT
126 $query = "SELECT name FROM #__sections WHERE id='$section'";
127 $database->setQuery( $query );
```

Listing 4.11: Mambo: `administrator/components/com_categories/admin.categories.php`, (revision 1754)

```
84 // @RawSQLUse, unclassified
85 $database->setQuery( "SELECT c.title, a.* FROM #__comment as a
       "
86    . "\n LEFT JOIN #__content AS c ON a.articleid = c.id"
87    . (count( $where ) ? "\n WHERE " . implode( ' AND ', $where
          ) : "")
88    . "\n ORDER BY a.id DESC"
89    . "\n LIMIT $pageNav->limitstart,$pageNav->limit"
90 );
91 $rows = $database->loadObjectList();
```

Listing 4.12: Mambo: `administrator/components/com_categories/admin.comment.php`, (revision 1754 (locally modified))

I used the *requiresFeatures* markers again, because they appeared helpful in the WordPress episode. Additionally, I added the string `CONCEPT` to some annotations. As explained in the *issue document*[62] I gave to the Mambo developers, I expected these cases to profit more from an "intentional" abstraction than a

[61] see the instructions section A.1.1 in the appendix for access to the actual annotations

[62] `http://mambo-manual.org/display/~mambocms@noroute.de/Annotations+for+raw+SQL` and [mambo:Annotations+for+raw+SQL.html]

literal one. A literal translation is one that just wraps the SQL query one-to-one, making the statement type the method name and columns and values the parameters of that method (e.g `select(column,value)`). In contrast, "intentional" abstraction hides the internal representation used in the database and interacts with the database in business logic terms (e.g. `getUser(id)`). I had made the proposition of "intentional" abstractions in the WordPress episode ([wp:25374]) and it attracted some interest. It also provides a much cleaner separation of concerns than just transforming the query into a method call.

After finishing the annotations, I created a patch that did not include the `unclassified` ones. I wanted to keep it really simple and the `unclassified` annotations might have been distracting to the developers. Since the Mambo team did not use the mailing list much, I put the *issue document* into the development wiki[63] and only sent an announcement to the list[64].

The reply[65] to that announcement by a project member agreed on the idea of creating a more robust design by the use of annotations and explained that the use of objects already encapsulates many cases of SQL use in the common base class mosDbTable and the derived classes of the respective objects. He argued that this already reduced the number of raw SQL uses and the annotations could be used to drive that number further down and make the use of SQL even more consistent[66]. In fact, Mambo already had something like a simple data model and architectural traits that made SQL injection handling easier (see the code analysis in section 4.2.2).

In the next chat session[67] I met some more members of the Mambo development team and asked them about the annotation idea. They also thought favorably of it. In the same session, developers were already talking about specifics of CakePHP, the framework that the next version of Mambo should be built upon. I had been told that it would still take months for the development to really start with CakePHP[68], but the developers were already seriously preparing the move. Another developer told me that after 8 years of development, the team felt that a general overhaul was necessary[69].

> *NOTE:*
> The chat transcript of 2009-02-11 was partly censored upon request of the Mambo project members. I respect their decision not to make certain statements public as they were intended for internal use only. The parts that were censored are clearly marked in the transcript.

After the very conservative notion of WordPress, the willingness of Mambo developers to modernize architecturally appeared to be a fundamental difference between the two projects and I pro-

[63] `http://mambo-manual.org/display/~mambocms@noroute.de/Annotations+for+raw+SQL` and [mambo:Annotations+for+raw+SQL.html]
[64] [mambo:mos-initial-sql.txt]
[65] [mambo:mos-initial-reply-andphe.txt]
[66] Ibid.
[67] [mamboIRC:2009-02-11]
[68] [mamboIRC:2009-02-09]
[69] [mamboIRC:2009-02-11]

structural conservatism
→p. 97

moted it to a concept. The concept is called *Structural Conservatism* and describes different levels of willingness to make fundamental changes to a project.

The developer also told me about the habari (see section 4.3.2 for complete discussion) web-logging application, which was started by some former WordPress developers because the WordPress project would not make some (as they felt) long-needed changes. In February I got an email saying that my annotations had been committed to the development branch for Mambo 4.6.

In the weeks after that, nothing happened with Mambo development. The development for Mambo 4.6 had been slow for some time. By the end of April 2009, there had only been 3 commits to the source code repository dating to 2009, including my annotations (!!!). I am not sure if someone will pick up the annotations and work on them. Following up and asking the developers why they think the annotation approach did not catch on would be a topic for further research.

Annotations for HTML encoding I had committed to follow up with annotations for HTML encoding in an early chat session[70]. After the annotations for SQL went into the source code repository, I made a basic proposal for what the annotations for HTML encoding would look like and the developer who also committed the earlier annotations agreed that the annotations would be useful[71].

I drafted an *issue document*[72] for the annotations, which stated that all variables which were part of an `echo()` call had to be sanitized directly in the `echo()` call. This proposal would interfere with a strict MVC architecture as it would introduce sanitation code in the *view*. Mambo uses *themes* for layouting, which do not directly contain variables, but arrange and call *components* (also see the code analysis in section 4.2.2). Annotations would only appear in the *components'* code which contains few layout instructions and is thus mostly static when redesigning a page. This approach is a bit of a compromise caused by the dependence on context for XSS prevention (cp. fundamentals in section 2.3.3 on page 32).

Mambo contains a huge amount of cases where variables are used in an `echo()` method, so I limited the annotations to a few files in the Mambo backend. I annotated only five *components* in the backend (there are 33 of them in a basic install of Mambo 4.6), which already produced 304 annotations. I chose the mnemonic *issueName* of `EncodeForHTML`. Since none of the sanitation methods that would be needed were present at that time and the effort for all types of sanitation would be similar, there was no need for an *effortIndicator*. Mambo used variables in different contexts (contexts relevant to how JavaScript is interpreted, again see fundamentals section 2.3.3 on page 32) so different types of sanitation would be needed. These were realized with the

[70] [mamboIRC:2009-02-05]

[71] [mamboIRC:2009-02-25]

[72] `http://mambo-manual.org/display/~mambocms@noroute.de/Consistent+encoding+for+HTML+against+XSS+attacks` and [mambo:Consistent+encoding+for+HTML+against+XSS+attacks.html]

requiresFeatures construct. The types I used are analogous to the different sanitations advised by the OWASP encoding guide [Wik09e]:

`plain` — `plain` was used for text appearing inside of HTML elements. There were 200 annotations.

`html` — `html` was used for function calls that did not return a simple value but a complex set of HTML by themselves. Sanitation is undefined for this case as a rule-set of allowed tags for each context would be necessary. There were only 5 of these cases.

`JavaScript` — `JavaScript` signifies a JavaScript value context. 64 annotations were made.

`attribute` — `attribute` indicates that the variable is part of a HTML attribute value and should be sanitized accordingly. There were 18 of these cases.

`URL` — `URL` shows that the value inserted is supposed to be part of a URL and accordingly needs special treatment (again, see section 2.3.3 on page 32 or the OWASP guide [Wik09e]). There were 14 cases.

`CSS` — `CSS` is used when the variable is part of a CSS expression. There were 3 cases.

Having this strict definition of what to annotate, it was easy to use a string search for `echo()` to find candidates for annotations. The classification was trivial, too, because I was dealing with the actual HTML template, where the context in which the data will appear is clearly visible. Listing 4.13 shows an example of a `plain` annotation.

```
289 // @EncodeForHTML, plain
290 echo $admin_comments_length; ?>" />
```

Listing 4.13: Mambo: `administrator/components/com_comment/admin.comment.php`, (revision 1711 (locally modified))

I put the accompanying *issue document* for the HTML encoding annotations into the Mambo development Wiki[73] again and sent an announcement email to the list[74]. To this day (April 14, 2009) I have not heard of the Mambo developers and it stays unclear what happens to the annotations.

[73] `http://mambo-manual.org/display/~mambocms@noroute.de/Consistent+encoding+for+HTML+against+XSS+attacks` and [mambo:Consistent+encoding+for+HTML+against+XSS+attacks.html]

[74] [mambo:mos-announce-html.txt]

Aftermath

The episodes for Mambo were more successful than the one with WordPress. The developers were more open to contributions and actually integrated the annotations into the code. Unfortunately, Mambo is a very small project, limiting the relevance of this partial success.

Furthermore, it remains unclear if someone will take on the annotations and put them to good use, but the annotation introduction for SQL consistency reached the third phase as defined in section 3.2.5 on page 51. See the discussion in section 4.5 on page 93 for the relevance of the concepts observed and chapter 5 for conclusions on what this means for the annotation approach.

4.3 Project Analysis

Following the innovation introduction attempts, this section provides discussion of additional open source web application projects. The discussion only considers the source code and public material from the web pages of the respective projects as well as a project questionnaire sent to the project communities, as lined out in the procedure descriptions in section 3.3 on page 51. This section provides additional grounding for the concepts discovered in the previous section and also adds new concepts.

No innovation introductions were attempted for these projects because all these projects had more good practices in use than WordPress or were at least planning to move to better practices. I did not see a big desire in these projects to use the annotation practice. In part, this is a consequence from the resistance I encountered in WordPress. In retrospect, I should probably have approached these projects more aggressively, motivating them to make substantial architectural improvements. Most of the changes planned by these projects were relatively minor and only enablers of future architectural change. Approaching web application projects more confidently to make projects consider more long-term improvements should be a focus for future research.

4.3.1 Joomla

Joomla![75] forked from the Mambo code base after policy-related disputes in 2005. This makes Joomla! interesting for a comparison with Mambo. It also is a web CMS and licensed under the GPL.

The fundamental feature set of Joomla! is similar to that of Mambo, only the *mambots* have been renamed to *plug-ins* (cp. section 4.2.2 on page 67). The Joomla! community seems larger (by the amount of community web pages available) and more active (looking into the Subversion repository[76], there are often multiple commits each day) than Mambo's.

The Joomla! community got me interested by having a few items in their issue tracker that were titled "Refactor XYZ"[77]. I expected a good fit for the annotation process and wrote an email to the development team to find out more[78]. I did not get any useful replies to my original question, but I was given an option to present my annotation idea[79]. Nobody expressed interest in the annotations either, but I was able to learn some details about the projects not documented on the web page. References are provided where appropriate.

[75] `http://www.joomla.org/`

[76] see section A.1.2 in the appendix for the repository used to retrieve the files discussed here

[77] e.g. `http://joomlacode.org/gf/project/joomla/tracker/?action=TrackerItemEdit&tracker_item_id=10747`

[78] [joomla:j-ref-1_refactor-initial.txt]

[79] [joomla:j-ref-3_presentation-annot.txt]

Code structure

The base directory layout looks very similar to the one used in Mambo, but the Joomla! project abstracted the framework part of the project into a library called Joomla! framework. The framework provides database access, data sanitation, session management, caching, authentication, etc. The Joomla! framework, along with all other external libraries resides in the `libraries` folder.

The *components*[80] use a much more visible MVC structure than Mambo. *Model* and *view* have their own subdirectory, called `model` and `view` respectively, with page templates for every view (which is done in multiple methods per file, which produce HTML in Mambo, see section 4.2.2 on page 70)[81]. While this results in the same functionality, Joomla!'s structure seems much cleaner and easier to read.

Input Handling

Joomla! has a defined method to access parameters, environment variables, or cookies in a safe way[82],JRequest::getVar(). Similar to Mambo's mosGetParm() (see section 4.2.2) it does filtering and ensures the correct type of the input. The difference to Mambo is its consistent usage. All functions that use parameters call JRequest::getVar() for every parameter in the first lines of the function, storing the result in local variables[83]. Sticking to this behavior enables quick assessment what kind of input filtering was applied for developers reading the code.

Database Access

Joomla! only supports MySQL as a database backend, but with both the mysql and mysqli drivers. mysqli seems to be included only because it is needed for newer versions of MySQL. Features for *Prepared Statements* in mysqli are not used in Joomla!.

SQL access is similar to Mambo: The model classes use raw SQL for their functionality[84]. Building the queries for loading and storing of business objects is done in the model files instead of the database layer, but this seems to be the consequence from completely separating framework and application (the framework is business object agnostic). The interesting difference between Joomla! and Mambo lies in the way SQL queries are constructed. In Joomla!, type conversion and escaping always happens directly where the string is constructed. An example is shown in excerpt 4.14. This behavior, which seems to be consistent in the Joomla! core makes code much easier to review since it is always clear where sanitation measures have to happen.

---

[80] which hold the contents of the different CMS elements

[81] see Joomla! `components/com_contact/views/contact/tmpl` (rev. 11772) for an example

[82] Joomla! `libraries/joomla/environment/request.php` (rev. 10919)

[83] see Joomla! `components/com_content/controller.php`:282 (rev. 11386) for an example

[84] see Joomla! `components/com_content/models/article.php`:438 (rev. 11646) as an example

