E-Mail Header Injections

An Analysis of the World Wide Web

by

Sai Prashanth Chandramouli

A Thesis Presented in Partial Fulfillment
of the Requirement for the Degree
Master of Science

Approved April 2016 by the
Graduate Supervisory Committee:

Dr. Adam Doupe, Chair
Dr. Gail-Joon Ahn
Dr. Ziming Zhao

ARIZONA STATE UNIVERSITY

May 2016

ABSTRACT

E-mail header injection vulnerability is a class of vulnerability that has been around for a long time but has not made its way to popular literature. It can be considered as the email equivalent of HTTP Header Injection Vulnerability. Email injection is possible when the mailing script fails to check for the presence of email headers in the form fields that take in email addresses. The vulnerability exists in the reference implementation of the builtin mail functionality in popular languages like PHP, Java, Python, and Ruby.With the proper injection string, this vulnerability can be exploited to inject additional headers and/or modify existing headers in an E-mail message.

To understand and quantify the prevalence of E-Mail Header Injection vulnerabilities, we used a black-box testing approach, where we crawled 'x' URLs in order to find the URLs which contained form fields. Our system used this data feed to classify the forms which had e-mail fields which could be fuzzed with malicious payloads. Amongst the 's' forms fuzzed, our system was able to find 'y' vulnerable URLs among 'z' domains, which proves that the threat is/isn't widespread and deserves future research attention.

To my mother and father, for giving me the life I dreamt of,

To my sister, who constantly made me do better just to keep up with her,

To my family in Phoenix, for always being there,

To God, for making me so lucky, for letting me be strong when I had nothing, and making me believe when no one else would have.

ACKNOWLEDGEMENTS

A project of this size is never easy to complete without the help and support of other people. I would like to take this opportunity to thank some of them.

This thesis would not have been possible without the help and guidance of my thesis advisor, and committee chair - the brilliant Dr. Adam Doupe. This project was his brainchild, and he held my hand through the entire project. Thank you for everything you did, Adam.

I would like to thank Dr. Gail-Joon Ahn, for being part of the committee, for all his help, and his valuable input on the changes to be made to make the project more impactful.

I would also like to thank Dr. Ziming Zhao, for being a part of my committee, and for the constant motivation.

I would like to thank the members of the SEFCOM, for their support, and would like to especially thank Mike Mabey, for helping set up the infrastructure for this project, and Marthony Taguinod, for helping me document this project.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

The World Wide Web has single handedly brought about a change in the way we use computers. The ubiquitous nature of the Web has made it possible for the general public to access it anywhere, and on multiple devices like Phones, Laptops, Personal Digital Assistants, and even on TVs and cars. This has ushered in an era of responsive web applications which depend on user input. While this rapid pace of development has improved the speed of dissemination of information, it does come at a cost. Attackers have an added incentive to break into user E-Mail accounts more than ever. E-Mail accounts are usually connected to almost all other online accounts of a user, and E-Mails continue to serve as the principal mode of official communication on the web for most institutions. Thus, the impact an attacker can have by taking over just a single E-Mail account of an unsuspecting end user is of a large magnitude.

Since attackers are typically users of the system, if user input is to be trusted, then developers need to have proper sanitization routines in place. Many different injection attacks like the ever popular SQL injection or cross-site scripting (XSS) OWASP (2013) are possible due to improper sanitization of user input.

Our research focuses on a lesser known injection attack known as E-Mail Header Injection. E-Mail header Injection can be considered as the E-Mail equivalent of HTTP Header Injection Vulnerability. The vulnerability exists in the reference implementation of the built-in mail functionality in popular languages like PHP, Java, Python, and Ruby. With the proper injection string, this vulnerability can be exploited to inject additional headers and/or modify existing headers in an E-Mail message.

E-Mail Header Injection attacks have the potential to allow an attacker to perform E-Mail spoofing, resulting in vicious Phishing attacks that can lead to identity theft. The objective of our research is to find if this vulnerability is widespread on the World Wide Web, and if so, how wide the impact is, and whether further research is required in this area.

