

WEB ENGINEERING AND THE SOCIAL INTERNET:  
AN ANALYSIS OF SECURITY AND ENGINEERING TECHNIQUES  
EMPLOYED IN THE DEVELOPMENT OF SOCIAL NETWORKING WEBSITES

By  
Michael P. Dippery

A Thesis  
Presented to the Faculty of  
Bucknell University  
In Partial Fulfillment of the Requirements for the Degree of  
Bachelor of Arts with Honors in Computer Science

May 5th, 2008

Approved By: \_\_\_\_\_  
Dr. Xiannong Meng,  
Thesis advisor, and  
Chair, Department of Computer Science

## Acknowledgments

I'd like to take a moment to thank my honors thesis committee: Professor Xiannong Meng, for providing sage advice during the writing of this thesis; Professor Luiz Felipe Perrone, for being an instructor, mentor, and friend—and for encouraging me to write a thesis in the first place; and Professor Matthew Higgins, for representing the Honors Council on my committee.

Additionally, I'd like to thank Chris Burroughs, for passing along some helpful papers and other materials about software engineering practices (as well as being an avid Lisper).

Finally, I'd like to thank my parents, who have supported me in my academic endeavors, even though they don't understand a bit about what I study; and Caitlin Russell, who helps remind me that there are more important things in life than computer science.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Background . . . . .	4
1.1.1	The Birth of the World Wide Web . . . . .	4
1.1.2	Web Technologies . . . . .	7
1.1.3	Classes of Attacks . . . . .	10
1.2	Thesis Statement . . . . .	14
<b>2</b>	<b>Methods</b>	<b>15</b>
2.1	Developing Facebook Applications . . . . .	16
2.1.1	Facebook’s Framework . . . . .	16
2.1.2	Creating an Application . . . . .	24
<b>3</b>	<b>Discussion</b>	<b>29</b>
3.1	Exploiting the Social Graph . . . . .	29
3.2	Spam . . . . .	30
3.3	FBML and JavaScript . . . . .	32
3.4	Sandboxing Malicious Web Pages . . . . .	33
<b>4</b>	<b>Conclusion</b>	<b>35</b>
4.1	Future Work . . . . .	42
<b>A</b>	<b>Code Listing</b>	<b>56</b>

## **Abstract**

Social networking websites, which allow individuals to build and strengthen social bonds between one another, are becoming increasingly common on the World Wide Web. One such social networking site is Facebook, a site that is particularly popular amongst college students across the United States. Facebook recently opened up an application programming interface (API) that allows third-party developers to write web applications that run in conjunction with Facebook, as though they had been created by Facebook's own developers. In doing so, they exposed the data behind Facebook, which gives an excellent opportunity for others to study issues concerning the security and privacy of Facebook.

Using Facebook and its public API as a case study, this thesis assesses the use of modern software engineering techniques in the field of web applications development. I attempt to discover and report on best practices for the development of web-based applications from a perspective of security and reliability, as well as encourage these best practices in future web-related software development.

While the results of the study show that Facebook employed many positive and pro-active techniques in designing, implementing, and deploying its API to third-party developers, I have found that there is much room for improvement in the world of web applications development. The study of Facebook and social networking has yielded insight into the way software design methodologies can be applied to the web to create reliable, secure applications; and how web technologies can be used to create desktop applications of similar reliability and security.

# Chapter 1

## Introduction

In less than twenty years, the World Wide Web has grown from a twinkle in the mind's eye of a handful computer scientists to a global marketplace for the sharing of information, ideas, data, and even tangible goods. Myriad applications for keeping in touch with friends, family, and former classmates, as well as applications that connect users to people with similar interests and aspirations, have followed in the wake of the explosion of the World Wide Web. Each day, millions of people log onto social networking sites like Facebook, MySpace, del.icio.us, and Flickr, in order to communicate with acquaintances all around the globe. This quick and easy social contact is unprecedented in human history.

Like all intimate human contact, this communication requires the exposure of personal information; but unlike face-to-face contact, this personal information is dispersed across a global network that was never intended to support the distribution of such information [42]. With this exposure of information comes a set of problems unique to web-based communication. Millions of people willingly turn their personal information over to companies with the expectation that their data will stay private. In most cases, their private information does stay private, but what happens when that information is leaked—or even worse, stolen?

Neither the World Wide Web nor the Internet itself were designed with security in mind [42]. This lack of security in the fundamental fabric of the global network requires

individual web application developers to build their applications with security in mind. Unfortunately, security is often an after-thought in software engineering [42]. A change is necessary in software engineering techniques themselves [42, 77].

Unfortunately, software engineering techniques are not often rigorously applied to the development of web-based applications, creating a crisis in the development of secure and robust web applications. Combined with the lack of major social networking web applications as little as ten years ago, there are very few examples to which we can turn to use as a model for “good designs” or “bad designs”. Even fewer web applications allow interoperation with third-party software developers via a public *application programming interface* (API).

Fortunately, there is one prominent example that is familiar to anyone in college today: Facebook [19]. Facebook is a social networking site that connects people to their family and friends all across the globe. In May 2007, Facebook opened its entire framework to third-party application developers. This interface gives developers access to the mother lode of Facebook’s data—access which can be used for potentially nefarious purposes.

Facebook has also systematically documented its efforts via its public blog [31], which gives outsiders an intimate look at the development of Facebook’s applications and third-party interface. These facts make Facebook an excellent case study in the design and implementation of public web-based APIs.

Facebook, along with Yahoo!, are the first major social networking sites to open up their framework [6, 59]; they will almost certainly not be the last. As the Web continues to grow, and social networking sites continue to blossom, the number of other sites offering such an intimate look at their data will continue to expand as well. Unfortunately, the dismal picture of software engineering, especially in regards to security, makes the future of such efforts look bleak [42]. Therefore, it is of crucial importance that software engineers develop a set of best practices and designs for exposing such systems to third-party developers [42, 77]. These efforts will be critical to prevent the exposure of private and personal data to malicious users.

Other studies of Facebook, such as those by Acquisti and Gross [3, 47] and Jones and Soltren [53], have focused on the *behavioral* aspects of social networking (i.e., they have studied the sort of information that social networking users *willingly* share). My study, on the other hand, focuses on social networking from an *engineering* perspective. Social networking sites cannot be expected to prevent exposure to information that users willingly share with other users, but they *can* be expected to prevent exposure to private information.

This thesis analyzes Facebook’s programming interface in regards to security and privacy to help develop a set of best designs and practices for building web applications. This thesis attempts to address the following questions:

- What did Facebook do *right* in designing the public API for the Facebook platform?
- What did Facebook do *wrong* in designing this API?
- What can a study of Facebook teach us about software design and engineering methodologies both in terms of web-based applications, and from a general perspective as well?

My original motivation for this thesis was a Facebook “game” called *Zombies*.<sup>1</sup> *Zombies* is an application which allows a user to “bite” other Facebook users, thus turning them into zombies [15]. Essentially, this is a metaphor for getting another user to add the application to his own user profile page. Thus, the application spreads similarly to a computer *virus*, although the exact mechanism is different.

The idea of distributing a virus or *worm* disguised as a Facebook application is not purely fictional or hypothetical. There is at least one instance of Facebook application that was really a worm-like vector for installing a piece of spyware and adware called *Zango* [40]. The concept of a “social worm” is a scary but real concept that has already been used “in the wild” [39].

---

<sup>1</sup>As we shall see in §2.1, Facebook allows users to add small applications to their profile pages.

## 1.1 Background

### 1.1.1 The Birth of the World Wide Web

Social networking is the latest buzz on the Internet. Social networking websites like MySpace and Facebook make the news on a nearly weekly basis, whether reporters are discussing the social merits of the site, the latest increase in market value, or the latest crime facilitated by information placed on such sites. It is hard to imagine the World Wide Web without social networking (and, indeed, to imagine the Internet without the World Wide Web), but if one takes a trip back in time twenty years, one will find an Internet without social networking or even the ubiquitous prefix *www*.

In 1980, Tim Berners-Lee, a British computer programmer, took up a position as a contract worker at CERN, the European Particle Physics Laboratory in Geneva, Switzerland [8]. Berners-Lee found the organization of CERN to be a complex web of people, projects, and information. He began to develop a program called Enquire that allowed him to record connections between people, the projects on which they worked, the computer programs they used the most, and the machines on which they stored their data [8]. Berners-Lee quickly saw that the most important aspect of data was not the information itself, but rather the connections between information and people [8].

Six months after arriving at CERN, Berners-Lee left, and Enquire was eventually lost. However, the idea was resuscitated in 1984, when Berners-Lee returned to CERN [8]. Berners-Lee once again saw a need for a system in which both information *and* connections could be stored. Unfortunately, the computing environment at CERN was much more heterogeneous than before. Berners-Lee quickly realized that any system he created had to run on—and talk to—a multitude of systems. Anything else required people to adapt their research habits to the system, and thus was doomed to failure [8].

Berners-Lee's idea rested on two key concepts: *hypertext*, or text that provided a connec-

tion or *link*, to a larger piece of related information [8]; and *remote procedure calls* (RPC) [10], a method in which procedures and functions could be invoked across a network between systems that may be running on different computing platforms [8]. Most importantly, Berners-Lee’s new hypertext-based system was decentralized. Berners-Lee wanted anyone to be able to add a node to the network at any time, without having to ask an authority figure. This had the side effect of making the system *scalable*—its size would not be limited by the resources of a single machine or a collection of servers [8].

By 1990, Berners-Lee turned his information storage idea into the familiar World Wide Web [8]. The rest of this history is apparent to most World Wide Web users, and shall be left to the history books, rather than being repeated in this thesis. Interested readers are encouraged to read Tim Berners-Lee’s book, *Weaving the Web*, which tells the story of the World Wide Web [8].

## Social Networking

In its simplest description, social networking is the abstract representation of individuals and the relationships between them. Often times, social networks are represented as mathematical *graphs* in which people are the vertices and relationships are the edges [38].

However, in the more colloquial sense, a social network is any website that connects people through various relationships, including personal, professional, or through common interests. Social networks especially help bring together people who might not otherwise have any interactions in “real life” [54].

Given the fact that humans are social creatures, it should come as no surprise that the web developed into such a medium of communication. The only thing surprising about social networking is the amount of time the concept took to become popular. In a sense, social networking is the encapsulation of Berners-Lee’s original idea: building connections between people and information using the Internet.

## Facebook

Created by a Harvard student named Mark Zuckerberg, Facebook debuted in February 2004 at Harvard University [22, 60]. The site was originally only open to Harvard students, but eventually grew to encompass other colleges, then high schools, and finally the general world community.