```
126 $query = 'INSERT INTO #__bannertrack ( track_type, banner_id,
        track_date )' .
127         ' VALUES ( 1, '.(int) $item->bid.', '.$db->Quote(
              $trackDate).' )'
128         ;
```

Listing 4.14: Joomla!: `components/com_banners/models/banner.php`, (revision 11393)

HTML Output Encoding

HTML output encoding seems largely unchanged from the original Mambo source, apart from the clean split of the view into multiple template files. This means that the template files themselves do not provide any escaping or sanitation functionality, and all of this is done in *view* code, preparing the data[85]. As argued before, this makes it impossible to know which context the data will be used in at the time of sanitation but provides a clean split between layout and business code. Unfortunately, there is not clear codification on what data may contain HTML and which is safe for attributes (e.g.). Joomla! has sanitation functionality for both normal HTML element context and for attribute context, but as I was told by the developers, developers tend to disagree on what may contain HTML[86]. This qualifies as *Inconsistent Use Of Sanitation*.

Inconsistent Use Of Sanitation
→p. 95

In addition, the actual sanitation for HTML is quite complex and (as in Mambo) uses blacklisting[87].

Further Observations

In answer to my original request about the refactorings I was told that Joomla! does not have codified best practices on where and when to do sanitation[88]. This potentially leads to more *Inconsistent Use Of Sanitation*. A relatively consistent approach to sanitation seems to be in place, but it looks like these best practices are not codified explicitly and are passed on by the developers orally[89].

Use Of Inferior Method
→p. 22

Inconsistent Use Of Sanitation
→p. 95

In the discussion about the refactorings a developer asked me if I wanted to use my ideas about security for a Google Summer Of Code[90] project[91], where Google sponsors students to work on open source projects. The proposal was not concrete at all, but it sounded like at least part of the Joomla! community would be in favor of stronger codification of sanitation.

[85] see Joomla! `components/com_contact/views/contact/view.html.php` (rev. 11393) as an example

[86] [joomla:j-ref-5_sanitation-reply.txt]

[87] Joomla! `libraries/joomla/filter/filterinput.php` (rev. 11324) and Joomla! `libraries/joomla/filter/filteroutput.php` (rev. 10707)

[88] [joomla:j-ref-5_sanitation-reply.txt]

[89] as exemplified in [joomla:j-ref-5_sanitation-reply.txt]

[90] `http://code.google.com/soc/`

[91] [joomla:j-ref-7_gsoc-offer.txt]

Additionally, the Joomla! community announced in 2008 that starting with version 1.6, Joomla! will require PHP 5[92]. They state that the switch does not mean they will refactor Joomla! immediately to make use of the new features (prepared statements would then be possible in all supported configurations), but it is a move that opens up the possibilities to use state of the art methodology.

Technology Improvement Planned
→p. 98

Summary

Although the Joomla! project does neither use data modeling to enable the framework to deal with type-specific validation and filtering nor explicit codification of sanitation rules, they reach high level of consistency in data handling. XSS is still a big problem, though, because there is no sanitation process all data passes through.

The Joomla! case also shows that data modeling and modularization play an important role to enable powerful mitigations for SQLIA and XSS. The separation of the Joomla! framework and the application does only expose a well-defined API to both the Joomla! core application and the third party additions, since both act as equal consumers. The simple data model and use of the MVC pattern support clear separation of concerns. Combined with the predictable code structure achieved by explicit (for SQL) and implicit (for input filtering) rules, it is apparently possibly to reach a tolerable level of robustness against SQLIA and XSS even without a comprehensive data model.

The introduction of prepared statements and the introduction of a well-defined filtering layer could make the application even more robust.

[92] see `http://developer.joomla.org/coordinator-blog/75-joomla-goes-php-5.html`

4.3.2 habari

habari[93] is a web-logging software released under the Apache License 2.0 [Wik09f] license, which is approved as an open source license by the OSI (cp. section 1.2 on page 2 in the introduction). Its initial release dates to April 2007. The project explicitly states that it is targeting "modern web hosting environment"[94] and was designed "with a firm understanding of the current state of blogging"[95]. The Frequently Asked Questions (FAQ) section of the web-site also explicitly mentions an object-oriented data model and *Prepared Statements*[96] as some core technologies that are in use in the project.

Prepared Statement
→p. 19

I was pointed to the habari project by a Mambo developer in a chat session (see the Mambo aftermath in section 4.2.1 on page 66). The design of the application, which is in clear contrast to platforms like WordPress, sounded appealing for a closer analysis. The habari project also points out in their FAQ that forking an existing open source web logging application had not been an option since none of them had been mature enough. This looks like a deliberate side blow in the general direction of WordPress, even more so when considering the cue of a Mambo developer that habari would not exist of it had not been for WordPress (cp. section 4.2.2 on page 76).

habari requires PHP 5.2 and supports MySQL, PostgreSQL[97] and the serverless SQLite[98] as databases. It enforces UTF-8 encoding throughout the application and uses PHP's multi-byte extensions, so it should not be vulnerable to attacks that exploit encoding mismatch (see section 2.2.4 on page 17 for details on multi-byte issues).

Code Structure

The parts of habari's source code (see section A.1.2 in the appendix on how to access the code used in here) which are interesting here reside in the `htdocs/system` subdirectory, which includes all the PHP files used in the application, excluding third-party code. The business object classes of the application reside in the `classes` subdirectory therein[99]. There is no (folder based) separation of the actual business classes and classes concerned with databases, actions or authentication.

Plug-ins habari supports plug-ins through a hook-based approach. The habari code contains calls to named hooks for all operations. Plug-ins may register *filters* or *actions* that are then executed through the hook. *Actions* can perform

[93] `http://www.habariproject.org`
[94] `http://wiki.habariproject.org/w/index.php?title=FAQ&oldid=3076`
[95] Ibid.
[96] Ibid.
[97] `http://www.postgresql.org/`
[98] `http://www.sqlite.org/`
[99] habari `htdocs/system/classes` (rev. 3483)

a certain function, when triggered (like a notification, when a blog post is published), *filters* are also allowed to modify the data involved with the operation (like re-formatting the text of a blog post). Hooks are the only way for plug-ins to interact with habari and provide a clean interface. Note that this approach does not allow plug-ins that place *widgets*[100] on the application's pages, as is supported by other web-log applications[101].

habari uses a thin controller that dispatches each request to an *ActionHandler*[102]. *ActionHandler*s implement all business logic (they constitute the actual *controller* layer in the MVC pattern). The particular *ActionHandler* is chosen by parsing the URL of an incoming HTTP request[103].

For each business entity (posts, comments and the like), there are classes named after the singular and the plural of the respective entity (*Posts*, *Post*, etc.). The singular versions model single instances of a business entity including the actual data and functionality for saving and deletion[104]. These correspond to a data access object [Wik09i]. The plural versions support querying for instances of a certain class using different criteria[105].

Fetching or persisting of objects is done using custom raw SQL queries for each object[106]. There is no global data model that includes relations between the different objects, which would enable an abstraction for these operations.

Missing Data Modeling →p. 97

Input Handling

habari uses so-called SuperGlobals[107] which filter the HTTP request parameters in place. Developers accessing these parameters automatically profit from the input filtering done by SuperGlobals. As stated by an habari developer (in response to my project questionnaire), this is also helpful for plug-ins, which are able to use the "standard" PHP way of accessing parameters[108]. HTTP parameters are all filtered, regardless of content. There is no notion of types for input filtering. Developers have to explicitly request an unfiltered version of the parameters if they need them. This favors consistency which can prevent many cases of SQLIA and XSS attacks. Unfortunately, habari tries to filter for "any use"[109], leading to potential problems with *Early Escaping*. However, input filtering in habari uses extensive white-lists and even uses a custom HTML tokenizer to thoroughly filter HTML contents. The project could use HTMLPurifier (see section 2.3.3 on page 32), gaining well- maintained filtering capabilities, but apparent license incompatibility (HTMLPurifier is licensed under the Lesser General Public License (LGPL) which would technically allow the use inside a non-GPL application, but habari developers seems to be

Early Escaping →p. 18

[100] `http://codex.wordpress.org/Plugins/WordPress_Widgets`
[101] e.g. WordPress
[102] example: habari `htdocs/system/classes/userhandler.php` (rev. 3421)
[103] habari `htdocs/system/classes/rewriterules.php`:19 (rev. 3421)
[104] example: habari `htdocs/system/classes/post.php` (rev. 3421)
[105] example: habari `htdocs/system/classes/posts.php` (rev. 3421)
[106] example: habari `htdocs/system/classes/posts.php`:54 (rev. 3421)
[107] habari `htdocs/system/classes/superglobal.php`:29 (rev. 3421)
[108] [habari:habari-que-2-ans1.txt]
[109] Ibid.

wary of GPL code[110]) keeps habari from using the library. Licensing issues can keep open source project from using best practice methods, so I promoted this finding to a *concept*: *Licensing Issue*.

Licensing Issue →p. 98

Validation is not provided by the input layer because it is completely type agnostic. I have not seen any extra validation of input parameters. In habari, SQL statements which cannot be inserted using prepared statement place-holders and have to be validated explicitly[111] are never derived directly from user input. In this case, no further validation is necessary for SQL.

Database access

Database access always happens with the help of PDO, the PHP Data Objects. PDOs provide support for *Prepared Statements*, which is used throughout the application. There are many cases of concatenated raw SQL throughout the code, but all occurrences use the place-holder notation ("`?`"). Excerpt 4.15 shows an example of SQL use in habari.

Prepared Statement →p. 19

