2010

A Social approach to security: Using social networks to help detect malicious web content

Michael Robertson

ROCHESTER INSTITUTE OF TECHNOLOGY

A Social Approach to Security

Using Social Networks to Help Detect Malicious Web Content

Michael J. Robertson

May 2010

Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in
Networking & System Administration

B. Thomas Golisano College of
Computing and Information Sciences
School of Informatics
Department of Networking, Security, and Systems Administration

Rochester Institute of Technology

B. Thomas Golisano College
of
Computing and Information Sciences

Master of Science in
Computer Security and Information Assurance

Thesis Approval Form

Student Name: Michael Robertson

Thesis Title: A Social Approach to Security:
Using Social Networks to Help Detect Malicious Web Content

Thesis Committee

Dr. Yin Pan
Chair

Dr. Bo Yuan
Committee Member

Professor Bill Stackpole
Committee Member

Abstract

In the past six years, tremendous growth in the size and popularity of social networking has fundamentally changed the way we use the Internet. As social aspects to the Internet continue to expand in both quantity and scope, security of the users of social networking sites and the data generated by them will ultimately become an unavoidable concern. This is a realization that malicious users have already had, as viruses, spyware, and phishing scams continue to propagate through social networks at an alarming rate. It is now becoming increasingly critical that the average user also understands this potential for the exploitation of trust among the social networking community. Yet, the security industry has been slow to respond in the act of providing adequate tools for protecting the user.

This thesis describes the development of a proof-of-concept application that uses social networking data to aid in the detection of malicious web content as it propagates through the user's network. While this methodology certainly has its limitations, including user impersonation and false positive rates, the results of testing this application against known phishing and malware sites with real-world user profiles have shown surprisingly positive results.

Acknowledgements

I would like to thank the following people whose support helped to make this thesis possible:

Dr. Yin Pan, for providing me with the opportunity to work on such an interesting topic and for being a constant source of inspiration.

Dr. Bo Yuan, for his support and for teaching some of the most interesting classes offered at RIT.

Professor Bill Stackpole, for always keeping me motivated in my time at RIT and for helping to raise the overall quality of this project.

All of the people who contributed data to the test results for this thesis, without whom the project could not be completed.

Jim and Rae Demme, for their lifelong support and for making my entire education a possibility.

Jessica, for her love and patience throughout this project.

Contents

Appendix

List of Tables

List of Figures

1
Introduction

1.1 Background

In the past six years, the tremendous growth in both the size and popularity of social networking sites, such as MySpace[1] and, most recently, Twitter[2], have allowed people all over the globe to virtually connect, communicate, and share their lives with one another [14, 39]. Of the many examples of these sites, one of the most popular is Facebook[3]—an online community of which at least 85% of college undergraduates in the United States are a part of [38]. In fact, since it was founded in 2004, Facebook's user base has grown to a staggering 400 million people from countries throughout the world [18]. Unfortunately, the proliferation of computer malware, such as viruses and phishing attacks, has also continued to rise in this same span of time [44]. As a result, malicious users and virus writers have found a natural progression between using traditional Internet technologies, such as web browsing and email, and exploiting the inherent trust and size of social networks to help spread their attacks [21].

One of the most famous examples of this new breed of attacks is the Koobface virus, which has plagued users of both Facebook and MySpace. Essentially, the Koobface virus spreads through hyperlinks that appear to come from one of your friends, usually advertising a funny video. When the victim clicks the link to watch the video, they are met with a pop-up message stating that they need to update their Adobe Flash player. When the user clicks to download the "update", they are actually downloading a Trojan horse which installs both a web proxy and a backdoor on the victim's system. At this point, the virus masquerades as the victim and continues to replicate itself throughout the victim's social network [21]. The reason that the Koobface virus has been so successful in spreading is simply because of the inherent level of trust that users put into their use of social networks. Since the links appear to come

[1] http://www.myspace.com
[2] http://twitter.com/
[3] http://www.facebook.com

from their friends, most users will click them and download the virus without ever stopping to consider the consequences of their actions.

1.1.1 Traditional Techniques

In an attempt to combat this problem, computer security researchers have developed tools and techniques to help users identify and avoid malicious content. Static methods, such as blacklists, often have a tremendous success rate when it comes to identifying attacks, though they are often difficult to maintain due to the short lifespan of malware and phishing sites [6, 24, 43, 44]. Dynamic detection methods avoid this problem by allowing the computer and/or browser to make a real-time decision about the site's intent based on attributes such as the server's geographic location, domain name, or offered content [6, 19, 42, 43, 44]. Yet, these methods are not without their own shortcomings as accuracy often becomes a concern. In addition, malware authors are able to design attacks that circumvent these analysis methods, thus rendering the tools essentially useless [6].

1.1.2 Using Social Networks

While several projects have been successful in the area of malware and fraud detection, including examples such as CANTINA, Netcraft, and SpoofGuard, none to date have utilized the tremendous wealth of information found in social networks [6, 28, 43, 44]. The work of this thesis seeks to change this by creating a Facebook application that uses a person's social network to help make a more educated decision about the presence or lack of malicious intent in web content that they are exposed to. Examples of the information that can be garnered from the Facebook databases that would be useful in this application are such statistics as the online identity of someone who sends you a link, the number of mutual friends you have with that sender, and the type of online interactions you have had with that person [17]. In addition, careful analysis of the data made available to a Facebook application can garner other useful information, such as how often a user tends to see messages from the sender. The underlying theory behind this approach is that a link that is sent to you by someone you talk often with, or have many mutual contacts with, is much less likely to be malicious than a link you receive from someone you rarely talk to or who is at the far reaches of your social network.

The inclusion of this information in malware detection techniques is a radically different approach than some of the more classic methods because it seeks to essentially customize the decision process to each individual user in the way that it uses personalized data as decision criteria. As a result, it should be

much more difficult to bypass an application of this type because an attacker would first need to enter a user's social network and gain the trust of the victim, not to mention some of the victim's friends, in order to effectively fool the detection process. All of this must be done in a period of time that is short enough to avoid detection by the site's administrators or the victim, which would likely prompt an account lockout or password change.

1.2 The Facebook Environment

For the purposes of this thesis, Facebook was chosen as the proof-of-concept platform for a variety of reasons. One of these reasons is that the easily accessible API and support for third party applications makes it possible to obtain so much of the data that is critical to the success of this concept. In addition, the user base of Facebook provides an undeniable opportunity to reach large sets of data that would otherwise be unavailable from some of the less popular social networking sites.

While 400 million people is an enormous user base, especially when compared to other popular sites like Digg and YouTube that have a reported 40 million and 24 million users respectively, it is not appropriate to assume readers have a detailed understanding of the Facebook environment [1, 7]. Thus, the following section will give a brief overview of the site and its features.

1.2.1 The Profile

When a user creates a Facebook account they are asked to setup their profile page, which will contain all of the information they want to present about themselves to other users. An example of a typical user profile is shown in Figure 1.1. The user's profile often contains some basic information, such as name, gender, hometown, and contact details. However, some users choose to be surprisingly indiscriminate in both the type and amount of personal information that they post about themselves. While Facebook does offer fairly granular privacy settings to help users choose who can see the data they post, these settings have, historically, been severely misunderstood by their users [36]. The user may also post status messages to their profile that provide updates on their thoughts or activities.

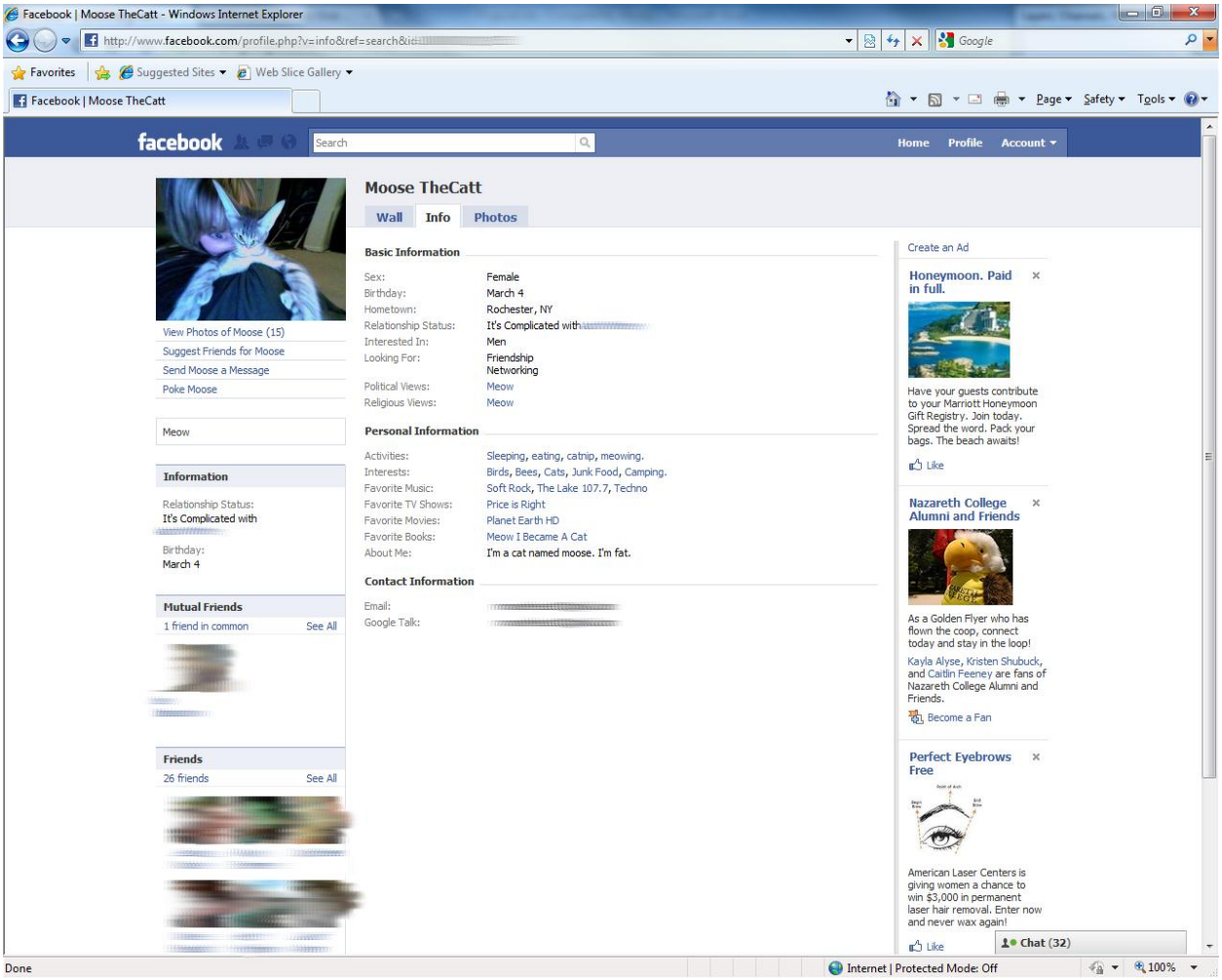

Figure 1.1: Facebook User Profile

1.2.2 The Wall

A user's wall, as shown in Figure 1.2, is a place for their friends to send publicly-viewable messages directly to the user. Unlike a status message, wall posts tend to be more targeted at the Wall's owner. The Wall will also contain a brief "recent activity" section that summarizes updates to the user's profile and their latest actions on the site.

Unfortunately, due to limitations in the Facebook API, the content of a user's Wall is not accessible to a third party Facebook application at this time. As a result, the application developed for this thesis is unable to detect malicious content that is posted directly to a user's Wall unless it shows up inside of a News Feed.

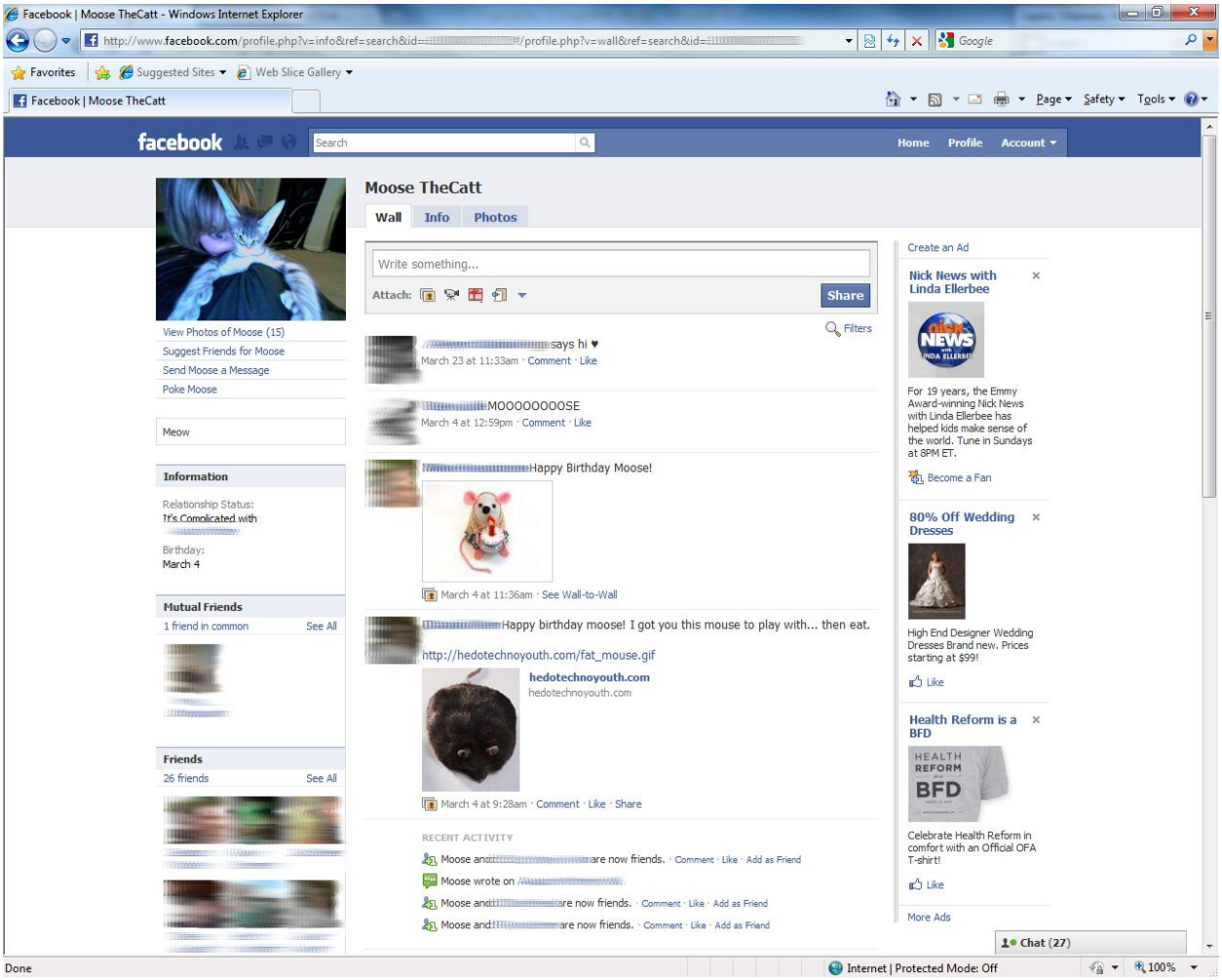

Figure 1.2: Facebook Wall

1.2.3 The News Feed

Like the Wall, the user's News Feed, or stream as it is called in the Facebook API documentation, is a collection of messages, status updates, and shared media posted by others in the user's social network. However, unlike the Wall, which is comprised of messages all directed at the user, the News Feed is an aggregation of content from all of the user's friends. For example, if Bob were to post a status message with a link to a funny video he is watching, this would not show up on Alice's wall. It would, however, be eligible to show up in Alice's News Feed along with all of the other news from her friends. Depending on a user's privacy settings, wall posts between mutual friends may also show up in a user's News Feed. The News Feed is the central focus for this thesis since it is the primary source of content that is automatically scanned for malicious links.

1.2.4 Applications

With the release of the Facebook API, the site began to allow third party developers to extend the functionality of Facebook. In order to create a compelling and useful experience for the user, the API allows an application to access most areas of the site, including user profiles and News Feeds, with the same permissions as the user who is running it. This allows users to post status updates and access information from their friends through individual applications.

Since the release of the Facebook API, applications of every conceivable function, useful or otherwise, have been created to run inside the Facebook environment. Examples of some popular Facebook applications include *Music*, which allows users to post songs to their profiles that can be streamed by others, *Astrology Daily*, which updates the user's status with a daily horoscope based on their birth date, and *Farmville*, a farm simulation game similar to the *SimCity* series that includes social networking aspects to the gameplay.

When a user chooses to run a given application, it is "installed" to their profile by becoming associated with their Facebook account after it is manually approved, as shown in Figure 1.3. Once this is complete, the application is able to execute within the context of the user's profile and perform whatever functionality it has been designed for.

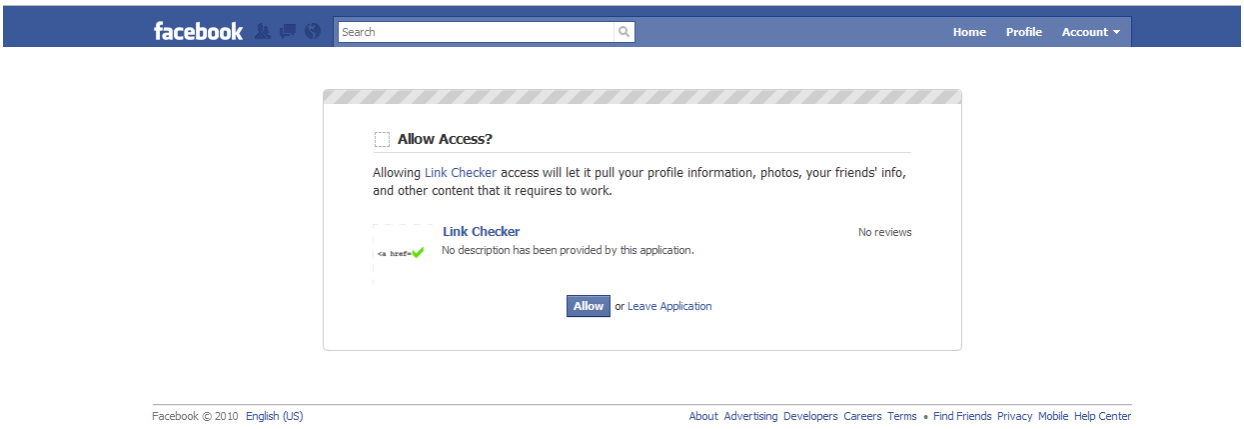

Figure 1.3: Application Approval Request

In a small number of cases, users must explicitly opt-in to certain application functions by approving requests for special permissions. Some of these special permissions include the ability to update the user's status, the ability to send email to the user, and the ability to upload videos and photos to the

user's profile [15]. An example of the special permission request to read the user's News feed, as required by this thesis, is shown in Figure 1.4. Unfortunately, beyond these standard and relatively nondescript opt-in screens, the user has no further insight into what the application is designed to do except for what the developer chooses to reveal in its description. This prevents a very real security concern for users of Facebook, more of which will be discussed in Section 5.3.

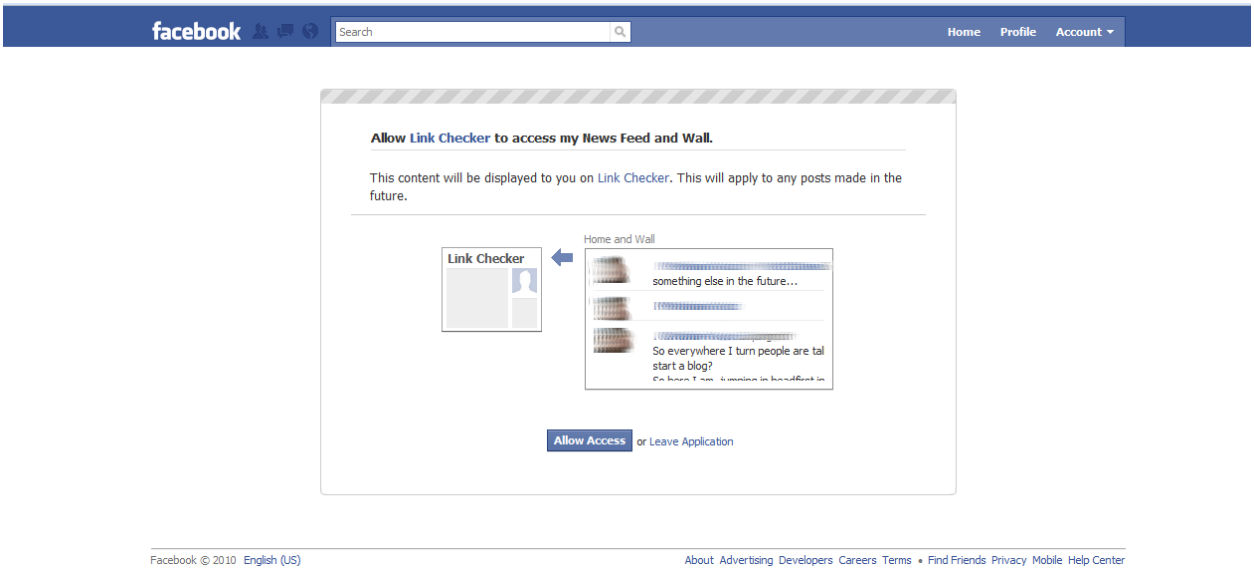

Figure 1.4: Special Permission Request

The proof-of-concept performed by this thesis is developed as a Facebook application. The application relies heavily on the Facebook API, both to provide an avenue for testing as well as access to a user's social networking data.

1.3 Work Presented

This thesis uses a set of PHP and BASH scripts to create a Facebook application that acts as a proof-of-concept for the method of using social networking data to aid in the detection of malicious web content. Data was collected over the course of four weeks from Facebook profiles of real users and analyzed to determine its ability to improve the success rates of some traditional detection methods. The following sections will present detailed descriptions and analysis of these methods, the application's design, and the findings and conclusions of this thesis.

2
Related Work

2.1 Overview

Unfortunately for both social networking users and the security industry, little work has been done to study ways in which malware protections can be applied specifically to online communities. Most of the historical methods have taken the approach of monitoring a user's web browser activity in general, while ignoring the additional data that can be gleaned from social networks to help personalize the protection experience. In addition, nearly all previous efforts have been focused on stopping the spread of either malware or unsolicited (spam) email. It has only been in very recent months that developers have started to look at some of a user's social networking data to help software make more educated and user-tailored decisions. These methods will be outlined in the following sections and the differences proposed by this thesis will be explained.

2.2 The Historical Approach to Security

Although malware research has been taking place since almost the time of the first personal computers, most of what is still considered relevant to the heuristics-based approach to malware detection has been created in the past five to six years. In 2004, researchers from Stanford University released the first iteration of the SpoofGuard plugin for the Internet Explorer browser [6]. The plugin is designed to perform dynamic content analysis for sites a user browses to and attempts to quantify the probability of whether or not the site is a fraud with a value called a "spoof index". In the event that the spoof index for a site exceeds a user-defined threshold, a visual warning is displayed to alert the user to the potential hazard. The spoof index itself is calculated by performing a series of tests, such as searching for URLs that contain suspicious characters like "@" or many subdomains, sites that require passwords but do not have valid SSL certificates, and searching for hashes of images that exist in other legitimate domains, such as E-Trade's etrade.com. Each test returns a result of "0", which indicates a safe score, or "1", which indicates a suspicious score. The test results are then weighted and aggregated to determine the overall score for the website a user is visiting. The result is a relatively effective spoofing detection

algorithm that can also monitor SSL sessions, given that it is in line with the HTTPS transaction. However, since SpoofGuard relies entirely on dynamic heuristics for detection, it is easy for an attacker to adjust the properties of their page to fool SpoofGuard's detection mechanisms.

A Carnegie Mellon University paper released in 2006 described an algorithm for unwanted email detection, called "phishing identification by learning on features of email received", or PILFER [19]. The PILFER algorithm examines URLs for a variety of suspicious characteristics, including ones that are based on an IP address, mismatches between the text of a hyperlink and its destination, the number of "." or subdomains in a link, and the age of the domain that is being linked to, since newer domains tend to have a higher chance of being malicious. When the algorithm was tested against corpora consisting of both legitimate and malicious emails, researchers found that it had a 99% accuracy rate and a less than 1% rate of false positives, which is simply unheard of in the field of automatic malware detection. The only potential disadvantage of such a successful algorithm is actually success itself. In other words, due to the success rate of PILFER, it is likely that spammers and malware writers would find ways to study its detection methods and write malicious code designed to fool these methods. Unfortunately, this is something that is easily done with most dynamic methods.

Just a year later in 2007, researchers released the Carnegie Mellon Anti-Phishing and Network Analysis Tool (CANTINA), which is a security toolbar for Internet Explorer based on the PILFER algorithm [44]. CANTINA uses a somewhat unique approach of utilizing both heuristic-based methods developed for PILFER as well as content analysis. To do this, CANTINA uses the Term Frequency/Inverse Document Frequency (TF-IDF) [11] and Robust Hyperlinks [30] algorithms to generate a list of keywords associated with the content of a site, which is then queried via Google to make a determination about which domain name should likely be associated with that keyword list. The idea behind this detection method is that a keyword list of Windows, Vista, and Office might return a Google search result containing references to Microsoft.com. If the site a user is looking at contains these keywords but is not hosted in the Microsoft.com domain, it may be potentially fraudulent. When tested against a database of 200 fraudulent and legitimate URLs, the CANTINA toolbar was able to detect about 95% of the known phishing sites. Yet, the authors cited performance problems and the limitation of only searching sites written in English as some of its key flaws. These limitations would ultimately have to be resolved before CANTINA was useful for widespread implementation.