Facebook is a social networking site: a website specifically devoted to linking together a collection of users through social relationships such as geographic proximity, professional background, or hobbies [43]. Facebook's original goal was to link college students together through interests, hobbies, shared classes, and even residence, but has grown to encompass other social groups (such as high school students or coworkers) as well.

Facebook is now estimated to be worth over \$1 billion [60]. Both its popularity and size are astounding: it attracts the second highest number of visitors of any PHP site on the Internet [22], and boasts one of the largest MySQL installations in the world [22]. From a technical view, Facebook is an interesting study in sheer scalability and usage.

**Third-Party Applications** In May 2007, Facebook decided to open its web application to third-party developers by releasing an application programming interface (API) that can be used to build applications that exist outside of Facebook. These applications can be written in numerous programming languages on nearly any computing platform. They have nearly full access to Facebook's vast amounts of personal data, and communicate with Facebook via *HyperText Transfer Protocol* or HTTP, the protocol used to communicate with web servers on the World Wide Web. The details of this mechanism of communication are discussed in §2.1.1.

End-users may add these applications to their Facebook profile. The application is then given "permission" to access the user's personal data and manipulate it in some way. This manipulation ostensibly adds additional functionality to the user's Facebook experience, be

it in a purely utilitarian or purely entertaining manner, or some degree of the two.

Details about Facebook's third-party framework and the development of Facebook applications are discussed in §2.1.

### 1.1.2 Web Technologies

Although websites vary greatly in their design, they are all built from a similar set of structural elements. The basic technologies used to build web pages include:

**HyperText Markup Language** (HTML) is the language used to specify the structural elements of a web page, like headers, paragraphs, and lists, as well as content like texts and images. Syntactically, the language is structured in a hierarchical fashion using blocks of text enclosed in *tags*, or delimiters used to describe the textual data [94].

**Cookies** are small pieces of data stored by a website on the client's computer. They are commonly used to identify users. Cookies are designed to overcome the stateless nature of HTTP by allowing web applications to maintain some information about user interaction with an application [62].

Typically, when a website wants to maintain some sort of state, the web server will write data to a cookie, which is stored as a file by a user's web browser on the user's local machine. The website can then request the data from the cookie at a later time when it requires the stored data [62].

**Cascading Style Sheets** (CSS) specify the fonts, colors, and other visual elements of a web page. Style sheets can be used to provide a consistent *theme* for a web page [93].

**JavaScript** is used to dynamically alter web content.<sup>2</sup> Unlike a *server-side* language like

---

<sup>2</sup>JavaScript should not be confused with the programming language *Java*. Although both languages share a name and syntax, they are unrelated technologies.

PHP, JavaScript is executed in the client's own browser. JavaScript is commonly used in conjunction with AJAX (described below) to provide a web page that looks and acts somewhat like a desktop application [84].

**Extensible Markup Language** (XML) is an HTML-like language that can be used to structure almost any sort of information. One goal of XML is to be “human-legible and reasonably clear” [12]. XML marks up the data in a human-readable format, and looks similar to HTML, although its uses extend far beyond the markup of web content. XML is *extensible* in the sense that it allows extensions to the base language [12, 83]; that is, authors may create their own “tags”. Facebook uses an extension of XML called *Facebook Markup Language* in its API, as described in §2.1.1.

**Extensible Hypertext Markup Language** (XHTML) is a version of HTML that is expressed using XML [83]. Since XHTML is syntactically more rigid than HTML, it can be more easily parsed. It otherwise shares the same design and purpose of HTML [83].

## Dynamic Web Content

As first designed, the World Wide Web was *stateless* and *static*. In other words, connections were not maintained between client and server (“stateless”), and once marked up, the same web content was served to all clients (“static”). However, by the mid-1990s, some web developers saw a need for *stateful*, *dynamic* content; that is, content that acted like a desktop application and could be delivered to a user based on certain criteria. For example, web authors wanted a way to control or customize the display of a web page based on certain criteria such as the user or his geographic location. Web designers also wanted a way to alter a web page based on user interaction without having to refresh the web page (i.e., allow web pages to look and feel much the same way as a typical desktop application).

Out of this need arose a number of technologies for delivering *dynamic web content*.<sup>3</sup>

**Perl** is primarily a language for finding patterns, or *regular expressions*, in strings of text [66, 67]. Since creating and serving dynamic web content consists mostly of text processing, Perl was a logical choice for scripting web content—at least until more robust technologies were released [66]. Even with the advent of languages like PHP and Ruby, Perl is still occasionally used to build web pages [66], and Perl continues to be actively developed.<sup>4</sup> It is still used as a scripting language on many computing platforms, as well [66].

**PHP** was released shortly after Perl [68]. It was inspired by Perl [67, 68], and thus the two languages share many similar elements. Like Perl, PHP is open source, and the PHP community has released many libraries for interacting with a plethora of databases.

**Structured Query Language** is a language used to communicate with databases [88]. SQL is an old technology, but has found widespread use in database applications. Its implementations include MySQL, PostgreSQL, and SQLite [88].

**Java** is an object-oriented programming language released by Sun Microsystems in the early 1990s [13]. Java is hardly limited to web applications; it can be used to create full-featured desktop applications as well [13, 79]. However, Java failed to become popular until support for Java *web applets* was added to the Netscape browser in 1995 [13]. Since then, Java has found a strong home on the web. Java can be used to write *applets*, miniature applications that run inside a user’s web browser [79].

**Python** is an object-oriented scripting language that was released in 1991 [67, 72].

---

<sup>3</sup>This list is hardly exhaustive, and is merely a list of the technologies referenced in this thesis.

<sup>4</sup>In fact, at the time of this writing, the Perl community is getting ready to release Perl 6.

**Ruby** is an object-oriented scripting language [76]. Released around 1995 [76], Ruby rapidly gained popularity as a web scripting language with the release of *Ruby on Rails* [32], a Ruby-based framework that is used to rapidly create web applications.

## Web 2.0

Many of the above technologies are grouped under the umbrella term *Web 2.0*, a term describing a concept in which traditional web technologies are mixed with dynamic web content in order to give control over the content of web pages to users rather than authors [45, 65]. Essentially, the user *becomes* the author.

Several key technologies are used by Web 2.0 authors to develop their web applications:

**Asynchronous JavaScript and XML** (AJAX) is the name for a set of technologies that can be used to create dynamic web pages [41]. AJAX-based sites combine XHTML, JavaScript, and XML with dynamic web languages like PHP and Ruby, and database languages like MySQL, into web applications that attempt to look and feel like desktop applications. They often exchange data between client and server using XML or *JSON* [41].

**JavaScript Object Notation** (JSON) is a markup language used to transfer data between client and host [1, 92]. JSON is a way of marking up data such that it is easily turned into JavaScript *objects* [92]. JSON is often used instead of XML because it is a more compact way of representing data. Data marked up in the JSON format can be transmitted and parsed faster than XML [85, 92].

### 1.1.3 Classes of Attacks

Before discussing specific attacks against Facebook's platform, it will be useful to understand *classes* of attacks that may be generally used against web applications.

## HTML

HTML is a *markup language*, a language used to provide formatting and content for a document, generally in a *cross-platform* manner. By itself, HTML is rarely malicious. This is due to the fact that HTML documents are not *executed*, but rather parsed for display to the end-user. HTML is not *Turing-complete*.<sup>5</sup> A language is said to be *Turing complete* when it is capable of computing anything that a *Turing machine* can compute [11]. A Turing machine is a machine based on finite-state machines with a single unbounded read/write tape [48]—essentially, the abstract definition of our modern computing architectures. This is all just a formal way of saying that HTML is limited in what it can compute. However, a number of problems related to HTML exist:

- Arbitrary JavaScript can be embedded in HTML markup, which can cause security vulnerabilities on a website; this issue is described in greater detail below. Less maliciously, arbitrary JavaScript can cause “pop-up” or “pop-under” browser windows to appear.
- HTML can allow for “annoying” behavior, such as the pop-up windows described above, or the automatic playing of music or video content.

## SQL Injections

Communications between applications and databases is often performed using a language known as *Structured Query Language*, or SQL. SQL is a language standard implemented by numerous database systems, including MySQL, PostgreSQL, and SQLite.

SQL structures database queries in a way that it similar to the English language. For example, to query a database for all users, one might use a query like *SELECT user FROM*

---

<sup>5</sup>This, by itself, is not necessarily a limiting factor; SQL is not Turing-complete, but can wreak havoc in the wrong hands, as we shall see in §1.1.3.

*users*. The simplicity of SQL's syntax, combined with the complexity and power of its operation, allow for a set of database exploits known as *SQL injections*, an exploit in which an attacker takes advantage of SQL's syntax to perform malicious operation without the express consent of the application.

For example, pretend that a developer has a database of user-password pairs. This data is stored in the table *users*, and consists of two columns: *user*, which contains the username; and *password*, which contains a hash of the user's password. Assume again that this database backs a web application, which allows a user to log in via an HTML form that contains the fields *loginUser* and *loginPassword*. The web application might construct a SQL query using the string shown in Figure 1.1.

```
SELECT COUNT(*) FROM users WHERE user = ' + loginUser + ' AND  
password = ' + loginPassword + ';
```

Figure 1.1: A typical SQL string.

This query would return the number of users with the specified user-password pair. If, for example, the user entered the username *me* and the password *password*, the web application would construct the SQL query shown in Figure 1.2.

```
SELECT COUNT(*) FROM users WHERE user = 'me' AND  
password = 'password';
```

Figure 1.2: A SQL query after string interpolation.

SQL statements are terminated with a semicolon. This allows more than one statement to be sent to the database at a time. This property of SQL's syntax can easily be exploited by malicious users. For example, assume now that the user enters the username *me* and the password *a'*; *DROP TABLE users; SELECT \* FROM users WHERE user LIKE ' [89]*. Certainly

this is a weird password, but it is constructed for a very sinister purpose: to delete the *users* table. Figure 1.3 shows the SQL query that is constructed by the web application.

```
SELECT COUNT(*) FROM users WHERE user = 'me' AND password = 'a';  
DROP TABLE users;  
SELECT * FROM users WHERE user LIKE '';
```

Figure 1.3: A SQL query after a successful SQL injection.

