MQP-CEW-1301

Monitoring Web Ads

a Major Qualifying Project Report
submitted to the Faculty of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

____

Andrew Feeney

____

Matthew Mancuso

April 22, 2013

____

Professor Craig E. Wills

Contents

Abstract 4
1 Introduction 5
2 Background 7
2.1 Web Advertising 7
2.2 Behavioral Targeting 7
2.3 The Technology 9
2.3.1 Hypertext Transfer Protocol (HTTP) 9
2.3.2 Data Anonymization 11
2.4 Implications 12
2.5 Summary 12
3 Design 13
3.1 Previous Data Collection Methods 13
3.2 Requirements 14
3.3 Why a Virtual Machine? 14
3.4 Summary 15
4 Research & Tools 16
4.1 Browser Automation 17
4.1.1 Selenium WebDriver 18
4.2 Proxies 19
4.2.1 Fiddler 21
4.2.2 BrowserMob-Proxy 21
4.2.3 OWASP Zed Attack Proxy (ZAP) 22
4.3 Summary 23
5 Implementation & System in Action 24
5.1 File Formats 24
5.1.1 URL-Scripts 24
5.1.2 Header Output Files 27
5.1.3 Summary Output Files 27
5.2 Data Collection 29
5.2.1 Input 30
5.2.2 Processing 31

5.2.3 Output ... 32
5.3 Data Parsing ... 32
5.3.1 Input ... 33
5.3.2 Processing ... 34
5.3.3 Output ... 37
5.4 Summary ... 37
6 Results ... 38
6.1 Testing Environment ... 38
6.2 Initial Results ... 38
6.3 Implications ... 39
6.4 Summary ... 39
7 Future Work ... 40
7.1 Maintenance and Improvements ... 40
7.2 Known Issues ... 40
7.2.1 ZAP ... 40
7.3 Additional Features ... 41
7.3.1 Website Interaction ... 41
7.3.2 Optical Character Recognition (OCR) ... 42
7.4 Summary ... 42
8 Conclusion ... 44
References ... 45
Appendix ... 46
List of proxies researched ... 46

Figures

Figure 1 - A comparison of two HTTP request headers from Facebook.com (top) and WPI.edu (bottom). 10
Figure 2 - A comparison of two different yet equally valid HTTP response headers from Facebook.com (left) and WPI.edu (right). 11
Figure 3 - A complete diagram of data interaction of the project. 17
Figure 4 - Interaction process for Selenium. The program makes calls to the Selenium API, which uses a plugin to communicate with the Web Browser. 19
Figure 5 - Interaction with the proxy server. HTTP requests are made through the web browser, redirected to the proxy, recorded, and forwarded to the Internet. Responses from the internet are directed to the proxy, recorded, and forwarded to the browser. 20
Figure 6 - The basic representation of the flow of commands when using the data collection tool. 23
Figure 7 - A small subset of a sample URL input list. 25
Figure 8 - A simple of example of more complex interaction. This script will navigate to pages and then log in to each. 26
Figure 9 - An example entry in the header output file. 27
Figure 10 - A representative sample from the summary output file. 28
Figure 11 – The data collection module, indicated by the red arrow. 30
Figure 12 - Data collection loop. 32
Figure 13 – The data processing module, indicated by the red arrow. 33
Figure 14 - Main data processing loop. 35
Figure 15 - The summary file generation loop. 36
Figure 16 - The header file generation loop. 37

Abstract

Many modern advertising techniques now implement tracking techniques to “watch” users in order to serve up relevant advertising. Although benign on the surface, this technique raises privacy concerns. This project is designed to allow automated data collection and processing from regular web browsing behavior in order to determine what data is being collected and derive what advertisers really know about their users.

1 Introduction

This project is designed to automate manual procedures currently in use for collecting HTTP data from the Internet. Through analysis these data will be able to show trends in advertising presences on the Internet and how advertisements being served change depend on the presence of user behaviors and browsing histories. Particularly, we work to eliminate the necessity for direct human interaction in order to navigate through a set of Web pages, perform interactions as needed, and return usable data for further analysis.

To meet these goals, we decided on an approach that allows a combination of tools to perform these actions and synchronize the activities. This approach necessitates writing a “command” module that organizes and coordinates these tools, as well as manages the data flow between processes.

Automating this process leads to a level of consistency not inherent to a human-dependent process and allows for the re-use of input data to see how the output varies over time. Automated tools also allow for parallelization of the data collection process, leading to data sets many times the current size for little additional effort. Automation also allows the human researchers involved to dedicate their time toward more efforts involved in processing and analyzing the data, rather than simple manual collection behaviors.

The remainder of this report is organized into several chapters. In Chapter 2, we describe some of the background involved in this project. This background includes basic explanations for technologies that we reference and analyze, along with a motivation for the research and a high-level description of modern advertising techniques. In Chapter 3 we discuss the design philosophy we adopted for the project, and talk about the procedures and tools we are working to replace with our project. In Chapter 4 we discuss the research done for this project and the tools we decided to use and integrate into our project. In Chapter 5 we provide examples of the functional system and discuss implementation details,

as well as discuss the procedures we use to gather and process data. Chapter 6 shows early results of several tests to prove the validity of our approach and the functionality of our project. In Chapter 7 we discuss possible future work that can be done to further enhance functionality provided by our project. Our paper ends with Chapter 8, where we provide a short conclusion of our work.

2 Background

In order to understand this project and the value of the software that was developed, a basic understanding of the Internet and how it works is required. However, in addition to technical knowledge it is important to understand how advertising works and what we mean to demonstrate and analyze with the software we created. With this in mind, we have included a simple background in advertising techniques and how they are implemented.

2.1 Web Advertising

The Internet is ubiquitous in our modern economy. There are thousands of online retailers and businesses that are visited every day that sell products that range from food and baking supplies to golf clubs. However, virtually every online business has something in common—advertising.

Web advertisements come in many different sizes, shapes, and experiences. Many are less invasive than others. Ads can vary from a simple text-based ad encouraging a user to “Click here to download!” to the multi-colored, flash-based advertisements that demand a user’s attention.

The project is designed to target all of these types of ads from different places. Whether ads are text-based, image-based, or flash-based is a question of implementation rather than content, and each of these types is critical to seeing the whole advertising picture.

2.2 Behavioral Targeting

Everyone has seen advertisements on the Web, and likewise, advertisements have seen everyone too. An advertising practice that has continually grown in popularity throughout the early 21st century is that of behavioral targeting.

When using this practice advertisers try to individualize and make their ads relevant to the user that is currently viewing them. In concept there is nothing inherently wrong with this—it makes perfect

sense for an online retailer such as Amazon to serve up suggestions for items similar to what you have already purchased or viewed. This increases the likelihood that users will be interested in the suggestions made.

However, as this technology and technique has grown in popularity, so too has the data that are used to profile users. There are several different popular approaches to targeting these advertisements, and we categorize them into three different types: product-based, demographic, and individual.

Product-based advertisements are categorized and served up on sites that offer those specific types of products. The most well-known example of this type of service is Google's AdSense platform, which scans the current page for keywords and content to search for similar or related ads to show on that page.

The idea of demographic advertising is adapted from the television advertising market. Television has famously used the concept of demographics to target both programming and advertising, and independent companies, most notably the Nielsen Corporation, have been monitoring television demographics for decades. The basic idea is to create several templates into which the majority of the population can be sorted. Each of these broad categories is a "demographic." For example, a middle-aged white male who lives in a suburb is likely projected to have a spouse, children, several cars, a mortgage, and other common factors with many other suburban white males. By creating many of these archetypical profiles and polling a sample of television watchers who fit each of these profiles, advertisers are able to determine which "types" of people watch certain television shows or channels, and can cater their advertisements to those groups. This principle is largely the same when it is transferred to the Web advertising model. Demographic data is polled from users that go to certain sites in order to build profiles of what their typical user is interested in.

The particular portion of behavioral targeting that we will be focusing on primarily in this project is one of individual advertising. In this style of targeted advertisement data is collected on a per-user basis to build a profile detailing the interests for that user. Given the increased granularity of the technique it is the most accurate form of advertising, however the privacy concerns grow significantly with this method. In nearly every end-user license agreement (EULA) from an advertiser, a claim is made that data is anonymized[1] and that "sensitive data," such as a user's sexual orientation, ethnicity, or other "personal" data will not be stored or used to build profiles.

However, this approach to anonymity is at best a flawed concept. How is this data being collected and aggregated? Is it possible to separate the "sensitive" data from the standard profile? These are difficult questions to answer without access to a large amount of data. We provide the ability to gather and process this data in this project.