The Netcraft Toolbar is another popular security plugin for Internet Explorer that seeks to warn users about visiting potentially hazardous websites [28]. The Netcraft Toolbar largely relies on a community maintained blacklist, making it more up-to-date than traditional blacklists that rely on a single entity for updates. Netcraft also provides some basic heuristic detection techniques for sites that are not listed in the database, such as looking for IP addresses in URLs and non-standard port numbers. One of the advantages of using Netcraft as touted by the developer is that it is unsusceptible to DNS poisoning, since it resolves all URLs into their IP address for domain analysis. However, it does rely solely on visual cues in the toolbar area, which means that some users may miss its warnings if they are not careful or alert.

As of 2007, there were well over a dozen security toolbars claiming to offer users protection against malicious websites. This is likely what spurred a research paper designed to test many of them in real-world scenarios to see which were the most effective [43]. While the experiment included no PILFER-based tools, such as CANTINA, researchers found that SpoofGuard, EarthLink, and Netcraft were the top three performers in terms of success rates. However, SpoofGuard, the most successful of the three, also had the highest false positive rate. Google's Safe Browsing blacklist was also included in the study and although it was not able to identify some of the newest threats, which is a limitation of any static blacklist, its results vastly improved in just 12-24 hours of a site's existence. This would indicate that Google's Safe Browsing API would be an invaluable resource for tools that need to query a blacklist without maintaining their own.

2.3 The Social Approach to Security

In 2005, before most people widely associated the term "social network" with sites like MySpace and Facebook, researchers from the University of Florida and the University of California at Los Angeles proposed an algorithm that could be used to construct a visual mapping of a user's social network based on the header information from emails in their inbox, such as the To:, From: and CC: fields [5]. The network created by this algorithm was then used to make an estimation of which emails were likely spam and which were likely legitimate, based on where the senders lie within the map. For example, when a user lies on the far edge of a network where no common nodes are shared, that user is more likely to be a spammer than someone at the center of the network, with whom one may share many common nodes. The authors of this algorithm were able to successfully implement it in a real-world trial and found that they were able to correctly identify spam about half of the time (56%). However, success

with identifying legitimate emails was much lower at 34% with 47% of the emails examined being marked as unclassifiable [5]. Thus, the algorithm in its initial stages is too underdeveloped to be practical for an average user.

In the latter half of 2009, Twitter finally acknowledged the problem that malicious sites were being spread by exploiting the trust people place in its service [29]. Although Twitter's technical process for deciding what sites are malicious has yet to be determined, users are now blocked from sharing links that are known to contain malware. Facebook also quietly supports a similar feature, which will prevent users from sending certain links in a message if they are reported as "abusive" by the Facebook community, as seen in Figure 2.1. In attempts to contact both Twitter and Facebook for comments on these protections and how they work, only form replies were received (see Appendices K and L).

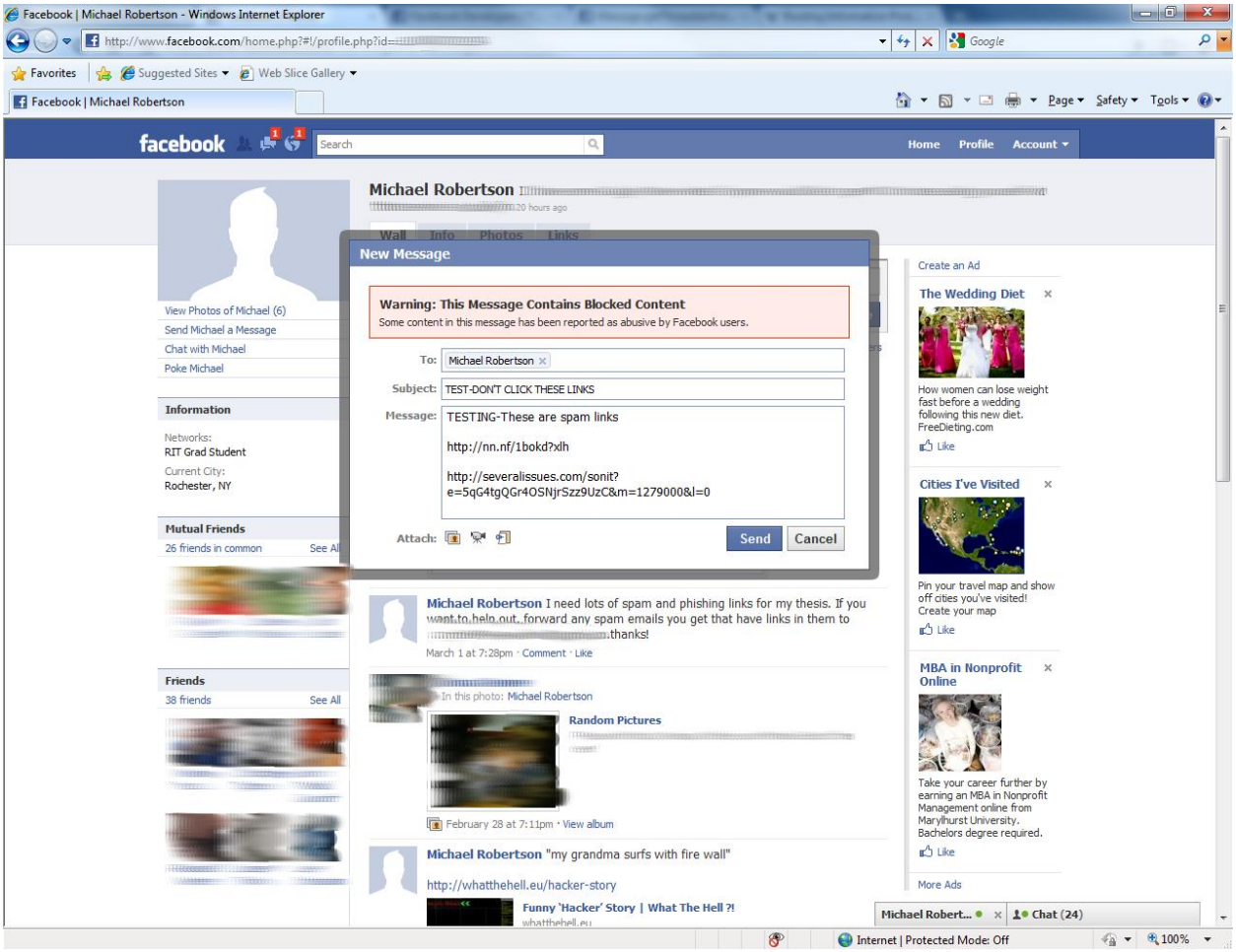

Figure 2.1: Facebook's URL Filtering

Although at the time of this writing shortened URLs, such as those created using TinyURL, are typically ignored by URL filtering services, some services like bit.ly have released similar filters to prevent users from obfuscating the presence of malware with URL shortening services [29]. This is important because URL shortening services make heuristic detection of suspicious URL characteristics difficult and, thus, malware authors have recently depended on them to help spread their attacks.

While it is not exactly an example of Internet security, a new social review site called Unvarnished uses information from a user's Facebook account, which is required for access to the service, to perform some basic levels of fraud protection. In the event that a user was to create a fake Facebook profile to access Unvarnished, the site could detect this and block the user from posting on the site. To do this, the fraud detection looks at both when the Facebook account was created and the number of friends that the user has [35]. Presumably, a profile that was recently created and does not have any friends can be considered fraudulent and access to Unvarnished could be blocked to prevent defacement. As an increasing number of sites allow users to login using a Facebook account through Facebook Connect, these types of heuristics for security and fraud detection have the chance to become more commonplace [16].

To those who are active in technology and social networking circles, it would come as no surprise to state that the use of social networking services has simply exploded in the past six years. However, what may be an unexpected side effect of this statistic is the fact that the spread of malware and phishing attacks through these sites has increased at a rate at or above the legitimate use of these sites. In Microsoft's Security Intelligence Report Volume 7, the company reported seeing a tremendous increase in social networking-based phishing scams in the early- to mid-portions of 2009. While the social networking category comprised only about 10% of the total number of phishing sites in April 2009, this figure jumped to about 50% in May 2009. By June 2009, just one short month later, 76% of all phishing sites categorized by Microsoft's report were in the category of social networking [4].With these alarming statistics, it is shocking to find such few efforts focused specifically on the threats targeting social networking communities.

2.4 New Methods

Much of the aforementioned research has provided a significant theoretical basis for the work performed for this thesis. The heuristic and content analysis methods are all critical components of the

application that was developed and tested. In addition, the experience of past researchers in the successes and limitations of these methods was useful in understanding the best approach to building and testing the tool. However, the key difference that sets this research apart is the use of social networking data from services like Facebook and Twitter to aid in malicious site detection and help reduce the number of false positives produced by the tool. Unfortunately for the security industry, there has yet to be any significant research performed to test the effectiveness of these methods.

Sites like Facebook and Twitter offer an enormous wealth of information that is being automatically compiled and is also more up-to-date than anything a static blacklist would reasonably be able to provide within a set period of time. With the APIs released for these sites, this previously untapped knowledge base can now be queried with relative ease. While the novel use of social networking data has been shown to be effective in many interesting areas of research, including accurately predicting box-office revenues for films [3], it has been left relatively untouched for security research, which is a detriment to the industry.

3

Methodology

3.1 Purpose

The purpose of this thesis was to test the effectiveness of augmenting traditional heuristic-based methods for malware detection with a user's social networking data via a new Facebook application.

3.2 Environment

The application for this thesis was developed and tested on a 1.6 GHz AMD Athlon XP 2000+ PC with 256 MB of RAM. The system used the following software versions:

Name	Purpose	Version
CentOS	Operating System	5.4 (2.6 Kernel)
Apache HTTPd	Web Server	2.2.3
MySQL	Database Server	5.0.77
PHP	Script Processor	5.1.6

Table 3.1: Software Versions

Apart from the software listed above, the application requires few resources. The application must be accessible via HTTP from the client's system.

3.2 Application Design

The methodology of this thesis was coarsely segmented into two separate categories: development of the Facebook application and testing the application's effectiveness. First, the application was developed using the rough process outline depicted in Figure 3.1.

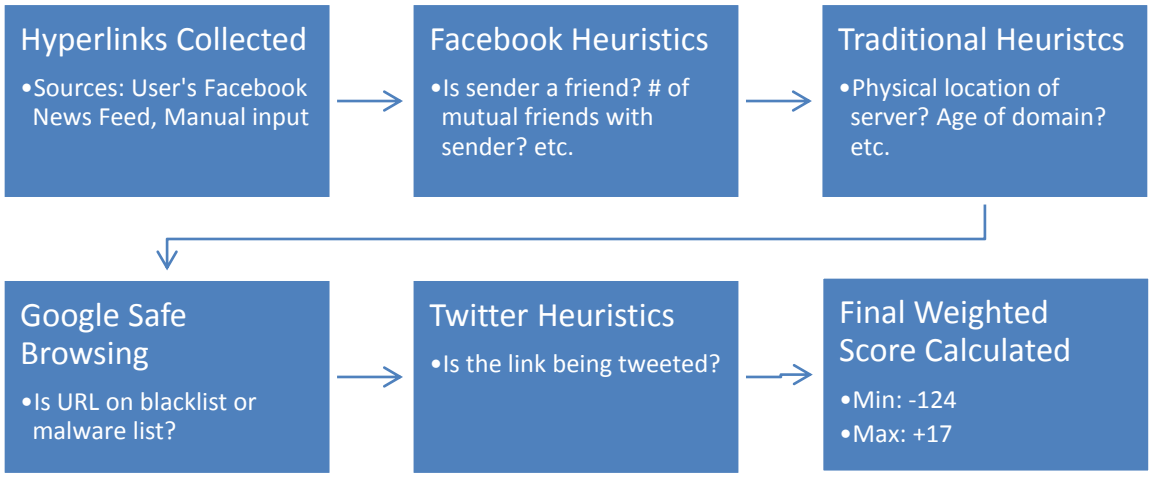

Figure 3.1: Application Process Flow

In order to implement the design shown in Figure 3.1, the application was built from five core scripts: index.php, scanner.php, fb-checks.php, url-checks.php, and manual-scan.php. In addition, several ancillary scripts were required to accomplish some of the primary goals of the application. Each of these scripts is described in more detail in the following sections.

3.2.1 index.php

The index.php script comprises the main page of the application that a user will visit. Upon loading, the script browses through the last 100 posts in the user's News Feed and uses a regular expression to find URLs that may have been posted there. While it does this search, it also keeps a tally of the stream posts seen from each friend. These totals are added to a local database so that it is possible to track the relative frequency with which a user communicates with each of their friends. Unfortunately, because this information is not available directly through the Facebook API, the application must keep track of this on its own in order to provide more effective scanning functionality.

The application will parse both the News Feed posts, as well as their attachments, for evidence of any hyperlinks. When found, the link is added to an array that is eventually sent, along with some metadata about the post and its author, to the scanner.php script. When parsing, the application will keep close

track of the links that are found to prevent duplicate URLs from being scanned. For the purposes of this application, a link is considered a duplicate if the exact URL appears more than once in the same message, including its attachments. Finally, the application will also keep track of the time the scan is being run and purposely not consider URLs which were found prior to time of the last performed scan.

The full source code for index.php can be found in Appendix A.

3.2.2 scanner.php

The scanner.php script is responsible for initiating each of the test functions and will also create the final score report. The test functions are invoked for each hyperlink in the array passed from index.php. Once each test function completes and returns a weighted version of the $GOOD_SCORE or $BAD_SCORE variables (or $NO_SCORE in the event of an error), the script aggregates the scores and a report is printed to the screen for the user's perusal. In addition, a delimited version of the score report is saved to the application server, where it can be further parsed and analyzed by administrators.

In total, the application submits each link to a series of 15 different tests. Of these 15, seven are fully dependent on data mined from social networking content.

The full source code for scanner.php can be found in Appendix B.

3.2.3 fb-checks.php

The fb-checks.php script contains each of the test functions for all Facebook-related checks. As mentioned previously, each test should return a weighted version of $GOOD_SCORE or $BAD_SCORE to be added to the link's final score. In total, six test functions were written that depend on the Facebook API for data. The tests that comprise this script can be found in Table 3.2.

Test Name	Purpose
senderIsFriend	Checks to see if the sender of the link is a friend of the user.
hasMutualFriends	Checks the number of mutual friends the user has with the sender. This test is based on research by

	[5].
attendsEvents	Checks to see if the user is attending/has attended any events with the sender. The theory is that the user does not attend events with malicious users.
photoIsTagged	Checks to see if the user is tagged in any photos with the author. The theory is that the user does not appear in pictures with malicious users.
chatsOften	Checks to see if the user sees posts by the author often. The theory is that a link which comes from an author who does not post links often may be malicious.
isInTrustedList	Checks to see if the author is in the user's trusted friend list that they selected when the application started.

Table 3.2: Facebook-related Tests

The full source code for fb-checks.php can be found in Appendix C.

3.2.4 url-checks.php

The url-checks.php script contains each of the test functions for all URL-related checks. As mentioned previously, each test should return a weighted version of $GOOD_SCORE or $BAD_SCORE to be added to the link's final score. In total, eight test functions were written that analyze the URL for suspicious characteristics. In addition to the six socially-dependent tests written for fb-checks.php, this script adds another, twitterSearch, which is also dependent on social data from Twitter. The tests that comprise this script can be found in Table 3.3.

Test Name	Purpose
serverLocation	Checks the geographic location of the web server. Based on research by [43] and [4].
domainAge	Checks the age of the domain name. Based on research by [44] and [6].

suspiciousCharacters	Checks the URL for the presence of "@" or many (> 1) "-" characters. Based on research by [44] and [6].
nonStandardPort	Checks to see if the website is being hosted on a non-standard port. Non-standard ports are considered to be anything other than TCP/80 or TCP/443. Based on research by [43].
manySubDomains	Checks the URL for the presence of many subdomains (> 3). Most legitimate sites will keep the amount of subdomains to 3 or less for the sake of ease for their visitors (i.e. mycourses.rit.edu, mail.google.com, etc.). However, malicious sites may use many subdomains to trick visitors by using keywords that are associated with other sites (i.e. www.microsoft.com.test.net). Based on research by [44].
hasIPAddress	Checks the URL for the use of IP address instead of domain name. Few legitimate sites will neglect to register a domain name for their site since it makes it easier to remember for users. Malicious sites that will likely be short lived may not go to the trouble of registering a domain name. Based on research by [44].
usesSSL	Checks to see if the URL is using HTTP or HTTPS. Few malicious sites will take the time to setup an SSL version of their site. In addition, this test considers a site built with SSL as more secure. Based on research by [6].
checkSafeBrowsing	Checks the URL hash against the Google Safe Browsing databases (both blacklist and malware) to see if the site is known to be malicious.

twitterSearch	Searches Twitter for the full URL and, if it is not found, the domain used in the URL to see if it is being shared there. The theory is that popular links and well-known domains should appear in Twitter search results. Malicious or obscure sites should not appear in Twitter search results.

Table 3.3: URL-related Tests

The full source code for url-checks.php can be found in Appendix D.

3.2.5 manual-scan.php

The manual-scan.php script was created to provide a manual interface for the URL scanner. Unlike index.php that automatically scans a user's News Feed and extracts links that have been posted there, this script allows the user to pass any arbitrary URLs to the same tests used in the automated scan without requiring that they are first made available within the user's stream.

This interface has benefits for both the user and the researcher. First, the manual scan gives the user the ability to test URLs with the application that were seen in sources other than the News Feed, such as the user's Wall, which is currently inaccessible via the Facebook API, or an email message external to the site. In addition, the researcher is able to test certain URLs that are known to be malicious and should generate a negative score, which would otherwise be blocked from being posted directly on Facebook (see Figure 2.1). This is made possible since any URLs submitted for testing via the manual scan interface are only passed to the application server and are never seen by Facebook or its users.

The full source code for manual-scan.php can be found in Appendix E.

3.2.6 create-db.sql

The application also uses two MySQL databases with several simple tables to keep track of certain information, such as the last time the user scanned their News Feed with the application and the active hashes for the Google Safe Browsing data. This script was run to initially setup these databases on the server. In theory, this script should only have to be run once on the server that is hosting the application data.

```
CREATE DATABASE linkchecker;
USE linkchecker;
CREATE TABLE updateTime (userID varchar(32) not null primary key,
updateTime int);
GRANT SELECT, INSERT, UPDATE, CREATE ON linkchecker.* TO
'linkchecker'@'localhost' IDENTIFIED BY 'linkchecker';
FLUSH PRIVILEGES;

CREATE DATABASE safebrowsing;
USE safebrowsing;
CREATE TABLE updates (id int not null primary key, type varchar(16),
updateTime datetime, version varchar(32));
INSERT INTO updates (id, type, updateTime, version) VALUES (1,
'blacklist', '2010-04-21 12:21:03', -1);
INSERT INTO updates (id, type, updateTime, version) VALUES (2,
'malware', '2010-04-21 12:21:03', -1);
CREATE TABLE blacklist_hash (hash varchar(32) primary key,
INDEX(hash));
CREATE TABLE malware_hash (hash varchar(32) primary key, INDEX(hash));
GRANT SELECT, INSERT, UPDATE, DELETE, DROP, CREATE, INDEX ON
safebrowsing.* TO 'googlesb'@'localhost';
GRANT FILE ON *.* to 'googlesb'@'localhost';
FLUSH PRIVILEGES;
```

3.2.7 sb-black-updates.sh & sb-malware-updates.sh

The sb-black-updates.sh and sb-malware-updates.sh scripts are BASH script designed to periodically check for and apply updates from Google's Safe Browsing API. As part of the API's Acceptable Usage policy, clients must update their lists every half hour and include mechanisms for backing off after an error [12]. These scripts include this functionality and ensure that the database used to store the Safe Browsing information remains up-to-date.

The full source code for sb-black-updates.sh and sb-malware-updates.sh can be found in Appendices G and H.

3.2.8 getcurrentver.sql

The getcurrentver.sql script is a simple script used to retrieve the version of the last update which was applied to the Google Safe Browsing hash database. Each time an update to the hash database is received from Google, the application keeps track of this in the “updates” table of the “safebrowsing” database. The source code for the script is as follows:

```
USE safebrowsing;
SELECT version FROM updates;
```

Upon completion, the script should return the following result, which lists both the blacklist (line 2) and malware list (line 3) versions last received from Google:

```
version
54853
20634
```

3.2.9 inserthashes-black.sql & inserthashes-malware.sql

The inserthashes-black.sql and inserthashes-malware.sql scripts are used by sb-black-updates.sh and sb-malware-updates.sh (see Section 3.2.7) to insert any new hashes received in a Safe Browsing update from Google into the local hash database. The source code for theses scripts is as follows:

```
-- inserthashes-black.sql
USE safebrowsing;
LOAD DATA INFILE '/tmp/blackadds.tmp'
INTO TABLE blacklist_hash
FIELDS TERMINATED BY '+'
(@skip, hash);

-- inserthashes-malware.sql
USE safebrowsing;
LOAD DATA INFILE '/tmp/malwareadds.tmp'
INTO TABLE malware_hash
FIELDS TERMINATED BY '+'
(@skip, hash);
```

3.2.10 recreate-black-hash.sql & recreate-malware-hash.sql

Typically, when a client requests a Safe Browsing update, the API will respond with a "diff" version that lists the changes that must be made to the client's database to bring it up-to-date. In some cases, the client could be so exceptionally out-of-date that the diff list would be larger than if the entire list was sent again. In these situations, the client will drop the old version of the table and recreate it using the recreate-black-hash.sql and recreate-malware-hash.sql scripts. The source code for theses scripts is as follows:

```
-- recreate-black-hash.sql
USE safebrowsing;

DROP TABLE blacklist_hash;
CREATE TABLE blacklist_hash (hash varchar(32) primary key,
INDEX(hash));

-- recreate-malware-hash.sql
USE safebrowsing;

DROP TABLE malware_hash;
CREATE TABLE malware_hash (hash varchar(32) primary key, INDEX(hash));
```

3.2.11 Extensibility

The application was designed to be as modular as possible, allowing new heuristic methods to be added in the future. To add new tests, a developer simply needs to write the testing function, ensuring that it returns a weighted version of the $GOOD_SCORE or $BAD_SCORE variables, (or $NO_SCORE if it encounters an error) and add their function call to the scanner.php script (see Section 3.2.2). This can be done by adding the following lines to the script, which will enable the test and include it in the final score report:

```
$result = <scan_function>;
echo "<scan_name>: " . $result . "<br />";
$output .= "^" . $result;
$score += $result;
```

3.3 Application Testing

Once the application was built, testing began by volunteers who approved the application for use on their profiles and ran the automatic scan on their News Feeds several times. In order to reach the largest amount of potential users in the shortest amount of time, brief descriptions of the application and this thesis, as well as a link to the application, were posted as status messages on two different Facebook profiles. While this method had the potential to reach a total of 377 individuals, only 22 took the time to participate in the study. While the reasons for this are unknown, likely causes include lack of user patience with the slow performance of the application (see Section 5.2) and lack of user interest in participating. In addition, lack of awareness may also be a factor, since users with large networks may see hundreds of News Feed posts in a day and a single request for volunteers may go unnoticed by relatively inactive users.

After several weeks of users performing the automated scans, requests were made of some of the most active users to run manual scans with a pre-compiled test data set. This allows analysis of the application's effectiveness against data that is known to be safe or suspicious. This data set was comprised of 100 URLs, half of which were known to be safe and half of which were known to be suspicious. The 50 safe links were selected at random from the safe data set released by 3Sharp in their 2006 evaluation of various Anti-Phishing tools for the Windows Operating System [34]. This also happens to be a portion of the data set used by [44]. The remaining 50 suspicious URLs were taken at random from two different sources on April 8, 2010: MalwarePatrol's Aggressive Malware Block List and PhishTank's validated phishing list [8, 31]. Once the data set was compiled, it was split into two separate tests, in the interests of performance and reducing the application load. Each user scanned the first data set with a sender of their choosing (if the user had a trusted friend list configured for their Facebook profile, it was asked that they select a friend from this list). At a later time, each user was asked to scan the second data set with a different sender (this time not in the trusted friend list, if one was configured). The results from these tests were collected and analyzed in the same way that data from the automated scans was.

The effectiveness of the application was judged on the number of sites that it can correctly categorize, given a chosen score threshold, including considerations for false positives and negatives. With the values of $GOOD_SCORE and $BAD_SCORE chosen for testing, a link can receive a score between -124 and +17. It should be noted, however, that because scores from the Google Safe Browsing check are

weighted so heavily, a link that is not found in the Safe Browsing database will typically receive a score between -24 and +17.

3.4 Validity

Although 22 people approved the application for use on their Facebook profile, score reports were only received from eight individuals. While the reasons for this are unclear, the external validity of this thesis may be rightly examined.

Likely due to the dangers of Facebook applications and the barrage of information that users see in their News Feed from both friends and other applications, it can be difficult to obtain a large user base for the testing of such an application. For the purposes of this thesis, attempts were made to offset this problem in several ways. First, the application was designed to be as automated as possible so that users only need to click a few times to generate the data needed for this research. In addition, this thesis foregoes seeking a large user base in favor of larger data sets. In other words, rather than trying to get 50-100 people to run the application, the goal instead was to get a handful of people to run the application many times in order to get a large data set. This technique has been used by researchers in the past in other successful studies related to malicious web content [5, 6, 13, 19, 43, 44]. As a result, this thesis was able to generate result sets of well over 800 URLs tested against eight user profiles. While these methods help to overcome the problems of achieving validity, a more formal study may need to be conducted in order to understand how a more diverse set of Facebook users would affect the results of the tests.