This is not one, but *three* SQL statements. The first one checks the database for a valid user-password pair; the second *deletes the entire table of users!*<sup>6</sup>

Of course, to perform a successful SQL injection, the attacker must know the names of the columns and tables contained in the database. Although at first this seems difficult, it is not as hard as one might expect. Application developers often use common names (e.g., *users* for the table of users or *password* for the column of passwords). Attackers can also use tools that probe an application to determine the the structure of a database using trial-and-error techniques. Finally, the structures of databases used by open-source software *are* known to attackers. For example, one can examine the PHP source code of the popular *phpBB* bulletin board software to discover the standard names of databases, tables, and table columns used by the software.<sup>7</sup>

While crippling, SQL injections are all too easy to perform, and thus quite common. Luckily, however, application developers can take simple precautions to prevent many SQL injections. For example, developers can make sure to *sanitize database input*; functions for sanitizing database inputs are built into many web scripting language libraries, including PHP [69, 70, 71].

---

<sup>6</sup>The third is placed so that the SQL statement is syntactically valid; it does nothing useful in this case.

<sup>7</sup>Of course, open-source software has the benefit that many more developers are examining the software for vulnerabilities. Indeed, I have found no evidence to suggest that open-source web applications are more susceptible to SQL injections than “closed-source” software.

## Cross-Site Scripting

A *cross-site scripting* (XSS) attack occurs when arbitrary scripting code is injected into the output of a web application [49, 82]. This scripting code is then executed by the web server (in the case of server-side scripting, such as ASP or PHP) or web browser (in the case of client-side scripting like JavaScript). The script then performs a malicious action, ranging from something relatively harmless (like changing the color scheme of the web page) to something potentially damaging (such as siphoning off private data to a third party, or copying or modifying a cookie).

Such attacks are relatively easy to perform on any site that allows user-submitted content, a feature relatively common on blogs, news sites, and social networking websites. Instead of submitting an innocuous comment, a user may submit text that contains arbitrary JavaScript. When this “comment” eventually gets posted to the website, the website will include the harmful JavaScript, which will get executed by the client’s web browser [49, 82].

Luckily, cross-site scripting attacks are relatively easy to prevent. The web developer must simply *sanitize* any inputs by removing “illegal” scripting commands. Unfortunately, many web authors do not even take this relatively simple step [49].

## 1.2 Thesis Statement

The increased use of computing applications over the web has created a crisis in the application of software engineering techniques to such applications. Using Facebook as a case study, this thesis seeks to analyze the design decisions in notable web applications in order to develop a set of best practices for creating applications used over the World Wide Web.

# Chapter 2

## Methods

The study consisted primarily of the development and use of Facebook applications to “probe” Facebook’s web page via the application programming interface, as well as test various theories about the structure, design, and general nature of Facebook’s model of social networking.

While I was an experienced web developer, and had also written desktop applications for a number of years, initially I did not know the specifics of developing applications that worked in conjunction with Facebook’s framework. I began by studying Facebook’s developer documentation in an effort to gain an understanding of the nature of Facebook’s framework [20]. Naturally this not only taught me how to write a Facebook application, but also gave me a look at the underpinnings of Facebook’s API. Facebook maintains a blog for developers which helped detail the development of the framework [31], and a perusal of this blog gave an inside look at the engineering behind the Facebook API. I wrote *PyCatalyst* and *Friend Plucker* to explore the framework.

After learning how to write Facebook applications, my study of the framework itself began. This essentially involved a “two-pronged” attack: writing a few scripts (such as *Grapher*, described in §3.1 and listed in Appendix A) to test various hypotheses about the security

and privacy of Facebook’s framework, as well as analyzing the framework in a qualitative sense to draw various conclusions about the architecture, design, and methodology.

The focus of the analysis of the methodology behind the design and implementation of web applications was not in trying to “hack” a web application. Rather, I attempted to understand and analyze the decisions behind the design of Facebook’s framework, and look for ways to improve the process in the future, or apply the gained knowledge to other similar areas of software development.

The exact implementation and test of various theories is described more closely in Chapter 3. Source code for the scripts is listed in Appendix A.

## 2.1 Developing Facebook Applications

Facebook is notable because it is one of the first prominent social networking site to release an API to a platform on which “third-party” developers can build applications [6, 59]. These applications leverage the power and breadth of Facebook’s own data servers to build useful, fun, or even annoying applications for Facebook’s end-users.

### 2.1.1 Facebook’s Framework

Facebook’s external architecture interface consists of three primary components: a REST-based application programming interface (API), a custom markup language (Facebook Markup Language), and a custom database query language (Facebook Query Language).

Facebook allows for two types of applications: *Website* and *Desktop*. A *Website* application is one that ties directly into Facebook’s web interface. Generally this is content written in a web scripting language like PHP, Python, or Java ServerPages (JSP), and hosted on the application developer’s own servers. The application generates web pages that are grafted into Facebook’s user web pages. The application’s web pages can either be generated

on the developer's server and presented to the Facebook user via an HTML *frame*, or can be marked up using FBML, a restricted subset of HTML. Under this second method, the FBML markup is retrieved by Facebook's servers, parsed, and rendered along with surrounding markup code [35]. Web-based applications may also place a small "widget" on a user's profile page, which acts as an interface for interacting with the application or displaying data provided or processed by the application. Widgets *must* be written in FBML [35].

A *Desktop* application is an application written in a programming language such as C++ or Java, and executed directly on a user's own machine. The application then communicates with Facebook's servers using Representational State Transfer, or REST.

The particular type the application will have is a decision that is based upon the goals of the application's developers. A Website application generally adds content to the user's profile, or enhances his experience while using Facebook. Some Website applications allow Facebook users to rate music or movies, send each other electronic "gifts", or play games like Scrabble or Tic Tac Toe. A Desktop application usually serves a more utilitarian role. For example, some desktop applications run on a user's computer in the background as *dæmons*, and alert the user when they've received a new message or "poke" via Facebook. This allows a user to stay in touch with friends without continually checking Facebook via the web interface.

Facebook seems to prefer integration via web applications. Its developers' documentation database contains much more thorough information for the creation of web applications; even though desktop applications debuted before web applications, most examples are written for web applications. The integration of web applications with Facebook is more seamless. For example, desktop applications must first redirect the user to a login page, then have the user switch back to the desktop app. Web applications login "automatically" and thus do not have this break between start and login.

Facebook's API is so powerful that the company's own engineers now implement most

of Facebook’s new features as applications that can be selectively enabled and disabled by the end user. Thus the framework provides an interface for internal and external developers alike, as well as giving users fine-grained control over their own application experience. Furthermore, not only can users decide which applications to use and which ones to remove, but also whether those applications can append information to their public profiles, publish “stories” in their news feed, or show up in the user’s internal navigation bar. With this design, Facebook has raised the bar for web applications by maximizing customization.

### Service Architecture

Facebook is built on a large array of over 200 servers [46]. Each different network (e.g., university) has its own *domain* (generally a virtual domain); for example, Bucknell’s user profile pages exist at *bucknell.facebook.com*. This helps isolate networks from each other and prevent cross-site scripting attacks, since JavaScript cannot act across domains in most browsers, even virtual domains, unless explicitly given permission [81].

Facebook also makes heavy use of XML and JSON, which it uses as the format for replies to API queries.

**REST-based Architecture** Facebook uses a REST-based architecture for communication with its internal servers. REST, or *Representational State Transfer*, is an architecture commonly used for distributed hypertext-based systems like the World Wide Web [37]. This allows third-party applications to communicate with Facebook’s REST server using simple HTTP requests [17].

REST is similar to SOAP, another method for communicating with web-based applications [86], but differs in its approach to communication: REST focuses on resources (“nouns”), whereas SOAP focuses on functions and methods (“verbs”) [86]. Essentially, Facebook’s REST server allows access to Facebook’s database objects through a set of Uni-

form Resource Locators, or URLs (e.g., web addresses). Resources are manipulated via these URLs, and navigation through Facebook's system is done through a set of hyperlinks. The primary benefit in REST is that it provides a common, uniform interface through which all of Facebook's publicly available resources can be accessed via HTTP. Any programming language capable of communicating via HTTP can communicate with Facebook's REST server.

### **Application Programming Interface**

Facebook releases an "official" API library for PHP and Java, but other parties have written libraries in numerous other languages that are sanctioned by Facebook and released on Facebook's developer website. Libraries are offered for the following languages [20]:

- ASP.NET
- ASP (VBScript)
- ColdFusion
- C++
- C#
- D
- Emacs Lisp
- Java
- Lisp
- Perl
- PHP
- Python
- Ruby
- VB.NET

**Services Provided by the API** A number of web design-related services are provided by the API to the Facebook applications developer:

**Facebook Markup Language (FBML)** is a subset of HTML. FBML provides a way for applications developers to write content to the Facebook user’s profile page. Essentially, the FBML interface allows authors to write information directly to a user’s profile. In order to prevent malicious actions, Facebook applications are limited to writing only FBML tags.

It is important to note that FBML is *only* used on profile pages; applications may still create their own pages on which they can write anything they want.

**Facebook Query Language (FQL)** is a restricted subset of SQL. FQL allows applications to query Facebook’s databases as though they were querying a local database. To prevent *SQL injections*, a type of attack commonly used against web applications, Facebook has designed a limited language.

**Facebook JavaScript (FBJS)** is a restricted subset of JavaScript, a language used to provide dynamic-content facilities to web pages. FBJS was designed to prevent certain kinds of attacks known as *cross-site scripting* attacks.

### Facebook Markup Language

Facebook provides its own markup language, called *Facebook Markup Language* (FBML), to allow client applications to easily mark up the web pages (or web page snippets) that they return after processing. Facebook describes FBML as an “evolved subset of HTML” that can be used to position profile elements on the user’s profile page. [24]. These profile elements are represented as tags in an FBML document, and are automatically styled in accordance with Facebook’s “theme” when presenting the resulting web page to the user. This makes it

easy to design a web page that integrates seamlessly with Facebook’s own layout and design, which allows third-party applications to look and feel like Facebook’s internal modules.

FBML also limits the web markup that can be used in a web page. FBML is limited to a subset of HTML elements; for example, elements like `<embed>` and `<object>` have been removed. This is an attempt to prevent application developers from embedding multimedia content (e.g., movies and music that automatically play) in their applications’ web pages. To include multimedia content, application authors must use standard FBML tags like `<fb:mp3>` and `<fb:swf>`, which allows Facebook to render the specifics of that markup; for example, Facebook can easily prevent such content from playing automatically.

FBML allows some use of conditional statements and boolean expressions to dynamically render content on the page. For example, the tag combination `<fb:if-user-has-added-app>` and `<fb:else>` allow the application author to present certain content to users who have added his application, and other content to users who have not yet added his app [24]. Much of this dynamic content can be done application-side, but some can only be done after the page has been rendered on Facebook’s servers.