```
247 // let's make sure we only insert an integer
248 $internal = intval( $internal );
249 DB::query( 'INSERT INTO {poststatus} (name, internal) VALUES
        (?, ?)', array( $status, $internal ) );
```

Listing 4.15: habari: `htdocs/system/classes/post.php`, (revision 3421)

Although type casting seems to be done close to the string concatenation for SQL queries (see the example) in most cases, this is not a codified practice and not followed in all cases, leading to hard to review code and qualifies for *Inconsistent Use Of Sanitation*.

Inconsistent Use Of Sanitation →p. 95

HTML Output Encoding

The input layer already filters for XSS with a white-list (see section 4.3.2 above). This way, habari prevents all use of scripting through user input (HTML tags for formatting still pass the white-list). Data is output in so-called *theme* files, which determine the layout of the actual pages in habari. In these files, dynamic data is acquired from object attributes which were filed by the respective handler before[112]. There is no output filtering in the templates. While the input filter may prevent reflected XSS attacks (see XSS classification in section 2.3.4 on page 35), this is a case of *Reliance On Input Filtering* as it ignores type 2 XSS exploit code already contained in the database.

Reliance On Input Filtering →p. 95

Summary

habari shows many good approaches to SQLIA and XSS prevention. The use of an MVC-based architecture, PDO for prepared statements, the use of a mandatory input filter and white-listing support the mitigation of SQLIA and

[110] [habari:habari-users-license.txt]
[111] such as table names or LIMIT, see 2.2.4 on page 17
[112] example: habari `htdocs/system/themes/mzingi/entry.single.php` (rev. 3421)

XSS effectively. The ability to use a white-list is a direct consequence of up-front planning for data types and usage patterns. Media types other than text are supported through the use of so-called *silos*, which enable the inclusion of videos or pictures and are isolated from the habari core. This way, the core does not have to include (complex) white-listing for all conceivable media types (which would make the use of white-lists to cumbersome at some point).

Only the missing support for a validation layer, the reliance on input filtering, the inability to use HTMLPurifier, and mostly non-existent documentation[113] give reason to think about improvements.

Note that habari and WordPress are not directly comparable, although they share use cases. habari was designed to be more minimal and does not support the use of widgets or large modifications of the behavior, like WordPress does. Whether this is a disadvantage for habari depends on the requirements for the specific use case.

Addendum After I finished my analysis of habari, a few additional messages showed up in the thread initiated by my project questionnaire (see section 3.3.1 on page 52). Developers suggested habari should use the CakePHP framework in future releases[114]. Although probably partly in jest, I would consider this a welcome addition to add more consistency to sanitation in the habari project.

[113] also mentioned in a questionnaire response: [habari:habari-que-2_ans1.txt]

[114] [habari:habari-que-5_ans4.txt], [habari:habari-que-7_ans6.txt],[habari:habari-que-9_ans8.txt]

4.3.3 phpBB

phpBB[115] is a popular web forum application in PHP, licensed under the GPL. The original release appeared in 2000, the current version is phpBB 3, which underwent significant refactoring since version 2 and was published at the end of 2007. Prior versions were infamous for security problems. The code for version 3 received a security audit by PHP security expert Stefan Esser of SektionEins[116]. phpBB supports multiple database backends, including MySQL and PostgreSQL. The current version of phpBB requires PHP 4.3.3.

phpBB supports so-called MODs [sic!] which modify the original code to add new functionality or change the behavior of the original application. It does not support plug-ins. phpBB uses a custom HTML-based template language for forum posts and can thus avoid allowing any HTML in user input.

The phpBB community seemed friendly but not very open. The channel for communication with developers is the project's IRC channel. The forums on the web page are mostly concerned with helping users. There is no public mailing list. My requests in IRC about where to post the project questionnaire prompted a core developer to offer responding to the questionnaire by himself. Unfortunately, I did not get any replies. This is why I don't have any project questionnaire feedback for phpBB.

Code Structure

The actual phpBB code is located in the root directory of the source tree. Utility code used by multiple files is located in the `includes` directory, the templates are located below the `styles` directory, in the directory for the respective theme.

For every function phpBB supports (viewing a forum[117], viewing a topic[118], etc.), there is a specific "handler" file. Common code to all requests is situated in `common.php`[119]. Since business entities are implicitly defined through the actions which are supported by phpBB, there is no data modeling.

Missing Data Modeling →p. 97

Input Handling

There is no centralized input sanitation. Methods accessing request parameters need to use the request_var()[120] function to retrieve variables. request_var() takes a default value that acts as the return value if the named parameter is not found and also determines the return type if the parameter *is* found (by type-casting to the type of the default value). HTML escaping is done by the

[115] `http://www.phpbb.com`
[116] `http://www.sektioneins.de`
[117] phpBB `viewforum.php` (rev. 9003)
[118] phpBB `viewtopic.php` (rev. 9138)
[119] phpBB `common.php` (rev. 8760)
[120] phpBB `includes/function.php`:63 (rev. 9153)

function if the parameter is a string. This is sensible for phpBB as it does not support HTML user input. It uses its own mark-up language, called *bbcode* that provides very limited formatting and uses square brackets instead of angle brackets (`[i]foo[/i]`).

As the input sanitation in phpBB does not imply any specific use of the data afterwards (removing HTML by default is well-grounded if the application itself relies on all user input not containing HTML), there is no case of early escaping here.

Database Access

The database abstraction layer defines a common access class[121] from which the respective database drivers inherit[122]. SQL queries are concatenated from strings throughout the code. There is no support for prepared statements. Non-string variables used in the SQL queries are not validated or filtered when constructing the query string[123]. The developers rely on the type conversion and filtering done by request_var(). This is reasonable as the calls to request_var() can always be found either at the top of the respective function or (if not inside a function) at the head of the file. For strings, the sql_escape[124] method is used, which does database-type specific escaping.

Although there is no documentation as to where the use of request_var() should happen, the coding guidelines advise developers to sanitize all potential user input, return values from functions and parameters given to a function[125]. It looks like common sense is enough to provide a consistent coding style, making it easy to look up which sanitations were applied to a variable.

HTML Output Encoding

Pages and page snippets are generated using templates, regular HTML files[126] with keywords that provide control statements or variable insertions. The template files do not contain any PHP code or filtering directives. Escaping for HTML is done in the (in MVC terms) "controller" classes[127] that fill objects with attributes which are later used in the templates. Short examples are provided by excerpts 4.16 (defining values) and 4.17 (use).

```
560 // Send vars to template
561 $template->assign_vars(array(
562     'FORUM_ID'              => $forum_id,
563     'FORUM_NAME'    => $topic_data['forum_name'],
```

Listing 4.16: phpBB: `viewtopic.php`, (revision 9138)

[121] phpBB `includes/db/dbal.php` (rev. 9178)
[122] MySQL for example: phpBB `includes/db/mysql.php` (rev. 8815)
[123] see example at phpBB `viewtopic.php`:60 (rev. 9138)
[124] phpBB `includes/db/mysql.php`:310 (rev. 8815)
[125] `http://area51.phpbb.com/docs/coding-guidelines.html#general`
[126] e.g. phpBB `styles/prosilver/template/index_body.html` (rev. 8479)
[127] phpBB does not actually follow the MVC pattern

```
42 <fieldset>
43   <input class="inputbox search tiny"  type="text" name="
         keywords" id="search_keywords" size="20" value="{
         L_SEARCH_TOPIC}" onclick="if(this.value=='{
         LA_SEARCH_TOPIC}')this.value='';" onblur="if(this.value
         =='')this.value='{LA_SEARCH_TOPIC}';" />
44   <input class="button2" type="submit" value="{L_SEARCH}" />
45   <input type="hidden" value="{TOPIC_ID}" name="t" />
```

Listing 4.17: phpBB: `styles/prosilver/template/viewtopic_body.html`, (revision 9136)

The keywords used for variables (e.g. `L_SEARCH`) contain the type of data they represent in their name. `L` is for *language*, signifying a localized string. There are markers for URLs, strings and (static) JavaScript code. This helps the template developers to see where it is safe to use a certain variable[128].

The functions filling the keywords used in the templates do not always encode the values for HTML. This makes the application vulnerable to existing XSS in the database and is an instance of the *Reliance On Input Filtering* and *Inconsistent Use Of Sanitation* concepts.

Reliance On Input Filtering →p. 95
Inconsistent Use Of Sanitation →p. 95

Summary

phpBB's simple structure of task-based files makes the sources easily navigable. The template engine is an elegant solution to decouple design and functional development. However, there are readily available template languages for PHP which could have been used. The input filtering mechanism and HTML sanitation are also custom developments for phpBB. The missing data model puts validation and filtering in the hand of developers, requiring lots of error-prone boilerplate code.

The project would definitely benefit from a framework and a data model.

[128] e.g. not using a keyword that does not start with a U (for URL) inside a `href` attribute

4.3.4 Zikula

Zikula[129] calls itself an application framework. It re-launched the popular PostNuke CMS in 2008[130]. It is developed in PHP, is licensed under the GPL and supports multiple database backends through the ADOdb database abstraction library. PHP version from 4.1.0 up are supported.

The Zikula community develops the Zikula framework and a number of modules, which provide the actual functionality. This is called the Zikula core. The core is augmented by third-party modules which provide additional content types for Zikula.

Code Structure

The Zikula code cannot deny its PostNuke legacy. The directory structure looks similar to PostNuke's and the prefix *pn* for variables or functions still appears throughout the code.

The actual framework resides in the `system`[131] subdirectory, the content modules are placed in `modules` and the themes for the page layout reside in `themes`. The `includes` directory (as it seems standard for PHP web applications) contains libraries included in the distribution and utility functionality.

Input Handling

Input parameters are accessed through the getPassedValue() method of the FormUtil class[132]. getPassedValue() removes escaping if PHP's magic_quotes_gpc() is turned on (cp. section 2.2.5 on PHP's false friends) and filters for unwanted input using black listing. Zikula considers HTML script, object, applet and framing tags undesirable. This filter primarily targets XSS, but only filters literal JavaScript in HTML elements. JavaScript for attributes or in link targets is not filtered here. This clearly is a case of *Early Escaping* because it is unknown if user input would be used inside attributes or URLs.

Use Of Inferior Method →p. 22
Early Escaping →p. 18

Database Access

Zikula uses the ADOdb library for database abstraction. Therefore, in principle all databases supported by ADOdb could be used with Zikula. At the moment, the project only supports MySQL, PostgreSQL and Oracle[133].

Access to the database always happens through the DBUtil[134] class, which provides utility functions to retrieve table rows marshalled to objects, escaping

[129] `http://zikula.org/`
[130] `http://www.postnuke.com/module-Content-view-pid-6.html`
[131] Zikula `system` (rev. 25745)
[132] Zikula `includes/FormUtil.class.php`:35 (rev. 24908)
[133] `http://www.oracle.com/database/index.html`
[134] Zikula `includes/DBUtils.class.php` (rev. 25453)

data for use in queries and helpers for the construction of SQL query strings. For some object types, the database access is further abstracted through utility classes which provide direct retrieval of an object instance by different criteria[135].

Zikula does not use data modeling for relations between the different business objects and lacks prepared statements. There is no uniform way to construct SQL queries and to include unsafe data into it. The Zikula DataUtils provide an escaping method for strings that are used in queries (formatForStore()[136]) which unfortunately only uses addslashes() instead of a database-specific method (see explanation of addslashes() in section 2.2.5 on page 22). For numeric values, type-casting is used, but developers don't seem to agree on *when* type-casting has to happen. In most cases, type-casting happens when a value is first fetched via getPassedValue() and no further escaping or validation is done[137]. Sometimes, type-casting is done on string concatenation for SQL queries[138]. This is consistent with the answers to the project questionnaire[139] which stated that there was no documented behavior for sanitation[140]. This is an instance of *Inconsistent Use Of Sanitation*.

Missing Data Modeling →p. 97
Use Of Inferior Method →p. 22
Inconsistent Use Of Sanitation →p. 95

HTML Output Encoding

Data for inclusion in the Zikula templates comes from object attributes set by the "controllers" for the specific action (which reside in the `pnuser.php` files of the respective module). An example is shown in excerpt 4.18.

```
55 // Assign the config vars
56 $pnRender->assign('enablecategorization',
       $enablecategorization);
57 $pnRender->assign('shorturls', pnConfigGetVar('shorturls'));
58 $pnRender->assign('shorturlstype', pnConfigGetVar('
       shorturlstype'));
59 $pnRender->assign('lang', pnUserGetLang());
60
61 // Return the output that has been generated by this function
62 return $pnRender->fetch('pages_user_main.htm');
```

Listing 4.18: Zikula: `modules/Pages/pnuser.php`, (revision 25012)

The template files used in Zikula are based on the Smarty[141] template language for PHP. Smarty provides access to variables made available to the template in a PHP-like syntax and supports filtering of output data with the `"|"` notation. In excerpt 4.19, the $content variable is filtered for HTML contents.

[135] e.g. Zikula `includes/CategoryUtil.class.php`:25009 (rev.)
[136] Zikula `includes/DataUtil.class.php`:271 (rev. 25016)
[137] e.g. Zikula `modules/Pages/pnuser.php`:77 (rev. 25012)
[138] e.g. Zikula `modules/Topics/admin.php`:179 (rev. 24342)
[139] for instructions retrieving this data, see A.1.3 on page 115 in the appendix
[140] [zikula:zikula-queger-6-englishque-ans.txt]
[141] `http://www.smarty.net/`