3.5 Limitations

3.5.1 API Related

Unfortunately, many of the disadvantages of this method stem from limitations in the Facebook API. For example, because information posted directly to a user's Wall is unavailable unless it happens to show up in a News Feed, there is a large amount of information that the application cannot parse for hyperlinks. This is especially troublesome because targeted attacks focused at a specific user would likely utilize the victim's Wall in order to spread the malicious link. Unfortunately, at the time of this writing, there is no remedy for this limitation provided by Facebook and thus software with more functionality potential, such as a browser plugin, would be required to scan this data.

Another limitation of the API that impacts the amount of information that can be effectively scanned by the application is its accessibility issues with certain data sources. Because there is no simple or efficient way to selectively parse comments to a News Feed post or a user's Facebook Inbox, an email-like environment within Facebook, the application's design forgoes scanning this information in favor of performance and limiting the amount of calls that need to be made to the Facebook API. However, since these are legitimate sources of potentially malicious links, they should still be addressed by future versions of the application.

3.5.2 Non-API Related

While the Facebook API certainly has its limitations, this detection method certainly has its own set of flaws which are unique to its theory. For example, circumvention of the application's scanning abilities could be severely prohibitive if a malicious user was able to impersonate an acquaintance of their victim. In this way, an attacker could exploit the inherent trust users place in social networking services and fool their victims into browsing to a malicious site, provided they were also able to circumvent the other heuristic tests configured for this application. Potential methods of prevention of user impersonation would be to associate some semi-identifiable information to a user's account that is unlikely to change, such as the geographic location of their public IP address or their hard drive's serial number. In this way, account activity that takes place outside of these known parameters could trigger a warning or account verification process that would indicate potential account fraud and prevent the exploitation of trust. Unfortunately, sufficient data of this type was not available through the Facebook API at the time of this writing, and thus was unavailable for inclusion in the application. However, a new privacy policy proposed by Facebook in March 2010 may offer this ability if it becomes approved. The proposed policy alludes to the ability for Facebook to collect, as well as share with third parties in some cases, information such as "your browser type, location, and IP address, as well as the pages you visit". In addition, the policy acknowledges the ability for some of this information to be used for authentication purposes by making "information about the location of your computer or access device and your age available to applications and websites in order to help them implement appropriate security measures" [32].

Shortened URLs also present a concern for any URL-based security analysis. URL shortening services, such as TinyURL and bit.ly, are designed to save characters when sharing long URLs by obfuscating the original address with a vanity URL (i.e. http://www.rit.edu/emcs/ptgrad/grad/1389.php3 becomes

http://bit.ly/bWvy7b). However, because the URL's original properties become masked, it is relatively difficult to perform any dynamic analysis on them. To combat this problem, the application uses the Long URL Please API to detect and expand shortened URLs before they are passed to the scanner. While Long URL Please supports 80 different shortening services, including bit.ly and TinyURL, it is unreasonable to expect it to support all of them [9]. As a result, obscure shortening services have the ability to circumvent most of the URL-based checks this application provides.

Finally, the capabilities of dynamic web technologies, such as JavaScript, present further challenges to a method based purely on URL characteristics. As any web developer will know, it is trivial to redirect a user to a new page once they visit an initial URL [41]. Because of this fact, an attacker could automatically redirect the victim to a malicious site by tricking them into visiting a seemingly harmless page. If the initial URL passes most checks performed by this application, the victim could still become infected by the page they are redirected to. In order to address this problem, future versions of the application would have to perform analysis of the page's content, in addition to its URL, in the way other tools have done in the past [6, 43].

4

Application Usage

4.1 Overview

In order to provide an effortless testing process, running the proof-of-concept application was designed to use as little user input as possible. The application is split into two main functional components: the automated scan and the manual scan. The following sections will provide descriptions of both components, as well as a walkthrough of their usage from the client's perspective.

4.2 Automated Scans

The automated scan is the main real-world data collection component of the application. It is also the first feature that a user will access when they add the application to their Facebook profile. To begin, the user will visit the main application page, which was hosted at http://apps.facebook.com/linkchecker for the course of this thesis. Once here, the user is presented with the prompt shown in Figure 4.1, which requires that the user approve the application for use on their profile.

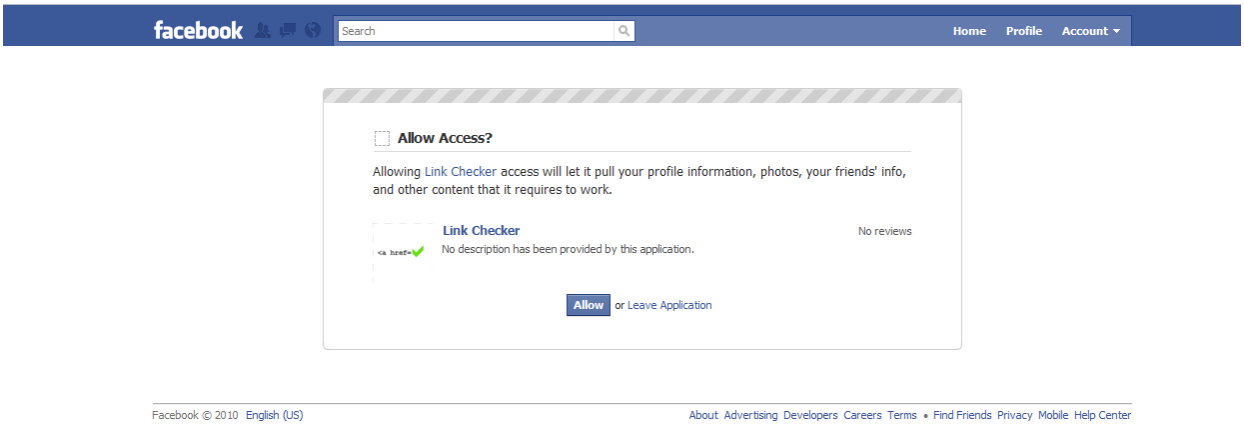

Figure 4.1: Application Approval Request

Once the application's basic functionality is approved, the user is asked to allow the application's ability to read through the user's News Feed, as shown in Figure 4.2. This functionality is required for the automated scan, since it is the primary source of content where the application will harvest URLs from.

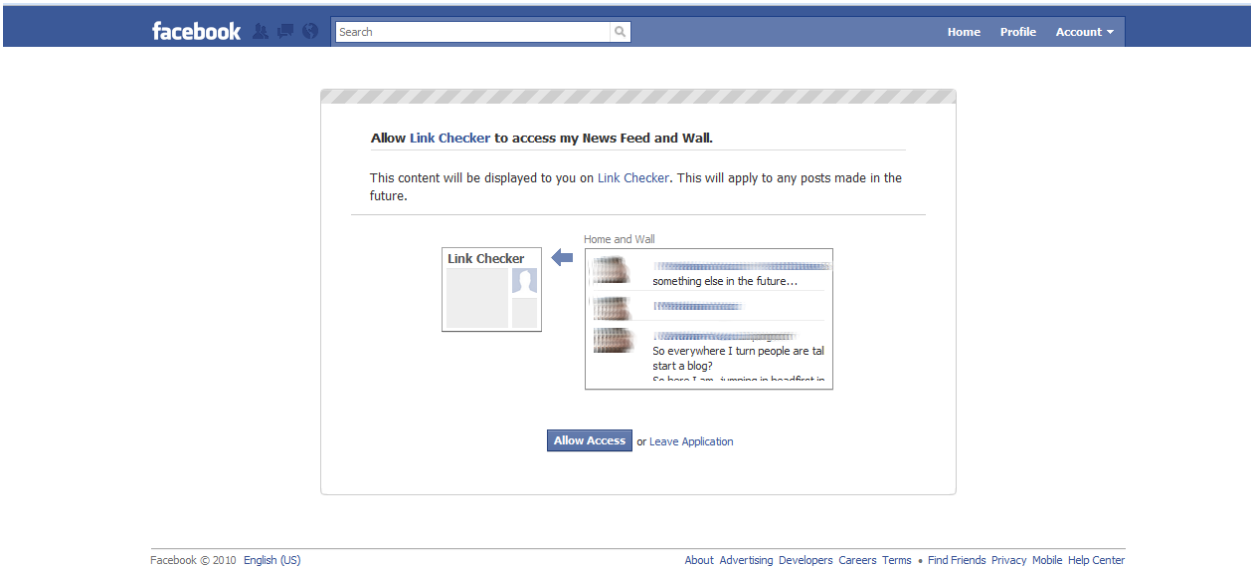

Figure 4.2: Special Permission Request

After all necessary permissions have been granted, the user enters the setup screen for the automated scan. On this page, the user is given a drop-down menu with any Friend Lists that they have configured for their Facebook profile. If the user elects to choose a Friend List on this page, they should select one that contains friends whom they trust content from, as seen in Figure 4.3. An option for "None" is also given, in case the user does not wish to choose a list or does not have any Friend Lists configured for their profile.

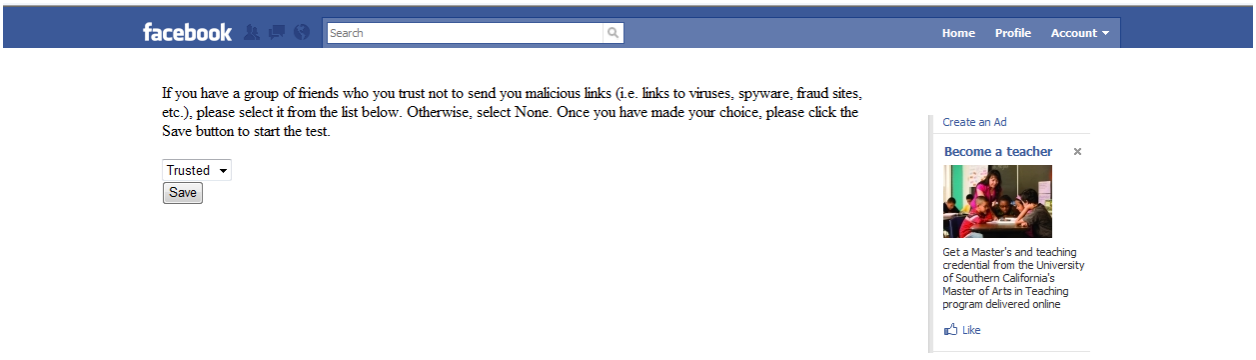

Figure 4.3: Selecting a Trusted List

The option selected by the user on this page is stored in a cookie for 30 days to alleviate the burden of selecting the list each time the application is executed, provided the user does not delete the cookie at the end of their browser session. Once the choice is saved, the automated scan begins. Therefore, the user can execute the automated scan in essentially one click during future sessions, since each of the previously mentioned steps are only required once and will be skipped in subsequent executions.

Unfortunately, due to the Facebook application architecture and the nature of the PHP scripting language, the user sees only a blank iframe inside a Facebook page wrapper while the automated scan is running. During this time, the application is collecting information from the user's News Feed, such as any links that are found and tallies of the number of posts seen from each friend. Depending on the size of the user's network and the number of links that are found in the News Feed, the automated scan can take anywhere from a few seconds to a few minutes to complete. However, subsequent scans should see a slight performance improvement since much of the initial profile creation is only done during the first execution. When the scan completes, the user will see both a message indicating this and receive a copy of the scan report, as shown in Figure 4.4.

Figure 4.4: The Scan Report

The scan report will include each of the links that were discovered, their scores for each test that was run, and a final aggregated score for each link. A delimited copy of the score report is also saved on the server for use in data analysis.

Since the application for this thesis is used strictly for data collection and analysis in a proof-of-concept manor, the application simply ends after the score report is generated. However, with more research and further application design, a decision can be made about the nature of the link (safe or malicious). When this decision is able to be made, the application can also take further steps to protect the user, such as sending an email alert or adding a comment to the News Feed post indicating the nature of the link. Fortunately, all of this functionality is available through the Facebook API. However, it was chosen to forego this behavior for the purposes of this thesis to prevent the application from spamming a News Feed in the event of false positives.

4.2 Manual Scans

The manual scan is designed to allow researchers and users to test links which were discovered outside of their Facebook News Feed, without requiring that they post them publicly. Links are collected manually, but then passed to the scanner and tested using the same scripts as the automated scan, which means that the results should be comparable to that of the automated scan. The user can initiate a manual scan by browsing to the manual-scan.php script, as shown in Figure 4.5.

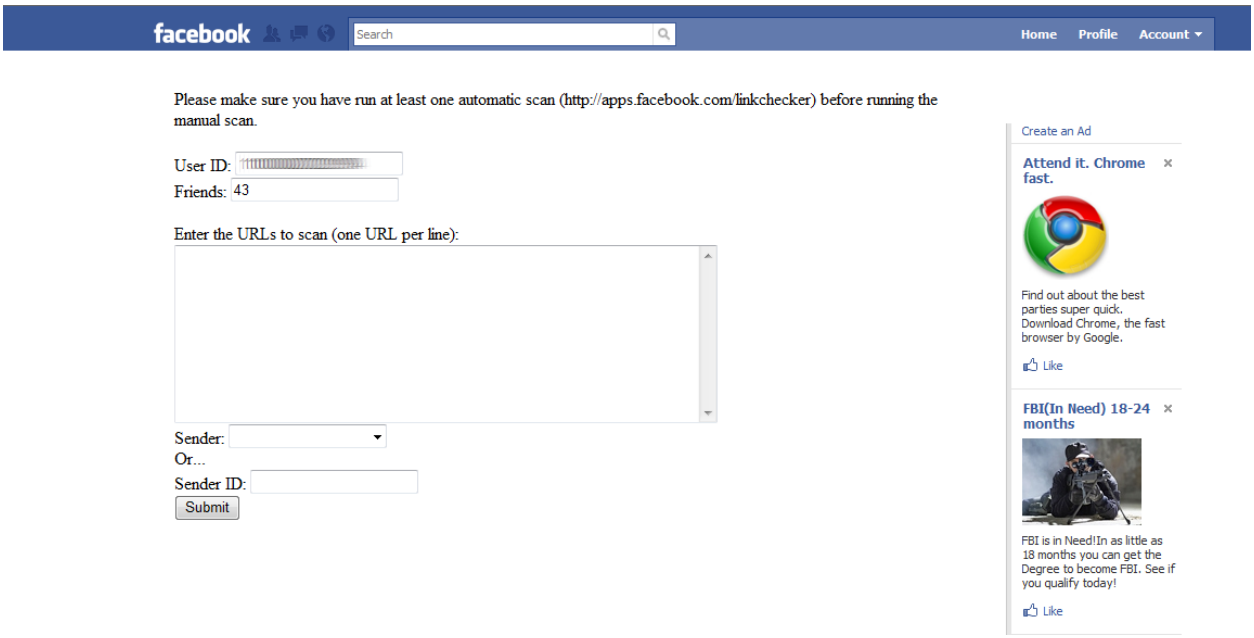

Figure 4.5: Manual Scan Setup

Although the manual scan runs completely independently of the automated scan, the user should have run at least one automated scan before attempting the manual version. This will allow the scan results to be more complete, since the automated scan takes the time to allow the user to choose a trusted Friend List and generates the database of post tallies from each of their friends.

On the manual scan setup page, the user is asked to enter the URLs to be scanned into a text box. The user is also asked to provide the Facebook user ID of the person who sent the link. The user may choose this from a drop-down menu of their Facebook friends list, or specify the user ID manually. This functionality allows the user to quickly and easily select a friend without knowledge of their user ID, or to enter the user ID of a Facebook user without necessarily adding them as a friend on their profile.

When the user finishes filling out and submitting the manual scan form, the URLs provided are parsed and sent to the same scanner that is used for the automated version of the scan. Therefore, application behavior is essentially identical from this point forward and a user will receive the same completion message and score report seen in the automated scan (see Figure 4.4). The only notable difference in these reports is that the post ID for each link will be shown as “0”, since the URLs were provided manually and not found in any News Feed post.

5

Results & Discussion

5.1 Data Analysis

The following sections seek to analyze the data collected by the tests described in Section 3.3. The analysis is broken into two main groups: data for the manual scans and data for the automated scans.

5.1.1 Manual Scan Analysis

Because the manual scan tests were performed with a known data set of safe and malicious URLs, these results can be considered far more empirical and thus will be examined first. For the manual tests, the data set of 100 links was comprised of the URLs listed in Appendices I and J. These 100 URLs were then tested against profiles of four different Facebook users who had also participated in the automated scan testing. The first test consisted of 50 links (25 safe, 25 malicious) and asked users to choose a random friend in their trusted list as the sender. The second test with the remaining 50 links was performed with a user not in their trusted list chosen as the sender. In total, this resulted in 400 data points, which are briefly summarized in Tables 5.1 and 5.2 for the safe and malicious URLs, respectively.

Score Measure	Value
# of Data Points	200
Minimum Score	-13.5
Maximum Score	+10
Mean Score	+3.2
Median Score	+3.5
Mode Score	+3.5

Table 5.1: Safe URL Score Summary – Manual Scan

Score Measure	Value
# of Data Points	200
Minimum Score	-103.5
Maximum Score	+9
Mean Score	-8.2
Median Score	+1.5
Mode Score	+1.5

Table 5.2: Malicious URL Score Summary – Manual Scan

As the high-level results in Tables 5.1 and 5.2 seem to indicate, the application does a decent job of identifying safe and malicious URLS. On average, the mean score for a safe URL is a positive number and the mean score for a malicious URL is a negative number. However, to further understand the effectiveness of the application, one must understand the success and failure rates of its algorithm. To do this, an appropriate scoring threshold must be chosen that will allow the application to make a definitive estimate of the URL's intent. As shown in Figure 5.1, a scoring threshold of +2.5 was chosen for these tests, since this value is exactly in between the median scores for safe (+3.5) and malicious (+1.5) URLs. This means that any score greater than, or equal to, +2.5 will be considered a safe URL and scores less than +2.5 will be considered malicious.

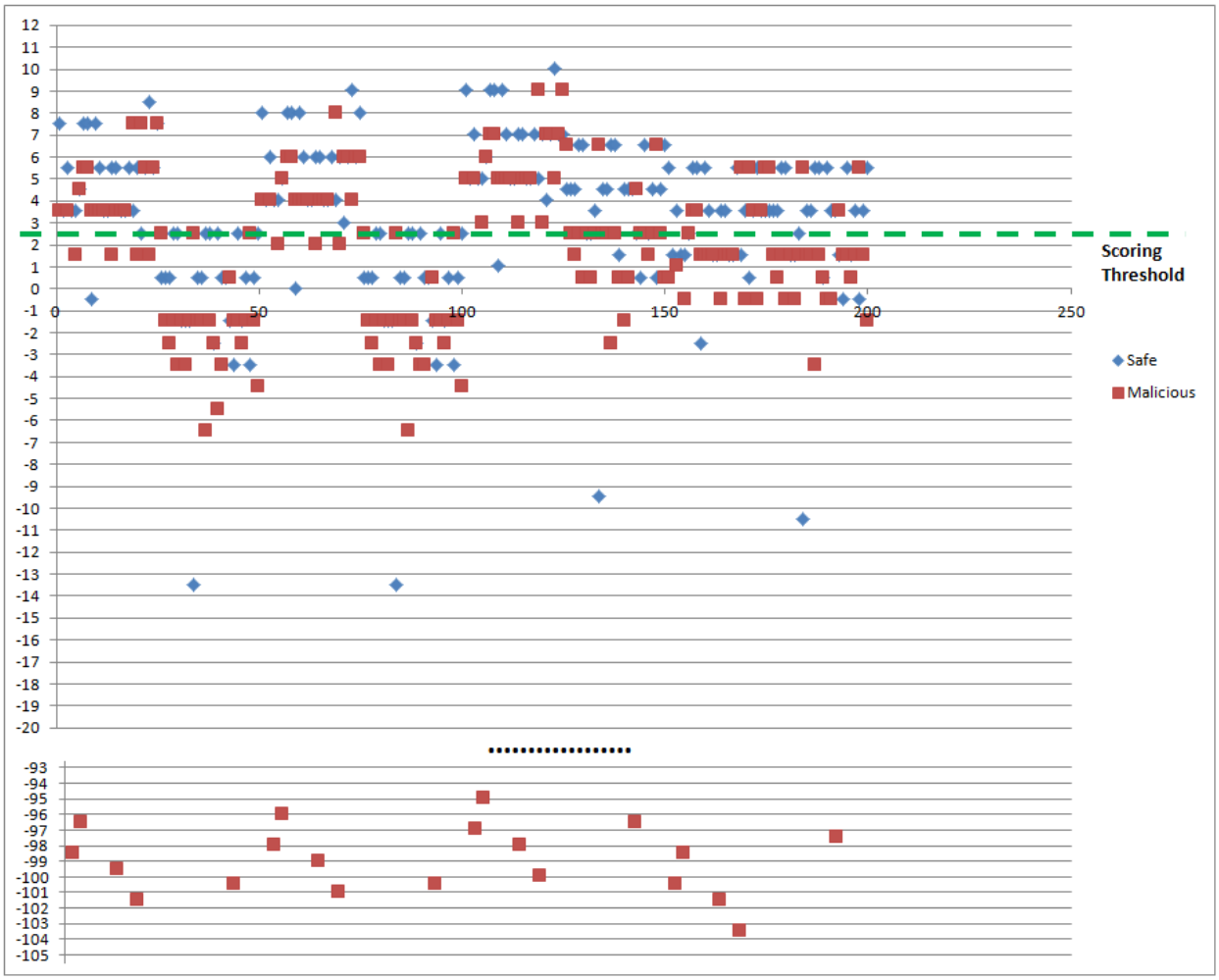

Figure 5.1: Manual Scan Test Scores - Unsorted

Apart from a small group of outliers in the malicious URL results, which was caused by the URL receiving a heavily weighted negative score for being found in the Google Safe Browsing blacklists, the majority of the data points are seen to be heavily grouped in the -2 to +7 range of scores. This is ultimately an indicator of less than adequate performance, since it is difficult to be sure of the results for each URL when it is so close to the scoring threshold. This problem becomes quite apparent when the scores are sorted in numerical order and graphed, as shown in Figure 5.2.

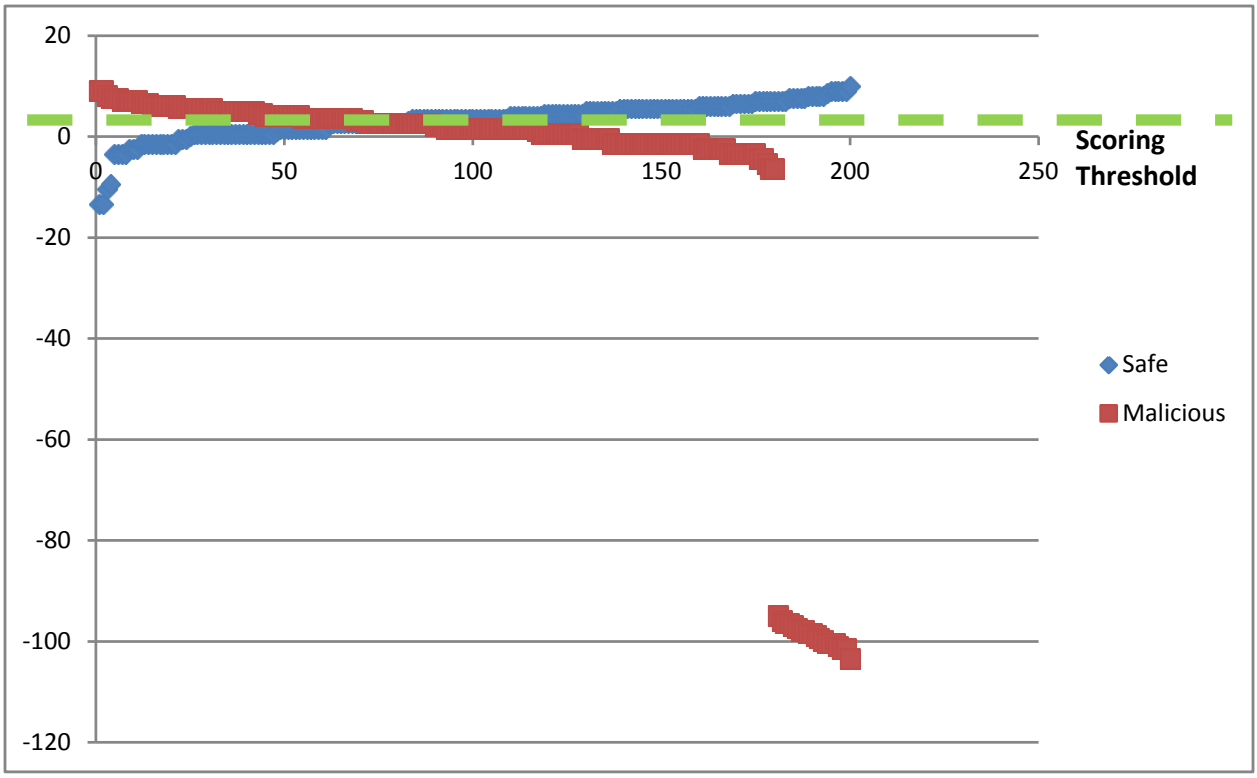

Figure 5.2: Manual Scan Test Scores – Sorted

The tight grouping problem could likely be solved in a number of ways. First, the addition of more tests to the application will allow for more granular scoring and may increase the variance in scores between safe and malicious URLs. In addition, a more significant score weighting system may help to greater the distance between a safe and a malicious score. Finally, increasing the size of the data set will help to determine what typical scores should be for safe and malicious URLs.