Part of the power of FBML comes in the fact that is rendered server-side; Facebook’s own servers process the FBML into HTML, CSS, and JavaScript, and present the resulting web page to the user. This serves three primary goals:

**Security** Facebook can enforce its markup security policies by having the “final say” on a page’s markup. This ostensibly prevents applications from slipping in illegal content, `<embed>` tags or JavaScript. For example, to embed multimedia content, authors must use Facebook-supplied tags like `<fb:mp3>` for MP3 files, or `<fb:swf>` for Flash movies; by rendering the actual content serverside, Facebook can ensure that such content does not play automatically.

**Validation** Facebook’s servers are able to *validate* the markup before it is presented to the

user, ensuring that the resulting web page will be correctly rendered to the user.

**Style** Facebook can apply its own style to the resulting web pages. This means that every application-generated page is consistent with Facebook’s site, which enhances the overall user experience. Rendering the page styles serverside also allows Facebook to provide FBML tags for common page elements; for example, `<fb:submit>` provides a Facebook-styled submit button for forms, so the developer doesn’t have to worry about correct styling [24].

FBML allows application developers to design pages using a consistent, uniform markup language that provides only the features necessary to them, and eliminates all the other features that application developers should not care about.

FBML is a valid XML-based language. FBML is “an additional namespace for HTML”, and well-formed FBML should validate against the FBML Document Type Definition (DTD) [25]. Facebook’s web pages are marked up using XHTML. XHTML, like any XML-based language, can include other XML-based languages in its markup; thus, Facebook is able to mix XHTML and FBML freely in its web documents.

### Facebook Query Language

*Facebook Query Language* (FQL) is a SQL-like language designed by Facebook and used to communicate with its databases. It is a wrapper around more complex API method calls [21]. FQL provides a consistent, uniform way for accessing data, similarly to how FBML provides a uniform interface for structuring web pages. FQL seeks to improve application latency by reducing the number of API calls that must be made to return a set of data. Instead of making an API call, getting a return string, then passing *that* data to *another* API call, applications can make a single FQL query; the query is turned into the appropriate calls, then the *final* form of data is passed back to the caller. Since the biggest bottleneck in any

networked application is communication over the network, this structure can greatly improve the performance of Facebook applications [21].

The biggest benefit of FQL, however, is security. Any communication with a SQL database opens up the possibility of the class of exploits known as *SQL injections*, as described in §1.1.3. In order to minimize the serious impact that the use of SQL can have on the security of a web application or framework, Facebook has designed its own query language called *Facebook Query Language* (FQL). FQL is a subset of SQL that is designed specifically to minimize SQL-based exploits.

FQL limits all database queries to *SELECT* statements. In effect, this creates a one-way conduit for Facebook’s data: information is *read-only* (it can only go *out* of the databases, but cannot be put back *in*). This effectively eliminates the most malicious SQL injections, since they rely on a two-way data stream to allow *DELETE* and *DROP* statements.

Interestingly, Facebook’s developer documentation makes little note of the security benefits of FQL. This omission may be due to the fact that the API does not provide a way to talk directly to Facebook’s databases, which effectively eliminates SQL-based attacks; thus, FQL does not prevent SQL injections any more than the API already did. However, it is still important to note that Facebook’s wrapping of its database in both API calls and FQL is a major security benefit.

## Facebook JavaScript

Facebook provides a mechanism, called *Facebook JavaScript* (FBJS), for developers to embed JavaScript code into their web pages [23, 57]. Instead of running JavaScript in a *sandbox*, or a partitioned runtime environment, Facebook simply prepends a unique identifier (the application’s ID number) onto the beginning of all JavaScript function names when rendering the web page. This provides a unique “namespace” for JavaScript functions [23].

Furthermore, Facebook provides its own way of traversing a website’s *Document Object*

*Model* (DOM) [23]. The DOM is the internal representation of a web page created by a browser during the parsing phase. The representation is a tree structure in which the nodes of the tree are the tags, attributes, and free-form content of the HTML document. JavaScript running in a browser generally has access to this DOM; this allows JavaScript to dynamically alter page content without having to refresh the page from the server. This ability to dynamically alter a web page's content, without having to communicate with the web server, is often referred to as *AJAX*.

Facebook modifies how JavaScript can access a profile page's DOM [23]. Because these AJAX calls are wrapped in another layer, the Facebook server is able to process them first.

There are a number of reasons for and implications to restricting JavaScript usage in this way. The primary goal is to maintain user's privacy [57]. As noted in §1.1.3, sites that use JavaScript are especially prone to possible cross-site scripting attacks.

### 2.1.2 Creating an Application

Creating a desktop or web application for Facebook is easy. Every Facebook application requires an *API key* and a *secret*, which are used to identify a Facebook application. Every application also requires some basic URLs for communication with the Facebook servers; web applications require even more information.

The aspiring Facebook developer should begin by visiting the Facebook Developers homepage [20]. The site guides a user in creating his first application. A helpful (and prominent) "Get Started" link leads the developer to a page that demonstrates the creation of Facebook applications.

The first step in creating a Facebook application is adding the *Developer* application to one's profile. A link to this application is provided on the "Get Started" page described above. The addition of this application provides the developer with a link to his application

in his left-hand navigation. This, in turn, allows the developer to browse documentation, as well as apply for the API and secret keys necessary to create an application.

To start an application, one should log into Facebook and click on the Developer application’s link in the left-hand navigation bar. Clicking “Set Up New Application” allows one to register an application. A form used to initialize an application will be presented to the user. Most of the settings on this form are straightforward, but the more prominent settings are described below:

**Application Name** Every application requires a unique name.

**Application Keys** Every Facebook application requires two keys: an *API key*, which is a unique identifier for an application; and a *secret*, which is used to authenticate requests [27].

**Callback URL** Every application—web or desktop—must have a *callback URL*, a page to which the user is directed upon logging in to Facebook. *The trailing slash should be included in this URL*, since Facebook *concatenates* page addresses onto this URL.

**Canvas Page** The URL of a page that generates content for the Facebook *web* application.<sup>1</sup>

**Application Icon** This is displayed in the application user’s left-hand navigation box. This is not necessary for desktop applications, since they will never be displayed in the left-hand navigation.

To complete the creation of the application, one must click “Submit”. If the applicant made any errors in the filling out of the application, he will be prompted to fix them. If the application has been completed correctly, he will be presented with a page that lists the basic pieces of information about his application, as well as supplying him with the crucial API key and secret.

---

<sup>1</sup>This does not seem to be necessary for *desktop* applications, although Facebook asks for it anyway.

While filling out the form, the developer selected whether he wanted to build a web or desktop application. Building a desktop application is fairly straightforward—one writes an application that can be run on the target platform (e.g. Linux, Mac OS X, or Windows), and uses networking libraries to “talk” to the Facebook servers via HTTP.

Building a web application is a bit more work. One must have access to a web server and build his application using a language that can be run on the server (e.g., PHP or Ruby).

The completion of the above steps results in a functioning Facebook application. The only step left is the actual coding of the application. There are many tutorials available on the World Web Web to describe how to design and build a Facebook application. The following sections describe the steps I went through in building my own Facebook applications.

### **PyCatalyst: A Desktop Application**

The first application I wrote was called *PyCatalyst* (listed in Appendix A), and was written in the Python programming language. Introducing me to the Facebook API was the primary goal of PyCatalyst. I selected Python because it is a simple language that allowed me to write a script fairly quickly. Initially I planned on writing an application in Java, but the Java library supplied by Facebook was cumbersome and poorly documented; it also had a number of dependencies, such as the Java JSON library, which did not ship with the library. Furthermore, the dynamic nature of web applications made working with Java cumbersome.<sup>2</sup>

The library that I used to build PyCatalyst was *PyFacebook* [2]. PyFacebook is developed by third-party developers. PyFacebook provides a class, *Facebook*, that acts as a mediator between a client application and the Facebook service. Facebook has performed no security audit or other analysis of these third-party frameworks; it merely lists them as a courtesy to developers [28]. However, I do not see this as a problem, since Facebook is relying on its

---

<sup>2</sup>For example, a lot of Facebook data is returned as lists and dictionaries—two data structures that are integral to Python, but require the creation and manipulation of complex objects in Java. Python also features an *interactive interpreter* that makes developing scripts especially easy.

own architecture to protect user’s private data, and not the integrity of third-party software.

PyCatalyst was fairly simple in its operation: It allowed a user to log in to Facebook, then made calls to Facebook’s REST server and “grabbed” information about the logged-in user. For example, it used the API to obtain the user’s name, his list of friends, and any upcoming events that the user planned to attend.<sup>3</sup>

### **Friend Plucker: A Web Application**

Unfortunately, I found the documentation for desktop applications to be sparse; furthermore, I quickly realized that the World Wide Web was full of tutorials on writing *web* applications that integrate into a user’s profile, but few tutorials existed for desktop applications. This made creating an advanced application rather difficult.

Because of this situation, I made the decision to write a web application. At the same time, I decided to write an application that *did* something “fun”, rather than merely “use” the API—because fun applications are a lot more fun to write than stuffy, boring *academic* applications (not that the two are mutually exclusive, of course).

I wanted to write my application using Python or Ruby on Rails, but decided against these languages for a number of reasons:

1. Bucknell’s Unix webserver did not have support for *mod\_python* and *mod\_ruby*, the Apache modules that allow for easy Python and Ruby integration;
2. While I can write *desktop* applications in both Python and Ruby, I am not experienced in writing *web* applications in these languages.

I decided to write my application using PHP instead. Facebook produces a library for PHP 4 and 5, and provides a helpful tutorial detailing the creation of a simple database-

---

<sup>3</sup>Facebook offers a calendar, as well as an event-creation system that allows users to invite other users to events via the Facebook interface.

backed PHP application [18]. I used this tutorial to create my PHP-based web application, called “Friend Plucker”.

Friend Plucker’s operation was simple. It pulled a list of all of the user’s friends, then randomly selected a friend from the list. The script then retrieved several pieces of information for that friend, including the user’s name, birthdate, picture, and the time of the user’s last profile update. It then displayed this information in an aesthetically pleasing manner. The goal of Friend Plucker was to get a feel for Facebook’s API, rather than do anything truly “useful” or “exciting”.

# Chapter 3

## Discussion

### 3.1 Exploiting the Social Graph

Facebook essentially uses the data contained within its databases to build a *social graph*. In a branch of mathematics known as *graph theory*, a *graph* is a set of *vertices* and *edges* that connect those vertices. An obvious example of a graph is a map of cities and the roads that connect them.