```
34 <div class="pages_page_body">
35     <!--[$content|pnvarprephtmldisplay|pnmodcallhooks:"Pages"
           ]-->
36 </div>
```

Listing 4.19: Zikula: `modules/Pages/pntemplates/pages_user_display.htm`, (revision 24588)

While filtering in templates is often used, I could not find a simple pattern of what values had to be filtered in the template. In some cases, values were already returned from utility functions in sanitized form[142]. This is another case of *Inconsistent Use Of Sanitation* which makes it hard to read the code and get secure sanitations applied in all cases.

Inconsistent Use Of Sanitation →p. 95

Further Observations

Zikula features a lot of *Code Reuse* from open source projects[143]: ADOdb, Smarty, SafeHTML and various AJAX libraries are used for the core functionality, additional libraries are used for specific features like RSS.

Code Reuse →p. 40

Furthermore, the road-map for Zikula 2.0 contains the use of a PHP ORM library, doctrine[144] and the replacement of SafeHTML by HTMLPurifier. This is a case of *Technology Renewal Planned* and shows willingness by the Zikula team to use best practices.

Technology Improvement Planned →p. 98

Summary

The Zikula project shows good approaches, especially by the extensive reuse of code and centralization of sanitation code in utility classes. However, Zikula is missing a comprehensive data model which makes consistent and abstract data handling difficult and results in unnecessary hand-written code. Moving to an ORM layer requires such a comprehensive data model and probably enables Zikula to become a really well-structured web application with state of the art technology support.

[142] e.g. Zikula `modules/News/pnuserapi.php`:389 (rev. 25471)
[143] `http://code.zikula.org/core/wiki/development/externalcomponents`
[144] `http://www.doctrine-project.org/`

4.4 Projects In Brief

Projects In Brief contains short discussion of three more projects. I included these projects because they represent very popular projects (Drupal and TYPO3) or are significantly different to the other projects presented here (riotfamily). All these projects also exhibit at least one concept I considered important for the summarizing discussion.

The discussion here only includes data that could be gathered by analysis of project documentation available from the web pages and a very short source code inspection.

4.4.1 Typo3

TYPO3[145] is an immensely popular, GPL-licensed CMS written in PHP. Its distinguishing feature is *TypoScript*, a configuration language that can represent arbitrary data structures for use in web page templates. TYPO3 can be used to create very diverse web sites but the flexibility comes with large complexity.

The current version, TYPO3 4, does not support prepared statements and a comprehensive data model. The latter is partially caused by the high configurability of the system, where the actual applications really only arise during customization.

Use Of Inferior Method →p. 22

Now, the TYPO3 community started a framework project, called Flow3[146], which will be the foundation of the next version of TYPO3, TYPO3 5. The framework boasts features like a persistence framework backed by Domain-Driven Design principles [Eva03, cp.], a full-fledged MVC framework for structuring web applications and an explicit validation layer, supported by rich domain objects. This constitutes a big technological and structural leap and is a case of large *Technology Renewal Planned*.

Missing Data Modeling →p. 97

Technology Improvement Planned →p. 98

4.4.2 Drupal

Drupal[147] is another popular CMS written in PHP. Like TYPO3, the current version of Drupal, Drupal 6, does neither have a comprehensive data model nor does it use prepared statements. A discussion on IRC revealed that the current database abstraction is completely unsuitable for the use of prepared statements[148]. The project will introduce prepared statements in Drupal 7, using the much-improved data abstraction layer currently under development, a large technological improvement for the project.

Missing Data Modeling →p. 97

Use Of Inferior Method →p. 22

4.4.3 Riotfamily

riotfamily, being written in Java, is the only non-PHP project in this thesis (for implications of that fact, see the discussion of validity and relevance in section

Technology Improvement Planned →p. 98

[145] http://typo3.org/
[146] http://flow3.typo3.org/
[147] http://drupal.org/
[148] [drupalIRC:2009-03-11]

5.2). riotfamily is licensed under the Mozilla Public License (MPL) and consists of a framework and different modules.

The most interesting features of riotfamily are the WYSIWYG editing feature of web pages and the possibility to use a rich, structured data in the business objects, which are fully customizable.

Data Modeling →p. 21

riotfamily is developed using the Spring framework, the Hibernate persistence layer, the FreeMarker templating language and Direct Web Remoting (DWR) for AJAX interactions, a case of major *Code Reuse*.

ORM →p. 21

Using these libraries enables the use of *Prepared Statements*, and consistent HTML escaping for a certain type throughout the application[149].

Code Reuse →p. 40

Although I did not do an elaborate code review of riotfamily, it looks like the project makes use of best practices and shows that it is in fact possible to avoid most of the adverse concepts by choosing a good modeling approach and mature technology.

Prepared Statement →p. 19

[149] `http://static.springframework.org/spring/docs/2.5.x/reference/spring.tld.html`

4.5 Concepts Observed

This section discusses the concepts discovered during the case studies. While the concepts covered in chapter 2 are all technical in nature and correspond to known best practices or anti-patterns, the concepts introduced here were distilled from the analysis of the source code and the interaction with the projects. None of these concepts are specific to the projects discussed in this thesis and may occur in many other (especially web) applications not discussed in this thesis.

Table 4.1 on the following page serves as a guide to look at a particular concept observed in a project. The table lists the page numbers where an instance of a concept is described inside the project discussion. Only concepts actually observed in at least one of the projects are included. Note that the concepts in this table are sorted alphabetically and only serve as a reference and must not be interpreted quantitatively. Concepts do not have equal impact on projects and not all of them are negative. See the discussion of particular concepts in section 4.5 on page 95 for details.

Concept	WordPress	Mambo	Joomla!	habari	phpBB	Zikula	TYPO3	Drupal	riotfamily
Code Reuse						90			92
Data Modeling									92
Early Escaping	56	69		82		88			
Fear Of Loss Of Power	65								
Inconsistent Use Of Sanitation	60,65	68,70	79,79	83	87	89,90			
Inferior Method	56,58	69	79			88,89	91	91	
Legacy Constraint	58								
Licensing Issue				83					
Missing Data Modeling	66			82	85	89	91	91	
Missing Interface	60,65	71							
Non-Uniform DB Access	57								
ORM									92
Prepared Statements				81,83					92
Reliance On Input Filtering	60	70		83	87				
Structural Conservatism	58	74							
Technology Renewal		71	80			90	91	91	

Table 4.1: Concept Overview

This table lists the page numbers where particular concepts appear in the respective projects. **Note that no information on the relative robustness against security vulnerabilities can be inferred from the information in this table alone.**

Concepts In Detail

Non-Uniform Database Access This concept describes a situation were there are multiple methods to access a database without a clear deprecation for all but one method. In such a situation, data being transmitted to the database can have undergone a variety of treatments. If sanitation against SQLIA or validation is done in the database access methods, it may be possible to bypass sanitation. With concurrent access methods, maintenance effort rises and developers may unintentionally use a method which is inappropriate for their use case.

A library that provides programmatic construction of SQL statements from simple method calls but also supports direct access through raw SQL access would be an example of this concept. In this case, there should be a clear project policy for which cases (if at all) direct use of raw SQL is allowed. *Missing Data Modeling* may cause the use of Non-Uniform Database Access, since in this case an application framework cannot be used to abstract database access (cp. figure 4.2 on page 98).

Inconsistent Use Of Sanitation The *Inconsistent Use Of Sanitation* concept describes the absence of a defined process to handle data sanitation on input or output. If sanitation can happen in multiple places or in different ways which are not completely specified, data in the system is in an undefined state (since it is unknown which sanitations were applied). This can lead to security vulnerabilities if necessary sanitations are (inadvertently) not applied or if the interplay of sanitations itself creates vulnerabilities.

Having explicit layers (often provided by application frameworks) can be used to provide consistent sanitation without relying on codified policy. *Missing Data Model* can lead to *Inconsistent Use Of Sanitation* because in that case sanitation has to be done by hand and can not be delegated to the framework (cp figure 4.2 on page 98).

Reliance On Input Sanitation *Reliance On Input Sanitation* describes the use of input sanitation without output sanitation. Relying on input sanitation is inappropriate if the input sanitation cannot (or should not because of the business logic of the application) sanitize for all cases of output or if there is a possibility of unsanitized data in the persistent storage of the application.

Having a comprehensive data model can enable automation of type-specific input and output sanitation and therefore make it more affordable for software projects to do both. *Missing Data Model* makes it more likely that either input or output sanitation is left out, because it has to be done by error-prone custom code (cp. figure 4.2 on page 98).

Missing Interface *Missing Interface* is the concept behind a situation where parts of an application that are used by a third party expose their internals. In this case, changing the application always involves changing

Figure 4.1: Separation of observed concepts into problem and solution space

the third party application, too. If there is not just a customer/supplier relationship (cp. [Eva03, pp. 356]) between the application and the third party (e.g. a plug-in), the development of the application can be severely impeded.

Fear Of Loss Of Power *Fear Of Loss of Power* labels instances of a social phenomenon where developers dismiss changes to their application which would disable functionality or specific ways to program the application. An example would be the introduction of an API which hides implementation details and thus also limits access to internals.

This becomes a problem when developers insist they should always be able to do *everything*. Limiting access leads to better separation of concerns and is especially important for parts of the application relevant for security. *Fear Of Loss Of Power* may hinder progress in projects, but the exact impact of the concept would have to be determined by further research.

Missing Data Model The concept *Missing Data Model* is found in applications where no explicit and comprehensive data model is visible. The data model has to define all business-relevant data types for the application and their relations centrally. Cases of *Missing Data Model* have to manually establish the type and handling of data wherever data is touched in the system. This makes consistent handling for every case burdensome and error-prone. *Missing Data Model* makes *Inconsistent Use Of Sanitation*, *Reliance On Input Filtering* and *Non-Uniform Database Access* more probable, since the abstraction of sanitation is not possible and has to be done manually (cp. figure 4.2 on the next page).

Structural Conservatism I found that the willingness to make necessary, fundamental changes radically differs between projects. This seems to be a very important factor for the acceptance of innovations, as exemplified by the annotations idea. Structural conservatism may influence a lot of the other concepts, since it may hinder or support refactoring towards a new architecture (perhaps establishing more *modularization* or making *consistent sanitation* possible) or control the readiness to live with legacy constraints. This concept may be influenced by *Fear Of Loss Of Power*.

Legacy Constraint A *Legacy Constraint* concerns situations when a project is kept from using a desired and known good technology because the technology is incompatible with the present version of another component used in the project and the project team is unable or unwilling to update this component.

A prime example in this thesis is the mandatory compatibility with PHP 4 in web applications, which bars the application from using features like a persistence layer such as PDO. This way, *Legacy Constraint* can be a major cause for e.g. *Use Of Inferior Method*, cp. figure 4.3 on the following page.

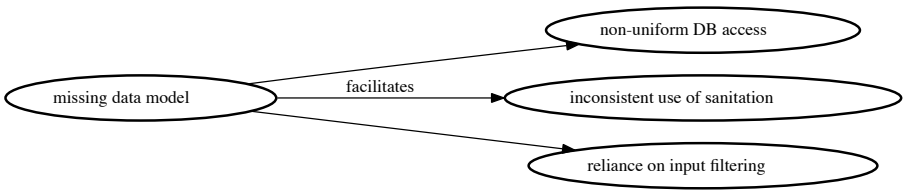


Figure 4.2: Possible consequences of *Missing Data Model*

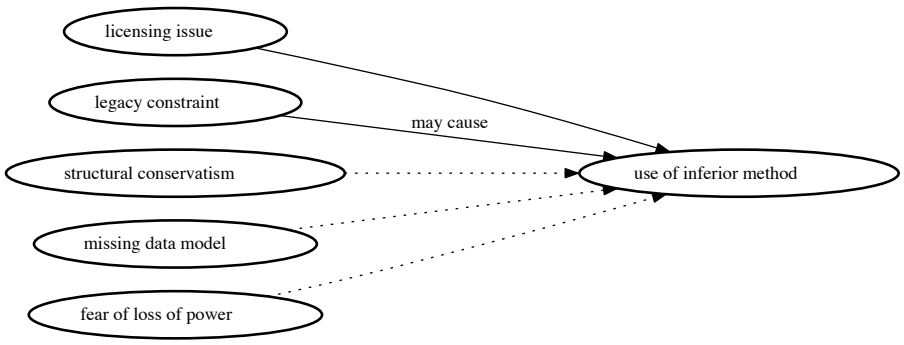


Figure 4.3: Possible reasons for *Use Of Inferior Method*

Licensing Issue Not all licenses of open source are compatible with each other. Especially "viral" licenses such as the GPL, which demands that software that incorporates code which was released using the GPL also has to be licensed under the same license, can bar other projects from using that code.

This is critical especially for libraries which then can only be used by GPL code, excluding projects that could make good use of the code otherwise. Therefore, most libraries are licensed under the LGPL, which does not have the "viral" characteristics of the GPL. Nevertheless, there seem to be provisos against the LGPL in projects that chose a "more free" library for political reasons (as exemplified by the habari project in section 4.3.3 on page 85). *Licensing Issue* can be the reason for *Use Of Inferior Method*.

Technological Improvement Planned *Technology Improvement Planned* labels the intent of a project community to make significant technological improvements which also have consequences for prevention of the vulnerabilities described in this thesis in the near future. The concept applies to upgrading a major component of the software, adding a new component, or restructuring the software towards better maintainability or a data model.

4.6 Assessment Of The Concepts

This section contains the concluding assessment of the concepts observed in the projects and assesses their influence on the prospects for the process innovation presented in chapter 3. The assessment will relate to research question Q3 about the applicability of the annotation innovation, presented in section 1.4.1 on page 4.

First of all, a project using comprehensive data modeling usually leverages the model to delegate sanitation tasks to a framework. It does therefore not exhibit the structural weaknesses abetted by *Missing Data Model* (cp. figure 4.2 and section 4.5). In this case, the annotations — targeting structural deficiencies — are of little use. Of the projects discussed in this thesis, only one, riotfamily, falls into that category (see section 4.4.3 on page 91).

Second, concepts that concern the inability of a project to make progress can be a major obstacle for the annotations. *Licensing Issue*, *Legacy Constraint*, *Fear Of Loss Of Power*, and *Structural Conservatism* fall into this category. While *Licensing Issue* only came up as an aside in the habari project (see section 4.3.2 on page 82), the other three, first and foremost *Structural Conservatism* seemed to be big inhibitors in the WordPress episode. Technical incompatibility (*Legacy Constraint*), combined with unwillingness of the project team to innovate (*Structural Conservatism* and *Fear Of Loss Of Power*) led to a stasis, which the annotation — requiring technological and process changes — could not break. Note that *Legacy Constraint* (the forced compatibility with PHP 4 that disabled the use of some useful libraries) was not the determining factor that the annotations could not be applied. Although not optimal, the changes lined out in the *issue document* would have been possible without the use of a newer version of PHP. The actual impact psychological concepts such as *Fear Of Loss Of Power* or *Structural Conservatism* could not be determined here. They require more in-depth research.

The episode with Mambo (cp. 4.2.2 on page 71) shows that the process innovation presented in this thesis can be adopted by a project with legacy software versions if the project's community is interested in having good architectural foundations.

Therefore, I think it is fair to say that the main reason for the refusal of the annotation proposal in WordPress (cp. section 4.2.1 on page 64) was not technical but accountable to the mind-set of parts of the WordPress community who value a somewhat working status quo above a re-design that would improve maintainability for years to come. The involvement of Matt Mullenweg and his company automattic, which is directly dependent on the number of users of WordPress, also seems to have influenced the decision not to do anything which could (temporarily) make the life of users harder. Missing compartmentalization in the WordPress code makes large changes especially difficult for the project.

It is not the technical limitations of projects that inhibit innovations, it is the project's community structure and its processes. It looks like more involvement into the project community could have helped the annotation approach. I did not implement any of the suggested changes and only provided the annotations

and instructions in form of an *issue document*. Apparently, these alone were not appealing enough for developers to work on my innovation. My personal interest for architecture and security infrastructure in applications is obviously not shared by developers in general. Therefore, future innovation approaches should include more commitment for actual implementation by the person doing the innovation introduction.

Chapter 5

Conclusion

> Science is always wrong. It never solves a problem without creating ten more.
>
> George Bernard Shaw