Since the malicious status of each link in the manual scan is known, the success and failure rates of the application can be accurately calculated for these tests. These rates are shown in Table 5.3.

Success/Failure Indicator	Value	Percentage
Success Rate	250 of 400	62.5%
False Positive Rate (Safe URLs w/ Malicious Score)	61 of 200	30.5%
False Negative Rate (Malicious URLs w/ Safe Score)	89 of 200	44.5%

Table 5.3: Manual Scan Success/Failure Indicators

As the results from Table 5.3 show, the application was correct 62.5% of the time when using a scoring threshold of +2.5. However, the application also showed fairly significant false positive and false negative rates, which would certainly impact the effectiveness and trustworthiness of the scoring system in a production environment.

In order to fully understand whether or not the use of social networking data is successful in helping to predict the presence of malicious content in URLs, the scores from all social networking-related tests can be removed and the success/failure rates can be re-calculated. The high-level summary of data for the recalculated scores is described in Tables 5.4 and 5.5.

Score Measure	Value
# of Data Points	200
Minimum Score	-9
Maximum Score	+7
Mean Score	+4.9
Median Score	+5
Mode Score	+7

Table 5.4: Safe URL Score Summary – Manual Scan without Social Networking Data

Score Measure	Value
# of Data Points	200
Minimum Score	-100
Maximum Score	+7
Mean Score	-5.8
Median Score	+4
Mode Score	+5

Table 5.5: Malicious URL Score Summary – Manual Scan without Social Networking Data

Using the same method of selecting a scoring threshold half way between the medians of the safe and malicious scores, the newly calculated scores are graphed with a scoring threshold of +4.5, as shown in Figure 5.3.

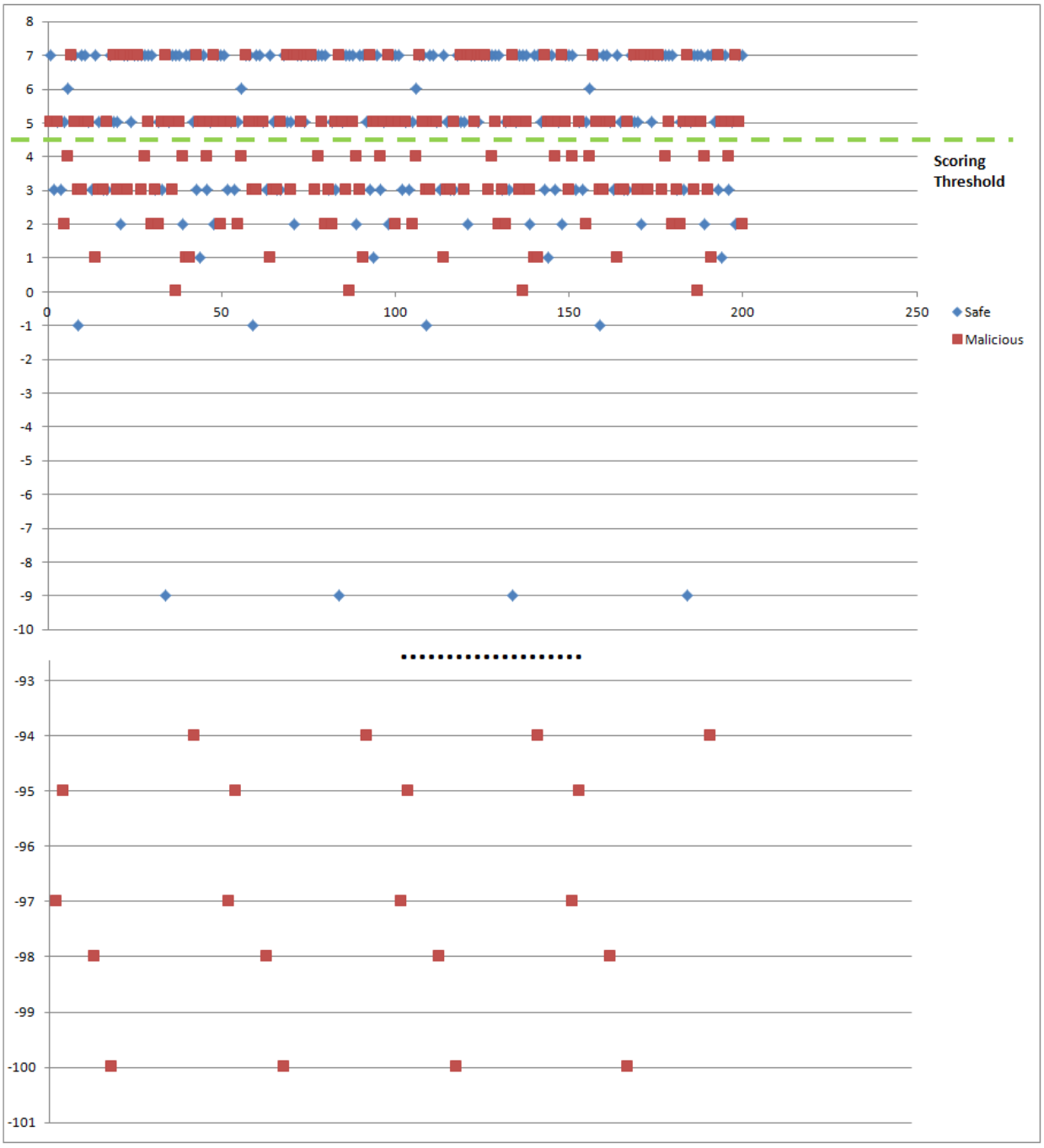

Figure 5.3: Manual Scan Test Scores without Social Networking Data

When compared to Figure 5.1, the scores in Figure 5.3 appear to be much more linear and have a fewer range of values that they occupy. Yet, the success and failure indicators described in Table 5.6 do not seem to indicate that this is beneficial to the effectiveness of the application.

Success/Failure Indicator	Value	Percentage
Success Rate	243 of 400	60.75%
False Positive Rate (Safe URLs w/ Malicious Score)	60 of 200	30%
False Negative Rate (Malicious URLs w/ Safe Score)	97 of 200	48.5%

Table 5.6: Manual Scan Success/Failure Indicators without Social Networking Data

Based on the comparison of the success and failure indicators described in Tables 5.3 and 5.6, it is unreasonable to make an assumption about the effectiveness of the use of social networking data in malicious URL prediction for this sample size, given that the values are so close. However, the results do indicate that there is potential for this method to be improved upon to the point where it does become effective. As the data shows, the use of social networking data allowed the application to identify an additional seven URLs that were malicious that would not have been detected otherwise. While the social networking data did induce a false positive for one additional URL, this behavior could likely be resolved through more appropriate score weighting. Ultimately, a larger sample size would need to be obtained to determine if these results are statistically significant, or even typical. However, it is clear, given the slight increase in effectiveness, that this application and its methods should be studied further.

5.1.2 Automated Scan Analysis

Although the results from the automated scan are much less empirical since the intent of the URLs is not known, analysis of these results is still of interest because it shows how the application performs in a real-world scenario. In total, 421 URLs were collected from eight unique Facebook profiles. The high-level summary of these results is shown in Table 5.7.

Score Measure	Value
# of Data Points	421
Minimum Score	-2
Maximum Score	+11
Mean Score	+6.2
Median Score	+6.5
Mode Score	+6.5

Table 5.7: URL Score Summary – Automated Scan

Again, since the true intent of the URLs is not known, it is impossible to select a scoring threshold that exists between the medians of the safe and malicious scores. In this case, an arbitrary scoring threshold of 0 is selected for the purposes of analysis. This means that any positive score is considered safe and any negative score is considered malicious, since the majority of URLs in a News Feed should be safe. As shown in Figure 5.4, this scoring threshold would likely still result in a handful of false positives.

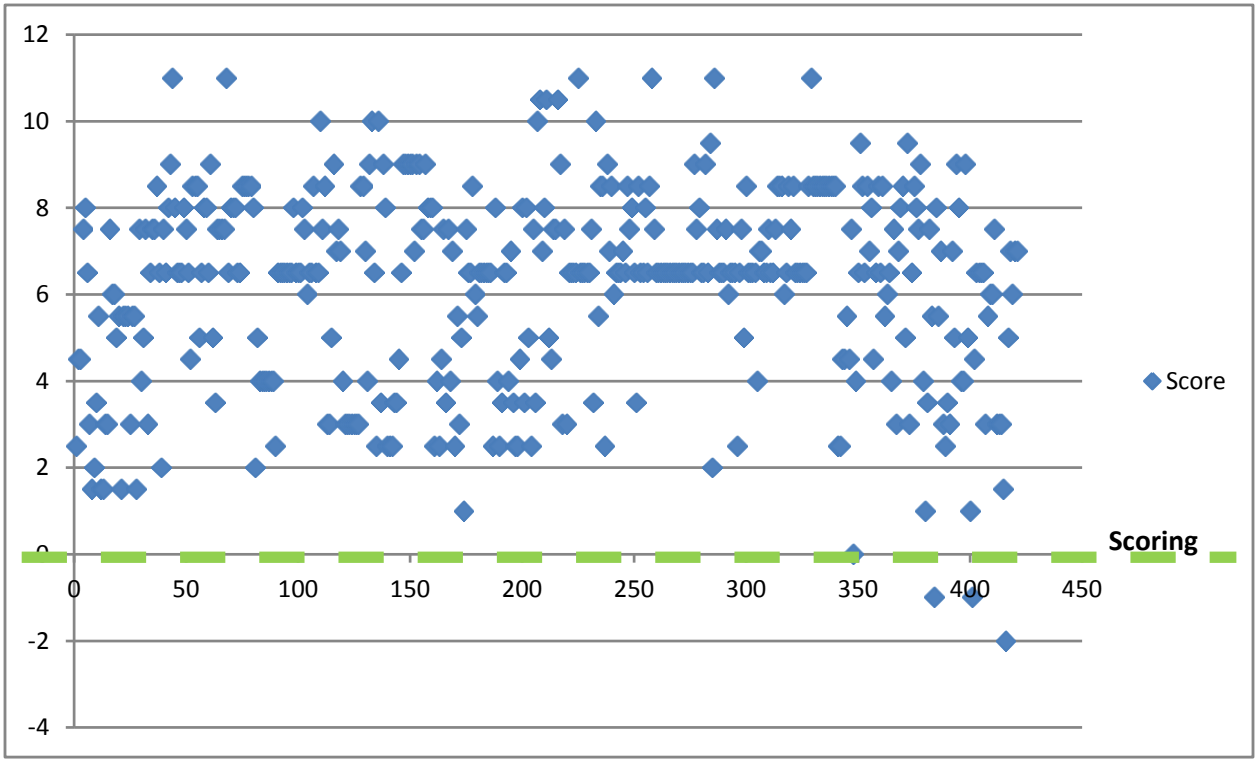

Figure 5.4: Automated Scan Test Scores

Based on the selection of these scoring criteria, it is clear that false positives have been included. For example, www.findyourburt.com, which is a promotional site maintained by the beauty products company Burt's Bees, received a "malicious" score of -2. Though this site clearly does not contain malware, it is unclear whether this should be considered spam/unsolicited content or a false positive. While it is impossible to determine the automated scan's true success and failure rates without further analysis of each URL, it would likely provide the main functionality for a production version of this proof-of-concept application in the future. Therefore, further work must be done to improve its effectiveness and scoring criteria.

5.2 Remaining Issues

The following sections describe the various issues with the application that are still outstanding and could be addressed in future studies or development endeavors.

5.2.1 Performance

With any application, performance is typically a primary concern. In the case of the development for this thesis, some performance was spared in favor of modularity to help aid future additions and changes to the application as APIs evolve and ideas for more tests become available. Therefore, although each link is submitted to each test function independently, this ultimately slows down the application from returning a prompt result when testing many links.

While performance of this application is fairly slow, with some scans of less than 20 links taking several minutes to complete, scan times are still within usable limits, especially considering the hardware they were executed on. In addition, the scan times are certainly within reason when compared to what most users experience with traditional application-based virus scanners. Yet, this issue may seem exacerbated by the fact the user sees nothing but a white iframe while the scan is running. Unfortunately, there is no easy way to present a user with a loading or intermediate screen while the scan is in progress, since the web server will not return any result until the entire PHP script finishes processing. At this point, however, the scan will be complete and the score report will be shown.

More modern server hardware and improvements in the efficiency of the scripts and their algorithms may help to improve application performance and generate a better experience for the user.

5.2.2 Scope

The Facebook application itself is also an issue because of the scope it is allowed to run in. Although the Facebook development platform was chosen deliberately for its API and large user base, the same functionality could be built into a desktop application or browser plugin. While these concessions are acceptable for the proof-of-concept application that is this thesis, a production implementation of this research should have a more overarching scope and not be limited to the data accessible through APIs provided by companies like Facebook, Twitter, and Google, who all have a vested interest in keeping certain data to themselves. This would also allow the application to be exposed to data outside of the Facebook domain, such as a URL found on a friend's blog, and take more active steps to protect the user from stumbling onto a malicious website, such as redirecting the browser or displaying a pop-up warning message.

5.2.3 API Limitations

As mentioned previously, limitations in the APIs used by this application, especially the Facebook API in particular, served to cripple certain functionality of the application that was originally planned for, such as the ability to scan a user's Wall posts. Ultimately, the usefulness of this application is limited by the amount of data that is allowed to be gleaned from the vast information bases maintained by social networking sites. Since this data is the crux of this thesis' core functionality, it becomes a significant issue when it cannot be accessed totally or easily.

It does ultimately seem that the Facebook API provides enough data to third party developers to be able effectively prove that the concepts presented by this thesis are plausible. Yet, cooperation from sites like Facebook and Twitter would be required before an application like this was effective enough for widespread and active use by the Internet community.

5.2.4 Response

Of the 22 users who approved the Facebook application for use on their profile, the score reports received were only a compilation of those from eight individuals. While this is a detriment to the results set obtained for this thesis, it is also an issue that needs to be solved because it means that 14 users could have potentially sought security protection from this application and not have received it. As with any security application, user trust is directly related to its effectiveness, since users who do not trust an application will not allow themselves to be protected by it [42]. While the reasons for the discrepancy in

the number of users versus the number of reports generated could be caused by a number of issues, including slow performance, bugs in the reporting functions of the application, or change of heart by the user after seeing the permission requests from Facebook, more investigation and testing is needed to ensure 100% of the users who run the application receive some type of result.

5.3 Security Concerns

Throughout the development of this thesis, several security concerns became apparent that are specific to Facebook applications. Although they are technically ancillary to the scope of this thesis, they are nonetheless important and directly impact the effectiveness of the proof-of-concept application discussed herein. Thus, they will be described in the following sections.

5.3.1 Information Disclosure

While it was likely a beneficial business decision for Facebook to open their platform with an API and allow third party developers to create content that extends and enhances the user experience, it seems that many Facebook users are fairly indiscriminate when they select applications to add to their profile. In fact, 228 of the 421 URLs scraped from the News Feeds of users in this study by the automated scan were related to Facebook applications of various kinds and, more importantly, various developers. The reason this can be a security concern is that these applications, and subsequently their developers, have access to an enormous amount of the personal information that users add to their Facebook accounts. In many cases, because the application runs within the user's own profile, this access to information is often in lieu of any privacy settings the user may think are protecting them. Worse, some of this access even extends into the information posted by the user's friends, which means personal information could be harvested by an application without requiring the friends also approve the application's access [40]. To be fair, the Facebook API documentation clearly states the various types of information that are accessible to third parties and even provides a sandbox environment to allow users to test what information from their profile is returned by various API calls [17]. However, it is likely safe to assume that most users will never take the time to read the API documentation, especially those who have no knowledge or interest in Facebook application development. Instead, Facebook must do a better job of educating users about the dangers of third party applications and provide more accessible documentation on the exact nature of information being disclosed about them to applications run by themselves and their friends.

5.3.2 Development Environment

Another enormous threat to the security of Facebook application users is inherent in the way the application development environment is designed. When a developer creates a Facebook application, they are given the opportunity to create a vanity URL for the apps.facebook.com domain, as was used by this thesis with http://apps.facebook.com/linkchecker. However, hosting for the application code is the sole responsibility of the developer and visiting the vanity URL simply displays the application within an iframe on the Facebook site. What this means is that Facebook does not maintain any control over the application's functionality or developer's intent whatsoever, apart from what was agreed to in the developer's terms of service[4]. In fact, Facebook never has visibility into the application's code at all since it is hosted entirely outside their domain. As a result, a developer can create a page in the apps.facebook.com domain and host their application within minutes without any approval or review from Facebook required, at which point the possibilities for functionality are limited only by the language they have chosen to develop with.

Presumably, Facebook made a conscious effort to create such a development environment in order to provide a unified and seamless experience for the end-user. In fact, they have accomplished this since from the end-user's perspective there is almost no indication that the application has been developed or hosted by anyone other than Facebook while the browser continues to make calls to third party web servers that may not be trusted. However, what Facebook has also accomplished is to create an environment that is simply ideal for social engineering. While the application developed for this thesis was presented to the user as something akin to a "malware scanner" for their Facebook profile, it could have just as easily contained code to present a spoofed Facebook login form with a message indicating to the user that their "session" has expired and they need to re-enter their credentials. From there, the application can collect the user's input, store it to a database, and redirect the user back to their profile without any indication that a successful phishing attack has taken place. While this would certainly violate the developer's terms of use, there is nothing technically in place that can be used to prevent it from happening. Furthermore, the effectiveness of such an attack would likely be magnified by the fact that, from the user's perspective, their browser's address bar still points to a site in the facebook.com domain.

[4] http://developers.facebook.com/policy/

It is clear that such an environment, while certainly providing a seamless experience to the user, must be changed given its purely misleading functionality and ability for exploitation. This design may also hamper the efforts of security applications, including the one developed for this thesis, since the URL remains in what some may consider a trusted domain (facebook.com), even though the content is being served from a potentially malicious web server elsewhere. Thus, all URL-related tests become moot for Facebook applications unless further analysis is performed into the actual content the URL is pointing to.

In addition to these changes, however, Facebook must also consider changes to the freedom provided to third party developers since they are currently given free rein to serve any functionality they desire. Yet, it is possible that a solution may not yet exist for this problem, given that the industry has only recently been faced with it on a widespread scale. Only since the arrival of platforms like Apple's iPhone and Google's Android operating systems have we had to deal with the problem of software being developed by third parties but sold through avenues provided by the operating system companies, such as the iTunes App Store or the Android Marketplace. While the personal computer certainly allowed for this type of functionality, users were typically aware that they were purchasing software from a third party that was not endorsed by Microsoft or Apple since they could not purchase the product from them directly. As a result, sites like Facebook are now offering an app-based platform but have chosen to continue to leave the onus of awareness on the non-technical user for lack of a better solution. However, with the amount of personal information entrusted to Facebook and the clear opportunity for malice, this seems like an irresponsible decision.

Apple's model for dealing with this problem in their iTunes App Store, which requires that all applications sold there be reviewed and approved by the company before they are available, seems ineffective and inefficient. With the sheer amount of apps being submitted to Apple for review, which is now well over 100,000, it does not seem possible for the company to effectively review the functionality of each line of code in a timeframe that would be acceptable to developers hoping to quickly find a customer base [23]. This is doubly true for Facebook, which states there have been over 500,000 applications developed with its API [37]. In addition, exerting such tight control over the development process could potentially stifle innovation and allow reviewers to potentially exhibit favoritism as they approve apps and allow them to be published to the store.

On the other hand, Google's Android marketplace takes the completely opposite approach, and one that is similar to Facebook's methods, by allowing developers to immediately release their software, even on their own site, without any approval from Google whatsoever. While this is certainly a more cost effective and realistic approach, it does little to provide assurance to Android users that the application is safe. To offset this, the Android platform will require special permission be granted to applications which perform extended functionality that could be potentially dangerous, such as the ability to access the Internet or the user's contact list, similar to the methods used by Facebook applications. However, this still does not provide any indication of the actual functionality performed by the application since access to the Internet could, in the benign case, allow the application to serve dynamic advertisements or, in the malicious case, download a piece of malware to the device.

It may be possible that a middle ground approach could be the most effective for the Facebook development platform. For example, a voluntary approval process could be offered, whereby applications would undergo thorough screening by Facebook and considered verified only if they are found to be safe for users. This is similar to the methods used by Microsoft with their "Certified for Windows" program [22]. While this would certainly increase the length of the development cycle, users could seek out applications which are specifically vetted by Facebook and anxious developers could choose to release an unverified application and leave discretion in the user's hands. This, of course, also depends on how much a user trusts the Facebook verification process. In addition, Facebook could consider some type of certificate-like model. Similar to the way we use certificates signed by reliable authorities to judge the trust worthiness of unknown entities, Facebook could offer some type of developer certification program and issue certificates to those who are deemed trustworthy. Finally, certificates should also be used by developers to sign their applications, especially when they are being hosted on third party servers, so that a user can know the application code has not been tampered with. This is one example that is being actively used by the Android development guidelines. While these methods are certainly not fool-proof, they would provide a marked improvement over the current model.

6
Conclusions

It is undeniable, regardless of the application, that social networks contain an enormous amount of data that has largely been untapped. As social aspects to the Internet continue to expand in both quantity and scope, security of this data and the users of these sites will ultimately become an unavoidable concern.

While more research must be done to prove the usefulness of social networking data in the field of security and web content analysis, this thesis makes it clear that studying the concept is worthwhile. In addition, this thesis shows that it is in fact possible to detect malicious web content with the help of heuristics based on social networking data.

The success rates for the tests performed with this proof-of-concept application certainly do not rival the real-world success rates achieved by researchers such as Fette et al. [19] or Zhang et al. [44]. However, this thesis was able to exceed the 34% success rate for detecting legitimate content achieved by Boykin and Roychowdhury in their study of using social networks to predict the presence of spam in email by a factor of 35.5%, since it correctly identified 69.5% of safe URLs. In addition, the study's 56% success rate for detecting unwanted content is extremely close to this thesis' 55.5% for detecting malicious URLs [5]. This is an encouraging result as future studies will likely be able to improve this proof-of-concept application and bring its effectiveness to a point that is reliable enough for use in real-world environments.

7
Future Work

Although the application for this thesis was purposely developed as a proof-of-concept, the application is usable as-is by anyone with a Facebook account. Still, there are many ways in which this application can be improved upon that will benefit its functionality, performance, and effectiveness. The following sections describe some of the improvements that can be made by future work.

7.1 Bugs

While the code for this application did undergo a fairly extensive period of testing, several bugs that still exist became apparent while the results of the tests were being analyzed. First, occasionally a test might return a score of the $NO_SCORE variable, which indicates that an error occurred for that particular test. This is a difficult problem to test for, since it appears only a very select number of URLs will trigger the bug. Second, even with the use of the Long URL Please API, several shortened URLs were not correctly scored given their obfuscated nature. Although some of the URLs affected by this problem were created with a service not supported by the Long URL Please API, there were several results which might be considered invalid due to the use of popular shortening services like bit.ly. More testing with the Long URL Please API must be performed before it can be understood why these links were not reversed into their full form correctly. Finally, there is potentially a bug in the way that score reports are generated and stored on the server, since the testing period only yielded reports from eight individuals out of the 22 who approved the application for use on their Facebook profiles. While this may be a legitimate software flaw or simply a lack of good communication with the user about what is required, more verbose logging needs to be done to understand why score reports were not generated for certain trials.

7.2 Functionality

Perhaps the most promising of the potential improvements to this project is the ability to add more tests to the application's functionality. Currently, the application puts each URL through a series of 15 tests, seven of which are directly related to social network-based data. However, there certainly exists the potential for the creation of an even greater of number of tests, which would likely improve the effectiveness of the application as a whole. In addition, as the APIs for sites like Facebook and Twitter continue to evolve, this will certainly make more interesting and useful tests possible.

7.3 Interface Design

Because this application was designed to be a proof-of-concept, its development lacked the significant front-end design considerations that would be found in typical production software. Certainly, the application has plenty of room to improve in this area. Interface design changes could likely lead to a more user-friendly experience that allows users to grasp the concept and execution of the software more easily.

7.4 Platform Choice

Although the Facebook application platform was certainly ideal for proving the concept of this thesis while simultaneously having the best chance for in-depth access to significant amounts of social networking data, it may not be the best choice for an effective security tool used in real-world situations. Instead, future projects may consider porting the application to a platform with better insight into the user's Internet habits, such as a desktop application or a browser plugin. Making this change would allow the application to more effectively monitor web content from sources other than the Facebook News Feed, yet continue to use APIs to access the social networking data that is critical to the success of this concept.

7.5 Machine Learning

It is likely that the inclusion of a machine learning algorithm with this application would be useful in improving the scoring mechanism. Currently, an arbitrary scoring threshold must be chosen based on the median scores for data with a known safe or malicious rating. While this is suitable for a proof-of-concept application, a future production version could use machine learning to dynamically determine the scoring threshold to be used in making an assumption on whether a link is safe or malicious. This

decision would be based on a learning process that uses known data sets to develop an ideal scoring threshold for the environment in which the application is executed. It is likely that this process would significantly improve the scoring algorithm and produce more effective results, while allowing for customized scoring thresholds for different users.