The concept behind Facebook rests on the idea of a *social graph*—that is, a graph of social connections. In a social graph, people are the vertices of the graph, and relationships between these people are edges.

Just like any graph, Facebook’s social graph can be traversed using various *graph traversal* methods, which are algorithms for visiting some or all vertices in a graph by way of the edges. I hoped to investigate Facebook’s social graph with a short program called *Grapher* (listed in Appendix A).

Grapher is a Python script that was written to investigate the possibility of “breaking” the protection mechanism provided by Facebook Query Language. The goal of Grapher was to exploit the concept of the social *graph* in Facebook by using a simple graph traversal to

gain access to profiles to which the application’s user would not normally have access. FQL was selected as an attack vector because it appeared that FQL would allow easy access to profile data, simply by supplying a valid user ID [29].

This attack rested on a facet of application permissions in Facebook: under certain conditions, a third-party application can have access to a user’s profile information *even if that user has not added the application*. This can occur if a friend of the user adds the application; if a friend adds the application, the application has access to some of his friends’ data, even if they have not explicitly added the application themselves.

Theoretically, the attack algorithm worked as follows:

1. Supply a “target” friend, and a friend that the user and the target have in common.
2. Get the UID of the mutual friend.
3. Get a list of all the friends of the mutual friend.
4. Search this list for the *target* friend.
5. Use FQL or the API methods to get the target friend’s information.

In practice, Facebook prevented this possible security breach. Facebook does not allow an application to get the friends of a friend of the person using the application—even through FQL. This layer of prevention was unexpected, but is certainly a positive feature.

## 3.2 Spam

Facebook has two particularly interesting FBML tags: `<fb:wall>` and `<fb:wallpost>`. The Facebook wall is a feature that allows users to post short, public messages on the profile pages of their friends. The tags `<fb:wall>` and `<fb:wallpost>` mimic the wall, as well as individual posts on the wall. While this tag does not allow an application to write to the

“official” Facebook wall, an application can effectively draw a pane on the user profile that *looks* like a wall, potentially tricking unwary users.

Moreover, `<fb:wallpost>` allows an application to make an entry *from any user*. Anwar developed an example in which an application could effectively fake wall posts from other users, as shown in Figure 3.1 [5].

```
<fb:wall>
  <fb:wallpost uid="[victim id goes here]">
    Fady ownz me
  </fb:wallpost>
</fb:wall>
```

Figure 3.1: A piece of FBML code that could be used to create a fake wall post.

A related problem appearing on Facebook is *wall spam*. This annoyance is similar to email spam: automated robot software makes wall posts on user’s profile pages. Initially this problem seemed to be linked to a flaw in Facebook’s API (either a security flaw or a design flaw), but the problem seems to be due to a much less sophisticated *phishing* attack. According to one report, the spam is caused by users who had “entered their login information on a fake page that looked like [Facebook’s] standard login page” [80]. This “attack” is representative of a flaw in the general nature of the Web, not specifically Facebook’s API.

Spam in general has been a problem on Facebook almost since the release of the API. The Facebook API allows applications to invite friends; many applications began to do this automatically, in an effort to increase their vulnerability. This “spammed” the target users’ notification boxes with application add requests. Facebook curtailed this feature in February 2008 [52]. Another problem was misleading notifications. While there was no technical way to prevent misleading application notifications, they were forbidden under Facebook’s Terms of Service [30, 63].

### 3.3 FBML and JavaScript

Felt describes an attack that takes advantage of a vulnerability in Facebook’s markup language, FBML. Some FBML tags, including `<fb:swf>`, allow the web developer to insert Cascading Style Sheet (CSS) commands into the tag. Some browsers, including Mozilla-based browsers (such as the popular Firefox web browser) and Microsoft’s Internet Explorer, provide a way to call a JavaScript file from a style sheet [34, 35]. This attack represents a classic cross-site scripting attack.

Felt uses this feature to inject arbitrary JavaScript into a Facebook profile page. She does this by taking advantage of an application’s ability to write content to a page via FBML. Facebook’s servers do not normally allow the full set of JavaScript functionality, and strip out any JavaScript in content written to a profile page [35]. Felt’s attack takes advantage of `<fb:swf>`, a tag that contains an optional parameter for referencing a Cascading Style Sheet. Instead of referencing a CSS file, though, she instead references a specially crafted XML file. This XML file contains an arbitrary piece of JavaScript code that is executed by the browser [34, 35]. Felt demonstrated this attack using Mozilla Firefox, but it could also work in Microsoft’s Internet Explorer browser, albeit using a different attack vector.<sup>1</sup>

This attack vector was possible because, when rendering FBML, Facebook’s servers only stripped out special characters, and not JavaScript references [34].

Felt’s demonstration was not malicious, but the nature of JavaScript could allow more for attacks far less mundane than Felt’s. In her paper, Felt describes a scenario in which an attacker could inject a JavaScript script that accesses a user’s cookies and impersonates a user in various Facebook actions such as wall posts [35].

Luckily, Facebook patched this vulnerability within a month of Felt’s publishing of his report [34]. It removed the `style` attribute of the `<fb:swf>` tag, which prevents styling

---

<sup>1</sup>Internet Explorer allows CSS to include an `expression()` function that can execute JavaScript.

information from being added to the tag [26]. Instead of having an “all-encompassing” styling tag, Facebook now provides an attribute for the Flash animation’s icon only.

This change effectively prevents Felt’s attack. Therefore, I was unable to recreate her attack, although I did confirm that the *style* attribute in `<fb:swf>` was no longer allowed.

### 3.4 Sandboxing Malicious Web Pages

In computer security, a *sandbox* is a mechanism by which possibly malicious code is allowed to run only in a controlled environment. A sandbox can provide a test bed for untested programs, or provide a safe area to run untrusted applications [55, 78, 87].

Sandboxes are used quite often in computer science and software engineering. One of the most recognized sandboxes is the Java virtual machine (JVM) [44, 78, 87]. The JVM provides a platform on which programs compiled to Java bytecode can be run. Java applications are often embedded in web pages, where they are referred to as “applets”. Applets often come from untrusted sources, such as a web page. To prevent an untrusted applet from wreaking havoc on a system, these applets are run in a “sandbox”, which means they are prevented from accessing, e.g., the file system, reading from and writing to disk, or making outbound calls over a network connection [44]. Essentially, these applets are given nothing more than a small area of the screen on which to present data. The concept of a sandbox has been expanded to other technologies, such as Adobe’s Flash animation software and Microsoft’s Silverlight web technology [87].

Sandboxes are useful, as they allow untrusted code to run on a trusted platform. Facebook utilizes sandboxing in its incarnation of *inline frames*. Facebook applications are allowed to write user content in one of two ways: using FBML or writing into an *iframe*. An *iframe*, short for *inline frame* and named after the HTML tag `<iframe>`, is a mechanism by which one web page can embed another web page.

The use of an inline frame essentially sandboxes an application from the rest of Facebook, at least as far as JavaScript vulnerabilities are concerned [35]. Although the web page looks like *one* page, it is really *two*. Since the pages exist in completely different domains (one at facebook.com, the other at the individual developers' domains), the pages are limited in how easily they can talk to each other. This means that JavaScript in the application's page cannot access DOM elements on the Facebook page, which limits the ability of JavaScript to disrupt the main Facebook site. This mechanism greatly reduces the ability of malicious web application authors to use cross-site scripting attacks to disrupt Facebook.

# Chapter 4

## Conclusion

The results of my study have led to a much better understanding of the nature of Facebook's framework and third-party API. This knowledge has allowed me to make a fair assessment of Facebook's engineering prowess, as well as provide recommendations for future web authors endeavouring to build and deploy a similar framework.

Most importantly, my study has given me better insight on the software engineering disciplines, as well as a better theoretical understanding of how to improve the creation of software.

Since the specific results of my research are documented in Chapters 2 and 3, I instead discuss the ramifications of those results here.

As outlined in the previous two chapters, I can find little fault with the implementation of Facebook's API. The design of the API is strong; it is flexible, yet builds walls around data in such a way that the private information contained in Facebook's vast data store is safe from all but the most tenacious of crackers.

The beauty of Facebook's design lies at least partly in the use of *safe subsets* of existing technology. Consider the problem of cross-site scripting vulnerabilities, as discussed in §1.1.3. Sites traditionally have limited user-generated content to a few HTML tags, such as `<a>`

(the *anchor*, or link, tag); `<b>`, `<i>`, and `<u>` (bold, italics, and underline tags); list creation tags like `<ul>`, `<ol>`, and `<li>`; and select other tags. Obviously this approach would have greatly limited the abilities of authors to create rich content on user profile pages, so Facebook took a two-pronged approach: like other sites, it greatly limited the available palette of HTML markup, but in place of the missing tags, it created an XML-based markup language that provided the functionality of the missing set of tags while eliminating the possibility of authors to do damaging things (inadvertently or purposefully) with those tags. This is a creative solution that sets a precedent in web applications design—a solution that is definitely not used in similar social networking websites such as MySpace.

This idea of utilizing a safe subset of an existing technology is extended gracefully to JavaScript (by way of Facebook JavaScript) and SQL (by way of Facebook Query Language). Together, all three of these technologies are an effort at allowing developers to write applications in familiar languages while still keeping these applications safe for the handling of private data. This is an exemplary solution to the problems discussed in §1.1.3.

## Data Harvesting

One area that particularly concerns me is the idea of a data harvester masquerading as an application. When added to a profile, an application has access to nearly all of a user's profile information. An application could easily masquerade as a fun game or useful utility, but have the dual purpose of storing users' data in the developer's own databases.

Such an action specifically violates Facebook's terms of service [30], but as the old saying goes, it's only illegal if you get caught. Facebook can revoke an application's access if the application is storing data in violation of the terms of service, but if the application does so secretly, the author could presumably get away with it.

This very problem is discussed by Felt [33], who posits that many applications do not

even *need* access to all the data given to them by Facebook’s platform [33]. Felt argues that Facebook’s API should be developed around the *principle of least privilege*, which states that an object should have access only to the data it needs to complete its task [33]. Felt suggests a mechanism, *privacy-by-proxy*, in which an application is given a “stub” of data, rather than the data itself; when the application attempts to display the data, Facebook’s servers will fill in the appropriate pieces of information [33]. Felt points out that this is similar to what Facebook already does when rendering FBML [33]. This is an elegant solution to the problem that is functionally similar to security methods already implemented by Facebook.

The larger question is whether this is a new problem specific only to web-based applications, or the extension of a classic problem into the World Wide Web. Malicious applications have been around since the first viruses appeared in the early 1980s. Users of personal computers have long been told to be mindful of the applications they install and run, because applications are not always what they seem to be.