Following the argumentation in chapter 2, one can only conclude that while it is not trivial to prevent SQLIA and especially XSS attacks, highly effective mitigations exist, which only lack consistent application.

Innovations targeting the consistent application of preventive measures through architecture seem to work in principle. The failed innovation introduction for WordPress (section 4.2.1 on page 55) and the only partially successful one for Mambo (section 4.2.2 on page 67) show that the approach is regarded as useful by some developers. Now Mambo is a very small project, so the inclusion of the annotations into the project sources does not provide very valuable evidence. Also, no data on the actual improvement of security of web applications could be collected.

Nevertheless, I think the annotation approach is a sensible idea. Examples like the one in the WordPress episode (cp. section 4.2.1 on page 61) where issues in the issue tracker were not fixed because they appeared in big chunks, are an ideal match for the annotation technique. For the projects that planned technology renewal, the renewal either involved only a small technological change which would bring the project up to par with technology that has been considered as standard for new projects for some time; or it resulted in radical changes such as the foundation of a new project (habari, cp. 4.3.2 on page 81) or major rewrites (TYPO3, cp. 4.4.1 on page 91). In these cases, annotations could be useful to get projects to use evolutionary improvement instead of throwing away large parts of the source code. Literal implementation of the annotation introduction may not work, but, as lined out in the concept assessment in section 4.6 on page 99, more involvement of the innovator in the actual development could be used to increase the acceptance rate and lead to annotations that are actually used in projects. As the annotation idea appears to me as a powerful approach for architectural corrections, there should be more research to identify more concrete determining factors that can make this process successful.

As for the criteria of projects that can successfully be targeted by process innovations, the analysis in section 4.6 on page 99 hints that the willingness to

innovate in the community is much more important for a successful innovation introduction than the technical properties of the project. Unfortunately, this willingness is much harder to assess from outside the project. In the WordPress episode, I consulted the mailing list archives to get a feeling for how the community reacts to suggestions from outside. This kept me from unsubstantially suggesting the same things that pop up on the mailing list from time to time.

Nevertheless, the reservations in the WordPress community against practices they considered to complicated or destabilizing prevailed, and the annotations were refused. I consider it grounded to say that a strong presence of *Structural Conservatism* in projects will make the annotation introduction presented here unsuccessful. The criticism I received in the WordPress also points to the conclusion that low exposure to ideas of data modeling and architecture by the community can cause refusal of ideas that lead into this general direction. The other way around, the annotation approach should have a high probability of success in projects whose contributors are familiar with architecture and data modeling concepts. The partial innovation introduction would not have been possible if the Mambo project had not been interested in structural improvement of their application. For an innovation introduction to succeed, it always has to match the goals of the community at large. For Mambo, it did, and thus the annotations got included.

It is possible that the size of the projects played a determining role in this outcome. While the Mambo project was very small at the time I interacted with it (less than five active members, cp. section 4.2.2 on page 71), WordPress had numerous volunteers. The Mambo team seemed to be happy with all the help they could get. Also, the coordination process is much easier with a small project. Gaining support from one project member was enough to get the annotations into the project repository. With WordPress, three core developers supported my idea which was eventually shot down in the discussion on the mailing list.

Another important result of this thesis is the identification of *concepts* which seem to influence the architecture and technologies used in projects, and also their interaction. As lined out in the discussion of observed concepts, there are numerous concepts that inhibit innovation (*Fear Of Loss Of Power*, *Structural Conservatism* and *Legacy Constraints*). These concepts should be used in future research to better understand behavior of project participants. The definitions of the concepts in this thesis are quite limited because of the shallow insight possible with the projects at hand. These could be extended and refined by looking at other projects, specifically with these concepts in mind.

5.1 Encore: The State Of Open Source Web Application Security

The concept analysis of the nine open source web applications did not only deliver insights for innovation introduction approaches but also — with the best practices for SQLIA and XSS mitigation in mind — understanding of the state of open source web application security in general. While the MITRE

analysis mentioned in the introduction (cp. 1 on page 1) blamed the rise of XSS vulnerabilities on amateur developers writing toy software, the applications discussed in this thesis are written by large groups of reasonably experienced developers. Even though, they have their share of vulnerabilities.

In all nine projects, the recurring concepts have to do with inconsistencies (*Inconsistent Use Of Sanitation*, *Non-Uniform DB Access*, *Reliance On Input Filtering*, and *Early Escaping*). As already stated in the concept descriptions (see section 4.5, earlier in this chapter), *Data Modeling* is an appropriate way to minimize inconsistencies by delegating sanitation to a framework. Of the nine applications, only the Java-based riotfamily uses a third-party framework for a significant part of the application. Joomla! and Zikula have their own internal custom frameworks, but none of these applications use a general purpose framework (like CakePHP).

Apparently this is going to change, as indicated by the oft-mentioned plans to move to a proper framework (*Technology Renewal Planned* concept). Mambo wants to use CakePHP, habari considers the same (maybe partly in jest, cp. 4.3.2 on page 84) and TYPO3 will implement their own framework with support for current technologies. Joomla!, Drupal and Zikula will at least introduce new technology that makes the database layer consistent. For Zikula, this also means having an ORM implementation which requires a comprehensive data model. The only projects which are not planning any substantial improvement concerning their application architecture are WordPress and phpBB. This may be a reason for optimism for the future of open source web applications. However, all of these improvements are relatively minor and in the best case bring the projects to a level that would be expected from a newly implemented application. Maybe the annotation approach could be used to push projects forward more aggressively.

In summary, the following findings have substantial backing from the data gathered in this thesis:

1. The annotation method should be able to produce evolutionary architectural change in open source web applications, but its implementation has to be revised with more involvement by the innovator and a clearer vision.

2. Concepts like *Structural Conservatism* make the application of the annotation method very difficult.

3. Behavior of a project community can be better understood by applying concepts like the ones developed in this thesis to it.

4. The identified concepts help identifying projects unsuitable for the annotation approach.

5. Very few of the popular PHP web applications are thoroughly architected and use a comprehensive data model.

6. Most of the open source PHP web applications improve the technology used in the applications but only after they have been mature for a while and are considered compulsory.

7. Small projects can profit more from the annotation method than large projects unless the community of the large project really endorses the method.

8. The annotation method may help projects to integrate technological innovations quicker.

5.2 Validity And Relevance

The findings about the annotation method are mostly backed by the episodes with WordPress and Mambo, both PHP projects and very unequal. While WordPress is immensely popular and has lots of contributors, Mambo lives in the large shadows of Joomla! and has very few developers. Nevertheless, the ability to judge the usefulness of the annotation approach is not impeded by the small number of community members, and the Mambo project accepted the idea.

However, it cannot be denied that the actual introduction of the annotation approach may have to be totally different to reach projects bigger than Mambo. That's why this thesis cannot provide advice for a general approach to be used for the annotation introduction. The general finding that the annotation method is useful is therefore more an argument backed by rationality then by empiricism.

The concepts observed are valid and relevant given the definitions used in this thesis. The phenomena mentioned in the concept descriptions definitely exist in the projects discussed because they are based on the exact observations. There is nothing specific in the concepts that should limit them to the cases actually discussed in this thesis (e.g. to PHP projects). The prevalence of the concepts may be different for projects developed in other programming languages, but this does not affect the validity of the concepts. No evidence can be provided for the relevance of these concepts for projects using other programming languages than PHP, however, there is no reason to expect the concepts to be irrelevant for other environments. Further research should try to determine the exact relevance by applying the concepts to web applications in different languages.

General relevance is given because the concepts were developed within PHP projects, which are immensely popular for open source web applications.

5.3 Future Research

The research in done in this thesis could not definitely assess if the annotations will get used. The Mambo project is probably to small to exert big evidential leverage, and no other project included annotations into their source code. There is reason to believe that the annotations are beneficial for vulnerability prevention (see the conclusion), but the questions whether and how projects used the annotations is still open. This could be approached with another annotation introduction attempt with more commitment by the innovator.

More research is needed for really grounded answers as to whether the annotations improve security. This requires successful annotation introductions into projects and long-term monitoring of a web application project.

Another obvious direction for future research is to establish if the annotations can be used for prevention of other (web) application vulnerabilities. At some point, SQLIA and XSS will be largely under control, but new types of vulnerabilities appear permanently. Looking at OWASP's top ten [Wik09d], vulnerabilities like Cross-Site Request Forgery (CSRF) or *Insecure Direct Object Reference* look like they could be approached using structural or architectural means, probably a good fit for the annotation method.

The tree categorization provided by the CWE project (see section A.6 on page 128 with explanations on CWE in the appendix) also provides valuable pointers as to which vulnerabilities could be compatible with the annotation approach. In CWE's tree, both SQLIA and XSS are children of *Failure to Sanitize Data into a Different Plane (aka 'Injection')* [DVT09a]. Other children include LDAP injection, XML injection and general code injection. Since these share the same root problem (missing sanitation when changing execution planes), it is reasonable to expect the annotation approach to be useful for these vulnerabilities, too.

Going one step further up the CWE tree, *Improper Input Validation* is listed as the parent for the sanitation failure[TDV08]. The annotation approach presented in this thesis can probably also be used to enforce consistent input validation (either through the use of a framework or by codification), opening up a host of other vulnerabilities to target[1].

During the research I discovered concepts present in the projects which were not immediately obvious. These could use some more in-depth analysis of their impact, prevalence and consequences. Especially psycho-social concepts like *Structural Conservatism* or *Fear Of Loss Of Power* could have dire consequences on projects' ability to innovate.

There are more opportunities to build on the insight achieved during the making of this thesis, which were not to be foreseen. It became apparent while dealing with the different projects that the age of the applications seems to have a large influence on the maturity of the components used. While most projects discussed here have been around for some years, only the relatively young riotfamily, Zikula and habari score high in the domains of Code Reuse and the use of modern technology like ORM and strong object orientation. It would be interesting to see if projects of the same age and platform (programming language plus framework) adopt substantial technological innovations (like the availability of sophisticated modeling tools or an ORM library) at roughly the same time or if there are other significant factors involved, besides project age.

Then again, there seem to be differences between the development platforms. The riotfamily project exhibits significantly more best practices than all the other projects under discussion. riotfamily is a very recent project as is habari, but the platform used seems to be beneficial to the amount of technology available for use in the project, too. Java had prepared statements much earlier

[1] the CWE lists another 40 children to *Improper Input Validation*

than PHP, so even a year-old Java project could possibly outperform a quite recent PHP project in this domain. Is the use of data modeling and consistent sanitation techniques higher in (e.g.) Java applications in general? If this would indeed be the case, even further research should try to answer the question if this effect is caused by the technology itself or merely the sophistication of the development community attracted by the technology.

The last open question I present here is whether writing annotations using the technique described here serves as a good means to develop code reviewing skills for the two vulnerabilities discussed in this thesis (as in perspective-based reading [BGL+96]). I noticed that having to think through the data flow, one develops a good intuition for what spots in the code might be vulnerable. Developing these intuitive patterns enables oneself to skim large amounts of code quickly.

Bibliography

[ABC+08] Anurag Agarwwal, Daniele Bellucci, Arian Coronel, Stefano Di Paola, Giorgio Fedon, Alan Goodman, Christian Heinrich, Kevin Horvath, Gianrico Ingrosso, Roberto Suggi Liverani, Alex Kuza, Pavol Luptak, Ferruh Mavituna, Marco Mella, Matteo Meucci, Marco Morana, Antonio Parata, Cecil Su, Harish Skanda Sureddy, Mark Roxberry, and Andrew Van der Stock. *OWASP Testing Guide V3*. OWASP Foundation, 3 edition, December 2008.

[BBM96] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. How reuse influences productivity in object-oriented systems. *Communications of the ACM*, 39(10):104–116, October 1996.

[BGL+96] Victor R. Basili, Scott Green, Oliver Laitenberger, Filippo Lanubile, Forrest Shull, Sivert Sorumgard, and Marvin V. Zelkovitz. The empirical investigation of Perspective-Based reading. *Empirical Software Engineering*, pages 133–164, 1996.

[Bou09] Jad S. Boutros. Reducing XSS by way of automatic Context-Aware escaping in template systems, March 2009.

[Cha05] Victor Chapela. Advanced SQL injection, April 2005.

[Cha06] Kathy Charmaz. *Constructing Grounded Theory: A Practical Guide through Qualitative Analysis*. Sage Publications Ltd, 1 edition, 2006.

[chr04] chromatic. Perl code kata: Testing taint. `http://www.perl.com/pub/a/2004/10/21/taint_testing_kata.html`, October 2004.

[CM07a] Steve Christey and Robert A. Martin. Vulnerability Type Distributions in CVE. `http://cwe.mitre.org/documents/vuln-trends/index.html`, May 2007.

[CM07b] Steve Christey and Robert A. Martin. Vulnerability Type Distributions in CVE - Summary. `http://cve.mitre.org/docs/vuln-trends/index.html#summary`, May 2007.

[Coa06] Ken Coar. The open source definition. `http://www.opensource.org/docs/osd`, July 2006.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[Cor07] MITRE Corporation. About CWE. `http://cwe.mitre.org/about/index.html`, September 2007.

[Cow03] C. Cowan. Software security for open-source systems. *Security & Privacy, IEEE*, 1(1):38–45, 2003.

[CR05] William R. Cook and Siddhartha Rai. Safe query objects: Statically typed objects as remotely executable queries. In *Proceedings of the*, St. Louis, May 2005. ACM.

[Dam] Bernardo Damele. sqlmap: automatic sql injection tool.

[Dam09] Bernardo Damele. SQL injection: Not only AND 1=1, March 2009.

[DAVT08] Eric Dalci, KDM Analytics, Veracode, and CWE Content Team. CWE-89: failure to preserve SQL query structure (aka 'SQL injection') (1.3). `http://cwe.mitre.org/data/definitions/89.html`, July 2008.

[DGD+08] Robert Douglass, Larry Garfield, Marc Delisle, Ken Rickard, Dries Buytaert, and Simon Hobbs. GoPHP5.org - helping speed the transition to PHP 5.2. `http://gophp5.org/`, February 2008.

[DVT09a] Eric Dalci, Veracode, and CWE Content Team. CWE-74: failure to sanitize data into a different plane (aka 'Injection') (1.3). `http://cwe.mitre.org/data/definitions/74.html`, March 2009.

[DVT09b] Eric Dalci, Veracode, and CWE Content Team. CWE-79: failure to preserve web page structure (aka 'Cross-site scripting') (1.3). `http://cwe.mitre.org/data/definitions/79.html`, March 2009.

[DYI+05] Martin J. Dürst, François Yergeau, Richard Ishida, Misha Wolf, and Tex Texin. Character model for the world wide web 1.0: Fundamentals. `http://www.w3.org/TR/charmod/#sec-Escaping`, February 2005.