2.3 The Technology

In order to demonstrate what current advertising practices are being employed as well as the implementations involved, it is important to talk about the technology that is being utilized to generate these advertisements in the first place, as well as the techniques used to by advertisers to claim that the data collected from users is anonymous and non-identifying.

2.3.1 Hypertext Transfer Protocol (HTTP)

The standard protocol for communications over the World Wide Web is the Hypertext Transfer Protocol, or HTTP. Messages are either requests—messages sent to servers to ask for content—or responses, which carry the content that was asked for by the client. Messages to and from web servers are encapsulated in a set of plaintext data known as headers. These HTTP headers contain information

[1] http://arstechnica.com/tech-policy/2009/09/your-secrets-live-online-in-databases-of-ruin/

necessary for managing connections and describing the contents of the body of the HTTP message. HTTP has different types of headers for requests and responses over the protocol, as well as numerous possible fields for each, as HTTP headers are variable length, with few mandatory fields.

In Figure 1, the request headers for two different sites (Facebook.com and WPI.edu) look similar with the exception of a small number differing fields. This is because request headers are fairly standard throughout the Internet, and when particular fields are needed the client is told which additional fields to include from the server being queried. Initial request headers (such as the ones from visiting a website for the first time) are generated by the Web browser, which sends these standard fields.

```
accept: */*
accept-charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
accept-encoding: gzip,deflate,sdch
accept-language: en-US,en;q=0.8
cookie: lu=ghE_h-hQkjhV_a-QVPAN8XDQ; datr=ANpuUS_-10qCV9g3wFu5QxwD; hz_amChecked=1; c_user=1074939976; csm=2; fr=0iHNAa2
9268; sub=8192; p=46; presence=EM366662542EuserFA21074939976A2EstateFDsb2F0Et2F_5b_5dElm2FnullEuct2F136666194B0EtrFnul
%2C1366662549047%2C%22act%22%2C1366662549046%2C6%2C%22https%3A%2F%2Fwww.facebook.com%2Fkatherine.fitton%22%2C%22himp%2
%22%3A%225869770906697319706%22%2C%22mf_story_key%22%3A%22-8917992409833137094%22%2C%22has_expanded_ufi%22%3A%221%22%2
dnt: 1
host: www.facebook.com
method: GET
referer: https://www.facebook.com/
scheme: https
url: /ajax/notifications/get.php?time=1366647212&user=1074939976&version=2&locale=en_US&earliest_time=0&__user=107493997
user-agent: Mozilla/5.0 (Windows NT 6.2; WOW64) AppleWebKit/537.31 (KHTML, like Gecko) Chrome/26.0.1410.64 Safari/537.31
version: HTTP/1.1
x-svn-rev: 792438
```

```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Cache-Control: max-age=0
Connection: keep-alive
Cookie: __atuvc=15%7C16%2C22%7C17; __utma=48377184.1652228072.1366259170.1366482039.1366659790.3; __utmb=48377184.25.10.1366659790;
provided)
DNT: 1
Host: www.wpi.edu
User-Agent: Mozilla/5.0 (Windows NT 6.2; WOW64) AppleWebKit/537.31 (KHTML, like Gecko) Chrome/26.0.1410.64 Safari/537.31
```

Figure 1 - A comparison of two HTTP request headers from Facebook.com (top) and WPI.edu (bottom).

In contrast, responses from Web servers tend to be much less uniform. This variety of fields is clearly shown in Figure 2; the HTTP response headers for the response from Facebook.com contain significantly more data and several fields that are not present in the response from WPI.edu, such as the “X-Webkit-CSP” and “Transfer-Encoding” fields. The absence of these fields do not invalidate the WPI.edu headers, however. It is simply a case of using different data for different transactions.

HTTP request status: 200 (OK)

Name	Value
Pragma	no-cache
X-FB-Debug	tFXtNZqT3To/vrgImkdKo1o208XLjhJRbJfh7ROzgug=
Content-Encoding	gzip
X-Content-Type-Options	nosniff
Date	Wed, 20 Mar 2013 02:59:02 GMT
X-Frame-Options	DENY
P3P	CP="Facebook does not have a P3P policy. Learn why here: http://fb.me/p3p"
Cache-Control	private, no-cache, no-store, must-revalidate
Transfer-Encoding	chunked
Connection	keep-alive
Content-Type	text/html; charset=utf-8
X-WebKit-CSP	default-src *;script-src https://*.facebook.com http://*.facebook.com https://*.fbcdn.net http://*.fbcdn.net *.facebook.net *.google-analytics.com *.virtualearth.net *.google.com 127.0.0.1:* *.spotilocal.com:* chrome-extension://lifbcibllhkdhoafpjfnlhfpfgnpldfl 'unsafe-inline' 'unsafe-eval' https://*.akamaihd.net http://*.akamaihd.net;style-src * 'unsafe-inline';connect-src https://*.facebook.com http://*.facebook.com https://*.fbcdn.net http://*.fbcdn.net *.facebook.net *.spotilocal.com:* https://*.akamaihd.net ws://*.facebook.com:* http://*.akamaihd.net;
X-XSS-Protection	0
Expires	Sat, 01 Jan 2000 00:00:00 GMT

HTTP request status: 200 (OK)

Name	Value
Date	Wed, 20 Mar 2013 03:00:45 GMT
Content-Encoding	gzip
Server	Apache/2.2.11 (Unix) mod_ssl/2.2.11 OpenSSL/0.9.8e-fips-rhel5 mod_pubcookie/3.3.3
Vary	Accept-Encoding
Content-Type	text/html
Connection	Keep-Alive
Accept-Ranges	bytes
Keep-Alive	timeout=15, max=100
Content-Length	6163

Figure 2 - A comparison of two different yet equally valid HTTP response headers from Facebook.com (left) and WPI.edu (right).

2.3.2 Data Anonymization

An important point that many advertisers try to make clear is that the data the company collects about users is anonymous. When advertisers describe data as being anonymous, generally what is meant is that data is not stored with obvious identifying information, such as a name or social security number. This, however, is a misnomer, according to security researchers at Microsoft[2]. The researchers found that the content from just the user-agent field in the HTTP requests along with an IP address accurately identified a host over eighty percent of the time. This only increased as more data were given, and personal information, coupled with the header data, can easily be traced back to the end-user.

[2] http://www.networkworld.com/news/2012/020212-microsoft-anonymous-255667.html

2.4 Implications

By capturing the whole HTTP transaction (request headers, request content, response headers, and response content) we are able mine relevant data and examine the relationship between visited sites and subsequent requests. For a typical commercial website there are many of these subsequent requests, ranging in variety from requests made to content distribution servers (what we refer to as “second-party” requests) and to outside entities, such as advertisers (what we refer to as “third-party” requests). We are able to examine which websites are directing requests to these third-parties and discover what kinds of data are being transferred back and forth.

2.5 Summary

Many web advertising corporations prefer to use targeted advertising in order to serve up relevant products and services to the user, increasing the likelihood that the user would be interested in purchasing the product or service. However, in doing so the privacy of the user may be compromised, even with good intentions on the part of the advertisers. We will be analyzing the data produced by our automated collection tool in order to trace the data that advertisers must have in order to serve the ads we collected. With this knowledge we can see if sensitive data are indeed being stored.

Our analysis technique is reliant on the message metadata sent in HTTP headers, as well as the content of the ads themselves. These are the two main sources of usable information from the collection process.

3 Design

Our project underwent several focal shifts during its inception. Initial work was focused on processing advertisements in both text and image form. However, as work began and research into the technologies available began to show that this was a more complex challenge than initially determined, focus shifted to building an automated system that would be capable of both mass data collection and complex interactions with web sites to mimic human behavior. It was decided that processing of the data could be revisited and revised, and having the ability to collect data in the first place in an automated form was more important.

3.1 Previous Data Collection Methods

The motivation for the project came from two previous data collection methods Professor Craig Wills had used in previous studies on web advertising. One of these, a Firefox extension called PageStats, needed to be replaced due to age. The other was a lengthy manual collection of data that would save many man-hours with the automation of these tasks.

The PageStats extension was used to collect general data on the presence of web advertisers on popular sites. To this end, it provided a batch processing utility that would visit a list of sites and record information about the requests and responses made by each page. A summary of the data for each page was created when the processing was complete. This summary could then be processed to acquire information about web advertiser presence on the visited sites. Unfortunately, the extension is no longer compatible with more recent Firefox versions and is no longer under active development, so an upgrade or replacement was needed.