The same guidelines appear to applications that run entirely on a web-based platform such as Facebook. Since these applications have access to a user’s personal information, even more care should be used in making sure that the application comes from a trusted or otherwise reliable source. Determining what sources are “trusted” and “reliable” is, of course, up to the discretion of the user; common sense certainly applies here.

The difference in user perception between the safety of desktop and web-based applications would indeed be an interesting study unto itself, but would be more appropriate in a psychology-based study rather than this technical analysis.

## Web Engineering

Ginige and Murugesan describe the current state of web development methodology as “chaos” [42].

The way we current use the World Wide Web has stretched far beyond its intended pur-

pose, that being the sharing of scientific data [8, 42]; and web applications are becoming increasingly complex [42, 77]. Some applications are more complicated (in terms of design and architecture) than many of our desktop applications [77]. Due to reasons outlined below, the quality and security of web applications has not increased in relation to the popularity of the World Wide Web [42].

The examination of the technical aspects of Facebook’s third-party framework, as well as the methodologies used in its design, have allowed me to reflect greatly on the nature of web engineering. While I am certainly impressed with the thought and care that obviously went into the creation of Facebook’s framework, the study of the API has made me realize that, as the World Wide Web continues to grow, some fundamental changes may be necessary in the very nature of the tools we use to, and the manner in which we, develop web-based applications.

One major flaw is in the very way we build software. Current software methodology teaches software engineers to employ a technique known as the *waterfall model*, a technique described by W. W. Royce [75]. The exact methodology behind this technique is described in Royce’s paper, but it basically describes several steps necessary to build robust software [75]:

1. Gather and analyze requirements.
2. Design the program.
3. Write the code.
4. Test the software.
5. Deploy.

While the technique *sounds* reasonable, in the very paper describing the technique, Royce writes that the implementation he describes is “risky and invites failure” [75]. His primary concern is that the *entire system* is not tested until the *end* of development, which may require the entire project to be scrapped [56, 75]! Unfortunately, the waterfall model of

software development is one of the models most frequently taught in college curricula [16].

The waterfall method is simply not conducive to web development. One major drawback of the waterfall model is the fact that development time is fairly long [75]. Unfortunately, web applications developers often do not have a long period of time to get their idea onto the web—if one hesitates, one fails.

Take the case of ConnectU and Facebook, for example. ConnectU is a college-oriented social network, much like Facebook [73]. ConnectU’s founders originally worked with Mark Zuckerberg, who took the idea, ran with it, and created Facebook [73]. Would things have been different had ConnectU’s founders implemented their idea first?

Problems with adapting the software engineering model to the web are not new. Engineering problems persist due to four related reasons [42]:

1. It is very easy to make money on the web—much easier than in the world of desktop software. Few independent software vendors get rich by writing desktop software, but a few people with a relatively good idea can make a lot of money on the web. Google, Facebook and YouTube are but three examples.
2. Because of the highly dynamic nature of the web, and the fact that it is relatively easy to make a lot of money with little startup cost, there is often a great “race” to get a web-based application onto the World Wide Web. This means that developers often do not have time (or do not take the time) to rigorously test an application before it goes “live”.
3. Tools like Python, Ruby, PHP, and related frameworks make it easy to build a web application very quickly and with little formal training. More and more, people who have no formal training in software development or security are producing web applications.
4. The web is generally viewed as a print medium, rather than an application medium [42].

Often, developers view web applications development as an extension of graphic design—a form of *art*. While this is certainly true to an extent, a good web application employs both stunning visual elements and a rock-solid architectural foundation. Unfortunately, web applications developers are more likely to be artists rather than engineers [42].

These four reasons converge into two key problems in web engineering:

**Methodology** Even though I agree with the criticism of the waterfall method, as described above, I am reluctant to blame all problems in web engineering on a particular methodology (or any development technique, for that matter). Rather, I think the issue is *apathy*—a failure to rigorously apply engineering methodology in the production of software.

**Languages** Many of our computing technologies, including web technologies, are based on very old frameworks and, perhaps more importantly, are written in very old languages. Many of these languages, such as C, are unforgiving; a careless programmer can easily make mistakes.

Attempts have been made to allow programs to be written in a manner that is more conducive to formal reasoning and analysis. Much of the research in this realm have been in *functional languages*, which allow a program to be represented as a set of mathematical functions [50]. Functional languages are unique in that they have no concept of program *state*; the state of a program is developed explicitly, using functions that depend only on their parameters [50]. This idea is referred to as *referential transparency* [58]. This is in contrast to programs written in languages like C, which make heavy use of global state, and which have functions that are affected by parameters not explicitly passed to the function [58].

In essence, functional programming is the opposite of *object-oriented programming* [58], a form of programming in which the state of a program is stored in discrete packets of data called *objects*.

The debate over the “best” language has likely raged ever since the second programming language was created, but researchers are increasingly advocating the use of functional programming languages [58]. This is due at least partly to the fact that large functional programs are built from relatively simple components that are easy to put together and test. In a purely functional language, functions that work in isolation will work *exactly the same way* when used in conjunction with other functions [58]. This is not necessarily true of object-oriented languages, where much integration testing is necessary to ensure the reliability of a large system. Functional programming helps alleviate at least some of the problems faced by software engineers today, especially when writing large software systems. “There is no reason today not to choose a functional language to implement complex systems—even web applications—particularly in situations where development time is short and there is a need for clear, concise code with predictable behavior” [58].

Even if a functional language is impractical for a particular project, writing with an imperative language in a functionally-oriented way—for example, by keeping in mind the concept of referential transparency—can contribute greatly to the conciseness and correctness of a software program [58].

## Virtual Machines

Many desktop applications are traditionally written in a *compiled* language such as C or C++; whereas most web applications are written in an interpreted language like PHP, Ruby, or Python. This is partly due to the fact that compiled languages tend to have better performance than interpreted languages [7]. However, interpreted languages offer many many benefits; most have an interactive interpreter than can reduce development time, and many interpreted languages have specialized language features that are impossible or difficult to implement in a compiled language [7].

Interpreted languages have another benefit: they essentially run in a sandboxed environment. The interpreter can behave as a sort of “jail” that prevents applications from having direct access to particular sensitive system resources, such as the file system or network.

Increasingly, commercial and freely-available frameworks are allowing developers to use traditionally web-bound technologies on the desktop. The Adobe Integrated Runtime (AIR), for example, allows developers to use XHTML, CSS, and Flash in desktop applications [4]. These technologies have the potential to allow applications written in interpreted languages to be used easily on the desktop. This trend has two key benefits:

**Portability** Applications written in interpreted languages are generally more portable than compiled languages. Programs written in compiled languages must be re-compiled for each targeted platform, which sometimes involves re-writing portions of the application. On the other hand, interpreted languages can be run on any platform for which the interpreter has been written. The interpreter needs to be implemented on each target platform, but generally, the program itself can be run on any platform without modification.

**Security** As noted, interpreted languages are run in an environment—the language interpreter—that can limit the programs’ ability to access critical parts of the system, such as the file system or network. This can effectively prevent many vectors of attacks commonly found in desktop applications.

The viability of web applications on the desktop is discussed further in §4.1.

## 4.1 Future Work

Social networks are not likely to disappear any time soon; in fact, since social networks embody the very principles of the web—free sharing of information through a global web of

people—social networks could be described as the *future* of the World Wide Web. As such, there is ample room for future endeavours in this area.

## Web Applications on the Desktop

The use of traditionally web-bound technologies in desktop applications is growing. The trend began in 2003 with the release of Konfabulator by Arlo Rose and Perry Clarke [74, 91]. Konfabulator (now known as Yahoo! Widgets) is a *widget engine*, a runtime environment for small, utilitarian applications known as *widgets*.<sup>1</sup> Konfabulator was novel in that it allowed developers to create an application in which the user interface (UI) was composed of image files like JPEGs and PNGs, the UI was laid out using XML, and the logic of the application was expressed using JavaScript [91]. Thus, Konfabulator was little more than an XML processor married to a JavaScript interpreter, yet it provided a powerful environment for developers.

Konfabulator’s novelty lay in the fact that developers were able to write full-fledged applications in JavaScript, rather than cumbersome and “unsafe” languages like C, C++, or Objective-C. Furthermore, developers did not have to stick with the traditional rectangular windows of the Mac OS X graphical user interface (GUI); rather, they could design windows and interface controls with whatever style they wished, simply by “drawing” the UI as an image. This allowed a generation of authors who had “cut their teeth” on the web, using HTML, images, and JavaScript, to bring their expertise to the desktop.

This change in software development techniques is a good thing for the reasons listed earlier in the chapter. However, applications built in this manner do represent a marriage of the local exploits found on the desktop with the remote exploits found in web-based applications and other applications that primarily use the network for communication.

---

<sup>1</sup>Mac OS X and Windows users will recognize the concept of widgets from Dashboard and Windows Gadgets.

One interesting avenue of future research would be the creation of a runtime system that allows developers to mix scripting languages and simple image files like the JPEG and PNG formats commonly used on the web, much the same way as Konfabulator. This could bring new interface paradigms to the desktop, casting out the traditional *WIMP* (Window, Icon, Menu, Pointing device) design pattern that is so ubiquitous on the desktop [90], and ushering in UI technologies that are finding more widespread use in devices like gaming consoles and cell phones. The Apple iPhone, for example, has a novel interface that could potentially see more use on the desktop using such a system.

## File Distribution

One area which interests me is the potential for social networks to be more than a simple medium for the exchange of social information. Currently, we mostly use social networks to link people together, through geographic proximity, hobbies, musical interests, employers, schools, and so forth. Thus, the building of social contacts seems to be the primary purpose of these networks.

But the possibilities of social networks are much greater. Facebook has shown that using social networks as a *platform* for applications can add to their utility; however, such utilities are still limited to web-based applications. An expansion of social networking concepts to non-web applications could prove useful.

I am particularly interested in expanding the concept of the social network to file sharing. In 2000, Clark, et al., described the notion of a distributed, anonymous file sharing network called *Freenet* [14]. The goal of the project was to create a network for distributing files that was completely anonymous and thus would subvert attempts to prevent dissemination of such information, as might be done in totalitarian countries [14].

Last semester I worked on an unrelated personal project to leverage the power of social

networks to achieve the goal of secure file exchange. The project built a file sharing network by exchanging user information via a pre-existing social network. This allowed a group of users to build a distributed file sharing network. Security was achieved through a combination of public key encryption and symmetric key encryption using Bruce Schneier's Twofish algorithm [36], similarly to SSH's security mechanism [64].