[Eva03] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, August 2003.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, July 1999.

[Fel04] David Fells. What's new in PHP 5. `http://www.devshed.com/c/a/PHP/Whats-New-in-PHP-5/`, November 2004.

[FH06] Richard Ford and Michael A. Howard. A process for performing security code reviews. *IEEE Security & Privacy*, 4(4):74–79, August 2006.

[Fis09] Harrison Fisk. MySQL :: Prepared statements. `http://dev.mysql.com/tech-resources/articles/4.1/prepared-statements.html`, March 2009.

[FLS08] Mark Fallon, Bryn Liewellyn, and Howard Smith. How to write injection-proof PL/SQL, September 2008.

[Fou09] Django Software Foundation. Django | the web framework for perfectionists with deadlines. `http://www.djangoproject.com/`, February 2009.

[Fow02] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, November 2002.

[GE08] Jürgen Giesel and Stefan Esser. Bau sicherer LAMP Anwendungen, August 2008.

[Gro08] The PHP Group. PHP: downloads. `http://www.php.net/downloads.php#v4`, August 2008.

[Gro09a] Jeremiah Grossmann. Quick wins and web application security, March 2009.

[Gro09b] PostgreSQL Development Group. PostgreSQL 8.3: Data types. `http://www.postgresql.org/docs/8.3/static/datatype.html`, March 2009.

[Gro09c] The PHP Group. addslashes - Manual. `http://de2.php.net/addslashes`, March 2009.

[Gro09d] The PHP Group. PHP: Backward Incompatible Changes - Manual. `http://www.php.net/manual/en/migration5.incompatible.php`, April 2009.

[Gro09e] The PHP Group. PHP Data Objects - Introduction. `http://de.php.net/manual/en/intro.pdo.php`, March 2009.

[Han07] Robert Hansen. Charset vulnerabilities. `http://ha.ckers.org/charsets.html`, March 2007.

[Han08] Robert Hansen. Clickjacking details. `http://ha.ckers.org/blog/20081007/clickjacking-details/`, October 2008.

[Hei01] Josh Heidebrecht. What's new in JDBC 3.0. `http://www.ibm.com/developerworks/java/library/j-jdbcnew/`, July 2001.

[HL02] Michael Howard and David LeBlanc. *Writing Secure Code*. Microsoft Press Books, 2. a. edition, December 2002.

[HVO06] William G.J. Halfond, Jeremy Viegas, and Alessandro Orso. A classification of SQL injection attacks and countermeasures. In *IEEE Int'l Symposium on Secure Software Engineering*, 2006.

[LAAA06] V. T. Lam, S. Antonatos, P. Akritidis, and K. G. Anagnostakis. Puppetnets: misusing web browsers as a distributed attack infrastructure. In *Proceedings of the 13th ACM conference on Computer*

and communications security, pages 221–234, Alexandria, Virginia, USA, 2006. ACM.

[Lai98] Oliver Laitenberger. Studying the effects of code inspection and structural testing on software quality. In *Proceedings of The Ninth International Symposium on Software Reliability Engineering*, page 237. IEEE Computer Society, 1998.

[Mar08] Evan Martin. myspace worm. `http://community.livejournal.com/evan_tech/150019.html`, October 2008.

[McC04] Steve McConnell. *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press Corp., 2 edition, July 2004.

[Moo05] Nate Mook. Cross-Site scripting worm hits MySpace. `http://www.betanews.com/article/CrossSite-Scripting-Worm-Hits-MySpace/%1129232391`, October 2005.

[MS08] Ofer Maor and Amichal Shulman. Blind SQL injection, 2008.

[Mul07] Matt Mullenweg. On PHP, July 2007.

[Oba02] Dare Obasanjo. The myth of open source security revisited v2.0. `http://www.developer.com/open/article.php/990711`, 2002.

[Oez] Christopher Oezbek. *Introducing innovations into Open Source projects*. PhD thesis, Freie Universität Berlin, Berlin, Germany.

[OWA09] OWASP. OWASP Enterprise Security API. `http://www.owasp.org/index.php?title=Category:OWASP_Enterprise_Secur%ity_API\&oldid=57252`, March 2009.

[Rei09] Jeremia Reith. WordPress "Host" header RSS feed script insertion vulnerability. `http://secunia.com/advisories/32882/`, March 2009.

[RvW03] Mark R. von Wyk, Kenneth G. Graff. *Secure Coding, Principles & Practices*. O'Reilly & Associates, June 2003.

[Sch06] Bruce Schneier. *Beyond Fear: Thinking Sensibly About Security in an Uncertain World*. Springer, Berlin, illustrated edition edition, June 2006.

[Shi06] Chris Shiflett. addslashes() versus mysql_real_escape_string(), 2006.

[Sul09] Chris Sullo. Scrubbr - new stored XSS finder, February 2009.

[TDV08] CWE Content Team, Eric Dalci, and Veracode. CWE-20: improper input validation (1.3). `http://cwe.mitre.org/data/definitions/20.html`, July 2008.

[Tea08] CWE Content Team. CWE-652: failure to sanitize data within XQuery expressions (aka 'XQuery injection'). http://cwe.mitre.org/data/definitions/652.html, October 2008.

[TFH04] Dave Thomas, Chad Fowler, and Andy Hunt. *Programming Ruby: The Pragmatic Programmers' Guide, Second Edition.* Pragmatic Bookshelf, 2nd edition, October 2004.

[Wik08] OWASP Wiki. OWASP SQLiX project - OWASP. http://www.owasp.org/index.php?title=Category:OWASP_SQLiX_Project\&o%ldid=40827, September 2008.

[Wik09a] OWASP Wiki. About the open web application security project. http://www.owasp.org/index.php/About_OWASP, February 2009.

[Wik09b] OWASP Wiki. OWASP AntiSamy project. http://www.owasp.org/index.php?title=Category:OWASP_AntiSamy_Project%\&oldid=51770, 2009.

[Wik09c] OWASP Wiki. Top 10 2007. http://www.owasp.org/index.php?title=Top_10_2007\&oldid=56154, March 2009.

[Wik09d] OWASP Wiki. Top 10 2007-Insecure direct object reference. http://www.owasp.org/index.php?title=Top_10_2007-Insecure_Direct_O%bject_Reference\&oldid=52137, 2009.

[Wik09e] OWASP Wiki. XSS (Cross site scripting) prevention cheat sheet. http://www.owasp.org/index.php?title=XSS_%28Cross_Site_Scripting%%29_Prevention_Cheat_Sheet\&oldid=56462, March 2009.

[Wik09f] Wikipedia. Apache license - Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Apache_License\&oldid=28449%5805, April 2009.

[Wik09g] Wikipedia. ASCII — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=ASCII\&oldid=276091873, March 2009.

[Wik09h] Wikipedia. Cross-site scripting - Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Cross-site_scripting\&oldid%=279137374, March 2009.

[Wik09i] Wikipedia. Data Access Object - Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Data_Access_Object\&oldid=%279710644, March 2009.

[Wik09j] Wikipedia. ECMAScript - Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=ECMAScript\&oldid=277644739, March 2009.

[Wik09k] Wikipedia. Information security — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Information_security\&oldid%=274618521, March 2009.

[Wik09l] Wikipedia. Mambo (software) - Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Mambo_(software)\&oldid=277%164952, March 2009.

[Wik09m] Wikipedia. Relational algebra — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Relational_algebra\&oldid=2%72656872, February 2009.

[Wik09n] Wikipedia. SOAP - Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=SOAP\&oldid=279877292, March 2009.

[Wik09o] Wikipedia. Unicode — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Unicode\&oldid=273508720, March 2009.

[Wik09p] Wikipedia. Wikipedia:Academic use — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Wikipedia:Academic_use\&old%id=262252509, March 2009.

[Wil09] Jeff Williams. Don't Write Your Own Security Code: The OWASP Enterprise Security API, March 2009.

[Wor09a] Wordpress.org. Data Validation. http://codex.wordpress.org/Data_Validation, April 2009.

[Wor09b] Wordpress.org. WordPress › About » Requirements. http://wordpress.org/about/requirements/, April 2009.

[Yin08] Dr. Robert K. Yin. *Case Study Research: Design and Methods*. Sage Publications, Inc, 4th edition, October 2008.

[Zal09a] Michal Zalewski. Browser security handbook, part 2 - same-origin policy. http://code.google.com/p/browsersec/wiki/Part2#Same-origin_policy, March 2009.

[Zal09b] Michal Zalewski. Browser security handbook, part 2 - same-origin policy for XMLHttpRequest. http://code.google.com/p/browsersec/wiki/Part2#Same-origin_policy_f%or_XMLHttpRequest, March 2009.

Appendix A

Appendix

A.1 Guides To External Data

A.1.1 Annotation Diff Instructions

The source code for all innovation introductions was generated in *unified diff* format as produced by the Subversion[1] `svn diff` command. Differences were taken against the respective version of the application, as noted below. All `diff` files can be found on the enclosed CD-ROM in sub-folders of the directory `annotations`. The particular repositories which provide the source code the `diff`s were made from can be found in section A.1.2 on the following page.

WordPress

All the annotation `diffs` can be found in the `WordPress` subdirectory.

`wordpress-2.7_sqlannotations.diff` contains the first annotation attempt for WordPress, targeting SQL. The diff applies to `tags/2.7` in the WordPress repository.

`wordpress-trunk_20090124_sqlannotations.diff` contains a revised version of the `diff` above that applies to the `trunk` of the WordPress repository in the version as of January 24th, 2009. This version was requested on the mailing list ([wp:25294]).

`wordpress_sqlannotations_simple.diff` contains a version of the `diff` against the `trunk` as above but with only the `method_exists` and `trivial_implementation` annotations. This was an attempt to provide an easier-to-swallow annotation approach to WordPress ([wp:25394]).

`wordpress_delete_wpdb.diff` contains a proof-of-concept abstraction for SQL's DELETE statement and was used for illustration purposes ([wp:25394]).

[1] `http://subversion.tigris.org/`

Mambo

All the annotations `diffs` can be found in the `Mambo` subdirectory. All `diffs` apply to the `tags/465_release` tag in the Mambo repository (see repository directory in A.1.2).

`mambo465_administrator_annotations.diff` contains the annotations for abstraction of raw SQL for the `administrator` subdirectory of Mambo. It contains both the `trivial_implementation` and `unclassified` annotations used in the `diff` below.

`mambo465_administrator_annotations_trivialonly.diff` contains a version of the annotation above with only annotations marked as `trivial_implementation`. It was presented to the Mambo developers in [mambo:Annotations+for+raw+SQL.html]

`mambo465-insert-impl.diff` contains a proof-of-concept of how an abstraction of SQL's INSERT statement could look like. It was also part of the original announcement of the annotations in [mambo:Annotations+for+raw+SQL.html].

`mambo465_htmlencoding_com_*.diff` contain the annotations for the respective subdirectories treated by the HTML escaping annotations for Mambo. They were introduced in [mambo:Consistent+encoding+for+HTML+against+XSS+attacks.html].

A.1.2 Source Code Repositories

Code excerpts and references to the code throughout this thesis contain the project name, the path to the file references, the relevant line number and revision number. This list contains the base URLs to the source code repositories that were used to retrieve the code.

Source code files which are referenced in this thesis can be resolved by appending the paths given in a reference to the respective project base URL. Note that the paths don't specify the exact version of the files. In revision control systems like Subversion the exact version to retrieve has to be given in addition to the path to a file. Note that all project repositories beside Drupal use Subversion. The Drupal project uses CVS.

WordPress `http://svn.automattic.com/wordpress/trunk`

Mambo `https://mambo-developer.org/svn/mambo/tags/465_release`

Joomla! `http://joomlacode.org/svn/joomla/development/tags/1.5.x/1.5.10`

habari `http://svn.habariproject.org/habari/tags/0.6`

phpBB `http://code.phpbb.com/svn/phpbb/tags/release_3_0_4`

Zikula `https://code.zikula.org/svn/core/branches/zikula-1.1`

Drupal CVS: cvs.drupal.org:/cvs/drupal **module**:drupal **tag**:DRUPAL-6-10

riotfamily `http://svn.riotfamily.org/svn/riotfamily/tags/release-8-0`

TYPO3 `https://svn.typo3.org/TYPO3v4/Core/branches/TYPO3_4-2-6`

A.1.3 Mail Correspondence Instructions

This section explains how to access mail correspondence referenced throughout this thesis. Since the projects use different means for communication, accessing the sources is project-specific. All references to mail in this thesis have a common format and look like this: [projectname:identifier-or-filename].

Some mails from personal communication[2] are not available online and can only be found on the enclosed CD. Some mails are available online as well as locally. See the instructions for the respective project. Note that for mails supplied on the CD in normal text format, the mail addresses have been removed because of privacy and anti-spam considerations.

WordPress Mailing list data for the WordPress project is available online and locally as an mbox-format file. References for WordPress mailing list messages contain a unique id by GMane for each message. It can be used to retrieve the message from the online mail archive GMane[3]. The base URL for the WordPress development mailing list is `http://article.gmane.org/gmane.comp.web.wordpress.devel`. Individual messages can be retrieved by appending the unique id to the URL or (if viewing the Portable Document Format (PDF) version of the thesis, by clicking the link.

The GMane archive is also supplied as an archive in mbox format, containing all the archive contents as of April 19, 2009. The file can be found on the CD at `correspondence/wordpress/gmane.comp.web.wordpress.devel.mbox`

Mails from personal communication are identified by a filename. These messages are not available online and can be found in `correspondence/wordpress` on the CD.

Mambo Mambo does not use the development mailing list extensively, all references point to local files only. All references to messages are marked with a **mambo:** and include filenames. The files can be found in `correspondence/Mambo` on the CD. Note that for the Mambo project, chat transcripts were used extensively. The references are marked with **mamboIRC:** and are explained in the respective appendix section, A.1.4 on the next page.

Joomla! Messages concerned with the Joomla! project are available locally and online via Google Groups. The references contain the local filename which references files found in `correspondence/joomla`. The hyperlink associated to the reference points to the respective message in the Google Groups group. The base URL for the group is `http://groups.google.com/group/joomla-dev-framework`.

[2] all people whose mail appears in the context of this thesis gave their explicit permission for use

[3] `http://www.gmane.org`

habari The complete habari correspondence is available locally and via Google Groups, as for Joomla!. The files mentioned in the references can be found in `correspondence/habari`. References are click-able and point to the Google Group located at `http://groups.google.com/group/habari-dev`.

phpBB As I did not get responses from the phpBB team, there is no message data.

Zikula Messages exchanged with Zikula community members are available on the enclosed CD, in the directory `correspondence/zikula`. References contain the filename for the respective message.

TYPO3 As I did not get replies to my project analysis questionnaire, there is no data for TYPO3.

Drupal Replies to my project analysis questionnaire were not referenced, but can be found in `correspondence/drupal`.

riotfamily As I did not get replies to my project analysis questionnaire, there is no data for riotfamily.

A.1.4 Chat Transcript Instructions

Chat transcripts were used for the interaction with the Mambo project. They can be found on the enclosed CD in the `Mambo` sub-folder of `correspondence` directory.

The transcripts are available in two formats: First, in an XML format used by *Colloquy*, the tool I used to record the transcripts (ending in `.colloquyTranscript`) and second, a simple text format produced by the XML Stylesheet Language Transformations (XSLT) style-sheet found in `tools/colloquy.xml` on the enclosed CD (ending in `.txt`).

The filenames of the transcript files consist of the channel name followed by the date the transcript was recorded on. References to transcripts use date granularity. An example filename would look like this: `mos-cms_2009-02-11.txt`. A reference in the thesis text to this file would look like this: `mamboIRC:2009-02-11`.

A.2 Data Excerpts

A.2.1 Selected Correspondence

From: Florian Thiel
Date: January 16, 2009 8:20:49 PM GMT+01:00
To: Matt Mullenweg
Subject: Proactively enhancing WordPress security the lightweight way

Hello Matt,

My name is Florian and I'm doing research on open source projects and security for my diploma thesis. I would like to propose (and attach a patch for) a very lightweight, evolutionary approach to SQL Injection Attacks, some of which plagued WordPress in the past.

Previous attempts to "just make WordPress use data access abstraction" (as sometimes proposed on the mailing list) were understandably met with a lot of opposition as large structural changes don't come easy in an open source project (or any project). Since there is basic data access abstraction in the code (e.g. $wpdb->insert and $wpdb->update in wp-db.php) I think you would agree that having general data access abstraction would be a good thing from a security/robustness point of view. I'd like to make the path to general data access abstraction easier:

I produced a patch against WordPress 2.7 which annotates and classifies all uses of inline SQL. The classification tells you how much work it would be to get rid of the inline use of SQL. The patch can be found at
http://www.noroute.de/downloads/wordpress-2.7_sqlannotations.diff

From the 443 places where I found inline SQL, there are 85 places where an abstraction already exists and just had to be used. Furthermore, there are 172 places where a trivial implementation (trivial meaning that a very similar method already exists) would help get rid of the use of inline SQL. So by adding around 5 simple methods to wp-db.php you would get rid of more than 250 problematic uses of inline SQL. And the best part: We can start with the low-hanging fruit and gradually move to the harder ones while keeping the code working all the time!

I'm sending this mail to the WordPress core developers to gather your views on the topic before discussing with a general audience. Do you have any concerns about the proposed change? Do you think it would certain things worse? Not match the WordPress coding approach? We can certainly discuss all the points you consider problematic. If the community approves the idea and starts working on it, I will follow up with a similar approach for XSS, the other big security concern facing WordPress.

Hope to hear from you and all the best,
Florian

Listing A.1: initial mail to Matt Mullenweg of WordPress