A

Source Code – index.php

```
<?php
/*
* Author: Mike Robertson
* Application: Link Checker
* File: 'index.php'
*   The main application page. Upon loading, the script browses through the
*   last 100 posts in a user's stream and uses a regular expression to find
*   URLs that may have been posted there. While it does this search, it also
*   keeps track of the number of stream posts from each user's friend
*   (regardless of whether it contains a link or not). These totals
*   are added to a database (contacts_<user_id>) so that we can keep track of
*   the relative frequency that a user communicates with each of their
*   friends. This information is used as chatsOften function in
*   fb-checks.php, and is unfortunately not available at this time through
*   the Facebook API.
*
*     If the user has never used the application before (or has deleted their
*     cookie for the site), they are prompted to select one of their friend
*     lists as a group of "trusted" users. If a group is selected (the user
*     can choose a value of "none"), links sent by friends in this trusted
*     group are given a bonus point by the isInTrustedList function in fb-
*     checks.php. The idea in selecting this list is to place extra trust in
*     links sent by certain friends (i.e. family members or tech savvy people
*     who are unlikely to send you a malicious link).
*
*     Each link that is found in the stream (both the post's text and
*     attachments are checked) is added to an array which is sent, along with
*     some metadata, to the startScanner function in scanner.php. This
*     function will run the actual checks and create the report output. Once
*     the scan completes, this script will exit.
*
*     This file is part of Link Checker.
*
*     Link Checker is free software: you can redistribute it and/or modify
*     it under the terms of the GNU General Public License as published by
*     the Free Software Foundation, either version 3 of the License, or
*     (at your option) any later version.
*
*     Link Checker is distributed in the hope that it will be useful,
*     but WITHOUT ANY WARRANTY; without even the implied warranty of
*     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
*     GNU General Public License for more details.
*
*     You should have received a copy of the GNU General Public License
*     along with Link Checker.  If not, see <http://www.gnu.org/licenses/>.
*/
```

```
require_once '/data/www/api/facebook.php';      // Facebook API
include 'scanner.php';                    // Used to start the scanning
include '/data/www/linkchecker.conf';           // The global config file

// True if the "trusted_group" value is set in the HTTP POST data
// (i.e. the user has selected a trusted group and submitted the form)
if(isset($_POST["trusted_group"])) {
      // Allows IE to accept our 3rd party cookie by adding a
      // "compact privacy policy"
      // See [27] for more details
      header("P3P: CP=\"CAO PSA OUR\"");
      // Set a 30 day cookie to remember the user's trusted friend list
      // and reload the page to continue
      setcookie("trusted_group", $_POST["trusted_group"],
            time()+60*60*24*30);
      echo "<script>window.location.href=window.location.href;</script>";
}

// Initialize the Facebook object for API calls
$facebook = new Facebook($appapikey, $appsecret);
// An array of links found in the user's stream
$links = array();

// Expands shortened URLs using the Long URL Please API at [9]
// Returns: The expanded URL string if the URL was shortened
//                false if the URL was not shortened
function expandURL($url) {
      // Pattern adapted from Long URL Please Firefox extension [10]
      // We will only try to shorten the link if it matches the same pattern
      // used by the Long URL Please service (these are supported services)
      // This saves on the number of requests we need to make to their API
      $pattern =
            "/(http(s?):\/\/(307\.to|adjix\.com|b23\.ru|bacn\.me|bit\.ly|bloa
            t\.me|budurl\.com|cli\.gs|clipurl\.us|cort\.as|digg\.com|dwarfurl
            \.com|fb\.me|ff\.im|fff\.to|href\.in|idek\.net|is\.gd|j\.mp|kl\.a
            m|korta\.nu|lin\.cr|ln\s\.net|loopt\.us|lost\.in|memurl\.com|merk
            y\.de|migre\.me|moourl\.com|nanourl\.se|om\.ly|ow\.ly|peaurl\.com
            |ping\.fm|piurl\.com|plurl\.me|pnt\.me|poprl\.com|post\.ly|rde\.m
            e|reallytinyurl\.com|redir\.ec|retwt\.me|rubyurl\.com|short\.ie|s
            hort\.to|smallr\.com|sn\.im|sn\.vc|snipr\.com|snipurl\.com|snurl\
            .com|su\.pr|tiny\.cc|tinysong\.com|tinyurl\.com|togoto\.us|tr\.im
            |tra\.kz|trg\.li|twurl\.cc|twurl\.nl|u\.mavrev\.com|u\.nu|ur1\.ca
            |url\.az|url\.ie|urlx\.ie|w34\.us|xrl\.us|yep\.it|zi\.ma|zurl\.ws
            )\/[a-zA-Z0-9_-]+)|((http(s?):\/\/[a-zA-Z0-9_-]+\.notlong\.com)
            |(http(s?):\/\/[a-zA-Z0-
            9_]+\.qlnk\.net)|(http(s?):\/\/chilp\.it\/[?][a-zA-Z0-9_-]+)
            |(http(s?):\/\/goo\.gl\/fb\/[a-zA-Z0-9_-
            ]+)|(http(s?):\/\/trim\.li\/nk\/[a-zA-Z0-9_-]+)
            |(http(s?):\/\/url4\.eu\/[a-zA-Z0-9_-]+))[\/]?/";

      $num = preg_match_all($pattern, $url, $matches);
      // True if the URL matches a pattern supported by Long URL Please
      if($num > 0) {
            $matches = $matches[0];
            foreach($matches as $match) {
                  // Setup a cURL instance to shorten the URL using the Long
                  // URL Please API
```

```
                  $ch = curl_init("http://www.longurlplease.com/api/v1.1?q="
                        . $url);
                  // Return the results as an associative array,
                  // rather than printing them to stdout
                  curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);

                  // Execute the API call
                  $response = curl_exec($ch);

                  // Decode the JSON results into something we can
                  // parse with PHP
                  $results = json_decode($response);

                  // Return the full URL
                  return $results->$url;
            }
      }

      // Return false if the URL was not shortened
      return false;
}

// Checks links[] to see if a given link already exists in it
// to avoid scanning the same link twice.
// A link is considered a duplicate if its hash exists > 1 time
// in a single post.
// Returns: 1 if the link is a duplicate
//                0 if the link is not a duplicate
function checkForDupe($hash, $postID) {
      $GLOBALS["links"];

      foreach($GLOBALS["links"] as $link) {
            $hasHash = in_array($hash, $link);

            if($hasHash) {
                  $hasPost = in_array($postID, $link);
                  if($hasPost)
                        return 1;
                  else
                        continue;
            }
            else
                  continue;
      }
      return 0;
}


// Ask the user for permission to access their stream and store their user ID
$user_id = $facebook->require_login($required_permissions = 'read_stream');

// True if the user has not yet chosen a trusted friend list
if(!isset($_COOKIE["trusted_group"])) {
      echo "<span id=\"message\"><p>If you have a group of friends who you
            trust not to send you malicious links (i.e. links to viruses,
            spyware, fraud sites, etc.), please select it from the list
            below. Otherwise, select None.
```

```
            Once you have made your choice, please click the Save button to
            start the test.</p></span>";
      echo "<form action=\"index.php\" method=\"post\"><select
            name=\"trusted_group\"><option value=\"0\"
            selected>None</option>";

      // Returns a list of the user's friend lists, which is used to create
      // the drop-down menu
      $lists = $facebook->api_client->friends_getLists();
      if(is_array($lists)) {
            foreach($lists as $list)
                  echo "<option value=\"" . $list["flid"] . "\">" .
                        $list["name"] . "</option>";
      }
      else
            echo "<script>document.getElementById('message').innerHTML =
                  'There was an error getting your lists. Please select None
                  for your trusted list or contact the developer to report
                  this error.';</script>";
      echo "</select><br /><input type=\"submit\" value=\"Save\"
            name=\"save\" /></form>";
      exit(0);
}

// Otherwise, run the application
else {
      // Tell the user the test is complete (this will only display once the
      // scans have finished)
      echo "<b>The test is complete. See below for the scan report.</b><br
            /><br />";

      // Print some basic user information (user ID and number of friends)
      echo "User: " . $user_id . "<br />";
      $friends = $facebook->api_client->friends_get();
      $numFriends = count($friends);
      echo "Friends: " . $numFriends . "<br /><br />";

      // Get the current time, which is stored in a database to know when we
      // last scanned the stream
      $currentTime = time();

      // Connect to the database and select the linkchecker database.
      // Variables are accessed from the global config file
      $result = mysql_connect($server, $username, $password);
      if(!$result)
            die("Could not connect to database.");
      $result = mysql_select_db($contact_database);
      if(!$result)
            die("Could not open contact_database.");

      // Find the last time the user's stream was scanned
      $query = "SELECT updateTime FROM updateTime WHERE userID =
                  '" . $user_id . "'";
      $response = mysql_query($query);
      $result = mysql_fetch_assoc($response);

      // Set lastUpdate to be the time (in epoch) the stream was last
```

```
// scanned, or 0 if it's never been scanned
$lastUpdate = 0;
if(isset($result["updateTime"]))
      $lastUpdate = $result["updateTime"];

// Now that we have the latest update time, update the database with
// the current time since the stream is being scanned now
$query = "INSERT INTO updateTime (userID, updatetime) VALUES ('" .
            $user_id . "', '" . $currentTime . "') ON DUPLICATE KEY
            UPDATE updatetime='" . $currentTime . "'";
$result = mysql_query($query);

// Read the last 100 posts of the user's stream
$stream = $facebook->api_client->stream_get($user_id, '', '', '',
            '100', '', '');

// Loop through each post in the stream
foreach($stream["posts"] as $post) {

      if(isset($post["created_time"])) {
            // If we have not counted this post before, add +1 to the
            // user's contact frequency with this friend
            if($post["created_time"] > $lastUpdate) {
                  $query = "CREATE TABLE IF NOT EXISTS contacts_" .
                        $user_id . " (friendID varchar(32) not null
                        primary key, contactCount int default '0')";
                  $response = mysql_query($query);
                  if(!$response)
                        die("Could not create the database.");
                  $query = "INSERT INTO contacts_" . $user_id . "
                        (friendID, contactCount) VALUES ('" .
                        $post["source_id"] . "', '1') ON DUPLICATE KEY
                        UPDATE contactCount=contactCount+1";
                  $response = mysql_query($query);
                  if(!$response)
                        die("Could not update the database.");
            }
            // Skip this link if it has already been counted & scanned
            else
                  continue;
      }

      // Check the message text for links
      if(isset($post["message"])) {
            // Matches all of the URLs in a post
            $pattern = "/\b(http:\/\/|https:\/\/|www\.)(?:[a-zA-Z0-9\-
                  ])+(@|\.){1}[a-zA-Z0-9\-]+(?:\/?[a-zA-Z0-9\$\-
                  _\.\+!\*'\(\,):@=&\?])*\b/";
            $num = preg_match_all($pattern, $post["message"],
                  $matches);

            // True if any URLs were found
            if($num > 0) {
                  $matches = $matches[0];
                  foreach($matches as $match) {
                        if(isset($match[0])) {
                              // Enable for debugging only
```

```
                        //echo "Message: " . $match . "<br
                        //    />Posted by: " . $post["source_id"]
                        //    . "<br />Post #: " .
                        //    $post["post_id"]. "<br />";

                        // Expand the URL if it is shortened
                        $result = expandURL($match);
                        if($result != false)
                              $match = $result;

                        // Check to make sure the link is not a
                        // duplicate in this post
                        $isDupe = checkForDupe(md5($match),
                                    $post["post_id"]);

                        // Enable for debugging only
                        //echo "Is Duplicate: " . $isDupe . "<br
                        //          /><br />";

                        // True if the link is not a duplicate
                        if($isDupe == 0)
                              // Array format: [0] => URL hash,
                              [1] => URL, [2] => author's user
                              ID, [3] => post ID, [4] => score
                              $links[] = array(md5($match),
                                    $match, $post["source_id"],
                                    $post["post_id"], 0);
                  }
            }
      }
}

// Check the message attachment for links
if(isset($post["attachment"])) {
      $attachment = $post["attachment"];
      if(isset($attachment["media"]) &&
            is_array($attachment["media"])) {
            foreach($attachment["media"] as $data) {
                  if(isset($data["type"])) {
                        if($data["type"] == "link") {
                              $decoded_link = "";
                              // Links posted as attachments of
                              // applications other than Facebook
                              // will be raw URLs
                              if(strstr($data["href"],
                                    "http://www.facebook.com/l.ph
                                    p?") == false) {
                                    // Enable for debugging only
                                    //echo "Link: " .
                                    //    $data["href"] . "<br
                                    //    />";

                                    $url = $data["href"];
                              }
                              // Links posted as Facebook
                              // attachments will be encoded as a
                              // URL GET parameter
```

```
else {
      $query_data =
            parse_url($data["href"]
            , PHP_URL_QUERY);
      $url_param = explode("&",
            $query_data);
      $raw_link =
            substr($url_param[0],
            2);

      // Requires 2 calls to
      // urldecode (1st decodes the
      // "%", 2nd decodes the other
      // characters)
      $decoded_link =
            urldecode(urldecode($ra
            w_link));

      // Enable for debugging only
      //echo "Link: " .
      //    $decoded_link . "<br
      //    />";

      $url = $decoded_link;
}

// Enable for debugging only
//echo "Posted by: " .
//    $post["source_id"] . "<br
//    />";
//echo "Post #: " .
//    $post["post_id"] . "<br />";

// Expand the URL if it is
      // shortened
$result = expandURL($url);
if($result != false)
      $url = $result;

// Check to make sure the link is
// not a duplicate for this post
$isDupe = checkForDupe(md5($url),
      $post["post_id"]);

// Enable for debugging only
//echo "Is Duplicate: " . $isDupe .
//    "<br /><br />";

// True if the link is not a
// duplicate
if($isDupe == 0)
      // Array format: [0] => URL
      // hash, [1] => URL, [2] =>
      // author's user ID, [3] =>
      // post ID, [4] => score
      $links[] = array(md5($url),
```

```
                                                $url,
                                                $post["source_id"],
                                                $post["post_id"], 0);
                                    }
                                }
                            }
                        }
                }
        }

        // Get the total contact frequency we have seen
        $query = "SELECT SUM(contactCount) AS numContacts FROM contacts_" .
                    $user_id;
        $response = mysql_query($query);
        $result = mysql_fetch_assoc($response);
        $numContacts = -1;
        if(isset($result["numContacts"]))
                $numContacts = $result["numContacts"];

        // Start the scan if any links were found
        if(count($links) > 0)
                startScanner($user_id, $numFriends, $links, $facebook,
                        $numContacts);
}

mysql_close();

exit(0);

?>
```

B

Source Code – scanner.php

```
<?php
/*
* Author: Mike Robertson
* Application: Link Checker
* File: 'scanner.php'
*   Initiates all of the scanning functions and creates the scan report.
*     Adding a scan to this application only requires that the developer add
*     the following lines:
*
*           $result = <scan_function>;
*           echo "<scan_name>: " . $result . "<br />";
*           $output .= "^" . $result;
*           $score += $result;
*
*     Tests should return either GOOD_SCORE or BAD_SCORE, or a weighted
*     version of it. Tests can also return NO_SCORE on an error.
*
*     The scan report is both printed to the screen and stored as a delimited
*     text file on the server.
*     The text file has the following format for easy scripting/importing
*     (data is on a single line, but contains breaks here for formatting):
*
*     user_id^numFriends^hash^url^author_id^post_id^senderIsFriend^hasMutual
*           Friends^attendsEvents^photoIsTagged
*           ^chatsOften^isInTrustedList^serverLocation^domainAge^
*           suspiciousCharacters^nonStandardPort^manySubdomains^hasIPAddress
*           ^usesSSL^checkSafeBrowsing^twitterSearch^finalScore\r\n
*
*     This file is part of Link Checker.
*
*     Link Checker is free software: you can redistribute it and/or modify
*     it under the terms of the GNU General Public License as published by
*     the Free Software Foundation, either version 3 of the License, or
*     (at your option) any later version.
*
*     Link Checker is distributed in the hope that it will be useful,
*     but WITHOUT ANY WARRANTY; without even the implied warranty of
*     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
*     GNU General Public License for more details.
*
*     You should have received a copy of the GNU General Public License
*     along with Link Checker.  If not, see <http://www.gnu.org/licenses/>.
*/

include 'fb-checks.php';                  // Facebook-related scans
```

```
include 'url-checks.php';                  // URL-related scans
include '/data/www/linkchecker.conf';      // The global config file


// This is the main function that calls all of the checks and creates the
// report.
// It is called by both index.php and manual-scan.php
function startScanner($user_id, $numFriends, $links, $facebook, $numContacts)
{
      // Initialize the scan score to 0 and get the current time
      $score = 0;
      $startTime = time();

      // Loop through all of the links in links[]
      foreach($links as $link) {

            // Add some basic information to the report
            $output = $user_id . "^" . $numFriends;
            echo "Hash: " . $link[0] . "<br />";
            $output .= "^" . $link[0];
            echo "URL: " . $link[1] . "<br />";
            $output .= "^" . $link[1];
            echo "Author: " . $link[2] . "<br />";
            $output .= "^" . $link[2];
            echo "Post: " . $link[3] . "<br />";
            $output .= "^" . $link[3];

            // senderIsFriend (fb-checks.php): Checks to see if the author of
            // the post is a friend of the user
            // Returns: GOOD_SCORE if the author is a friend of the user,
            // BAD_SCORE if the author is not a friend of the user
            $result = senderIsFriend($user_id, $link[2], $facebook);
            echo "senderIsFriend: " . $result . "<br />";
            $output .= "^" . $result;
            $score += $result;

            // hasMutualFriends (fb-checks.php): Checks the number of mutual
            // friends the user has with the author.
            // This is based on research by [5]
            // Returns: GOOD_SCORE if the author has > NETWORK_CLUSTER_MAX
            // mutual friends with the author
            //          BAD_SCORE if the author has <
            //          NETWORK_CLUSTER_MIN mutual friends with the author
            //          GOOD_SCORE/2 if the author has >
            //          NETWORK_CLUSTER_MAX/2 mutual friends with the author
            //          BAD_SCORE/2 if the author has <
            //          NETWORK_CLUSTER_MAX/2 mutual friends with the author
            //          BAD_SCORE if the author has 0 mutual friends
            //          with the author
            $result = hasMutualFriends($user_id, $link[2], $facebook,
                        $numFriends);
            echo "hasMutualFriends: " . $result . "<br />";
            $output .= "^" . $result;
            $score += $result;

            // attendsEvents (fb-checks.php): Checks to see if the user is
            //    attending/has attended any events with the author.
```

```
//    The idea is that the user does not attend events with
//    malicious users.
// Returns: GOOD_SCORE if the author is attending/has attended at
// least 1 event with the user, otherwise NO_SCORE
$result = attendsEvents($user_id, $link[2], $facebook);
echo "attendsEvents: " . $result . "<br />";
$output .= "^" . $result;
$score += $result;

// photoIsTagged (fb-checks.php): Checks to see if the user is
//    tagged in any photos with the author. The idea is
//    that the user does not appear in pictures with malicious
//    users.
// Returns: GOOD_SCORE if the user is tagged in a photo with the
//    author, otherwise BAD_SCORE
// Notes: This test only works for photos owned by the user, due
//    to restrictions with the Facebook API
$result = photoIsTagged($user_id, $link[2], $facebook);
echo "photoIsTagged: " . $result . "<br />";
$output .= "^" . $result;
$score += $result;

// chatsOften (fb-checks.php): Checks to see if the user sees
//    posts by the author often. The idea is that
//    a link which comes from an author who does not post links
//    often may be malicious.
// Returns: GOOD_SCORE if the author posts with a frequency >=
//          total number of contacts*CONTACT_THRESHOLD.
//          BAD_SCORE if the author posts with a frequency <
//          total number of contacts*CONTACT_THRESHOLD.
//          BAD_SCORE*5 if the author has never posted.
$result = chatsOften($user_id, $link[2], $numContacts);
echo "chatsOften: " . $result . "<br />";
$output .= "^" . $result;
$score += $result;

// isInTrustedList (fb-checks.php): Checks to see if the author
//    is in the user's trusted friend list that they
//    selected when the application started.
// Returns: NO_SCORE if the user did not select a trusted list,
//    GOOD_SCORE if the author is in the trusted list,
//    BAD_SCORE if the author is not in the trusted list
$result = isInTrustedList($user_id, $link[2], $facebook);
echo "isInTrustedList: " . $result . "<br />";
$output .= "^" . $result;
$score += $result;

// serverLocation (url-checks.php): Checks the geographic
//    location of the web server.
// Returns: BAD_SCORE if the server is located in a country
//    listed in untrustedCountries[] (linkchecker.conf)
//          GOOD_SCORE if the server is not located in a country
//    listed in untrustedCountries[] (linkchecker.conf)
$result = serverLocation($link[1]);
echo "serverLocation: " . $result . "<br />";
$output .= "^" . $result;
$score += $result;
```

```
// domainAge (url-checks.php): Checks the age of the domain name.
// Returns: GOOD_SCORE*2 if the domain age is >= 3 years old
//          GOOD_SCORE if the domain age is between 1 year
//                (inclusive) and 3 years (exclusive) old
//          BAD_SCORE if the domain age is between 2 months
//                (inclusive) and 1 year (exclusive) old
//    Based on data [19]
//          BAD_SCORE*2 if the domain age is between 2 days
//                (inclusive) and 2 months (exclusive) old
//    Based on data from [2]
//          BAD_SCORE*10 if the domain age is < 2 days old
$result = domainAge($link[1]);
echo "domainAge: " . $result . "<br />";
$output .= "^" . $result;
$score += $result;

// suspiciousCharacters (url-checks.php): Checks the URL for the
//    presence of "@" or many (> 1) "-" characters.
// Returns: BAD_SCORE if the URL contains >1 "-" in the host
//    portion of the URL
//    BAD_SCORE if the URL contains "@"
//    GOOD_SCORE if the URL does not contain either of these
//    conditions
$result = suspiciousCharacters($link[1]);
echo "suspiciousCharacters: " . $result . "<br />";
$output .= "^" . $result;
$score += $result;

// nonStandardPort (url-checks.php): Checks to see if the website
//    is being hosted on a non-standard port. Non-standard ports
//    are considered to be anything other than TCP/80 or TCP/443.
// Returns: GOOD_SCORE if port 80 or 443 is specified in the URL,
//    or if no port is specified (browser will imply 80 or 443
//          based on the scheme [i.e. http or https])
//          BAD_SCORE if a port other than 80 or 443 is specified
//          in the URL
$result = nonStandardPort($link[1]);
echo "nonStandardPort: " . $result . "<br />";
$output .= "^" . $result;
$score += $result;

// manySubdomains (url-checks.php): Checks the URL for the
//    presence of many subdomains (> 3). Most legitimate sites
//    will keep the amount of subdomains to 3 or less for the
//    sake of ease for their visitors (i.e. mycourses.rit.edu,
//    mail.google.com, etc.). However, malicious sites may use
//    many subdomains to trick visitors by using subdomains
//    that are recognizable as other sites (i.e.
//    www.microsoft.com.test.net).
// Returns: BAD_SCORE if the URL contains > 3 subdomains
//          GOOD_SCORE if the URL cotains <= 3 subdomains
$result = manySubdomains($link[1]);
echo "manySubdomains: " . $result . "<br />";
$output .= "^" . $result;
$score += $result;
```

```
// hasIPAddress (url-checks.php): Checks the URL for the use of
//    IP address instead of domain name. Few legitimate sites
//    will neglect to register a domain name for their site since
//    it makes it easier to remember for users. Malicious
//    sites that will likely be short lived may not go to the
//    trouble of registering a domain name.
// Returns: BAD_SCORE if the URL contains an IP address
//          GOOD_SCORE if the URL does not contain an IP address
$result = hasIPAddress($link[1]);
echo "hasIPAddress: " . $result . "<br />";
$output .= "^" . $result;
$score += $result;

// usesSSL (url-checks.php): Checks to see if the URL is using
//    HTTP or HTTPS. Few malicious sites will take the time
//    to setup an SSL version of their site. In addition, this
//    test considers a site built with SSL as more secure.
// Returns: BAD_SCORE if the URL uses a scheme of HTTP
//          GOOD_SCORE if the URL uses a scheme of HTTPS
$result = usesSSL($link[1]);
echo "usesSSL: " . $result . "<br />";
$output .= "^" . $result;
$score += $result;

// checkSafeBrowsing (url-checks.php): Checks the URL hash
//    against the Google Safe Browsing databases (both blacklist
//    and malware) to see if the site is known to be malicious.
// Returns: BAD_SCORE*100 if the site's hash exists in either the
//                blacklist or malware Safe Browsing lists
//          GOOD_SCORE if the site's hash does not exist in
//                either the blacklist or malware Safe Browsing
//                lists
$result = checkSafeBrowsing($link[1]);
echo "checkSafeBrowsing: " . $result . "<br />";
$output .= "^" . $result;
$score += $result;

// twitterSearch (url-checks.php): Searches Twitter for the full
//    URL and, if it is not found, the domain used in the URL to
//    see if it is being shared there. The idea is that popular
//    links and well-known domains should appear in Twitter
//    search results. Malicious or obscure sites should not
//    appear in Twitter search results.
// Returns: GOOD_SCORE*2 if the full link appears on Twitter
//          GOOD_SCORE if the domain used in the link appears on
//                Twitter (i.e. rit.edu)
//          BAD_SCORE if neither the full link or domain appears
//                on Twitter
// Notes: This function relies on Twitter's malicious URL
//    filtering. The effectiveness of this function, however,
//    depends on the effectiveness of these filters since a
//    malicious link that is not filtered by
//    Twitter will receive a good score for this test.
$result = twitterSearch($link[1]);
echo "twitterSearch: " . $result . "<br />";
$output .= "^" . $result;
$score += $result;
```

```
            echo "Score: " . $score . "<br /><br />";
            $output .= "^" . $score;

            $output .= "\r\n";

            // Save the report in a file called
            //    /data/www/results/<user_id>_<start_time>.dat
            $f = fopen($GLOBALS["report_dir"] . $user_id . "_" . $startTime .
                  ".dat", "a");

            // output format:
            //    user_id^numFriends^hash^url^author_id^post_id^
            //    senderIsFriend^hasMutualFriends^attendsEvents^photoIsTagged
            //    ^chatsOften^isInTrustedList^serverLocation^domainAge
            //    ^suspiciousCharacters^nonStandardPort^manySubdomains
            //    ^hasIPAddress^usesSSL^checkSafeBrowsing^twitterSearch
            //    ^finalScore\r\n
            fwrite($f, $output);
            fclose($f);

            // Reset the score for the next link that will be scanned
            $score = 0;
      }
}

?>
```