This particular study, while outside the scope of this thesis topic, does provide insight on ways to use social networks for more than just socializing, and represents another possible avenue of study.

# Bibliography

- [1] JSON. [Online]. Available: <http://www.json.org/>
- [2] pyfacebook. [Online]. Available: <http://code.google.com/p/pyfacebook/>
- [3] A. Acquisti and R. Gross, “Imagined Communities: Awareness, Information Sharing, and Privacy on the Facebook,” in *6th Workshop on Privacy Enhancing Technologies*, ser. Lecture Notes in Computer Science, no. 4258, Privacy Enhancing Technologies. Heidelberg: Springer Berlin, Dec. 2006, pp. 36–58.
- [4] Adobe AIR. Adobe. [Online]. Available: <http://www.adobe.com/products/air/>
- [5] F. Anwar. (2008, Jan.) Fake Facebook Wall Posts Using FBML. [Online]. Available: [http://barmagy.com/blogs/infinite\\_loop/archive/2008/01/20/1054.aspx](http://barmagy.com/blogs/infinite_loop/archive/2008/01/20/1054.aspx)
- [6] M. Arrington. (2007, May) Facebook Launches Facebook Platform. Tech Crunch. [Online]. Available: <http://www.techcrunch.com/2007/05/24/facebook-launches-facebook-platform-they-are-the-anti-myspace/>
- [7] D. M. Beazley and P. S. Lomdahl, “Building Flexible Large-Scale Scientific Computing Applications with Scripting Languages,” 1997. [Online]. Available: <http://citeseer.ist.psu.edu/beazley97building.html>
- [8] T. Berners-Lee, *Weaving the Web*. Harper San Francisco, 1999.

- [9] I. Bicking. (2005, Mar.) Why Web Programming Matters Most. [Online]. Available: <http://blog.ianbicking.org/why-web-programming-matters-most.html>
- [10] A. D. Birrell and B. J. Nelson, “Implementing remote procedure calls,” *ACM Trans. Comput. Syst.*, vol. 2, no. 1, pp. 39–59, 1984.
- [11] R. S. Boyer and J. S. Moore, “A Mechanical Proof of the Turing Completeness of Pure Lisp,” The University of Texas at Austin, Tech. Rep. 37, May 1983.
- [12] T. Bray, J. Paoli, and C. Sperberg-McQueen, “Extensible Markup Language (XML) 1.0,” W3C Recommendation, Feb. 1998.
- [13] J. Byous. (2003, Apr.) Java Technology: The Early Years. Sun Microsystems. [Online]. Available: <http://java.sun.com/features/1998/05/birthday.html>
- [14] I. Clark, O. Sandberg, B. Wiley, and T. W. Hong, “Freenet: A Distributed Anonymous Information Storage and Retrieval System,” in *Proc. ICSI Workshop on Design Issues in Anonymity and Unobservability*, 2000.
- [15] B. Commagere and A. Olson. Zombies. [Online]. Available: <http://www.facebook.com/apps/application.php?id=2341504841&b&ref=pd>
- [16] (2007, Apr.) Educational Flaws: Programming with the Waterfall Model. CompSci.ca. [Online]. Available: <http://compsci.ca/blog/educational-flaws-programming-with-the-waterfall-model/>
- [17] API. Facebook. [Online]. Available: <http://wiki.developers.facebook.com/index.php/API>
- [18] Creating your first application. Facebook. [Online]. Available: [http://wiki.developers.facebook.com/index.php/Creating\\_your\\_first\\_application](http://wiki.developers.facebook.com/index.php/Creating_your_first_application)

- [19] Facebook. Facebook. [Online]. Available: <http://facebook.com>
- [20] Facebook Developers Wiki. Facebook. [Online]. Available:  
[http://wiki.developers.facebook.com/index.php/Main\\_Page](http://wiki.developers.facebook.com/index.php/Main_Page)
- [21] Facebook Query Language. Facebook. [Online]. Available:  
<http://developers.facebook.com/documentation.php?v=1.0&doc=fql>
- [22] Factsheet. Facebook. [Online]. Available:  
<http://bucknell.facebook.com/press/info.php?factsheet>
- [23] FBJS. Facebook. [Online]. Available:  
<http://wiki.developers.facebook.com/index.php/FBJS>
- [24] FBML. Facebook. [Online]. Available:  
<http://wiki.developers.facebook.com/index.php/FBML>
- [25] FBML DTD. Facebook. [Online]. Available:  
[http://wiki.developers.facebook.com/index.php/FBML\\_DTD](http://wiki.developers.facebook.com/index.php/FBML_DTD)
- [26] Fb:swf. Facebook. [Online]. Available:  
<http://wiki.developers.facebook.com/index.php/Fb:swf>
- [27] Getting started guide. Facebook. [Online]. Available:  
[http://wiki.developers.facebook.com/index.php/Getting\\_started\\_guide](http://wiki.developers.facebook.com/index.php/Getting_started_guide)
- [28] Resources. Facebook. [Online]. Available:  
<http://developers.facebook.com/resources.php>
- [29] Users.getInfo. Facebook. [Online]. Available:  
[http://wiki.developers.facebook.com/index.php/Users.getInfo#FQL\\_Equivalent](http://wiki.developers.facebook.com/index.php/Users.getInfo#FQL_Equivalent)

- [30] (2007, Jul.) Developer terms of service. Facebook. [Online]. Available:  
<http://developers.facebook.com/terms.php>
- [31] (2007, Nov.) The Facebook Blog. Facebook. [Online]. Available:  
<http://blog.facebook.com/>
- [32] B. Feld. (2006, Oct.) PHP vs. Ruby. [Online]. Available:  
[http://www.feld.com/blog/archives/2006/10/php\\_vs\\_ruby.html](http://www.feld.com/blog/archives/2006/10/php_vs_ruby.html)
- [33] A. Felt. Facebook Platform Privacy. [Online]. Available:  
<http://www.cs.virginia.edu/felt/privacy/>
- [34] ——. Facebook Project. [Online]. Available: <http://www.cs.virginia.edu/felt/fbook/>
- [35] ———, “Defacing Facebook: A Security Case Study,” University of Virginia, Tech. Rep., 2007.
- [36] N. Ferguson and B. Schneier, *Practical Cryptography*, 1st ed. Wiley, 2003.
- [37] R. T. Fielding, “Architectural Styles and the Design of Network-based Software Architectures,” Ph.D. dissertation, University of California Irvine, 2000.
- [38] T. Finin, L. Ding, L. Zhou, and A. Joshi, “Social networking on the semantic web,” *The Learning Organization*, vol. 12, no. 5, pp. 418–435, 2005. [Online]. Available:  
<http://www.emeraldinsight.com/Insight/ViewContentServlet?Filename=Published/EmeraldFullTextArticle/Pdf/1190120503.pdf>
- [39] (2007, Mar.) The State of Malware Today. FortiGuard. [Online]. Available:  
[http://www.fortiguardcenter.com/reports/roundup\\_mar\\_2007.html#1](http://www.fortiguardcenter.com/reports/roundup_mar_2007.html#1)
- [40] (2008, Jan.) Facebook Widget Installing Spyware. FortiGuard. [Online]. Available:  
<http://www.fortiguardcenter.com/advisory/FGA-2007-16.html>

- [41] J. J. Garrett, "Ajax: A New Approach to Web Applications," Adaptive Path, Tech. Rep., Feb. 2005.
- [42] A. Ginige and S. Murugesan, "Web Engineering: An Introduction," *IEEE MultiMedia*, pp. 14–18, Jan. 2001.
- [43] M. Glaser. (2007, Aug.) Your Guide to Social Networking Online. PBS. [Online]. Available:  
[http://www.pbs.org/mediashift/2007/08/digging\\_deeperyour\\_guide\\_to\\_so.1.html](http://www.pbs.org/mediashift/2007/08/digging_deeperyour_guide_to_so.1.html)
- [44] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers, "Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2," in *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Monterey, California, Dec. 1997.
- [45] P. Graham. (2005, Nov.) Web 2.0. [Online]. Available:  
<http://www.paulgraham.com/web20.html>
- [46] S. Grimm. (2007, May) Largest production memcached install? Email. [Online]. Available: <http://lists.danga.com/pipermail/memcached/2007-May/004098.html>
- [47] R. Gross and A. Acquisti, "Information Revelation and Privacy in Online Social Networks," in *Proceedings of the 2005 ACM Workshop on Privacy in the Electronic Society*. New York: ACM Press, 2005, pp. 71–80.
- [48] F. C. Hennie and R. E. Stearns, "Two-Tape Simulation of Multitape Turing Machines," *J. ACM*, vol. 13, no. 4, pp. 533–546, 1966.
- [49] D. Hu, "Preventing Cross-Site Scripting Vulnerability," Global Information Assurance Certification, Tech. Rep., 2004.

- [50] P. Hudak, “Conception, evolution, and application of functional programming languages,” *ACM Comput. Surv.*, vol. 21, no. 3, pp. 359–411, 1989.
- [51] J. Hylton. (2003, Sep.) A Plan for Improving Web Programming in Python. [Online]. Available: <http://www.python.org/~jeremy/weblog/030917.html>
- [52] P. C. Jeffries. Application Spam. Facebook. [Online]. Available: <http://blog.facebook.com/blog.php?post=10199482130>
- [53] H. Jones and J. H. Soltren, “Facebook: Threats to Privacy,” Thesis, Massachusetts Institute of Technology, Dec. 2005.
- [54] C. Lampe, N. Ellison, and C. Steinfield, “A face(book) in the crowd: social searching vs. social browsing,” in *CSCW '06: Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*. New York, NY, USA: ACM, 2006, pp. 167–170.
- [55] B. Lampson, “Protection,” in *Proc. 5th Princeton Conf. on Information Sciences and Systems*, Princeton, 1971, p. 437.
- [56] M. Langham. (2005, Apr.) Waterfall development model considered harmful. IT Analysis. [Online]. Available: <http://www.it-analysis.com/technology/productivity/content.php?cid=7865>
- [57] M. Laverdet. Javascript on Facebook. Facebook. [Online]. Available: <http://developers.facebook.com/news.php?blog=1&story=32>
- [58] K. C. Loudon, *Programming Languages: Principles and Practice*, 2nd ed. Thomson, 2003.
- [59] R. MacManus. (2007, May) Facebook Grows Up. Read/Write Web. [Online]. Available: [http://www.readwriteweb.com/archives/facebook\\_grows\\_up.php](http://www.readwriteweb.com/archives/facebook_grows_up.php)