```
From: Lynne Pope
To: Florian Thiel
Date: Thu, 29 Jan 2009 17:49:01 +0200
Subject: Re: [wp-hackers] Making WP more secure the evolutionary way

Hi Florian,
I've been quiet on the WP Hackers discussion because I've seen the issue of
abstraction being argued before. I am not sure you will get very far with
this!

However, I wanted to let you know that if you really want to be able to
contribute to a volunteer FOSS project, and don't mind if its not WordPress,
then Mambo is desperately needing developers with your skills.
The Mambo CMS is nearly 9 years old - 8 years as an open source project
under the GPL.

If you are interested in helping out with Mambo (where database abstraction
is wanted) the information about how to get involved and where to find the
mailing list and IRC channel is here:
http://mambo-manual.org/display/contrib/Home

The lead developer is usually found on the IRC channel. However, despite
just under 1.5 million downloads last year, the development team is very
small and 3rd party hackers don't hang out on the mailing list or channels,
so theres not a lot of outward signs of activity.

Just a thought ;)

Best Regards,
Lynne
```

Listing A.2: invitation to join Mambo

Hello Mambo developers,

as promised I took a look at how to make encoding for HTML to prevent cross-site scripting more consistent. I agreed with Andphe some time ago that doing the sanitation as late es possible (meaning: in the "echo"s) is the way to go. And here we are. I created annotations (similar to the ones for the unification on SQL handling, see XXXX) for the places that need changing. I limited myself to 5 folders in administrator/components (com_admin, com_banner, com_categories, com_checkin, com_comment) for the first run because there are a lot of annotations.

Patches for the respective folders (against Mambo 4.65) are attached.

The annotations can easily identified by searching for @EncodeForHTML. The annotation comes in different flavors since XSS prevention looks a bit different, depending on the context of the dynamic code. E.g, you don't need <>s if you can inject into an attribute. Urls and CSS also need different encoding, (IE can execute script code from CSS).

The different flavors are as follows:
* plain: body text, there should be no markup
* html: may contain html (often for return values that already build HTML)
* JavaScript data: data (not control) inside a JavaScript expression
* attribute: inside a HTML attribute
* URL: part of a URL
* CSS: becomes used as CSS

The vast majority of cases is "plain", which is good. I don't want to introduce much further complication. I would suggest that we introduce sanitation functions for all these cases. Some can (for now) do the same thing so that one can later switch the implementation if we have a good way for sanitation. Since they will be used often, the functions should have short names:

I suggest:
plain -> p()
html -> h()
JavaScript -> js()
attribute -> a()
URL -> u()
CSS -> css()

Mambo already has sanitation functionality, but I think it would be good to use a library that's maintained and updated externally so there will be no extra work. The Open Web Application Security Project (OWASP) maintains a library called "Reform" that has native support for our "plain" and "attribute" cases. Their approach is to only allow ASCII alphanumerics and basically HTML encode all the rest, which is really safe (whitelisting).