C

Source Code – fb-checks.php

```
<?php
/*
* Author: Mike Robertson
* Application: Link Checker
* File: 'fb-checks.php'
*   Provides scanning functions for all Facebook-related checks. Each
*     function in this script can accept any parameters
*     needed to run the tests. At the conclusion of the test, the function
*     should return GOOD_SCORE, BAD_SCORE, or a weighted
*     version of these variables. NO_SCORE can also be returned on an error
*     or other special case where the results are inconclusive.
*     See the comments above each function for a description of its test(s).
*
*     This file is part of Link Checker.
*
*     Link Checker is free software: you can redistribute it and/or modify
*     it under the terms of the GNU General Public License as published by
*     the Free Software Foundation, either version 3 of the License, or
*     (at your option) any later version.
*
*     Link Checker is distributed in the hope that it will be useful,
*     but WITHOUT ANY WARRANTY; without even the implied warranty of
*     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
*     GNU General Public License for more details.
*
*     You should have received a copy of the GNU General Public License
*     along with Link Checker.  If not, see <http://www.gnu.org/licenses/>.
*/

// Checks to see if the author of the post is a friend of the user
// Returns: GOOD_SCORE if the author is a friend of the user, BAD_SCORE if
//    the author is not a friend of the user
function senderIsFriend($user_id, $senderID, $facebook) {
      // Use the Facebook API to see if user_id and senderID are friends
      $result = $facebook->api_client->friends_areFriends($user_id,
                  $senderID);

      if(is_array($result)) {
            $areFriends = $result[0];
            if(isset($areFriends["are_friends"])) {
                  // True if the users are friends
                  if($areFriends["are_friends"] == 1)
                        return $GLOBALS["GOOD_SCORE"];
                  // True if the users are not friends
                  elseif($areFriends["are_friends"] == 0)
```

```
                        return $GLOBALS["BAD_SCORE"];
                  // Inconclusive/error
                  else
                        return $GLOBALS["NO_SCORE"];
            }
            else
                  return $GLOBALS["NO_SCORE"];
      }
      else
            return $GLOBALS["NO_SCORE"];
}

// Checks the number of mutual friends the user has with the author.
//          This is based on research by [5]
// Returns: GOOD_SCORE if the author has > NETWORK_CLUSTER_MAX mutual friends
//                with the author
//          BAD_SCORE if the author has < NETWORK_CLUSTER_MIN mutual friends
//                with the author
//          GOOD_SCORE/2 if the author has > NETWORK_CLUSTER_MAX/2 mutual
//                friends with the author
//          BAD_SCORE/2 if the author has < NETWORK_CLUSTER_MAX/2 mutual
//                friends with the author
//          BAD_SCORE if the author has 0 mutual friends with the author
function hasMutualFriends($user_id, $senderID, $facebook, $numFriends) {

      // Recommended clustering coefficients by [5]
      $NETWORK_CLUSTER_MAX = $numFriends * 0.10;
      $NETWORK_CLUSTER_MIN = $numFriends * 0.01;

      // Use the API to get a list of the users' mutual friends
      $result = $facebook->api_client->friends_getMutualFriends($senderID,
                  $user_id);

      // Evaluates to true if users have >= 1 mutual friends
      if(is_array($result)) {
            // Count the number of mutual friends
            $numMutualFriends = count($result);

            // True if the number of mutual friends is > 10% of the user's
            //    total friends
            if($numMutualFriends > $NETWORK_CLUSTER_MAX)
                  return $GLOBALS["GOOD_SCORE"];
            // True if the number of mutual friends < 1% of the user's total
            //    friends
            elseif($numMutualFriends < $NETWORK_CLUSTER_MIN)
                  return $GLOBALS["BAD_SCORE"];
            // True if the number of mutual friends is between 5% and 10% of
            //    the user's total friends
            elseif($numMutualFriends > ($NETWORK_CLUSTER_MAX/2))
                  return ($GLOBALS["GOOD_SCORE"]/2);
            // True if the number of mutual friends is between 1% and 5% of
            //    the user's total friends
            elseif($numMutualFriends < ($NETWORK_CLUSTER_MAX/2))
                  return ($GLOBALS["BAD_SCORE"]/2);
            else
                  return $GLOBALS["NO_SCORE"];
      }
```

```
	// Evaluates to true if users have 0 mutual friends
	else
		return $GLOBALS["BAD_SCORE"];
}

// Checks to see if the user is attending/has attended any events with the
//    author. The idea is that the user does not attend events with malicious
//    users.
// Returns: GOOD_SCORE if the author is attending/has attended at least 1
//    event with the user, otherwise NO_SCORE
function attendsEvents($user_id, $senderID, $facebook) {
	// Use the API to get a list of events the user has attended w/ sender
	$query = "SELECT eid FROM event_member WHERE uid = \"" . $user_id . "\"
		AND rsvp_status = \"attending\" AND eid IN (SELECT eid FROM
		event_member WHERE uid = \"" . $senderID . "\" AND rsvp_status =
		\"attending\")";
	$result = $facebook->api_client->fql_query($query);

	if(is_array($result)) {
		// True if the user has attended at least 1 event with the sender
		if(count($result) > 0)
			return $GLOBALS["GOOD_SCORE"];
		else
			return $GLOBALS["NO_SCORE"];
	}
	// Return NO_SCORE if the user has not attended any events w/ sender
	// We cannot be sure that this means the sender is malicious (users
	//    could be geographically separated)
	else
		return $GLOBALS["NO_SCORE"];
}

// Checks to see if the user is tagged in any photos with the author. The
//    idea is that the user does not appear in pictures with malicious users.
// Returns: GOOD_SCORE if the user is tagged in a photo with the author,
//          otherwise BAD_SCORE
// Notes: This test only works for photos owned by the user, due to
//          restrictions with the Facebook API
function photoIsTagged($user_id, $senderID, $facebook) {
	// Get a list of the photos owned by the user that are tagged with both
	//    the user and the sender
	$query = "SELECT pid FROM photo_tag WHERE subject=\"" . $user_id . "\"
			AND pid IN (SELECT pid FROM photo_tag WHERE subject=\"" .
			$senderID . "\")";
	$result = $facebook->api_client->fql_query($query);

	// Return GOOD_SCORE if there is a picture tagged with both the user
	//    and sender
	// Return BAD_SCORE if there are no pictures that match this criteria
	if(is_array($result)) {
		if(count($result) > 0)
			return $GLOBALS["GOOD_SCORE"];
		else
			return $GLOBALS["BAD_SCORE"];
	}
	else
```

```
            return $GLOBALS["BAD_SCORE"];
}

// Checks to see if the user sees posts by the author often. The idea is that
//    a link which comes from an author who does not post links often may be
//    malicious.
// Returns: GOOD_SCORE if the author posts with a frequency >= total number
//               of contacts*CONTACT_THRESHOLD.
//          BAD_SCORE if the author posts with a frequency < total number of
//               contacts*CONTACT_THRESHOLD.
//          BAD_SCORE*5 if the author has never posted.
function chatsOften($user_id, $senderID, $numContacts) {
      // The sender must post >= to this percent of the total posts the user
      //    sees to be considered safe
      $CONTACT_THRESHOLD = 0.10;

      // Set the database to the contact database
      $result = mysql_select_db($GLOBALS["contact_database"]);
      if(!$result) {
            echo "result: " . $result . "<br />";
            return $GLOBALS["NO_SCORE"];
      }

      // Get the number of posts the user has seen from this sender
      $query = "SELECT contactCount FROM contacts_" . $user_id . " WHERE
                  friendID='" . $senderID . "'";
      $response = mysql_query($query);
      $result = mysql_fetch_assoc($response);

      // Return BAD_SCORE*5 if the sender has never posted
      if(!$result)
            return $GLOBALS["BAD_SCORE"]*5;

      if(isset($result["contactCount"])) {
            // True if this sender has posted >= CONTACT_THRESHOLD% of the
            //    posts seen by the user
            if($result["contactCount"] >= $numContacts*$CONTACT_THRESHOLD)
                  return $GLOBALS["GOOD_SCORE"];
            // Return BAD_SCORE*5 if the sender has never posted
            elseif($result["contactCount"] <= 1)
                  return $GLOBALS["BAD_SCORE"]*5;
            // Return BAD_SCORE if the sender has posted more than once but <
            //    CONTACT_THRESHOLD% of the posts seen by the user
            else
                  return $GLOBALS["BAD_SCORE"];
      }
      else {
            echo "arr " . $result . "<br />";
            return $GLOBALS["NO_SCORE"];
      }
}

// Checks to see if the author is in the user's trusted friend list that they
//    selected when the application started.
// Returns: NO_SCORE if the user did not select a trusted list, GOOD_SCORE if
//               the author is in the trusted list,
//          BAD_SCORE if the author is not in the trusted list
```

```
function isInTrustedList($user_id, $senderID, $facebook) {
      // True if the user did not select a trusted group. Test cannot run in
      //    this case.
      if($_COOKIE["trusted_group"] == 0)
            return $GLOBALS["NO_SCORE"];

      // Use the API to get a list of members in the trusted list
      $query = "SELECT uid FROM friendlist_member WHERE flid='" .
                  $_COOKIE["trusted_group"] . "'";
      $result = $facebook->api_client->fql_query($query);


      if(is_array($result)) {
            // Loop through the list's members
            foreach($result as $member) {
                  if(isset($member["uid"])) {
                        // True if the sender is in the trusted group
                        if($member["uid"] == $senderID)
                              return $GLOBALS["GOOD_SCORE"];
                  }
            }
      }

      // Return if the sender was not found in the trusted group
      return $GLOBALS["BAD_SCORE"];
}
?>
```

D

Source Code – url-checks.php

```
<?php
/*
* Author: Mike Robertson
* Application: Link Checker
* File: 'url-checks.php'
*   Provides scanning functions for all URL-related checks. Each function in
*     this script can accept any parameters needed to run the tests. At the
*     conclusion of the test, the function should return GOOD_SCORE,
*     BAD_SCORE, or a weighted version of these variables. NO_SCORE can also
*     be returned on an error or other special case where the results are
*     inconclusive. See the comments above each function for a description of
*     its test(s).
*
*     This file is part of Link Checker.
*
*     Link Checker is free software: you can redistribute it and/or modify
*     it under the terms of the GNU General Public License as published by
*     the Free Software Foundation, either version 3 of the License, or
*     (at your option) any later version.
*
*     Link Checker is distributed in the hope that it will be useful,
*     but WITHOUT ANY WARRANTY; without even the implied warranty of
*     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
*     GNU General Public License for more details.
*
*     You should have received a copy of the GNU General Public License
*     along with Link Checker.  If not, see <http://www.gnu.org/licenses/>.
*/

// Use the MaxMind GeoIP API [26]
require_once "Net/GeoIP.php";

// Checks the geographic location of the web server.
// Based on research by [43] and [4]
// Returns: BAD_SCORE if the server is located in a country listed in
//               untrustedCountries[] (linkchecker.conf)
//          GOOD_SCORE if the server is not located in a country listed in
//               untrustedCountries[] (linkchecker.conf)
function serverLocation($link) {
      // Initialize the GeoLite database downloaded from [25]
      $geoip = Net_GeoIP::getInstance("/data/www/GeoIP.dat");

      try {
            // Get the domain from the URL and convert it to an IP address
            $hostname = parse_url($link);
```

```
            // Necessary for URLs without a scheme (i.e. http://), parse_url
            //    does not work correctly and stores host in ["path"]
            if(isset($hostname["host"]))
                  $host = $hostname["host"];
            else {
                  $link = "http://" . $link;
                  $hostname = parse_url($link);
                  $host = $hostname["host"];
            }

            // Use the GeoIP API to determine what country it is located in
            $country = $geoip->lookupCountryCode(gethostbyname($host));

            // Loop through the list of untrusted countries
            foreach($GLOBALS["untrustedCountries"] as $code) {
                  // True if the web server is hosted in an untrusted country
                  if($country == $code)
                        return $GLOBALS["BAD_SCORE"];
            }

            // If no match was found, the web server is hosted in a trusted
            //    country
            return $GLOBALS["GOOD_SCORE"];
      }
      catch (Exception $e) {
            return $GLOBALS["NO_SCORE"];
      }
}

// Checks the age of the domain name.
// Based on research by [44] and [6]
// Note: This test may not give expected results for domains in the co.uk
//    namespace. Some whois records do not reveal created dates.
// Returns: GOOD_SCORE*2 if the domain age is >= 3 years old
//          GOOD_SCORE if the domain age is between 1 year (inclusive) and 3
//                years (exclusive) old
//          BAD_SCORE if the domain age is between 2 months (inclusive) and 1
//                year (exclusive) old
//                Based on data from [19]
//          BAD_SCORE*2 if the domain age is between 2 days (inclusive) and 2
//                months (exclusive) old
//                Based on data from [2]
//          BAD_SCORE*10 if the domain age is < 2 days old
function domainAge($link) {
      $YEAR_IN_SECONDS = 31556926;        // 1 year in seconds
      $DAY_IN_SECONDS = 86400;            // 1 day in seconds

      // An array of months used to determine the month's number (i.e.
      //    January = 1)
      $months = array(1 => "jan", "feb", "mar", "apr", "may", "jun", "jul",
                  "aug", "sep", "nov", "dec");

      $hostname = parse_url($link);

      // Necessary for URLs without a scheme (i.e. http://), parse_url does
      //    not work correctly and stores host in ["path"]
      if(isset($hostname["host"]))
```

```
      $hostnameExp = explode(".", $hostname["host"]);
else {
      $link = "http://" . $link;
      $hostname = parse_url($link);
      $hostnameExp = explode(".", $hostname["host"]);
}

// Get the main domain and TLD of the URL (i.e. rit.edu)
$hostSize = count($hostnameExp);
$hostname = $hostnameExp[$hostSize-2] . "." .
      $hostnameExp[$hostSize-1];

// Check for *.co.uk domains, since this would cause an error without
//    the * subdomain
if($hostname == "co.uk")
      $hostname = $hostnameExp[$hostSize-3] . "." .
                  $hostnameExp[$hostSize-2] . "." .
                  $hostnameExp[$hostSize-1];

// Execute a whois lookup on the domain and extract the dates returned
exec("jwhois " . escapeshellarg($hostname) . " | egrep -o [0-9]{2}-[a-
      zA-Z]{3}-[0-9]{4}", $dates, $result);

if($result == 0) {
      // Loop through all of the dates from the whois results and
      //    convert them to epoch time
      // Determine which is the oldest date (i.e. smallest epoch
      //    number) and consider that the domain's creation date
      $smallest = "9999999999";     // Works until November 2286
      if(is_array($dates)) {
            foreach($dates as $date) {
                  $datePieces = explode("-", $date);
                  $whoisTime = mktime(0, 0, 0,
                        array_search(strtolower($datePieces[1]),
                        $months), $datePieces[0], $datePieces[2]);
                  if($smallest  > $whoisTime)
                        $smallest = $whoisTime;
            }
      }
      else
            return $GLOBALS["NO_SCORE"];

      // Get the current time in epoch
      $currentDate = time();

      // True if the domain is >= 3 years old
      if($smallest <= ($currentDate-($YEAR_IN_SECONDS*3)))
            return $GLOBALS["GOOD_SCORE"]*2;
      // True if the domain is between 1 year (inclusive) and 3 years
      //    (exclusive) old
      if($smallest <= ($currentDate-($YEAR_IN_SECONDS)))
            return $GLOBALS["GOOD_SCORE"];
      // True if the domain is between 2 months (inclusive) and 1 year
      //    (exclusive) old
      // Based on data from [19]
      if($smallest <= ($currentDate-($DAY_IN_SECONDS*60)))
            return $GLOBALS["BAD_SCORE"];
```

```
            // True if the domain is between 2 days (inclusive) and 2 months
            //    (exclusive) old
            // Based on data from [2]
            if($smallest <= ($currentDate-($DAY_IN_SECONDS*2)))
                  return $GLOBALS["BAD_SCORE"]*2;
            // True if the domain is < 2 days old
            else
                  return $GLOBALS["BAD_SCORE"]*10;
      }
      else
            return $GLOBALS["NO_SCORE"];
}

// Checks the URL for the presence of "@" or many (> 1) "-" characters.
// Based on research by [44] and [6]
// Returns: BAD_SCORE if the URL contains >1 "-" in the host portion of URL
//          BAD_SCORE if the URL contains "@"
//          GOOD_SCORE if the URL does not contain either of these conditions
function suspiciousCharacters($link) {
      // Only consider the host portion of the URL (i.e. www.rit.edu)
      $hostname = parse_url($link);
      // Necessary for URLs without a scheme (i.e. http://), parse_url does
      //    not work correctly and stores host in ["path"]
      if(isset($hostname["host"]))
            $host = $hostname["host"];
      else {
            $link = "http://" . $link;
            $hostname = parse_url($link);
            $host = $hostname["host"];
      }

      // Count the number of "-" characters
      preg_match_all("/[-]+/", $host, $hyphens);
      $numHyphens = 0;
      if(isset($hyphens[0]))
            $numHyphens = count($hyphens[0]);

      // True if the number of "-" is > 1
      if($numHyphens > 1)
            return $GLOBALS["BAD_SCORE"];

      // Check for "@"
      $hasAt = strpos($link, "@");

      // True if the URL contains "@"
      if($hasAt)
            return $GLOBALS["BAD_SCORE"];

      // Return a good score if the URL contained no suspicious characters
      return $GLOBALS["GOOD_SCORE"];
}

// Checks to see if the website is being hosted on a non-standard port. Non-
//    standard ports are considered to be anything other than TCP/80 or
//    TCP/443.
// Based on research by [43]
// Returns: GOOD_SCORE if port 80 or 443 is specified in the URL, or if no
```

```
//                port is specified (browser will imply 80 or 443 based
//                on the scheme [i.e. http or https])
//          BAD_SCORE if a port other than 80 or 443 is specified in the URL
function nonStandardPort($link) {
      // Check the URL for a port number (0 - 99,999 are matched, but
      //    anything over 65,535 is not a valid port)
      $result = preg_match("/:[0-9]{1,5}/", $link, $matches);
      if($result) {
            // True if the URL contains a port number
            if(isset($matches[0])) {
                  $port = explode(":", $matches[0]);
                  // True if the port is anything other than 80 or 443
                  if($port[0] != 80 && $port[0] != 443)
                        return $GLOBALS["BAD_SCORE"];
                  else
                        return $GLOBALS["GOOD_SCORE"];
            }
            else
                  return $GLOBALS["NO_SCORE"];
      }
      else
            return $GLOBALS["GOOD_SCORE"];
}

// Checks the URL for the presence of many subdomains (> 3). Most legitimate
//    sites will keep the amount of subdomains to 3 or less for the sake of
//    ease for their visitors (i.e. mycourses.rit.edu, mail.google.com,
//    etc.). However, malicious sites may use many subdomains to trick
//    visitors by using subdomains that are recognizable as other
//    sites (i.e. www.microsoft.com.test.net).
//    Based on research by [44]
// Returns: BAD_SCORE if the URL contains > 3 subdomains
//          GOOD_SCORE if the URL cotains <= 3 subdomains
//          NO_SCORE if the URL contains an IP address
function manySubdomains($link) {
      // Search for an IP address in the URL
      // Regex adapted from [20]
      $result = preg_match("/\b(?:\d{1,3}\.){3}\d{1,3}\b/", $link, $matches);
      if($result) {
            // True if the URL contains an IP address--we should not score
            //    this since it would be a false alarm
            if(isset($matches[0]))
                  return $GLOBALS["NO_SCORE"];
      }

      $hostname = parse_url($link);
      // Necessary for URLs without a scheme (i.e. http://), parse_url does
      //    not work correctly and stores host in ["path"]
      if(isset($hostname["host"]))
            $host = $hostname["host"];
      else {
            $link = "http://" . $link;
            $hostname = parse_url($link);
            $host = $hostname["host"];
      }

      // Count the number of subdomains
```

```
	// Return BAD_SCORE if the count is > 3
	// Otherwise return GOOD_SCORE
	$subdomains = explode(".", $host);
	if(count($subdomains) > 3)
		return $GLOBALS["BAD_SCORE"];
	else
		return $GLOBALS["GOOD_SCORE"];
}

// Checks the URL for the use of IP address instead of domain name. Few
//    legitimate sites will neglect to register a
//    domain name for their site since it makes it easier to remember for
//    users. Malicious sites that will likely be
//    short lived may not go to the trouble of registering a domain name.
// Based on research by [44]
// Returns: BAD_SCORE if the URL contains an IP address
//          GOOD_SCORE if the URL does not contain an IP address
function hasIPAddress($link) {
	// Search for an IP address in the URL
	// Regex adapted from [20]
	$result = preg_match("/\b(?:\d{1,3}\.){3}\d{1,3}\b/", $link, $matches);

	if($result) {
		// True if the URL contains an IP address
		if(isset($matches[0]))
			return $GLOBALS["BAD_SCORE"];
		else
			return $GLOBALS["NO_SCORE"];
	}
	else
		return $GLOBALS["GOOD_SCORE"];
}

// Checks to see if the URL is using HTTP or HTTPS. Few malicious sites will
//    take the time to setup an SSL version of their site. In addition, this
//    test considers a site built with SSL as more secure.
// Based on research by [6]
// Returns: BAD_SCORE if the URL uses a scheme of HTTP
//          GOOD_SCORE if the URL uses a scheme of HTTPS
function usesSSL($link) {
	// Get the URL scheme (i.e. http or https)
	$scheme = parse_url($link, PHP_URL_SCHEME);

	// True if the URL uses HTTP
	if($scheme == "http")
		return $GLOBALS["BAD_SCORE"];
	// True if the URL uses HTTPS
	elseif($scheme == "https")
		return $GLOBALS["GOOD_SCORE"];
	// True if the scheme is not specified
	else
		return $GLOBALS["NO_SCORE"];
}

function isInSafeBrowsing($hash) {
	// Check to see if the URL's hash exists in the blacklist table
	$query = "SELECT hash FROM blacklist_hash WHERE hash='" . $hash . "'";
```

```
	$response = mysql_query($query);
	if(!$response)
		return false;
	$result = mysql_fetch_assoc($response);
	if(isset($result["hash"]))
		return true;

	// Check to see if the URL's hash exists in the malware table
	$query = "SELECT hash FROM malware_hash WHERE hash='" . $hash . "'";
	$response = mysql_query($query);
	if(!$response)
		return false;
	$result = mysql_fetch_assoc($response);
	if(isset($result["hash"]))
		return true;

	// If we get this far, the hash was not found in either list
	return false;
}

// Checks the URL hash against the Google Safe Browsing databases (both
//    blacklist and malware) to see if the site is known to be malicious.
// Returns: BAD_SCORE*100 if the site's hash exists in either the blacklist
//               or malware Safe Browsing lists
//          GOOD_SCORE if the site's hash does not exist in either the
//               blacklist or malware Safe Browsing lists
function checkSafeBrowsing($url) {
	// Set the database to the Safe Browsing database
	$result = mysql_select_db($GLOBALS["sb_database"]);
	if(!$result)
		return $GLOBALS["NO_SCORE"];

	$parsed = parse_url($url);
	if(is_array($parsed)) {
		// Necessary for URLs without a scheme (i.e. http://), parse_url
		//    does not work correctly and stores host in ["path"]
		if(isset($parsed["host"]))
			$host = $parsed["host"];
		else {
			$url = "http://" . $url;
			$parsed = parse_url($url);
			$host = $parsed["host"];
		}

		// Hash and chech for various formats of the URL
		// See [12] for details on formats we should check
		if(isset($parsed["path"])) {
			if(isInSafeBrowsing(md5($host . $parsed["path"])))
				return $GLOBALS["BAD_SCORE"]*100;
			if(isset($parsed["query"])) {
				if(isInSafeBrowsing(md5($host . $parsed["path"] . "?"
				. $parsed["query"])))
					return $GLOBALS["BAD_SCORE"]*100;
			}
		}
		if(isInSafeBrowsing(md5($host . "/")))
			return $GLOBALS["BAD_SCORE"]*100;
```

```
            $domains = explode(".", $host);
            $numDomains = count($domains);
            $offset = -1;
            if($numDomains > 5) {
                  $numDomains = 5;
                  $offset = 1;
            }

            for($x=0; $x < $numDomains-1; $x++) {
                  $link = "";
                  for($y=$numDomains; $y > $x; $y--) {
                        if($y == $numDomains)
                              $link = $domains[$y+$offset];
                        else
                              $link = $domains[$y+$offset] . "." . $link;
                  }

                  if(isset($parsed["path"])) {
                        if(isInSafeBrowsing(md5($link . $parsed["path"])))
                              return $GLOBALS["BAD_SCORE"]*100;
                        if(isset($parsed["query"])) {
                              if(isInSafeBrowsing(md5($link . $parsed["path"]
                              . "?" . $parsed["query"])))
                                    return $GLOBALS["BAD_SCORE"]*100;
                        }
                  }
                  if(isInSafeBrowsing(md5($link . "/")))
                        return $GLOBALS["BAD_SCORE"]*100;
            }

            return $GLOBALS["GOOD_SCORE"];
      }
      else
            return $GLOBALS["NO_SCORE"];
}

// Searches Twitter for the full URL and, if it is not found, the domain used
//    in the URL to see if it is being shared there.
// The idea is that popular links and well-known domains should appear in
//    Twitter search results. Malicious or obscure sites should not appear in
//    Twitter search results.
// Returns: GOOD_SCORE*2 if the full link appears on Twitter
//          GOOD_SCORE if the domain used in the link appears on Twitter
//                (i.e. rit.edu)
//          BAD_SCORE if neither the full link or domain appears on Twitter
// Notes: This function relies on Twitter's malicious URL filtering. The
//          effectiveness of this function, however, depends on the
//          effectiveness of these filters since a malicious link that is not
//          filtered by Twitter will receive a good score for this test.
function twitterSearch($link) {
      // Setup a cURL instance to search, using the Twitter API, for the
      //    entire URL
      $ch = curl_init("http://search.twitter.com/search.json?q=" . $link);
      // Including a unique User Agent allows us to send more queries per
      //    hour per [33]
```

```
curl_setopt($ch, CURLOPT_USERAGENT, "Linkchecker/1.0
      (http://apps.facebook.com/linkchecker)");
// Return the results as an associative array, rather than printing
//    them to stdout
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);

// Execute the search
$response = curl_exec($ch);

// True if the search failed
if(!$response) {
      curl_close($ch);
      return $GLOBALS["NO_SCORE"];
}

// Decode the JSON results into something we can parse with PHP
$results = json_decode($response);

// Loop through the search results
foreach($results->results as $result) {
      // Return GOOD_SCORE*2 if the entire URL was found on Twitter
      if($result->text != "") {
            curl_close($ch);
            return $GLOBALS["GOOD_SCORE"]*2;
      }
}

// Search for an IP address in the URL
// Regex adapted from [20]
$result = preg_match("/\b(?:\d{1,3}\.){3}\d{1,3}\b/", $link, $matches);
if($result) {
      // True if the URL contains an IP address--we should not score
      //    this since it would be a false alarm
      if(isset($matches[0]))
            return $GLOBALS["NO_SCORE"];
}


// If we get here, the entire URL was not found on Twitter so we search
//    for the domain instead
$hostname = parse_url($link);

// Necessary for URLs without a scheme (i.e. http://), parse_url does
//    not work correctly and stores host in ["path"]
if(isset($hostname["host"]))
      $hostnameExp = explode(".", $hostname["host"]);
else {
      $link = "http://" . $link;
      $hostname = parse_url($link);
      $hostnameExp = explode(".", $hostname["host"]);
}

// Get the main domain and TLD of the link (i.e. rit.edu)
$hostSize = count($hostnameExp);
$hostname = $hostnameExp[$hostSize-2] . "." .
      $hostnameExp[$hostSize-1];
```

```
	// Search Twitter for the domain this time
	curl_setopt($ch, CURLOPT_URL,
		"http://search.twitter.com/search.json?q=" . $hostname);
	$response = curl_exec($ch);

	// Decode the JSON results into something we can parse with PHP
	$results = json_decode($response);

	// Loop through the search results
	foreach($results->results as $result) {
		// Return GOOD_SCORE if the domain was found on Twitter
		if($result->text != "") {
			curl_close($ch);
			return $GLOBALS["GOOD_SCORE"];
		}
	}

	// Return BAD_SCORE if neither the URL or domain were found
	curl_close($ch);
	return $GLOBALS["BAD_SCORE"];
}

?>
```