The second data collection method involved a human operator performing a series of steps to establish an identity with sensitive information (sexual orientation, health issues, etc.) and then visiting sites to record what ads were served. Data were recorded by directing the traffic through a local man-in-

the-middle proxy that recorded the HTTP transactions. Because there needed to be interaction with the sites, the PageStats extension was insufficient for automating the behavior. However, automation of this process was desired to enable more frequent and replicable data collection.

3.2 Requirements

Our project needed to be able to replace both of these data collection methods. In order to replace PageStats batch processing capabilities were needed. Adding scripting capabilities instead of only using a list of sites to visit would allow our utility to also replace a human operator interacting with the site. In this way both data collection methods could be performed by one utility.

3.3 Why a Virtual Machine?

We decided to develop the project in a virtual machine for four main reasons. First, it would enable easy transfer of the project. Users will not need to worry about setting up any dependencies; rather he or she will simply run the virtual machine from a copy of the image by using a virtualization program. Secondly, it will ensure consistency in the test environment. Results should be comparable across any machine running the tools we developed. Third, it gives the option to users to parallelize data collection easily. Duplication and distribution of the virtual machine is both simple and a process users are likely to be familiar with. Finally, by using a virtual machine as a test bed it meant that the project could be run from any platform that supported virtualization. This allowed us to avoid testing for platform-specific issues and instead focus on getting everything working with a single operating system.

We choose to run Ubuntu, an open-source Linux distribution we were both familiar with, on the virtual machine. This choice allows us to avoid potential licensing issues with other commercial operating systems, as well as providing an extensible operating system platform with an active community that will continue to provide support for the future.

3.4 Summary

Our project focus is to create a platform for mass data collection. These data are to include all HTTP traffic to and from Websites, particularly third-party traffic. We were designing a piece of software that would entirely replace manual human interaction with web sites, so we needed software capable of interacting with web pages in a non-trivial manner, while also supporting interaction with many web sites, producing a large set of data. We decided to create this tool in a virtual Linux environment in order to ensure consistency and compatibility with many different machines and users.

4 Research & Tools

The data collection portion of our project is based heavily on coordinating several software packages. It is necessary to understand what each of these programs do in order to realize how our software is interacting with them and the Internet at large. We coordinate mostly with three major outside pieces of software: a web browser, which is Mozilla's Firefox browser; a browser automation package; and a man-in-the-middle interceptor proxy to record traffic.

A complete diagram of the software used in this project is shown in Figure 3. The blue fields ("Data Collection Program" and "Data Summarizing Program") are components that are written by the authors as a way to coordinate the rest of the tools. The data flow is visible as well. The basic flow of our data collection program involves routing commands through Selenium, our automated browser API, to the browser, through the interceptor proxy, and data are saved in HSQL databases. Our data summarizing program takes these databases as input and creates plaintext, standard format outputs which can be read by existing tools.

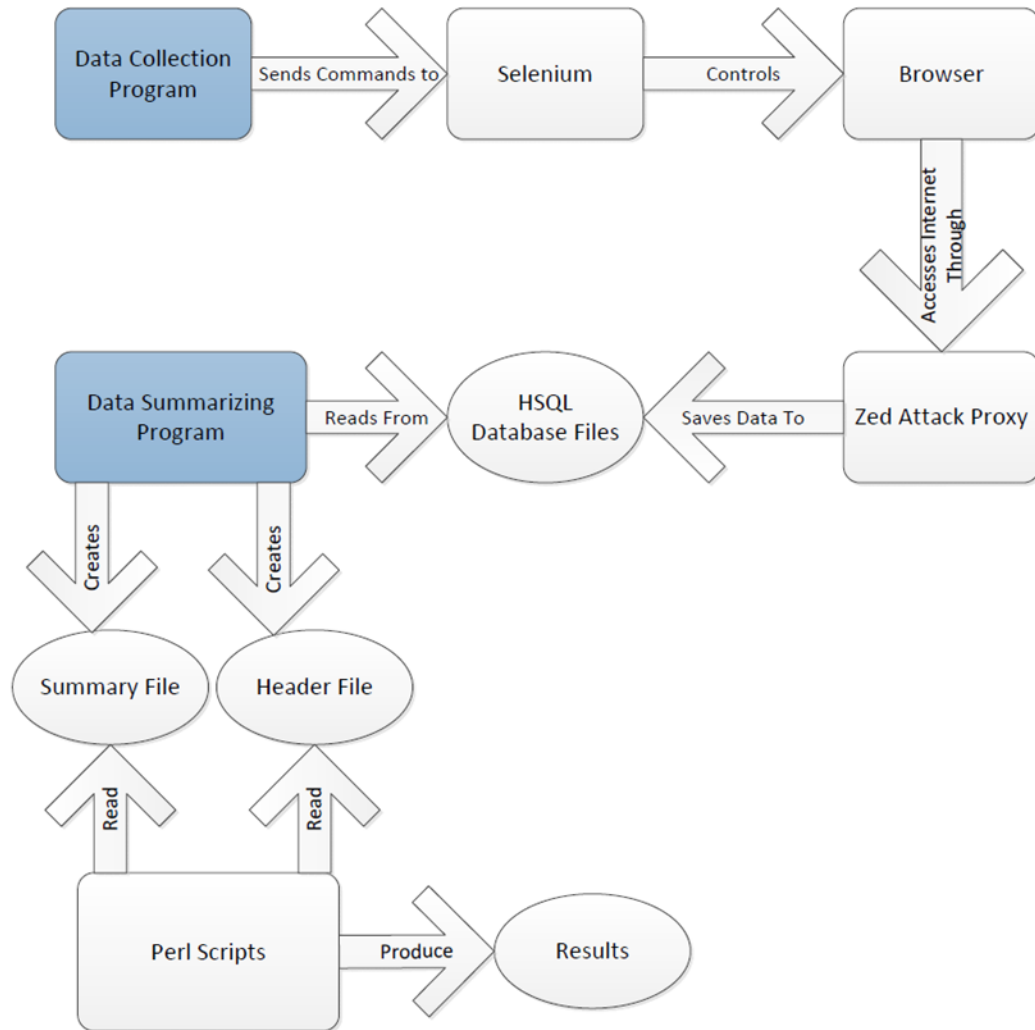

Figure 3 - A complete diagram of data interaction of the project.

4.1 Browser Automation

Few viable open-source options existed for the browser automation needs of this project. Research time and testing was put into several different options, including “Fourth-Party,” “Geb,” and “Fake.” All of these tools offered command APIs for interacting with browsers programmatically; however the only tool that offered the stability and command options we were looking for was “Selenium WebDriver,” which we quickly adopted early on in the process.

4.1.1 Selenium WebDriver

We needed to be able to access and interact with the web browser programmatically in order to automate data collection. We chose to do this with an open-source Java tool called Selenium. It is integrated into Firefox, among other browsers, through a plugin used to hook into the program. Since Selenium does not abstract away the browser to deal directly with traffic like other tools, we are able to be sure that that sites will behave exactly as they would for a normal end user and deliver the same content; it is now simply automated. This eliminated several concerns we had.

The first is how the tool handled cookies. If the way the tool handled cookies differed from how a standard browser would for an end-user then the tracking used by web advertisers may not work properly, leading to misleading and unreliable results from data collections.

Secondly, since many modern websites rely heavily on JavaScript in order to load content *after* the initial page load event, and JavaScript implementations can differ in how scripts are executed and managed; having the ability to ensure that we were using the same JavaScript engine that is used by a significant portion of end users meant we could be certain behavior would be consistent. This implementation of JavaScript support mitigates the potential for content to load out of order, or even not at all.

Additionally, by working through an actual browser we would have the same user-agent string that an actual end-user would have. Websites can issue different content based on the user-agent string, for example many websites have a mobile version of the site with modified content automatically loaded if a particular browser or environment is detected through the user-agent field. By using a standard desktop version of Firefox, we are able to ensure that the experience should be standard for each website as users would see it.

A final advantage lent by Selenium is that it allowed us to interact with pages by manipulating the document object model (DOM) of a webpage, enabling us to search for specific page elements and invoking events with them, rather than interacting directly with a human user. This ensures consistency and saves an enormous amount of time, as well as making the program more tolerant to changes in site layouts and running environments.

The way that Selenium is integrated into our project is shown in Figure 4. The two components of Selenium, the API and the browser plugin, are used to transfer commands from our data collection program to the Web browser. The plugin is installed and integrated into the browser, and the API is included in the data collection tool. This allows commands to be sent from the program to the plugin via the API, which then converts these general API calls to browser-specific commands in order to navigate with that particular browser.