```
As a first step, the implementation of h() would take us a step
forward. I don't know where (in the codebase) that should reside and
if we should use namespacing (against collisions, the names are quite
short).

h() could just use the implementation of HtmlEncode from Reform
(http://www.owasp.org/index.php/Category:OWASP_Encoding_Project) and
is therefore easy to implement.

What do you all think? Is this the way to go? Please comment on the
mailing list...

Florian
```

Listing A.3: announcement of the HTML escaping annotations for Mambo

From: Florian Thiel
To: joomla-dev-framework at googlegroups.com
Date: Tue, 31 Mar 2009 14:20:41 +0200
Subject: Process and technology questionnaire

Hello once again, Joomla developers,

I'm in the process of writing my diploma thesis on the prevention of (open source) web application security vulnerabilities and I'd like to know a bit
about your fine project, the way you see it. (I know some of these questions can be answered, at least roughly, from your web pages).

It would be great if you could take a couple
of minutes and think about the questions below. The questions are mostly open-ended. Elaborate and skip questions at will. I'd like to get answers by the community, not just the project lead. If answers diverge vastly on a topic, I'd like to follow up on these.

Thank you very much in advance. I will provide you with a link to the results of my thesis when it's done. Maybe you can benefit from some of the findings.

Florian

The questions:

About technical aspects:
- Are you using a web application framework? Which one?
- Do you use explicit data modeling for all business objects in the application?
- Do you have a specific layers for input/output validation/filtering? (If applicable) What does the input/output layer do (respectively)? How? Are you using external libraries? Why? Why not? (for HTML sanitation. object-relational mappers, database abstractions with prepared statements)?
- (If applicable) What responsibilities do the input/output layers have, respectively?
- How do you ensure that all input passed through validation/ filtering? Do you have an API that must be used?
- Do you provide services to independently developed modules/ components? Is there a defined API?
- Which other external libraries do you use?

About the development process:
- Is there public documentation about the responsibilities of the input/output layers?
- Is there public documentation about *when* input/output validation/ filtering should happen? (Like: "output filtering must always happen in the method that renders the data")
- Do you have automatic tests for the whole system?

```
Bonus question:
- Do you do manual code review?
```

Listing A.4: Project questionnaire sent to the Joomla! project

A.2.2 Additional Annotation Excerpts

```
640 * @RawSQLUse, algorithmic
641 */
642 function prepare_query() {
643     global $wpdb;
644     $this->first_user = ($this->page - 1) * $this->
            users_per_page;
645     $this->query_limit = $wpdb->prepare(" LIMIT %d, %d", $this
            ->first_user, $this->users_per_page);
646     $this->query_sort = ' ORDER BY user_login';
647     $search_sql = '';
648     if ( $this->search_term ) {
649         $searches = array();
650         $search_sql = 'AND (';
651         foreach ( array('user_login', 'user_nicename', '
                user_email', 'user_url', 'display_name') as $col )
652             $searches[] = $col . " LIKE '%$this->search_term%'
                    ";
653         $search_sql .= implode(' OR ', $searches);
654         $search_sql .= ')';
655     }
656     $this->query_from_where = "FROM $wpdb->users";
657     if ( $this->role )
658         $this->query_from_where .= $wpdb->prepare(" INNER JOIN
                $wpdb->usermeta ON $wpdb->users.ID = $wpdb->
                usermeta.user_id WHERE $wpdb->usermeta.meta_key =
                '{$wpdb->prefix}capabilities' AND $wpdb->usermeta.
                meta_value LIKE %s", '%' . $this->role . '%');
659     else
660         $this->query_from_where .= " WHERE 1=1";
661     $this->query_from_where .= " $search_sql";
662 }
```

Listing A.5: WordPress: `wp-admin/includes/user.php`, (revision 10323 (locally modified))

A.3 Syntax Guides

A.3.1 Python Syntax Used

Example listings in this thesis which explain SQL queries but do not exclusively consist of SQL, are written in Python[4]. Python is used because of its intuitive readability, even for people not familiar with the language details.

The main language constructs used in the examples are variable assignment and string templates. Variables in Python do not need any special markers and the language is dynamically typed, so assignments look like listing A.6.

```
variable = 3
variable2 = ''foo''
```

Listing A.6: Python variable assignment example

String templates are used to demonstrate SQL injections and look like the example in listing A.7.

```
"This is a string with a %s place-holder" % ("fancy")
```

Listing A.7: Python string template example

Similar to C format strings, %s is used to mark a place-holder for a string. The values following the % *after* the string are inserted (in order) into the place-holders, so the string above would read "This is a string with a fancy place-holder". Note that Python supports other place-holders than %s for types other than strings. If one would actually use Python to construct SQL statements, one would surely use these other place-holders to provide type safety.

[4] http://www.python.org

A.4 Legitimacy Of Online Sources

This thesis makes extensive use of online sources, including web-logs and Wikis. Using these is inevitable in order to provide an up-to-date discussion of the topics presented in this thesis. Scientific literature can not keep up with the pace exploit techniques — especially forXSS — are changing. Furthermore, web-logs allow me to cite the opinion of professionals who work with real-life vulnerabilities and mitigation solutions. The nature of this thesis' discussion requires real-life solutions which can be applied by open source web application projects, not theoretically perfect mitigations which only work with laboratory set-ups.

The people whose web-logs I cited have a background in security research and practice and state their professional opinions. While web-logs are not as thoroughly reviewed as scientific journals, the comment facilities allow corrections of errors or omissions by readers, which are often professionals as well. Furthermore, the professional reputation authors of web-logs gain by writing high-quality articles is at stake if they publish unfounded and possibly wrong information.

I think these arguments suffice to allow inclusion of certain kinds of non-reviewed source into this thesis. Note that neither Wikis nor web-log entries were cited for cases that need highly reliable data or are very controversial.

A.4.1 Citing Wikis

Wikipedia and OWASP are public Wikis, and as such, not regarded as a stable and trusted source for research by some (Wikipedia has its own take on the subject in an article about academic use of Wikipedia [Wik09p][5]).

Wikipedia and the OWASP Wiki are not strictly, scientifically reviewed. OWASP articles are mostly written by members of the OWASP, who are security professionals. Wikipedia is not written by a closed community and therefore the expertise of contributors varies wildly. Overall, I believe the quality of articles in Wikipedia, especially about computer science topics, is very high. Nevertheless, I only use Wikipedia articles in this thesis for background information for the reader. I do not cite Wikipedia for argumentation or data.

A.4.2 Linking to MediaWiki

Both OWASP and Wikipedia are Wikis using the popular MediaWiki[6] software.

To accommodate for the need to cite an exact version of an article, the MediaWiki software provides a little-known feature, called the *permanent link*. Each version of each article has a unique identifier which can be used to retrieve a specific version of an article. The identifier is independent from the article name and can as such be used to retrieve the exact version cited, unencumbered by renaming or restructuring. All citations from Wikipedia and

[5] This, again, is a citation of Wikipedia, which you may decide not to trust if you're a researcher.

[6] `http://www.mediawiki.org/wiki/MediaWiki`

OWASP include the URL with the identifier, somehow inappropriately called `oldid`. An example for the article about using Wikipedia in academia would look like this: `http://en.wikipedia.org/w/index.php?title=Wikipedia:Academic_use\&oldid=262252509`

A.5 Open Web Application Security Project (OWASP)

The OWASP[7] is a worldwide community focused on improving application security. As of their mission statement they are "dedicated to enabling organizations to develop, purchase and maintain applications that can be trusted" [Wik09a]. For this purpose they provide educational materials about application security, hold conferences and develop software tools.

In order to make application security issues visible, they publish top ten lists of common vulnerabilities. The 2007 version of this list [Wik09c] was one of the main inspirations for this thesis.

All documents, tools and other information provided by OWASP is available free of charge and open for reuse by anyone. Their large collection of How-Tos, presentations and guides proved to be an invaluable resource for research on the topic of web application security.

The documents provided by OWASP turn up in discussions among security professionals cited in this thesis (e.g. Jeremiah Grossman, Robert Hansen, etc.), providing credibility to material produced by OWASP.

[7] `https://www.owasp.org`

A.6 The Common Weakness Enumeration Project

Although this thesis is based on common weaknesses from the OWASP (see section A.5 on the preceding page) top ten list of 2007 [Wik09c], that list only provided the initial inspiration for the vulnerabilities to look at.

The CWE project supplies a much more interesting (from a research standpoint) summary discussion of the weaknesses discussed in this thesis, and security-related weaknesses in general.

The CWE project is a community-based effort to provide a structured catalog of software weaknesses. MITRE, a not-for-profit corporation which provides technical and operational expertise to government bodies, started categorizing software weaknesses in 1999 and published the Common Vulnerabilities and Exposure (CVE) list[8]. The list merely provides an identifier and a list of references for vulnerabilities which can be used to uniquely address a vulnerability. The list was maintained by a single body and its use was limited to redistribution. No changes to the list were allowed.

To be able to categorize and assess tools provided by the security assessment industry, a more structured approach was adopted in 2005. MITRE revised the CVE list for a government-funded endeavor, the Software Assurance Metrics and Tool Evaluation (SAMATE) project. The resulting document was called the Preliminary List Of Vulnerability Examples For Researchers (PLOVER) [Cor07]. It included the CVE names but featured conceptual grouping of vulnerability types. The final step was to add commonly acceptable definitions and descriptions to the weaknesses, resulting in the CWE list. In contrast to the CVE, the CWE list is maintained by a community process which includes researchers and practicioners. According to MITRE, the CWE provides "a common language for describing software security weaknesses [and] a standard measuring stick for software security tools"[Cor07].

The list provides a tree-structured collection of vulnerabilities. The leaves represent very concrete vulnerabilities, the entries on the path to the tree root are increasingly abstract, representing categories into which the vulnerabilities can be subsumed.

The tree-shaped structure of vulnerability definitions provided by CWE is useful for future research on adequate additional vulnerabilities which could be targeted by the annotation approach. See section 5.3 on page 104 on future research for ideas.

[8] `http://cve.mitre.org/`

A.7 Acronyms Used

CWE Common Weakness Enumeration

XSS Cross-Site Scripting

SQL Structured Query Language

OWASP Open Web Application Security Project

CVE Common Vulnerabilities and Exposure

SAMATE Software Assurance Metrics and Tool Evaluation

PLOVER Preliminary List Of Vulnerability Examples For Researchers

CMS Content Management System

SQLIA SQL Injection Attack

MAC Mandatory Access Control

DoS Denial of Service

ORM Object-Relational Mapping

DML Data Manipulation Language

FROC Front-Range OWASP Conference

HTML Hypertext Markup Language

XHTML Extensible Hypertext Markup Language

HTTP HyperText Transfer Protocol

DOM Document Object Model

AJAX Asynchronous JavaScript and XML

API Application Programming Interface

URI Uniform Resource Identifier

CSS Cascading Style Sheet

URL Uniform Resource Locator

XML eXtensible Markup Language

OSI Open Source Initiative

IEC International Electrotechnical Commission

ESAPI Enterprise Security API

MVC Model-View-Controller

ASCII American Standard Code for Information Interchange

IDE Integrated Development Environment

NVD National Vulnerability Database

CRM Customer Relationship Management

ERP Enterprise Resource Planning

GPL General Public License

RSS Really Simple Syndication

MVC Model-View-Controller

IRC Internet Relay Chat

XSLT XML Stylesheet Language Transformations

RPC Remote Procedure Call

WAF Web Application Firewall

FAQ Frequently Asked Questions

MPL Mozilla Public License

DWR Direct Web Remoting

LGPL Lesser General Public License

CSRF Cross-Site Request Forgery

PDF Portable Document Format

Listings

List of Figures

List of Tables