- [60] J. Markoff, "The tangled history of Facebook," Newspaper, International Herald Tribune, Palo Alto, CA, Aug. 2007. [Online]. Available:  
<http://www.iht.com/articles/2007/08/31/business/facebook.php>
- [61] I. Maurer. Python Web Development with Django. Xteric Technology Group. [Online]. Available: <http://itmaurer.com/clepy/htdocs/media/presentation/presentation.html>
- [62] L. Montulli, "Persistent Client State in a Hypertext Transfer Protocol Based Client-Server System," U. S. Patent 5 774 670, Oct. 6, 1995. [Online]. Available:  
<http://www.google.com/patents?hl=en&lr=&vid=USPAT5774670&id=Ni0gAAAAEBAJ&oi=fnd&dq=http+cookies>
- [63] D. Morin. Misleading Notifications to Users Will Be Blocked. Facebook. [Online]. Available: <http://developers.facebook.com/news.php?blog=1&story=26>
- [64] Public Key Encryption. National Center for Atmospheric Research. [Online]. Available: <http://www.cisl.ucar.edu/docs/ssh/guide/node4.html>
- [65] T. O'Reilly. (2005, Sep.) What Is Web 2.0. [Online]. Available:  
<http://www.oreillynet.com/pub/a/oreilly/tim/news/2005/09/30/what-is-web-20.html>
- [66] Perl FAQ. Perl Developer Community. [Online]. Available:  
[http://faq.perl.org/perlfaq1.html#What\\_is\\_Perl\\_](http://faq.perl.org/perlfaq1.html#What_is_Perl_)
- [67] Perl Timeline. Perl Developer Community. [Online]. Available:  
<http://history.perl.org/PerlTimeline.html>
- [68] PHP FAQ. PHP Developer Community. [Online]. Available:  
<http://us.php.net/manual/en/faq.general.php>
- [69] addslashes. The PHP Group. [Online]. Available:  
<http://us.php.net/manual/en/function.addslashes.php>

- [70] `mysql_escape_string`. The PHP Group. [Online]. Available:  
<http://us.php.net/manual/en/function.mysql-escape-string.php>
- [71] `pg_escape_string`. The PHP Group. [Online]. Available:  
<http://us.php.net/manual/en/function.pg-escape-string.php>
- [72] About Python. Python Developer Community. [Online]. Available:  
<http://www.python.org/about/>
- [73] D. Riley. (2008, Apr.) Facebook to Settle With ConnectU. Tech Crunch. [Online]. Available:  
<http://www.techcrunch.com/2008/04/07/facebook-to-settle-with-connectu/>
- [74] A. Rose and V. Brosgol. Konfabulator’s History. [Online]. Available:  
<http://www.konfabulator.com/cartoon/partFour.html>
- [75] W. Royce, “Managing the development of large software systems: concepts and techniques,” in *ICSE ’87: Proceedings of the 9th international conference on Software Engineering*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1987, pp. 328–338.
- [76] About Ruby. Ruby Developer Community. [Online]. Available:  
<http://www.ruby-lang.org/en/about/>
- [77] D. Scott and R. Sharp, “Developing Secure Web Applications,” *IEEE Internet Computing*, vol. 6, no. 6, pp. 38–45, Nov. 2002.
- [78] A. Singh. (2006) Sandboxing. Kernel Thread. [Online]. Available:  
<http://www.kernelthread.com/publications/security/sandboxing.html>
- [79] What is Java? Sun Microsystems. [Online]. Available:  
[http://www.java.com/en/download/whatis\\_java.jsp](http://www.java.com/en/download/whatis_java.jsp)

- [80] (2008, Jan.) Invasion of Facebook wall spam. Tech Generation. [Online]. Available: <http://techgeneration.net/2008/01/19/invasion-of-facebook-wall-spam/>
- [81] D. Theurer. Against the Browser's Will: Make Mashups Talk Across Domains. [Online]. Available: <http://www.devx.com/webdev/Article/30860/1954>
- [82] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis," in *Proceedings of the Network and Distributed System Security Symposium (NDSS'07)*, Feb. 2007.
- [83] W3C HTML Working Group, "XHTML 1.0: The Extensible HyperText Markup Language," World Wide Web Consortium, W3C Recommendation, Aug. 2002.
- [84] Wikipedia contributors. JavaScript. [Online]. Available: <http://en.wikipedia.org/wiki/JavaScript>
- [85] ——. JSON. [Online]. Available: <http://en.wikipedia.org/wiki/JSON>
- [86] ——. Representational State Transfer. [Online]. Available: <http://en.wikipedia.org/wiki/REST>
- [87] ——. Sandbox (computer security). [Online]. Available: [http://en.wikipedia.org/wiki/Sandbox\\_\(computer\\_security\)](http://en.wikipedia.org/wiki/Sandbox_(computer_security))
- [88] ——. SQL. [Online]. Available: <http://en.wikipedia.org/wiki/SQL>
- [89] ——. SQL injection. [Online]. Available: [http://en.wikipedia.org/wiki/SQL\\_injection](http://en.wikipedia.org/wiki/SQL_injection)
- [90] ——. WIMP (computing). [Online]. Available: [http://en.wikipedia.org/wiki/WIMP\\_%28computing%29](http://en.wikipedia.org/wiki/WIMP_%28computing%29)
- [91] ——. Yahoo! Widgets. [Online]. Available: <http://en.wikipedia.org/wiki/Konfabulator>

- [92] S. Willison. (2006, Dec.) Why JSON isn't just for JavaScript. [Online]. Available:  
<http://simonwillison.net/2006/Dec/20/json/>
  
- [93] Cascading Style Sheets. World Wide Web Consortium. [Online]. Available:  
<http://www.w3.org/Style/CSS/>
  
- [94] Introduction to HTML 4. World Wide Web Consortium. [Online]. Available:  
<http://www.w3.org/TR/html4/intro/intro.html>

# Appendix A

## Code Listing

The source code of significant scripts written in the course of the study are listed below, for reference.

### PyCatalyst

```
1  #!/usr/bin/env python
2
3  def main():
4      from facebook import Facebook
5
6      ### Global constants ###
7
8      _API_KEY = 'e066a2984f309d12bc61a5b283c661d4'
9      _SECRET  = '05122389a329f3da1334bf70a2feb520'
10
11     facebook = Facebook(_API_KEY, _SECRET)
12     facebook.auth.createToken()
13     facebook.login()
14
15     print 'Waiting for login...'
16     raw_input()
17
18     facebook.auth.getSession()
19
20     # Get info of logged-in user
```

```

21     info = facebook.users.getInfo(
22         [facebook.uid], ['name', 'birthday', 'sex'])
23     name = info[0]['name']
24     print '%s (%s)' % (name, info['uid'])
25     print '%s, born on %s' % (info['sex'], info['birthday'])
26     print ''
27
28     # Find out whether the application has been added
29     added = facebook.users.isAppAdded()
30     if added:
31         print '%s has added app' % name
32     else:
33         print '%s has not added app' % name
34     print ''
35
36     # Print user's groups
37     groups = facebook.groups.get([facebook.uid])
38     print '%s is a member of the following groups:' % name
39     for group in groups:
40         try:
41             print '    %s' % group['name']
42         except UnicodeEncodeError:
43             # These occasionally happen. Just ignore.
44             pass
45     print ''
46
47     # Get logged-in user's upcoming events
48     events = facebook.events.get([facebook.uid], ['name'])
49     print '%s has the following upcoming events:' % name
50     if not events:
51         print '    None (%s is a misanthrope!)' % name.split(' ')[0]
52     else:
53         for event in events:
54             print '    ', event
55
56     if __name__ == "__main__":
57         main()

```

## Friend Plucker

```

1  <?php
2  require_once 'setup.php';

```

```

3
4 $fbml = '<fb:subtitle>Pluck Your Friends</fb:subtitle>';
5 $fbml .= '<fb:header>Random Friend</fb:header>';
6
7 $facebook->api_client->profile_setFBML($fbml, $user);
8
9 $friends = $facebook->api_client->friends_get($user);
10
11 // Get a random friend
12 shuffle($friends);
13 $friend = $friends[0];
14
15 // Print the friend's info
16 $info = array(
17     'last_name',
18     'first_name',
19     'about_me',
20     'birthday',
21     'profile_update_time',
22     'pic_square'
23 );
24 $friend_info = $facebook->api_client->users_getInfo($friend,
25                                                     $info);
26 $friend = $friend_info[0];
27 echo '<img src=' .
28     $friend['pic_square'] .
29     ' style="float:left; padding:10px;">';
30 echo '<img src=' .
31     $friend['first_name'] .
32     ' ' .
33     $friend['last_name'] .
34     '</h1>';
35 echo '<p>Last profile update: ' .
36     date("F n, Y \a\\t g:i A", $friend['profile_update_time']) .
37     '</p>';
38 echo '<p>Born on ' . $friend['birthday'] . '</p>';
39 echo '<p>&ldquo;' . $friend['about_me'] . '&rdquo;</p>';
40 ?>

```

## Grapher

```

1 #!/usr/bin/env python

```

```
2 #
3 # Usage: python grapher.py "Target User" "User Friend"
4
5 import sys
6
7 def main():
8     import facebook
9
10     _API_KEY = "27440f56a23bdfafcdf260ad51eb7179"
11     _SECRET  = "7c3d895532bf7acce88bbe8bc7691d92"
12
13     target = None
14     friend = None
15     fid = 0
16
17     for arg in sys.argv[1:3]:
18         if target is None:
19             target = arg
20         else:
21             friend = arg
22
23     if target is not None and friend is not None:
24         print ("Searching for non-friend %s through friend %s"
25              % (target, friend))
26
27     fb = facebook.Facebook(_API_KEY, _SECRET)
28     fb.auth.createToken()
29     fb.login()
30
31     print "Waiting for login...",
32     raw_input();
33
34     fb.auth.getSession()
35
36     friends = fb.friends.get()
37     friends = fb.users.getInfo(friends, ['name'])
38     for frec in friends:
39         if frec['name'] == friend:
40             fid = frec['uid']
41             print "Found friend: %s [%d]" % (friend, fid)
42             break
43     else:
```

```
44         print "Could not find friend: " % friend
45         return 2
46
47     try:
48         targets = fb.fql.query("SELECT uid2 FROM friend "
49                               "WHERE uid1=%d"
50                               "% fid)
51     except facebook.FacebookError, (strerror):
52         print strerror
53         return 2
54
55     return 0
56 else:
57     print "Usage: python grapher.py ",
58     print '"Target User" "User Friend"'
59     return 1
60
61 if __name__ == "__main__":
62     sys.exit(main())
```