In order to do this, we performed an expansive crawl of the web, extracting forms that had E-Mail fields in them, and then injecting them with different payloads. We then audited the received emails to see if any of the injected data was present. This allowed us to classify whether a particular URL was vulnerable to the attack. This entire system works in a black-box manner, without looking at the web application's source code, and only analyzing the emails we receive based on the injected payloads.

Structure of document This thesis document is divided logically into the following sections:

- Chapter 2 discusses the background of E-Mail Header Injection, a brief history of the vulnerability, and proceeds onto enumerate the languages and platforms affected by this vulnerability.
- Chapter 3 discusses the System design, and enunciates the architecture and the components of the system, along with a detailed test plan to validate the system. It also enumerates the issues faced, and the assumptions made.
- Chapter 4 briefly describes the experimental setup and sheds light on how we overcame the issues and assumptions discussed in the previous section.
- Chapter 5 presents our findings, and our analysis of the said findings.

- Chapter 6 continues the discussion of the results, the lessons learned over the course of the project, limitations, and a suitable mitigation strategy to overcome the vulnerability.

- Chapter 7 explores related work in the area, and clearly shows how and why our research is different.

- Chapter 8 wraps up the document, with ideas to expand the research in this area.

We hope that our research sheds some light on this relatively less popular vulnerability, and find out its prevalence on the World Wide Web. In summary, we make the following contributions:

- A black-box approach to detecting the presence of E-Mail header injection vulnerability in a web application.

- A detection and classification tool based on the above approach, that will automatically detect such E-Mail Header Injection vulnerabilities in a web application.

- A quantification of the presence of such vulnerabilities on the World Wide Web, based on a expansive crawl across the Web, including 'x' URLs and 'y' forms.

The next chapter goes into the background of the problem at hand, and gives a brief history of E-Mail Header Injection.

Chapter 2

E-MAIL HEADER INJECTION BACKGROUND

2.1 Problem Background

E-Mail Header Injection belongs to a broad class of Vulnerabilities known simply as Injection attacks. However, unlike its more popular siblings, SQL injection (Halfond *et al.* (2006), Boyd and Keromytis (2004), Sadeghian *et al.* (2013)), cross-site scripting (XSS) (Jim *et al.* (2007) Klein (2005)) or even HTTP Header Injection (Johns and Winter (2006)), relatively little research is available on E-Mail Header Injection.

As with other vulnerabilities in this class, E-Mail Header Injection is caused due to improper sanitization (or lack thereof) of user input. If the mailing script fails to check for the presence of E-Mail headers in the form fields that take in user input to send E-Mails, a malicious user, using a well-crafted payload, can control the headers set for this particular E-Mail. Suffice it to say, that this can be leveraged to do a host of malicious attacks, including, but not limited to, spoofing, phishing, etc.

2.2 History of E-Mail Injection

E-Mail Header Injection seems to have been first documented over a decade ago, in a late 2004 Article on phpsecure.info (Tobozo (2004)) accredited to user tobozo@phpsecure.info describing how this vulnerability existed on the reference implementation of the mail function in PHP, and how it can be exploited. More recently, a blog post by Damon Kohler (Kohler (2008)) and an accompanying wiki article (Email Injection - Secure PHP Wiki (2010)) describe the attack vector, and outline a few defense measures for the same.

Since this vulnerability was initially found in the *mail()* function of PHP, E-Mail Header Injection can be traced to as early as early 2000's, present in the *mail()* implementation of PHP 4.0.

The vulnerability was also described very briefly (less than a page) by Stuttard and Pinto in their widely acclaimed book, "*The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaw*" (Stuttard and Pinto (2011)). A concise timeline of the vulnerability is presented in Table 2.1.

2.3 Languages Affected

This section describes the popular languages which exhibit this type of vulnerability. This section is not intended as a complete reference of vulnerable functions and methods, but rather as a guide that specifies which parts of the language are known to have the vulnerability.