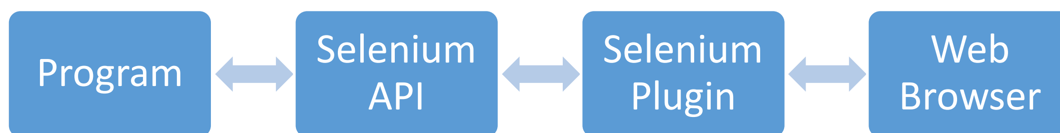

Figure 4 - Interaction process for Selenium. The program makes calls to the Selenium API, which uses a plugin to communicate with the Web Browser.

4.2 Proxies

It was clear from the start that we needed a reliable way to monitor and record HTTP transactions. Selenium offers simple manipulation of HTTP transactions, with abilities to create and filter HTTP requests and responses, however it does not offer the ability to save these transaction data. With this limitation in mind we agreed that the ideal way to save HTTP content is to set up an interceptor (man-in-the-middle) proxy server to route traffic from the browser to the Internet, a process that can be seen in Figure 5.

Figure 5 - Interaction with the proxy server. HTTP requests are made through the web browser, redirected to the proxy, recorded, and forwarded to the Internet. Responses from the internet are directed to the proxy, recorded, and forwarded to the browser.

We had decided that we wanted to use a proxy server that fulfilled the following requirements:

1. It must have an API available to control the proxy. This was essential, as our goal was to automate the whole process. Being able to control the proxy and save sessions and content programmatically is key to that goal.
2. The proxy would have to have a stable release or be under active development. In our research we found that there were many “dead” proxy projects. Although these packages may have suited our needs, it was decided that a project that was under development was better suited for long-term support in the case of compatibility issues in the future.
3. Traffic must not only be able to be viewed, but saved for future analysis. Many proxies allow users to view certain “sessions” when in use, but do not offer a standardized format to save the data. We decided it was necessary to have this feature

We spent a considerable amount of time researching, experimenting, and testing a wide variety of proxy servers, of which a complete list is available in the Appendix of this report. A smaller subset of proxies that we spent a great deal of time with (our final candidates) is what follows.

4.2.1 Fiddler

Fiddler[3] is a Windows-based web debugging proxy that we strongly considered. The proxy was already a known commodity, as it was in use by Professor Wills for data collection before the project even began. It offered an easy-to-use interface and had been reliable in previous data collections.

Fiddler would have been more complicated to extend as it required writing .NET 4 code, which neither of the authors is familiar with. An API was available, however it was packaged in a separate open-source project called FiddlerCore[4]. Also, Fiddler would require us to use a Windows system to run our tests. This requirement was the major factor in deciding not to use Fiddler; it would complicate our platform, as we had wanted to be able to create a virtual machine for testing. Using Windows could create licensing issues with distribution or copying of the machine, so after careful consideration Fiddler was rejected.

4.2.2 BrowserMob-Proxy

BrowserMob-Proxy[5] was the second proxy we spent major time researching and testing. It is an extension of the Selenium project and includes native support for many Selenium functions, which would make it ideal for us to use in conjunction with the Selenium WebDriver.

We had begun writing code to use and test this proxy, and it initial results were promising, so we had begun development on automation with the tool. However, as we wrote further, we began discovering serious bugs and missing features from the tool that were not immediately obvious. For example, at the time of testing there was no native support for saving sessions, though the function existed as a function stub in the source code.

---

[3] http://www.fiddler2.com/
[4] http://fiddler.wikidot.com/fiddlercore
[5] http://opensource.webmetrics.com/browsermob-proxy/

Given the infancy of the project and further anticipated issues with the proxy, we decided it was better to abandon the project and start fresh with a new proxy, rather than try to sort out the issues with BrowserMob.

4.2.3 OWASP Zed Attack Proxy (ZAP)

The third major proxy that we devoted significant development time to is the open-source Zed Attack Proxy[6] from The Open Web Application Security Project (OWASP)[7]. ZAP is designed with website security in mind, and is actually a proxy designed to act as a developer tool to test a website for potential vulnerabilities. This includes modifying packets, crawling through websites, and other security-related testing tasks. The functionality that we are interested in is simply the ability to record HTTP traffic, so we had disabled the ancillary functions for our testing purposes.

ZAP, although certainly not without its faults, admirably performed the tasks we laid out in our criteria. It is an open-source project implemented in pure Java, so modifications and examinations of the source proved simple if needed, and the API provided allowed us to programmatically save HTTP content in a Hyper SQL[8] database.

ZAP has been consistently updated during the development of this project, and has already had several major issues solved in recent releases. In the minds of the author ZAP is likely to continue improving and seems to be on track to have consistent, stable releases that will be easy to update the project to utilize.

For these reasons, we decided to use ZAP as our man-in-the-middle proxy, and it is the platform upon which our project is built.

[6] https://www.owasp.org/index.php/ZAP
[7] http://OWASP.org/
[8] Information on HyperSQL Database is available at http://hsqldb.org/

4.3 Summary

Figure 6 shows the entire data collection process. Commands are processed by the program, send to Firefox via the Selenium API, and traffic is routed through ZAP to get to reach the Internet. This process allows both programmatic interaction with the Web browser via the Selenium plugin and API and saving of HTTP transaction data with the Zed Attack Proxy (ZAP).

Figure 6 - The basic representation of the flow of commands when using the data collection tool.

5 Implementation & System in Action

Designing this project was an iterative process. As we went, we decided to focus on several features each week and thoroughly implement and test each feature. There were early iterations of the code that are completely excluded from the current release. File processing was initially done with Python scripting, and was switched to a much more robust and stable Java project several weeks into development. Here we will go into more detail on specific implementation details for the current release.

5.1 File Formats

It was necessary to define several different types of files in order to process data and produce output that was usable by Professor Wills. This includes created an input file format and simple scripting commands, as well as using formats for output that are consistent with previous data collection methods that were employed during manual data collection.

5.1.1 URL-Scripts

There are two main input formats that we support: a simple list of URLs, each of which will be visited by the Web browser; and a modified list of URLs, with support for adding additional navigation options and commands.

To allow for batch processing, the data collection program takes an input file of sites to visit. The file consist of the URLs to visit, separated by newline characters. This is the simplest form of input, and is generally used for large data sets (lists of several hundred URLs). A portion of one of these files can be seen in Figure 7.

```
http://000webhost.com
http://123rf.com
http://192.com
http://1up.com
http://24hourfitness.com
http://3fatchicks.com
http://4shared.com
http://4square.net
http://5min.com
http://6pm.com
http://888.com
http://aa.com
http://aafp.org
http://aarp.org
http://abc.go.com
http://abc.net.au
http://abcnews.go.com
http://abcnews.go.com/Technology/
http://abebooks.com
http://about.com
http://abovetopsecret.com
http://accorhotels.com
http://accuweather.com
http://acs.org
http://addictinggames.com
http://adobe.com
http://adpost.com
http://adwords.google.com
http://ae.com
http://afl.com.au
http://agoda.com
http://airberlin.com
http://aircanada.com
http://airliners.net
http://airtran.com
http://ajc.com
http://alarabiya.net
http://alaskaair.com
```

Figure 7 - A small subset of a sample URL input list.

In order to support advertising-inducing actions, such as logging onto websites, we designed a simple scripting language to use in conjunction with the input file. A short example of what this script format looks like is visible in Figure 8. Actions to be taken on a site are specified by indenting with a tab on the lines following a URL. The data collector currently supports two commands, *TYPE* and *BUTTON*, though it should be trivial to add additional support as needed, as these commands are primarily wrappers around existing Selenium API options. As seen in Figure 8 these commands are located on tab indented lines after the URL for the site the actions should be taken on.

```
https://accounts.google.com/Login
    TYPE    Email   mamqp1
    TYPE    Passwd  webadsmqp
    BUTTON  Sign    in
https://www.facebook.com/
    TYPE    email   mamqp1@gmail.com
    TYPE    pass    webadsmqp
    BUTTON  Log In
https://my.screenname.aol.com/_cqr/login/login.psp?sitedomain=startpage.aol.com&siteState=OrigUrl%3dht
    TYPE    loginId mamqp1
    TYPE    password    webadsmqp
    BUTTON  Sign In
https://register.go.com/global/abcnews/login?rd=true&appRedirect=http://abcnews.go.com/
    TYPE    username    mamqp1
    TYPE    gspw    webadsmqp
    BUTTON  Sign in
https://www.linkedin.com/uas/login?goback=&trk=hb_signin
    TYPE    session_key mamqp1@gmail.com
    TYPE    session_password    webadsmqp
    BUTTON  Sign    In
https://login.monster.com/Login/SignIn?r=http%3A%2F%2Fhome.monster.com%2F&ch=MONS&re=nv_gh_trnsi_%2F
    TYPE    EmailAddress    mamqp1@gmail.com
    TYPE    Password    webadsmqp1
    BUTTON  Sign In
http://www.kongregate.com/
    TYPE    username    mamqp1
    TYPE    password    webadsmqp
    BUTTON  Sign In
http://forums.penny-arcade.com/entry/signin
```

Figure 8 - A simple of example of more complex interaction. This script will navigate to pages and then log in to each.