E

Source Code – manual-scan.php

```
<?php
/* Author: Mike Robertson
* Application: Link Checker
* File: 'manual-scan.php'
*   A manual interface for the URL scanner. This page allows the user to pass
*     URLs to the scanner without requiring that they be posted to the user's
*     stream. This makes it easier to scan a large list of links or links
*     that otherwise be filtered out of the user's stream.
*
*     The user is asked to provide a list of links (1 per line) and to
*     specify the sender (either by selecting from a list of their
*     friends or entering the sender's Facebook user ID.
*
*     Note: The user must have run at least 1 automatic scan to develop some
*           of the heuristics, such as contact frequency and trusted
*           lists, which are required for the scans.
*
*     This file is part of Link Checker.
*
*     Link Checker is free software: you can redistribute it and/or modify
*     it under the terms of the GNU General Public License as published by
*     the Free Software Foundation, either version 3 of the License, or
*     (at your option) any later version.
*
*     Link Checker is distributed in the hope that it will be useful,
*     but WITHOUT ANY WARRANTY; without even the implied warranty of
*     MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
*     GNU General Public License for more details.
*
*     You should have received a copy of the GNU General Public License
*     along with Link Checker.  If not, see <http://www.gnu.org/licenses/>.
*/

$api = $app_dir . 'api/facebook.php';
require_once $api;                          // Facebook API
include 'scanner.php';                      // Used to start the scans
$conf = $app_dir . 'linkchecker.conf';
include '/data/www/linkchecker.conf';       // The global configuration file

// Initalize an array of links to scan
$links = array();

// Expands shortened URLs using the Long URL Please API [9]
// Returns: The expanded URL string if the URL was shortened
//          false if the URL was not shortened
```

```
function expandURL($url) {
      // Pattern adapted from Long URL Please Firefox extension [10]
      // We will only try to shorten the link if it matches the same pattern
      // used by the Long URL Please service (these are supported services)
      // This saves on the number of requests we need to make to their API
      $pattern =
            "/(http(s?):\/\/(307\.to|adjix\.com|b23\.ru|bacn\.me|bit\.ly|bloa
            t\.me|budurl\.com|cli\.gs|clipurl\.us|cort\.as|digg\.com|dwarfurl
            \.com|fb\.me|ff\.im|fff\.to|href\.in|idek\.net|is\.gd|j\.mp|kl\.a
            m|korta\.nu|lin\.cr|ln\-
            s\.net|loopt\.us|lost\.in|memurl\.com|merky\.de|migre\.me|moourl\
            .com|nanourl\.se|om\.ly|ow\.ly|peaurl\.com|ping\.fm|piurl\.com|pl
            url\.me|pnt\.me|poprl\.com|post\.ly|rde\.me|reallytinyurl\.com|re
            dir\.ec|retwt\.me|rubyurl\.com|short\.ie|short\.to|smallr\.com|sn
            \.im|sn\.vc|snipr\.com|snipurl\.com|snurl\.com|su\.pr|tiny\.cc|ti
            nysong\.com|tinyurl\.com|togoto\.us|tr\.im|tra\.kz|trg\.li|twurl\
            .cc|twurl\.nl|u\.mavrev\.com|u\.nu|ur1\.ca|url\.az|url\.ie|urlx\.
            ie|w34\.us|xrl\.us|yep\.it|zi\.ma|zurl\.ws)\/[a-zA-Z0-9_-
            ]+)|((http(s?):\/\/[a-zA-Z0-9_-
            ]+\.notlong\.com)|(http(s?):\/\/[a-zA-Z0-9_-
            ]+\.qlnk\.net)|(http(s?):\/\/chilp\.it\/[?][a-zA-Z0-9_-]+)
            |(http(s?):\/\/goo\.gl\/fb\/[a-zA-Z0-9_-
            ]+)|(http(s?):\/\/trim\.li\/nk\/[a-zA-Z0-9_-]+)
            |(http(s?):\/\/url4\.eu\/[a-zA-Z0-9_-]+))[\/]?/";

      $num = preg_match_all($pattern, $url, $matches);
      // True if the URL matches a pattern supported by Long URL Please
      if($num > 0) {
            $matches = $matches[0];
            foreach($matches as $match) {
                  // Setup a cURL instance to shorten the URL using the Long
                  //    URL Please API
                  $ch = curl_init("http://www.longurlplease.com/api/v1.1?q="
                              . $url);
                  // Return the results as an associative array, rather than
                  //    printing them to stdout
                  curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);

                  // Execute the API call
                  $response = curl_exec($ch);

                  // Decode the JSON results into something to parse w/ PHP
                  $results = json_decode($response);

                  // Return the full URL
                  return $results->$url;
            }
      }

      // Return false if the URL was not shortened
      return false;
}

// Checks links[] to see if a given link already exists in it
// to avoid scanning the same link twice.
// A link is considered a duplicate if its hash exists > 1 time
// in a single post.
```

```
// Returns: 1 if the link is a duplicate
//                0 if the link is not a duplicate
function checkForDupe($hash, $postID) {
      $GLOBALS["links"];

      foreach($GLOBALS["links"] as $link) {
            // Use 'true' to enable strict checking mode
            // This is needed when the array can contain a 0--this might
            //    cause in_array to return true
            $hasHash = in_array($hash, $link, true);

            if($hasHash) {
                  // Use 'true' to enable strict checking mode
                  // This is needed when the array can contain a 0--this
                  //    might cause in_array to return true
                  $hasPost = in_array($postID, $link, true);
                  if($hasPost)
                        return 1;
                  else
                        continue;
            }
            else
                  continue;
      }
      return 0;
}


// True if the user has filled out and submitted the scan form, so we can
begin the scan now
if(isset($_POST["start_scan"]) && $_POST["start_scan"] == 1) {
      // Set the variables that will be passed to the scanner
      $user_id = $_POST["user"];
      $numFriends = $_POST["numFriends"];
      $author = $_POST["friends"];

      if(isset($_POST["senderID"])) {
            if($_POST["senderID"] != "") {
                  if (is_numeric($_POST["senderID"]) &&
                  strlen($_POST["senderID"]) <= 32)
                        $author = $_POST["senderID"];
                  else
                        echo "<script>alert('Invalid Sender ID specified.
                              Starting over...');
                              window.location.href=window.location.href;</scr
                              ipt>";
            }
      }

      // Build the links array to be scanned by regex searching the list
      //    provided by the user
      $pattern = "/\b(http:\/\/|https:\/\/|www\.)(?:[a-zA-Z0-9\-
                  ])+(@|\.){1}[a-zA-Z0-9\-]+(?:\/?[a-zA-Z0-9\$\-
                  _\.\+!\*'\(\,):@=&\?])*\b/";
      $num = preg_match_all($pattern, $_POST["urllist"], $matches);
      if($num > 0) {
            $matches = $matches[0];
```

```
            // Loop through the URLs found by the regex search
            foreach($matches as $match) {
                  // Enable for debugging only
                  //echo "Message: " . $match . "<br />";

                  // Expand the URL if it is shortened
                  $result = expandURL($match);
                  if($result != false)
                        $match = $result;

                  // Check to see if the link is a duplicate
                  $isDupe = checkForDupe(md5($match), 0);

                  // Enable for debugging only
                  //echo "Is Duplicate: " . $isDupe . "<br /><br />";

                  // If the link is not a duplicate, add it to the links
                  //    array to be scanned
                  if($isDupe == 0)
                        // Array: [0] => URL hash, [1] => URL, [2] =>
                        //    author's user ID, [3] => post ID, [4] => score
                        $links[] = array(md5($match), $match, $author, 0, 0);
            }
      }

      // Carry over the user's session for API calls
      $facebook = new Facebook($appapikey, $appsecret);
      $facebook->set_user($user_id, $_POST["sKey"]);

      // Connect to the database and select the linkchecker database.
      // Variables are accessed from the global config file
      $result = mysql_connect($server, $username, $password);
      if(!$result)
            die("Could not connect to database.");
      $result = mysql_select_db($contact_database);
      if(!$result)
            die("Could not open contact_database.");

      // Get the total contact frequency we have seen
      $query = "SELECT SUM(contactCount) AS numContacts FROM contacts_" .
            $user_id;
      $response = mysql_query($query);
      $result = mysql_fetch_assoc($response);
      $numContacts = -1;
      if(isset($result["numContacts"]))
            $numContacts = $result["numContacts"];

      // If there are any links to scan, start the scanner
      if(count($links) > 0)
            startScanner($user_id, $numFriends, $links, $facebook,
                  $numContacts);

      mysql_close();
}

// True if this is the user's first visit to the page and they have not
submitted the form yet
```

```
else {
	// Initialize the Facebook object for API calls
	$facebook = new Facebook($appapikey, $appsecret);

	// Get the user's Facebook ID and the number of friends
	$user_id = $facebook->require_login();
	$sKey = $facebook->api_client->session_key;
	$friends = $facebook->api_client->friends_get();
	$numFriends = count($friends);

	// Users must run at least 1 automatic scan to build some of the
	//    heuristics first
	echo "<p>Please make sure you have run at least one automatic scan
		(http://apps.facebook.com/linkchecker) before running the manual
		scan.</p>";
	// Build the form. The hidden field is used to determine when the user
	//    submits the form and the scan can begin.
	echo "<form action=\"manual-scan.php\" method=\"post\"><input
		type=\"hidden\" name=\"start_scan\" value=\"1\" /><input
		type=\"hidden\" name=\"sKey\" value=\"" . $sKey. "\" />";
	echo "User ID: <input type=\"text\" name=\"user\" value=\"" . $user_id
		. "\" readonly /><br />";
	echo "Friends: <input type=\"text\" name=\"numFriends\" value=\"" .
		$numFriends . "\" readonly /><br /><br />";
	echo "Enter the URLs to scan (one URL per line):<br />";
	echo "<textarea name=\"urllist\" rows=\"10\" cols=\"60\"></textarea><br
		/>";
	echo "Sender: <select id=\"friends\" name=\"friends\"><option
		value=\"0\"></option>";

	// Build a drop-down menu of the user's friends to choose as the sender
	if(is_array($friends)) {
		$uids = "";
		$counter = 0;
		foreach($friends as $friend) {
			if($counter == 0) {
				$uids = $friend;
				$counter++;
				continue;
			}
			$uids.=", " . $friend;
			$counter++;
		}
		$name = $facebook->api_client->users_getInfo($uids,"last_name,
				first_name");
		$counter = 0;
		foreach($friends as $friend) {
			echo "<option value=\"" . $friend . "\">"  .
				$name[$counter]["first_name"] . " " .
				$name[$counter]["last_name"] . "</option>";
			$counter++;
		}
	}
	else {
		echo "<option>Unable to load friends</option>";
		echo "<script>document.getElementById('friends').disabled=true;
			</script>";
```

```
	}

	// The user can also enter a Facebook user ID of the sender if they are
	//    not in their friend list
	echo "</select><br />Or...<br />";
	echo "Sender ID: <input type=\"text\" name=\"senderID\"
		maxlength=\"32\" /><br />";
	echo "<input type=\"submit\" name=\"submit\" value=\"Submit\"
		/></form>";
}

exit(0);

?>
```

F

linkchecker.conf

```
/* Author: Mike Robertson
* Application: Link Checker
* File: 'linkchecker.conf'
*   A file containing some of the global configuration options for the
*     application.
*/

<?php

$appapikey = '0';      // Facebook API Key - Request from Facebook[5]
$appsecret = '0';      // Facebook Application Secret - request from Facebook[1]
$server = "localhost";  // MySQL Server (in relation to the web server)
$contact_database = "linkchecker";  // Name of the database that stores
                                    //    contact counts
$sb_database = "safebrowsing";      // Name of the database that stores
                                    //    Google Safe Browsing data
$username = "linkchecker";          // MySQL username for Link Checker
$password = "linkchecker";          // MySQL password for the Link Checker
$report_dir = "/data/www/results/"; // The local directory where reports will
                                    //    be stored

$GOOD_SCORE = 1;  // score to add if the test returns a positive result
$BAD_SCORE = -1;  // score to add if the test returns a negative result
$NO_SCORE = 0;  // score to add if the test returns an inconclusive result

// ISO 3166 Country Codes of untrusted countries
// Based on data from [4]
$untrustedCountries = array("CN", "RU", "CA", "IR", "MN", "ES", "GB", "CZ",
                        "UA", "RO", "BG", "BD", "TH", "VN", "MY", "ID",
                        "KR");

?>
```

[5] http://developers.facebook.com/

G

Source Code – sb-black-updates.sh

```
#!/bin/bash

# Copyright 2010 Michael Robertson
# This program is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program.  If not, see <http://www.gnu.org/licenses/>.

# Database username
USER="googlesb"
# Database name
DATABASE="safebrowsing"
# Safe Browsing API key – request from [12]
APIKEY="0"

# Get the current blacklist version that is in the database
RESULT=`mysql -u $USER < getcurrentver.sql`
BLACKVER=`echo $RESULT | cut -d" " -f2`

# Loop until the script is killed (new check every 30 minutes)
while true
do

# Loop until we get a successful response from Google
SUCCESS=0
while [ $SUCCESS -eq 0 ]
do
      # Logging message
      echo "`date` sb-updates: Attempting to update blacklist from
            sb.google.com" >> /var/log/messages

      # Get the next available version of the blacklist
      wget -O black\:1\:$BLACKVER
            "http://sb.google.com/safebrowsing/update?client=api&apikey=$APIK
            EY&version=goog-black-hash:1:$BLACKVER" 2> /tmp/black-result.tmp

      # Make sure we got a good response from Google
```

```
SUCCESS=`cat /tmp/black-result.tmp | grep -c "200 OK"`
# True if the request was a success
if [ $SUCCESS -eq 1 ]
then
      # Reset the error counter
      echo 0 > black-errors

      # Get the new blacklist version number
      BLACKHEADER=`cat black\:1\:$BLACKVER | head -n 1`
      NEWBLACK=`echo $BLACKHEADER | cut -d" " -f2 | cut -d"." -f2 |
                  cut -d"]" -f1`
      if [ -z $NEWBLACK ]
      then
            NEWBLACK=$BLACKVER
      fi

      # Check to see if the list sent from Google is just a "diff"
      # update or an entirely new list
      ISUPDATE=`echo $BLACKHEADER | grep -c update`

      # True if the list is just a "diff" update
      if [ $ISUPDATE -eq 1 ]
      then
            # Get a list of hashes that need to be added to the
            # database (marked with +)
            cat black\:1\:$BLACKVER | grep + > /tmp/blackadds.tmp

            # Load the new hashes into the database
            RESULT=`echo "LOAD DATA INFILE '/tmp/blackadds.tmp' IGNORE
                  INTO TABLE blacklist_hash FIELDS TERMINATED BY '+'
                  (@skip, hash);" | mysql -u $USER $DATABASE`

            # True if the load was successful
            if [ -z $RESULT ]
            then
                  # Update the updateTime field for the database
                  CURRENTTIME=`date +"%F %T"`
                  echo "UPDATE updates SET updateTime='$CURRENTTIME',
                        version='$NEWBLACK' WHERE id = '1';" | mysql -u
                        $USER $DATABASE

            # True if the load was unsuccessful
            else
                  # Log an error
                  echo "`date` sb-updates: Failed to update
                        blacklist_hash database" >> /var/log/messages
            fi

            # Get a list of hashes that need to be removed from the
            # database (marked with -)
            cat black\:1\:$BLACKVER | grep - > /tmp/blacksubs.tmp

            # Delete the hashes from the database
            while read LINE
            do
                  HASH=`echo $LINE | cut -d"+" -f2`
                  echo "DELETE FROM blacklist_hash WHERE hash =
```

```
                        '$HASH';" | mysql -u $USER $DATABASE
                done < /tmp/blacksubs.tmp

        # True if the list is an entirely new list
        else
                # Get a list of hashes to add (all of them since it is a
                # new list)
                cat black\:1\:$BLACKVER | grep + > /tmp/blackadds.tmp

                # Drop the old database and recreate it
                RESULT=`mysql -u $USER < recreate-black-hash.sql`

                # True if the database was successfully recreated
                if [ -z $RESULT   ]
                then
                        # Load the new hashes into the database
                        RESULT=`echo "LOAD DATA INFILE '/tmp/blackadds.tmp'
                                IGNORE INTO TABLE blacklist_hash FIELDS
                                TERMINATED BY '+' (@skip, hash);" | mysql -u
                                $USER $DATABASE`

                                        # True if the load was succesful
                                        if [ -z $RESULT ]
                                        then
                                                # Update the updateTime field
                                                # for the database
                                                CURRENTTIME=`date +"%F %T"`
                                                echo "UPDATE updates SET
                                                        updateTime='$CURRENTTIM
                                                        E', version='$NEWBLACK'
                                                        WHERE id = '1';" |
                                                        mysql -u $USER
                                                        $DATABASE
                                        # True if the load failed
                                        else
                                        # Log an error message
                                        echo "`date` sb-updates: Failed to
                                                update blacklist_hash
                                                database" >>
                                                /var/log/messages
                                        fi

                # True if the database could not be recreated
                else
                        # Log an error message
                        echo "`date` sb-updates: Failed to recreate the
                                blacklist_hash database" >> /var/log/messages
                fi
        fi
        # Break out of the loop so we can sleep for 30 minutes
        break 1

# True if Google sent an error
else
        # Keep track of the number of sequential errors received so far
        ERRORS=`cat black-errors`
        let PLUSONE=$ERRORS+1
```

```
            echo $PLUSONE > black-errors

            # Backoff algorithm per the API usage documentation
            if [ $ERRORS -lt 2 ]
            then
                  SUCCESS=0
                  echo "`date` sb-updates: Failed to get new blacklist from
                        sb.google.com. Backing off for 60 seconds" >>
                        /var/log/messages
                  sleep 60
            elif [ $ERRORS -lt 3 ]
            then
                  SUCCESS=0
                  echo "`date` sb-updates: Failed to get new blacklist from
                        sb.google.com. Backing off for 60 minutes" >>
                        /var/log/messages
                  sleep 3600
            elif [ $ERRORS -lt 4 ]
            then
                  SUCCESS=0
                  echo "`date` sb-updates: Failed to get new blacklist from
                        sb.google.com. Backing off for 180 minutes" >>
                        /var/log/messages
                  sleep 10800
            else
                  SUCCESS=0
                  echo "`date` sb-updates: Failed to get new blacklist from
                  sb.google.com. Backing off for 360 minutes" >>
                        /var/log/messages
                  sleep 21600
            fi
      fi
done

# Reset the successful update counter for the next try
SUCCESS=0

# Remove the temporary files
rm -f /tmp/black-result.tmp
rm -f /tmp/blackadds.tmp
rm -f /tmp/blacksubs.tmp

# Sleep for 30 minutes before updating again
sleep 1800

done
exit
```

H

Source Code – sb-malware-updates.sh

```
#!/bin/bash

# Copyright 2010 Michael Robertson
# This program is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program.  If not, see <http://www.gnu.org/licenses/>.

# Database username
USER="googlesb"
# Database name
DATABASE="safebrowsing"
# Safe Browsing API key – request from [12]
APIKEY="0"

# Get the current malware list version that is in the database
RESULT=`mysql -u $USER < getcurrentver.sql`
MALVER=`echo $RESULT | cut -d" " -f3`

# Loop until the script is killed (new check every 30 minutes)
while true
do

# Loop until we get a successful response from
SUCCESS=0
while [ $SUCCESS -eq 0 ]
do
      # Logging message
      echo "`date` sb-updates: Attempting to update malware list from
            sb.google.com" >> /var/log/messages

      # Get the next available version of the malware list
      wget -O malware\:1\:$MALVER
            "http://sb.google.com/safebrowsing/update?client=api&apikey=$APIK
            EY&version=goog-malware-hash:1:$MALVER" 2> /tmp/malware-
            result.tmp
```

```
# Make sure we got a good response from Google
SUCCESS=`cat /tmp/malware-result.tmp | grep -c "200 OK"`

# True if the request was a success
if [ $SUCCESS -eq 1 ]
then
      # Reset the error counter
      echo 0 > malware-errors

      # Get the new malware list version number
      MALHEADER=`cat malware\:1\:$MALVER | head -n 1`
      NEWMAL=`echo $MALHEADER | cut -d" " -f2 | cut -d"." -f2 |
            cut -d"]" -f1`
      if [ -z $NEWMAL ]
          then
                NEWMAL=$MALVER
          fi

      # Check to see if the list sent from Google is just a "diff"
      # update or an entirely new list
      ISUPDATE=`echo $MALHEADER | grep -c update`

      # True if the list is just a "diff" update
      if [ $ISUPDATE -eq 1 ]
      then
            # Get a list of hashes that need to be added to the
            # database (marked with +)
            cat malware\:1\:$MALVER | grep + > /tmp/malwareadds.tmp

            # Load the new hashes into the database
            RESULT=`echo "LOAD DATA INFILE '/tmp/malwareadds.tmp'
                  IGNORE INTO TABLE malware_hash FIELDS TERMINATED BY
                  '+' (@skip, hash);" | mysql -u $USER $DATABASE`

            # True if the load was successful
            if [ -z $RESULT ]
            then
                  # Update the updateTime field for the database
                  CURRENTTIME=`date +"%F %T"`
                  echo "UPDATE updates SET updateTime='$CURRENTTIME',
                        version='$NEWMAL' WHERE id = '2';" | mysql -u
                        $USER $DATABASE

            # True if the load was unsuccessful
            else
                  # Log an error
                  echo "`date` sb-updates: Failed to update
                        malware_hash database" >> /var/log/messages
            fi

            # Get a list of hashes that need to be removed from the
            # database (marked with -)
            cat malware\:1\:$MALVER | grep - > /tmp/malwaresubs.tmp

            # Delete the hashes from the database
            while read LINE
            do
```

```
                HASH=`echo $LINE | cut -d"+" -f2`
                echo "DELETE FROM malware_hash WHERE hash = '$HASH';"
                      | mysql -u $USER $DATABASE
          done < /tmp/malwaresubs.tmp

    # True if the list is an entirely new list
    else
          # Get a list of hashes to add (all of them since it is a
          # new list)
          cat malware\:1\:$MALVER | grep + > /tmp/malwareadds.tmp

          # Drop the old database and recreate it
          RESULT=`mysql -u $USER < recreate-malware-hash.sql`

          # True if the database was successfully recreated
          if [ -z $RESULT   ]
          then
                # Load the new hashes into the database
                RESULT=`echo "LOAD DATA INFILE '/tmp/malwareadds.tmp'
                      IGNORE INTO TABLE malware_hash FIELDS
                      TERMINATED BY '+' (@skip, hash);" | mysql -u
                      $USER $DATABASE`

                      # True if the load was successful
                      if [ -z $RESULT ]
                      then
                            # Update the updateTime field for the
                            # database
                            CURRENTTIME=`date +"%F %T"`
                            echo "UPDATE updates SET
                                  updateTime='$CURRENTTIME',
                                  version='$NEWMAL' WHERE id = '2';"
                                  | mysql -u $USER $DATABASE

                      # True if the load failed
                      else
                            # Log an error message
                            echo "`date` sb-updates: Failed to update
                                  malware_hash database" >>
                                  /var/log/messages
                      fi

          # True if the database could not be recreated
          else
                # Log an error message
                echo "`date` sb-updates: Failed to recreate the
                      malware_hash database" >> /var/log/messages
          fi
    fi

    # Break out of the loop so we canc sleep for 30 minutes
    break 1

# True if Google sent an error
else
    # Keep track of the number of sequential errors received so far
    ERRORS=`cat malware-errors`
```

```
            let PLUSONE=$ERRORS+1
            echo $PLUSONE > malware-errors

            # Backoff algorithm per the API usage documentation
            if [ $ERRORS -lt 2 ]
            then
                  SUCCESS=0
                  echo "`date` sb-updates: Failed to get new malware list
                        from sb.google.com. Backing off for 60 seconds" >>
                        /var/log/messages
                  sleep 60
            elif [ $ERRORS -lt 3 ]
            then
                  SUCCESS=0
                  echo "`date` sb-updates: Failed to get new malware list
                        from sb.google.com. Backing off for 60 minutes" >>
                        /var/log/messages
                  sleep 3600
            elif [ $ERRORS -lt 4 ]
            then
                  SUCCESS=0
                  echo "`date` sb-updates: Failed to get new malware list
                        from sb.google.com. Backing off for 180 minutes" >>
                        /var/log/messages
                  sleep 10800
            else
                  SUCCESS=0
                  echo "`date` sb-updates: Failed to get new malware list
                        from sb.google.com. Backing off for 360 minutes" >>
                        /var/log/messages
                  sleep 21600
            fi
      fi
done

# Reset the successful update counter for the next try
SUCCESS=0

# Remove the temporary files
rm -f /tmp/malware-result.tmp
rm -f /tmp/malwareadds.tmp
rm -f /tmp/malwaresubs.tmp

# Sleep for 30 minutes before updating again
sleep 1800

done
exit
```