TYPE allows interaction with text fields on a page. The command is followed by an identifier for the field to be typed in. This identifier is searched for by Selenium on the webpage, first by looking for an element ID of the same name, and then by element name. If neither exists, it will repeat the check once a second for up to thirty seconds to accommodate for AJAX page loading techniques; if both still do not exist than an exception is thrown to be caught by the code and move on to the next command. The final segment of a *TYPE* command is the input text to be entered into the field.

BUTTON allows issuing a JavaScript click event on a button element on the page. It is followed by the identifier the button to click. Again, this identifier is searched for on the page for a button element with that element ID or name. If it is found, a click event is generated for the button element, and if it does not exist an exception is thrown.

Together, these commands enable simple text field interaction with the login forms for websites. *TYPE* can be used to type in the username and password to the text fields, and *BUTTON* for submitting the form.

5.1.2 Header Output Files

The header output file is formatted in a straightforward manner that can be seen in Figure 9. The HTTP request header is inserted directly into the file, a line break is added, and the response header is inserted immediately after. The third item in the screenshot, the HTML response data, is only present in the data-processing mode (**level-of-detail** of 3) and if the data are not null.

```
GET http://000webhost.com/ HTTP/1.1
Host: 000webhost.com
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:14.0) Gecko/20100101 Firefox/14.0.1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Proxy-Connection: keep-alive
Content-length: 0

HTTP/1.1 301 Moved Permanently
Date: Tue, 05 Feb 2013 22:41:56 GMT
Server: Apache
Location: http://www.000webhost.com/
Content-Length: 234
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>301 Moved Permanently</title>
</head><body>
<h1>Moved Permanently</h1>
<p>The document has moved <a href="http://www.000webhost.com/">here</a>.</p>
</body></html>

---------------------------------------------------------- 1
```

Figure 9 - An example entry in the header output file.

5.1.3 Summary Output Files

The summary output file is formatted differently from the header output file. This file format mimics the output format of a previous tool used by Professor Wills during manual data collection in order to ensure compatibility with already existing processing scripts. An example of what this file can look like is shown in Figure 10. The line format is as follows:

Req. No   Start   Finish   Code   Content-Length   Content-Type   URL

```
415 0 0 200 1905 text/html http://cdn.interclick.com/op/9/swp.ic
416 0 0 200 43 image/gif http://ping.chartbeat.net/ping?h=1up.com&p=%2F&u=egrh0j5niuldhtfn&d=1up.com&g=21105&g0=HP.ROS-
ROS_HOMEPG&n=1&f=1&c=0&x=0&y=3864&w=826&j=45&R=1&W=0&I=0&E=0&r=&b=4969&t=n13qce18si0ylh6t&V=8&i=1UP.com%3A%20Video%20Gam
20More&_
417 0 0 200 711 text/html http://cdn.interclick.com/RSltPrc.aspx?o=11&e=15771
418 0 0 200 711 text/html http://cdn.interclick.com/RSltPrc.aspx?o=9&e=15755
419 0 0 200 238 text/html http://d.agkn.com/iframe/1731/!?camid=06272012&plaid=103&uuid=3810672d-ee94-40eb-b9f9-4e096f02
420 0 0 200 70 image/gif http://osmsync.interclick.com/IdSCb.aspx?
provider_id=39397&expiration=1361313733&TTL=1360104193&sigver=0&sig=7f017cbadf38e491ebc9910613b77786e0941ea5
421 0 0 0 0 N/A http://ad.yieldmanager.com
422 0 0 0 0 N/A http://ad.yieldmanager.com/v0
423 0 0 200 70 image/gif http://osmsync.interclick.com/ColDta.aspx
424 0 0 0 0 N/A http://cdn.interclick.com/op
425 0 0 0 0 N/A http://cdn.interclick.com/op/9
426 0 0 200 1150 image/x-icon http://www.1up.com/favicon.ico
427 0 0 0 0 N/A http://cdn.interclick.com/op/11
428 0 0 0 0 N/A http://osmsync.interclick.com
429 0 0 0 0 N/A http://d.agkn.com
430 0 0 0 0 N/A http://d.agkn.com/iframe
431 0 0 0 0 N/A http://d.agkn.com/iframe/1731
432 0 0 200 1150 image/x-icon http://www.1up.com/favicon.ico
433 0 0 200 0 image/gif http://tags.bluekai.com/ids?dest=132&id=3810672d-ee94-40eb-b9f9-4e096f026b15
434 0 0 0 0 N/A http://tags.bluekai.com
summary: http://1up.com/ 3232321 0 435 0

1 0 0 301 0 N/A http://24hourfitness.com/
2 0 0 0 0 N/A http://24hourfitness.com
3 0 0 200 3193 text/javascript http://choices.truste.com/ca?aid=attssl01&pid=mec01&cid=0112mec&js=st_1&sz=300x250&c=te-e
```

Figure 10 - A representative sample from the summary output file.

Req. No is the request index number for the given URL. What this means is that if a user navigates to a URL such as http://google.com/, it will be request number 1 for that given URL. Subsequent requests until the next input URL is reached are numbered sequentially, with the numbering resetting at 1 for the next URL in the input file.

Start and **Finish** are relevant timing metrics, however they are currently unused, so a placeholder value of 0 is entered in this field.

Code (HTTP response code), **Content-Length**, **Content-Type**, and **URL** are all fields of the same name from the request header.

Prior to the beginning of the next segment of input URLs and subsequent requests, a *summary* line is added to the file with relevant statistics for all the requests made due to navigating to a single

URL. A sample summary for all the request sent on behalf of http://1up.com/ is also visible in Figure 10. These summary lines are in the format:

summary: URL total-size total-time total-requests dead-requests

Here, the **summary** item is simply a keyword to denote that this is a summary line. **URL** is the URL that began the traffic, i.e. the entry in the input list. **Total-size** is the sum of all content-lengths for the URL entry, and represents the amount of data being sent to a client as a result of a single initial request. **Total-time** is unused, but would represent timing information regarding how long all the content took to load. **Total-requests** is the sum total of all requests made as a result of navigating to the URL. **Dead-requests** is the count of requests that were made but received no response. This could be due to several factors, including an error with saving the data, or a server that did not respond.

5.2 Data Collection

The data collection module ("WebCrawler") acts as the initial point-of-entry for the system. Figure 11 shows can see where this data collection module fits in with the rest of the project. This is the first program run during the data collection process, which transfers all commands to the Web browser and saves the HTTP transaction data with our interceptor proxy, ZAP.

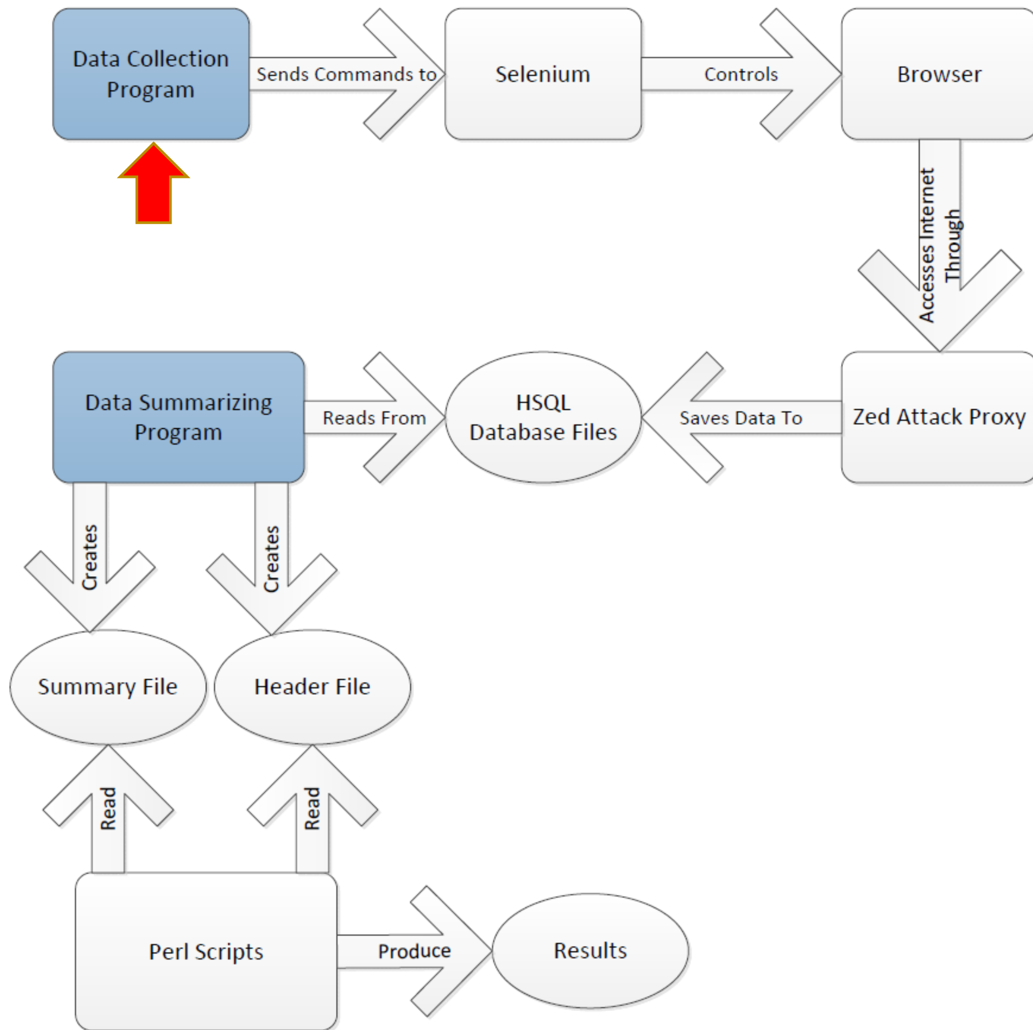

Figure 11 – The data collection module, indicated by the red arrow.

5.2.1 Input

The data collection module is invoked with the following shell command:

webcrawler.jar input-script

Here, the **webcrawler.jar** argument is simply the name of the compiled data collection Java module packaged into an executable JAR file. The **input-script** argument is a valid set of instructions in a URL-Script format, discussed in detail in Chapter 5.1.1.

5.2.2 Processing

Once invoked, the data collection module goes through a series of initialization steps. Due to a memory issue with ZAP (detailed in Chapter 7.2.1) the input URL-Script is segmented in order to process data in several passes. This data processing loop is shown graphically in Figure 12. Beginning at the top-most node on Figure 12, and working clockwise, the process begins with commands being split into segments of 15 URLs (with accompanying interaction details). This is an arbitrary division, however in practice 15 appears to be the average maximum number of sites to guarantee performance and stability from ZAP. In the next step, the module starts ZAP and opens a Firefox session configured to interact through ZAP on port 8080. After a few seconds to allow programs to start up completely, the loop continues and the program passes the list of URLs and commands to a script handler, which interprets the script file and generates Selenium calls to control Firefox. When the script handler has finished will visiting all web sites and performing interactions with each site, ZAP is instructed via an API call to save the current session as a HyperSQL database. Due to an issue with the API (detailed in Chapter 7.2.1) ZAP is then closed via a process kill command, and Firefox is also shut down via a Selenium API call. The processing then begins again from the top, and the next 15 URLs are loaded into the script processor with a new instance of ZAP and Firefox and continues until there are no more instructions.

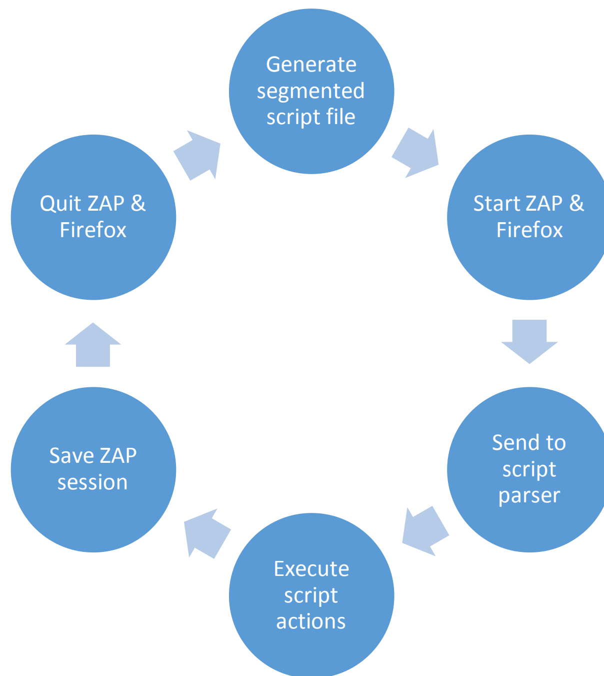

Figure 12 - Data collection loop.

5.2.3 Output

The output for the data collection process is a series of HyperSQL databases, one per every 50 URLs processed, which contain all HTTP transactions in their entirety from the session. These databases are stored on a separate data partition within the virtual machine, in a directory created for the input script given to the program.

5.3 Data Parsing

The data processing module ("HeaderParser") is used to build several text files from a collection of HyperSQL databases. This module is run after the WebCrawler to produce usable output files to process with Perl scripts written and maintained by Professor Wills. Figure 13 shows where this module interacts with the rest of the project. This is the second step, after the creation of the HSQL databases of HTTP transaction data. This program converts these data into a standardized plaintext format to be read by existing Perl scripts.

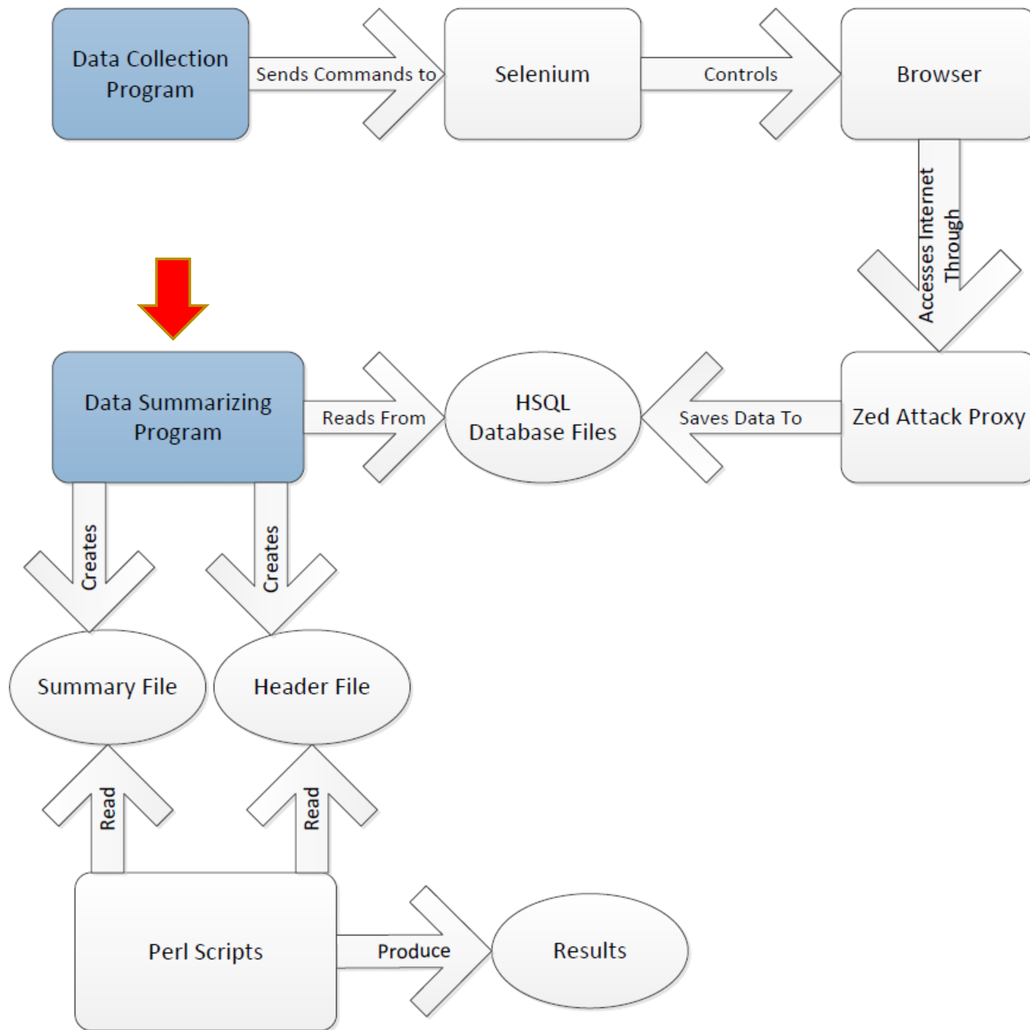

Figure 13 – The data processing module, indicated by the red arrow.