I

Test Data Set – Safe URLs

The following table contains the safe URLs portion of the test data set used with the manual scan tests described in Section 3.3. This data is a subset of the test data used by Robichaux and Ganger in their 2006 study of anti-phishing software [34].

http://fotolog.terra.com/
http://www.flogao.com.br/gatasdeksa
http://music.kapook.com/newrelease2.shtml
http://blueyspoker.easycash.com.au/p1.asp
http://bluewin.ch/
http://www.caf.fr/SimuLog.htm
http://www.friday-ad.co.uk/FullAdDetails.asp
http://www.satsw.net/
http://www.devochki.ru/showanketbynum.php
http://list4.auctions.yahoo.co.jp/jp/2084057161-categoryleaf.html
http://windowsmedia.com/Shop/FindAlbums.asp
http://www4.climatempo.com.br/climatempo/cep.php
http://board.n-age.in.th/forum_topics.asp
http://212.174.200.74/AirDev/login/loginIndex.jsp
http://aovivo.noticias.terra.com.br/noticias/aovivo/index2.htm
http://www.littlewoods-online.com/rf/navigation/home/index.do
http://www.publicanary.com/cristal/versos_tristes.htm
http://www.flyloco.de/DE
http://www.silver-world.net/auberge.php
http://www.poemasromancesyamor.com/principal.html
http://elearning.uws.edu.au/webct/ticket/ticketLogin

http://services.orange.co.th/ringtone/sounddemo.asp
http://lotteries.olgc.ca/consumer_wn_ct.jsp
http://www.ittakes2.com/login.php
http://www.mochisonline.com/foros/search.php
http://www.ancestry.com/s12955/t5299/landing/rd.ashx
http://www.bingobilly.com/
http://www.fife.gov.uk/themes/index.cfm
http://www.winantiviruspro.com/landingpage/wavpro_4in1.php
http://www.supreme.state.az.us/publicaccess/partylist.asp
http://www.neti.ee/cgi-bin/teema/HARIDUS_JA_KULTUUR/Haridus/
http://www.mariasearch.com/XML/index.php
http://hk.f522.mail.yahoo.com/ym/login
http://psc.disney.go.com/disneychannel/theproudfamily/games/musicvideo/index.html
http://fr.barbie.com/activities/games/softwared/2003/
http://clearstation.etrade.com/
http://ar.f331.mail.yahoo.com/ym/login
http://6arab.com/singers-ar/north/keffori/index-ar.shtml
http://www.compusa.com/default.asp
http://www.internacional.com.br/
http://www.mtv.com/onair/dyn/realworld-season16/series.jhtml
http://www.uh.edu/
http://www.tarjetasvirtuales.com/pe_es/
http://randki.o2.pl/
http://hub.benl.ebay.be/buy
http://www.aussiematchmaker.com.au/Default.aspx
http://www.lagrange-holidays.com/home.php3
http://www.portalcanoas.net/principal.asp
http://www.q8me.com/
http://www.zecheval.com/banque.php

Table I.1: Safe URL Test Data

J
Test Data Set – Malicious URLs

The following table contains the malicious URLs portion of the test data set used with the manual scan tests described in Section 3.3. This data is a random subset of data published by PhishTank and Malware Patrol [8, 31]. These links should **not** be opened in a web browser, since they likely contain malicious software or phishing scams.

http://wow.worldofwarcraft.com.zuoya.info/
http://yahoo08.t35.com/login.htm
http://get-doubled.tk/
http://www.facebook72.0fees.net/
http://zyngafanwinner.tk/
http://tboe-dvdrip-axxo.tk/
http://r123321.justfree.com/rs2/index2.php
http://www.orkuteiros.tk/
http://www.confeccionespanda.com/Logon.html
http://crm.mihnati.com/export/connexion/
http://74.208.109.36/facebook/
http://www.hgsc.gov.cn/postinfo1.html
http://xmandesign.com/media/formslogin.php
http://bunonline.com/abon/1.php?logon=myposte
http://santander-updates.com/
http://whtrainingaccount18.com/search.php
http://thelloydzinternet.com/
http://www.michaelsfence.com/p7pm/
http://www.5kant.ch/texte/
http://kaakibat2001.com/cgi-bin/IBlogin.php

http://www.worldofwarcrxaft-login.com/
http://64.22.67.45/~lk/online.hmrc.gov.uk/
http://www.amon.up.am/Confirm.htm
http://www.email-upgrade.us.ms/
http://www.hamptonhaddon.com/paypal.com
http://wywg.bianyaoyang.cn
http://appline.ieguide.co.kr
http://onsearch.co.kr
http://wywg.bbhhsp.cn
http://downlopaginvisualiz.com.sapo.pt
http://wywg.5207628.cn
http://cd-baidu.com.cn
http://wywg.bjyax.cn
http://rs47cg2.rapidshare.com
http://blogaofotos8.com.sapo.pt
http://wywg.xh-violet.com.cn
http://wywg.youyou365.cn
http://gmblog04.fileave.com
http://total.zerodoctor.com
http://wywg.go162.cn
http://uploaded.100free.com
http://wywg.hs1718.cn
http://komplex2.psyradio.org
http://wywg.gmjdsb.cn
http://wywg.huruoqi.cn
http://77.245.61.232
http://microsofj.com:88
http://ipod.imglobal.net
http://wywg.syzst.com.cn
http://nt1oo.8866.org:8866

Table J.1: Malicious URL Test Data

K

Request for Comments – Twitter

The following request was sent to Twitter on March 25, 2010 via their inquiry form[6]:

> *To whom it may concern,*
> *I am currently a graduate student at the Rochester Institute of Technology and as part of my MS thesis I am researching the spread of malicious web content through social sites like Twitter. I know that you have recently started filtering users from posting URLs that are known to be malicious, but I was wondering if you publish any information on how this filtering is actually accomplished (i.e. static blacklists or dynamic heuristics)? Also, I am curious about whether or not you have the ability to retroactively remove or block links which made it through the filters but were later found to be malicious? Finally, do you have a strategy to deal with shortened URLs that could potentially avoid the filters?*
>
> *Any information that you can provide would be extremely helpful for my research. Thanks in advance for your time and consideration.*
>
> *Regards,*
>
> *Michael Robertson*
> *Rochester Institute of Technology*
> *[Contact information removed]*

Within minutes of submitting the form, the following reply was received via email:

> *Hi ,*

[6] http://twitter.com/help/contact/make_press_inquiry

Thanks for your note. At the moment, we are not making our user statistics public. There are, however, several third party websites that make estimations of the number of twitter users, both in the U.S. and internationally. You may find useful information on those sites, since we cannot offer you this information.

You may also want to follow our @Twitter account for regular company updates, and read our blog for more in-depth information. We appreciate your interest in the company, and thanks for reaching out to us.

Thanks,

Twitter Communications Team

As of the time of this writing, no further response has been received from Twitter. Unfortunately, no detailed technical information could found on their website related to their URL filtering technology.

L

Request for Comments – Facebook

The following request was sent to Facebook on March 25, 2010 via email:

from *Michael Robertson*
to *press@facebook.com*
date *Thu, Mar 25, 2010 at 2:41 PM*
subject *Request for information about URL filtering*

To whom it may concern,

I am currently a graduate student at the Rochester Institute of Technology and as part of my MS thesis I am researching the spread of malicious web content through social sites like Facebook. I know that you have recently started filtering users from posting URLs that are known to be malicious, but I was wondering if you publish any information on how this filtering is actually accomplished (i.e. static blacklists or dynamic heuristics)? Also, I am curious about whether or not you have the ability to retroactively remove or block links which made it through the filters but were later found to be malicious? Finally, do you have a strategy to deal with shortened URLs that could potentially avoid the filters?

Any information that you can provide would be extremely helpful for my research. Thanks in advance for your time and consideration.

Regards,

Michael Robertson
Rochester Institute of Technology

[Contact information removed]

An immediate reply from Facebook followed, also via email:

from *Press <press-noreply@facebook.com>*
to *Michael Robertson*
date *Thu, Mar 25, 2010 at 2:41 PM*
subject *Thanks for contacting Facebook*

Hello,

Due to a high volume of requests, we are unable to respond to everyone immediately. We understand that you may be on deadline and will do our best to respond as quickly as possible. We also encourage you to visit our Press Page (www.facebook.com/press.php), where you will find general information such as the latest statistics.

If you are not a member of the press, please refer to the below resources and direct your inquiry appropriately:

** Help Center (http://www.facebook.com/help.php): If you are a user experiencing a problem with the site, or writing in with a suggestion*

** http://www.facebook.com/brandpermissions: If you would like to obtain permission to use Facebook's trademarks and/or copyrights for commercial and promotional purposes*

** http://www.facebook.com/facebook#/press/request.php: If you have a speaking request*

** http://www.facebook.com/advertising: If you are interested in advertising on the site*

** http://www.facebook.com/sponsorship: If you would like to request Facebook's sponsorship of an event*

Thanks,

The Facebook Corporate Communications Team

As of the time of this writing, no further response has been received from Facebook. Unfortunately, the links provided did not yield any detailed technical information about the site's URL filtering technology.

M

Supplementary File Contents

This section outlines the contents of the supplementary files included with this thesis.

- **<ZIP>\Results\ -** A directory containing copies of the raw result data obtained during the course of this thesis. Personally identifiable information has been sanitized with an “x”.
- **<ZIP>\src\ -** A directory containing electronic copies of all the source code listed in this Appendix, as well as a copy of the GNU General Public License under which this source code is being released.
- **<ZIP>\Test Data** - A directory containing the test data used by the manual scan. Note that the file titled test-data_BAD.txt contains URLs to real-world malicious websites. Please do not visit these links in a web browser.
- **<ZIP>\A Social Approach to Security.pdf** – An electronic copy of this document.
- **<ZIP>\A Social Approach to Security.pptx** – A copy of the defense slides for this thesis.
- **<ZIP>\Application Demo – Automated Scan.mp4** - A video demonstration of the proof-of-concept application’s automated scan function.
- **<ZIP>\Application Demo – Manual Scan.mp4** - A video demonstration of the proof-of-concept application’s manual scan function.

N

Application Installation Instructions

The following procedure can be used to install an instance of the Linkchecker application:

1. Install all of the prerequisite applications via their recommended installation procedure. The prerequisites that are needed are:
    a. Web Server (i.e. Apache httpd)

        *Note: The WWW root directory should be configured as /data/www/html. If another directory is desired, /data/www/html can be linked to the appropriate directory.

    b. PHP (5.1.6 or greater recommended)
        i. Must include MySQL support
    c. PEAR
    d. MySQL Client & MySQL Server
    e. jwhois Client
    f. cURL Client
    g. wget Client
2. If necessary, configure the system's firewall to allow inbound connections on the port used by the web server. In addition, disable SELinux by editing the /etc/sysconfig/selinux file.
3. Create the /data/www directory, if it does not already exist

    *Note: If it is not desirable to use the /data/www directory to host the application files, the user must do a find/replace in each PHP script to adjust any lines configured for /data/www with the appropriate directory
4. Copy the index.php, scanner.php, fb-checks.php, url-checks.php, and manual-scan.php scripts to /data/www/html/linkchecker/. These scripts should have permissions of 755.

5. Copy linkchecker.conf to /data/www/. This file should have permissions of 640 and belong to the same group as the account used by the web server.
6. Edit the $appapikey and $appsecret variables in linkchecker.conf to reflect the keys registered with the Facebook API.
7. Download a copy of the GeoLite Country database in binary format (GeoIP.dat) from http://www.maxmind.com/app/geolitecountry. Copy the database to the /data/www directory with permissions of 644.
8. Create the /data/www/results directory with permissions of 770. This directory should belong to the same group as the account used by the web server.
9. Create the /data/www/api directory.
10. Download a copy of the Facebook API from http://developers.facebook.com/ and copy it to the /data/www/api directory.
11. Execute the create-db.sql script on the MySQL database to setup the databases.
12. Copy the getcurrentver.sql, inserthashes-black.sql, inserthashes-malware.sql, recreate-black-hash.sql, and recreate-malawre-hash.sql scripts to the /root directory.
13. Copy the sb-black-updates.sh and sb-malware-updates.sh scripts to /root.
14. Update the APIKEY value in both the sb-black-updates.sh and sb-malware-updates.sh to reflect the API key registered with the Google Safe Browsing API.
15. Execute the sb-black-updates.sh and sb-malware-updates.sh scripts in the background to start the Google Safe Browsing update process. These scripts will continue to run and check for updates every 30 minutes until they are stopped.
16. Execute the `pear install Net_GeoIP-1.0.0RC3` command to install the Net_GeoIP PHP module. This module is necessary for the serverLocation test function.
17. Update the Facebook Application settings (http://www.facebook.com/#!/developers/) to point to the application server.

O

Works Cited

[1] Adelson, Jay. April 4, 2010. Update from Jay. Digg. http://about.digg.com/blog/update-jay. Accessed April 17, 2010.

[2] Anti- Phishing Working Group. March, 5, 2010. Phishing Attack Trends Report – Q4 2009. http://www.antiphishing.org/reports/apwg_report_Q4_2009.pdf. Accessed April 17, 2010.

[3] Asur, S. and Huberman, B. March 29, 2010. Predicting the Future With Social Media. http://arxiv.org/PS_cache/arxiv/pdf/1003/1003.5699v1.pdf. Accessed April 17, 2010.

[4] Boscovich, R., Campana, T. J., Canavor, D., et al. November 2, 2009. Microsoft Security Intelligence Report. Volume 7 – January through June 2009. Microsoft, Inc. http://www.microsoft.com/downloads/details.aspx?FamilyID=037f3771-330e-4457-a52c-5b085dc0a4cd&displaylang=en. Accessed April 17, 2010.

[5] Boykin, P. O. and Roychowdhury V. P. 2005. Leveraging Social Networks to Fight Spam. In *Computer*, 38(4), pages 61-68. IEEE. DOI= 10.1109/MC.2005.132.

[6] Chou, N., Ledesma, R., Teraguchi Y., Boneh, D., and Mitchell, J. C. 2004. Client-side defense against web-based identity theft. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium* (San Diego, California, February 2004). NDSS '04. ACM, New York, NY. http://crypto.stanford.edu/SpoofGuard/webspoof.pdf.

[7] comScore. October 27, 2009. 36 Million German Internet Users Viewed More Than 6 Billion Videos Online in August 2009. comScore, Inc. http://www.comscore.com/Press_Events/Press_Releases/2009/10/36_Million_German_Internet_Users_Viewed_More_Than_6_Billion_Videos_Online_in_August_2009. Accessed April 17, 2010.

[8] Correa, A. 2010. Aggressive Block List. Malware Patrol. http://www.malwarepatrol.net/cgi/submit-agressive?action=list&type=agressive. Accessed April 8, 2010.

[9] Curran, D. December 15, 2009. Long URL Please API. Peel Me A Grape. http://www.longurlplease.com/docs. Accessed April 17, 2010.

[10] Curran, D. 2010. longurlplease-firefox. http://code.google.com/p/longurlplease-firefox/. Accessed April 17, 2010.

[11] Dao T. 2005. Term frequency/Inverse document frequency implementation in C#. The Code Project. http://www.codeproject.com/KB/cs/tfidf.aspx. Accessed January 6, 2010.

[12] Developer's Guide. 2010. In *Google Safe Browsing API*. Google. http://code.google.com/apis/safebrowsing/developers_guide.html. Accessed April 17, 2010.

[13] Dhamija, R., Tygar, J. D., and Hearst, M. 2006. Why phishing works. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Montréal, Québec, Canada, April 22 - 27, 2006). R. Grinter, T. Rodden, P. Aoki, E. Cutrell, R. Jeffries, and G. Olson, Eds. CHI '06. ACM, New York, NY, 581-590. DOI= http://doi.acm.org/10.1145/1124772.1124861

[14] Ellison, N. B., Steinfield C., and Lampe, C. 2007. The Benefits of Facebook "Friends:" Social Capital and College Studens' Use of Online Social Network Sites. In *Journal of Computer-Mediated Communication*, 12(4), article 1. http://jcmc.indiana.edu/vol12/issue4/ellison.html. Accessed December 15, 2009.

[15] Extended Permissions. March 16, 2010. In *Facebook Developer Wiki*. Facebook. http://wiki.developers.facebook.com/index.php/Extended_permissions. Accessed April 17, 2010.

[16] Facebook Connect. 2010. Build and grow with Facebook Connect. Facebook. http://developers.facebook.com/connect.php. Accessed April 17, 2010.

[17] Facebook Developer Wiki. March 29, 2010. Facebook. http://wiki.developers.facebook.com/index.php/API. Accessed April 17, 2010.

[18] Facebook Factsheet. 2009. Facebook. http://www.facebook.com/press/info.php?factsheet. Accessed December 15, 2009.

[19] Fette, I., Sadeh, N., and Tomasic, A. June 2006. Learning to Detect Phishing Emails. In *Carnegie Mellon University Report CMU-ISRI-06-112*. Carnegie Mellon University. http://reports-archive.adm.cs.cmu.edu/anon/isri2006/CMU-ISRI-06-112.pdf. Accessed December 15, 2009.

[20] Goyvaerts, J. June 17, 2009. Sample Regular Expressions. http://www.regular-expressions.info/examples.html. Accessed April 17, 2010

[21] Greiner, L. 2009. Hacking social networks. *netWorker* 13, 1 (Mar. 2009), 9-11. DOI= http://doi.acm.org/10.1145/1516035.1516038.

[22] ISV Application Readiness and Certification. 2010. Microsoft. http://www.microsoft.com/windowsserver2008/en/us/isv.aspx. Accessed April 17, 2010.

[23] Kerris, N. and Pope, S. November 4, 2009. Apple Announces Over 100,000 Apps Now Available on the App Store. Apple, Inc. http://www.apple.com/pr/library/2009/11/04appstore.html. Accessed April 17, 2010.

[24] Kim, J., Chung, K., and Choi, K. July-August 2007. Spam Filtering with Dynamically Updated URL Statistics. In *IEEE Security & Privacy*, 5(4), pages 33-39. IEEE. DOI= 10.1109/MSP.2007.95.

[25] MaxMind. 2010. GeoLite Country. http://www.maxmind.com/app/geolitecountry. Accessed March 23, 2010.

[26] MaxMind. 2010. GeoIP PHP API. http://www.maxmind.com/app/php. Accessed April 17, 2010.

[27] Microsoft Support. April 21, 2006. Session variables are lost if you use FRAMESET in Internet Explorer 6. Microsoft. http://support.microsoft.com/kb/323752. Accessed April 17, 2010.

[28] Netcraft Toolbar. 2010. Netcraft, Ltd. http://toolbar.netcraft.com. Accessed December 15, 2009.

[29] Perez S. 2009. Twitter Starts Filtering Malicious URLs. ReadWriteWeb. http://www.readwriteweb.com/archives/twitter_starts_filtering_malicious_urls.php. Accessed January 6, 2010.

[30] Phelps T. A. and Wilensky R. 2000. Robust Hyperlinks and Locations. In *D-Lib Magazine*, 6(7/8). Corporation for National Research Initiatives. DOI= 10.1045/july2000-wilensky.

[31] Phish Search. 2010. PhishTank. http://www.phishtank.com/phish_search.php?valid=y&active=y. Accessed April 17, 2010.

[32] Privacy Policy. March 26, 2010. In *Facebook Site Governance*. Facebook. http://www.facebook.com/note.php?note_id=10150162286770301. Accessed April 17, 2010.

[33] Rate Limiting. April 14, 2010. In *API Wiki*. Twitter. http://apiwiki.twitter.com/Rate-limiting. Accessed April 17, 2010.

[34] Robichaux, P. and Ganger, D. L. September 2006. Gone Phishing: Evaluating Anti-Phishing Tools for Windows. 3Sharp LLC.

[35] Rusli, E. March 30, 2010. Unvarnished: A Clean, Well-Lighted Place for Defamation. TechCrunch. http://techcrunch.com/2010/03/30/unvarnished-a-clean-well-lighted-place-for-defamation/. Accessed April 17, 2010.

[36] Schnitt, Barry. April 6, 2010. Responding to Your Feedback. In *The Facebook Blog*. http://blog.facebook.com/blog.php?post=379388037130. Accessed April 20, 2010.

[37] Statistics. 2010. In *Press Room*. Facebook. http://www.facebook.com/press.php#!/press/info.php?statistics. Accessed April 17, 2010.

[38] Strater, K. and Lipford, H. R. 2008. Strategies and struggles with privacy in an online social networking community. In*Proceedings of the 22nd British HCI Group Annual Conference on HCI 2008: People and Computers Xxii: Culture, Creativity, interaction - Volume 1* (Liverpool, United Kingdom, September 01 - 05, 2008). British Computer Society Conference on Human-Computer Interaction. British Computer Society, Swinton, UK, 111-119.

[39] Tancer, B. July 11, 2006. MySpace Moves Into #1 Position for all Internet Sites. Hitwise Pty. Ltd. http://weblogs.hitwise.com/bill-tancer/2006/07/myspace_moves_into_1_position.html. Accessed December 15, 2009.

[40] Tynan, D. April 4, 2010. Facebook's Sneaky Apps and Privacy Issues. PCWorld. http://www.pcworld.com/article/193423/facebooks_sneaky_apps_and_privacy_issues.html. Accessed April 17, 2010.

[41] Window.location. March 27, 2010. Mozilla Developer Center. https://developer.mozilla.org/En/DOM/Window.location. Accessed April 17, 2010.

[42] Wu, M., Miller, R. C. and Garfinkel, S. L. 2006. Do security toolbars actually prevent phishing attacks?. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Montréal, Québec, Canada, April 22 - 27, 2006). R. Grinter, T. Rodden, P. Aoki, E. Cutrell, R. Jeffries, and G. Olson, Eds. CHI '06. ACM, New York, NY, 601-610. DOI= http://doi.acm.org/10.1145/1124772.1124863.

[43] Zhang, Y., Egelman, S., Cranor, L., and Hong, J. 2007. Phinding Phish: Evaluating Anti-Phishing Tools. In *Proceedings of the 14th Annual Network and Distributed System Security Symposium* (NDSS '07, February 28 – March 2, 2007). ACM. http://lorrie.cranor.org/pubs/ndss-phish-tools-final.pdf. Accessed December 16, 2009.

[44] Zhang, Y., Hong, J. I., and Cranor, L. F. 2007. Cantina: a content-based approach to detecting phishing web sites. In *Proceedings of the 16th international Conference on World Wide Web* (Banff, Alberta, Canada, May 08 - 12, 2007). WWW '07. ACM, New York, NY, 639-648. DOI= http://doi.acm.org/10.1145/1242572.1242659.