5.3.1 Input

The data processing module is invoked with the following shell command:

HeaderParser.jar input-url-list database-directory summary-output level-of-detail header-output

Here, the **HeaderParser.jar** input is the name of the compiled data processing module. The first argument, **input-url-list**, is a list of the URL's visited during the web crawl. These URLs are used to determine how to segment traffic viewed in the databases. The next argument, **database-directory**, is the location of the HyperSQL databases, as well as the name of the session (by default the title of the

URL-Script file used as input for the web crawl). The next argument, **summary-output**, is the desired output destination of the summary file (detailed in Chapter 5.1.3). This argument is followed by a **level-of-detail** argument. This is an integer between 1 and 3, inclusive. The levels are as follows:

1. Only the summary output file is produced. No HTTP body is processed, only the headers. **header-output** is not required as an argument.
2. The summary output file and a header output file (detailed in Chapter 5.1.2) is also generated at the location specified by **header-output**. HTTP data is not processed for the header file, but headers in their entirety are saved.
3. The summary file and a full header output file, including all data in HTTP bodies, is generated at the location specified by **header-output**.

The final argument, **header-output**, is optional if the **level-of-detail** field is equal to 1. Otherwise, it specifies the output destination for the header text file.

5.3.2 Processing

There are three major loops that make up the file processing program. The first loop is the control loop, shown in Figure 14, which iterates through the databases to process each one. The loop maintains an index of which database it is on, and begins by connecting to the database corresponding to the current index (topmost node in Figure 14). It then triggers the summary file generation loop, which is discussed in the next paragraph. If the **level-of-detail** argument was set to either 2 or 3, it then enters the header output file generation loop. When both the loops terminate, the connection to the database is closed, the index incremented, and processing continues on the next database.

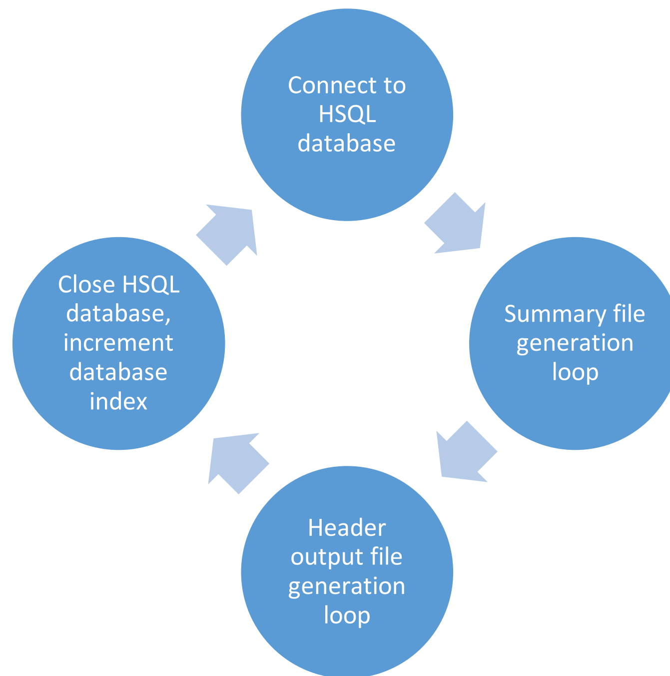

Figure 14 - Main data processing loop.

The summary file is a non-optional output; it will be generated each time the processing program is called. The process for creating this file is shown in Figure 15. At this point, a connection to the current database is already established in the control loop. At the start of the summary generation loop (topmost node in Figure 15) SQL queries are made to the database for the next HTTP transaction. The results are stored temporarily in corresponding variables, and fields with multiple kinds of data, namely the HTTP request and response fields, are matched against regular expressions in order to extract pertinent fields. Checking is done to determine whether the request/response pair is from a new URL in the input list, as we want to separate these data to be able to see which third party requests are made as a result of navigating to each input URL. If it is not a new entry, it is included in the current group and is included in the group statistics. If it is a new entry, then the group statistics are saved to the file and the entry written as a new group. This process repeats for each subsequent request/response pair.

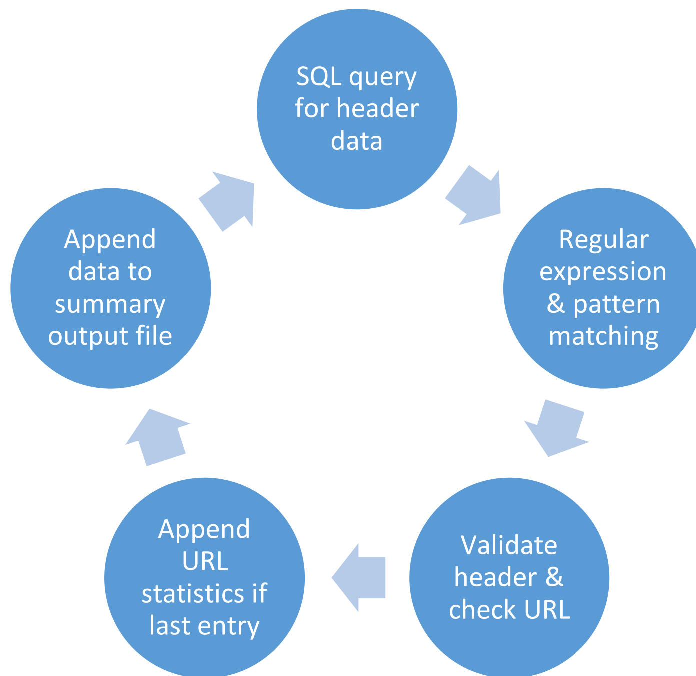

Figure 15 - The summary file generation loop.

The header file is only generated if the **level-of-detail** argument is either 2 or 3. The header file contains all the HTTP transaction data, and optionally the body of the request or response as well. The process is similar to the summary file generation, and is detailed in Figure 16. A database connection is already established in the control loop, and the header output file generation process begins by issuing SQL queries to access the data for the request/response pairs. The relevant fields and values are extracted from the results of these queries and are structured and formatted, and then written to the header data file. If the **level-of-detail** flag is set to 3, data handling is also done at this point. Simple text data is recorded directly into the file, and image data can be processed by OCR technologies or optionally saved. This same process is repeated for each entry.

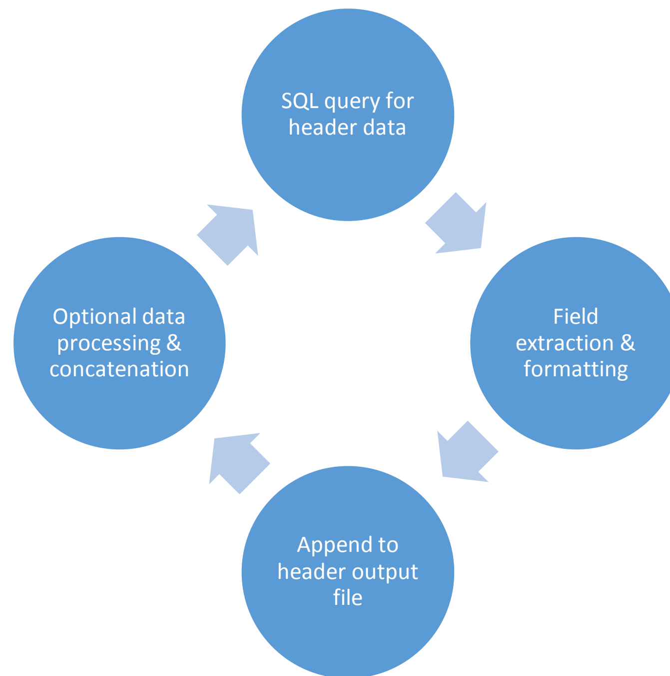

Figure 16 - The header file generation loop.

5.3.3 Output

This program generates a summary file (detailed in Chapter 5.1.3) and optionally a header output file (detailed in Chapter 5.1.2).

5.4 Summary

This project is segmented into two separate, yet related modules: a program designed to gather data from the Web, and a program to process that data. The data collection portion is made up of several pieces of unaffiliated software being coordinated by a Java module we wrote. Custom input is required for this module, and resulting HTTP traffic is saved to a series of HSQL databases. The processing module has several different output modes, all of which are designed to mimic the output of previous software that was used to manually collect data. By ensuring consistency in this output, tools did not have to be updated, and can simply seamlessly replace the human data-gathering.

6 Results

The results of the experiments conducted with our system are in three parts: the collection of raw site data from the web crawl resulting from the list of input URL's and site navigation commands; the results of processing these data with the parsing segment of the tool to produce one or more types of output described in Chapter 5.1.2 and Chapter 5.1.3; and the results of processing those results with the tools controlled by Professor Wills.

6.1 Testing Environment

The results discussed in this section have been produced in a controlled testing environment. A Windows-based research computer specifically designated for this project hosted an Ubuntu v12.04 virtual machine with VirtualBox, onto which our tools were loaded.

The input data used for stress-testing the system were two separate lists each containing approximately one thousand of the most popular websites from 2005 and 2012, respectively, compiled from information available at Alexa.com. No additional site commands were added to this list, and the tool was only directed to load each URL before moving onto the next one.

6.2 Initial Results

Initial tests conducted with this input and setup were consistent with previous results seen by Professor Wills during manual data collection. By consistent, we mean that the data were reasonable as compared to previous collections. Ad providers showed up in approximately the same percentage of sites and examination of a small subset did not demonstrate any worrying inconsistencies, thus indicating that the results of the data collection were valid.

6.3 Implications

With the positive indications from several trials of data collection it is reasonable to assume that the tool we created is viable for collecting large amounts of data to determine trends in advertisements on the Web. This will enable both Professor Wills and future users of the tool to design further experiments and more complicated data processing schemes, as much less manpower is required to collect the information, which would have otherwise been prohibitively time-consuming.

6.4 Summary

Initial testing has shown that our tool provides comparable results to manual data collection. This is important, as this implies that reliable data can be collected and processed with our tools, which will lead to tremendous amounts of time savings for humans involved in the process. The technique will be consistent, controlled, and has the potential to be parallelized to provide data in quantities that previously would not have been possible due to time constraints.

7 Future Work

This project is designed to be on-going. We are providing a platform on which functionality can be enhanced or modified as needed. We tried to organize the source in such a way that it should be clear where new functions can be introduced and existing functionality can be modified or removed. With this in mind, there are several tasks that the authors feel could be performed in the future to further develop this tool.

7.1 Maintenance and Improvements

Though the tools we have developed for this project will work as-is, it should be noted that the project is composed of several "moving parts" which may be necessary to update periodically. During our personal development, ZAP underwent a major update (to version 2.0.0) that corrected many issues and fixed several performance issues. Updates such as this should be monitored and installed for use on the virtual machine hosting the project. This creates opportunities to increase stability, reliability, and functionality.

7.2 Known Issues

As with any substantial project, there are several issues that had to be worked around. The majority of these issues are with interaction with the third-party software we are utilizing, and we will detail ongoing problems in this section.

7.2.1 ZAP

ZAP has presented several major challenges. Many of these have been fixed in recent releases, which we have updated the virtual machine to use, though several key issues remain.

The first major issue is that the ZAP API in practice has been somewhat unreliable. API commands seem to "hang," or fail unexpectedly at times and will not recover gracefully. An example of

this is when sessions are being saved; at times ZAP will save all the content successfully, but *not* terminate its connection to the database, causing an infinite loop.

A related issue with the API is that the *shutdown()* function does not work as intended. It inconsistently gives the program the command to terminate, and became such a problem that we implemented a workaround by using Java process handles so that we could manually terminate the ZAP process and restart it.

A third, major issue is that ZAP is simply not designed to handle as much data as we are giving it to process. ZAP is designed to operate on a single site for security testing, however we are passing a substantial amount of data through this during web crawls. As ZAP holds all these items in memory before being told to save them to a database, the ZAP process will run out of memory very quickly. This is the issue that has caused us to segment data collection in order to prevent ZAP from becoming unreachable.

It should be noted, however, that these issues are known to the ZAP development team and are currently in the bug-fix queue, so care should be taken to monitor the progress of development on ZAP and update the software as needed to possibly remedy these issues.

7.3 Additional Features

As the direction of the project shifted throughout the duration of the project, focus was shifted away from several key features that we believe could be useful.

7.3.1 Website Interaction

A more robust way to interact with websites may be beneficial to have, particularly given the wide variety functional designs on many Web pages on the Internet. Having the ability to search for particular types of elements or identify ad frames to further interact with them could be an important

asset for future research. This level of interaction was not required during our initial testing, though the authors believe it should be pursued.

One of the tools that we had previously looked into for the browser automation portion of this project, Geb[9], has been heavily updated since development began on this project. The authors suggest looking into replacing the Selenium WebDriver portion of this project with Geb, as it will likely make future development much simpler, as Geb currently supports complex operations that are difficult to do with Selenium, such as waiting correctly for asynchronous loading of page content.

7.3.2 Optical Character Recognition (OCR)

A significant amount of time was spent looking into OCR options to use in conjunction with advertising images. Support was added into the data processing code that would take image content, convert it to a .TIFF file (which is standard for most OCR engines available at the time of writing) and running it through Google's open-source Tesseract[10] OCR engine. However, it quickly became apparent that without significant amounts of training data the OCR would not be reliable. Due to the variety and inconsistencies among advertisements, ranging from background images to subtle color changes, it was difficult to get usable character recognition from this process. Thus, the functionality was removed. The authors believe that this functionality would be a tremendous asset to the project, and would like to encourage further research into the technology, though that research is now out of the scope of this project.

7.4 Summary

What we have created is not intended to be a finalized product; there is still work that can, and should, be done to enhance the functionality we have already provided. As better tools and new

[9] http://www.gebish.org/
[10] https://code.google.com/p/tesseract-ocr/

techniques are developed and released to the community, changes should be mirrored and added to the project in order to improve the quality of data we are collecting and processing.

In particular, the authors would like to encourage others to research, or create, an OCR solution that will allow the processing of images, as that would grant the most useful data.

8 Conclusion

Our project will save time for researchers and allow their research to stay current. Researchers who used PageStats can switch to using our programs instead of having to continue using an obsolete browser, switch to manual collection, or find and adjust to another tool. Additionally, because our program can provide outputs in the same format as existing tools there is no need to spend time modifying existing input files or output parsing scripts to work with a new tool. This will enable more time to be spent actually collecting and analyzing data.

Overall our project was successful at realizing its goals. It is capable of large scale data collection and has sufficient site interaction capabilities to log in to most sites on the internet. While the script processer's capabilities could be expanded, they are sufficient for current needs.

We took a couple of major lessons away from our work on this project. One is that even well supported third-party tools can have serious issues, particularly when using a portion of the code base that is less frequently utilized. Zap had errors with both its keeping all the data it had collected in memory and with its API for saving and starting new data collection sessions that required us to kill and restart it on a regular basis during large data collections. Selenium also had an error that would cause erroneous exceptions to be thrown and necessitate switching to a new Selenium WebDriver for future commands. Also, we discovered that the use of JavaScript on webpages has made the problem of determining when a webpage has loaded fairly complex. The JavaScript *onPageLoad()* event often fires before most of the data on the page has loaded, and some pages never stop making requests.

References

Craig E. Wills and Can Tatar. Understanding what they do with what they know. In Proceedings of the Workshop on Privacy in the Electronic Society, Raleigh, NC USA, October 2012.

Dedeo, Scot. "PageStats Extension." . Worcester Polytechnic Institute, n.d. Web. 15 Apr 2013. <http://web.cs.wpi.edu/~cew/pagestats/>.

Greene, Tim. "Microsoft researchers say anonymized data isn't so anonymous." *Network World*. 02 Feb 2012: 1-2. Web. 20 Apr. 2013. <http://www.networkworld.com/news/2012/020212-microsoft-anonymous-255667.html>.

Appendix

List of proxies researched

- Fiddler
- Paros
- BrowserMob
- WebScarab
- MembraneMontitor
- jProxy
- LittleProxy
- Andiparos
- HTTP Debugging Proxy
- Pymi Proxy
- Yacy
- Sahi
- HTTPRipper
- htfilter2
- MaxQ
- mitm-proxy
- Surfboard
- Zee Proxy
- Muffin
- ZAP