- PHP was one of the first languages found to have this vulnerability in its implementation of the *mail()* function. The early finding of this vulnerability can be attributed in part to the success and popularity of the language for creating web pages. According to W3techs (2016), PHP is used by 81.9% of all the websites in existence, thereby creating the possibility of this vulnerability to be widespread.

  PHP's low barrier to entry, and lack of developer education about the existence of this vulnerability have contributed to the vulnerability continuing to exist in the language. It is to be noted that after 13 further iterations of the language since the 4.0 release (the current version is 7.1), the *mail()* function is yet to be fixed. However, it is specified in documentation (PHP-Manual (2016)) that

Year	Notes
Early 2000's	PHP 4.0 gets released, along with support for the mail() function, which has no protection against E-Mail Header Injection.
Jul 04	Next Major version of PHP - Version 5.0 releases
Dec 04	First known article about the vulnerability surfaces on phpsecure.info
2005 - 2007	XSS and SQL steal all the limelight from our poor E-Mail Header Injection.
Oct 07	The vulnerability makes its way into a text by Stuttard and Pinto.
Dec 08	Blog post and accompanying wiki about the header injection attack in detail with examples.
Apr 09	Bug filed about email.header package to fix the issue on Python Bug Tracker
Jan 11	Bug fix for Python 3.1, Python 3.2, Python 2.7 for email.header package, backport to older versions not available.
Sep 11	The vulnerability is described with an example in the 2nd edition of the text by Stuttard and Pinto.
Aug 13	Acunetix adds E-Mail Header Injection to the list of vulnerabilties they detect, as part of their Enterprise Web Vulnerability Scanner Software.
May 14	Security Advisory for JavaMail SMTP Header Injection via method setSubject is written by Alexandre Herzog.
Dec 15	PHP 7 releases, mail function still unpatched.

Table 2.1: A brief history of E-Mail Header Injection

the *mail()* function does not protect against this vulnerability. A working code sample of the vulnerability, written in PHP 5.6 (latest well supported version), is shown in 2.1

```
$from = $_REQUEST['email'];
$subject = "Hello Sai Pc";
$message = "We need you to reset your password";
$to = "schand31@asu.edu";

// attack string => 'sai@sai.com\nBCC:spc@spc.com'
$retValue = mail($to, $subject, $message, "From: $from");
// E-Mail gets sent to both sai@sai.com AND spc@spc.com
```

Listing 2.1: E-Mail Header Injection - PHP

- Python - A bug was filed about the vulnerability in Python's implementation of the *email.header* library and its header parsing functions allowing newlines in early 2009, which was followed up with a partial patch in early 2011.

  Unfortunately, the bug fix was only for the *email.header* package, and thus is still prevalent in other frequently used packages like *email.parser*, where both the classic *Parser()* and the 'new and improved' *FeedParser()* exhibit the vulnerability even in the latest versions - *2.7.11* and *3.5*. The bug fix was also not backported to older versions of Python. There is no mention of the vulnerability in the Python Documentation for either Library. A working code sample of the vulnerability, written in Python 2.7.11, is shown in 2.2

```
from email.parser import Parser
to = input()
msg = """To: """ + to + """\n
From: <user@example.com>\n
Subject: Test message\n\n
```

```
Body would go here\n"""

f = FeedParser() # Parser.parsestr() also
# contains the same vulnerability
f.feed(msg)
headers = FeedParser.close(f)

# attack string => 'sai@sai.com\nBCC:spc@spc.com'

# both to:sai@sai.com AND bcc:spc@spc.com
# are added to the headers
print 'To: %s' % headers['to']
print 'BCC: %s' % headers['bcc']
```

Listing 2.2: E-Mail Header Injection - Python

- Java seems to have been the latest 'big' language to have a bug report about E-Mail Header Injection filed against its JavaMail API. A detailed write-up by Alexandre Herzog is available at Herzog (2014), complete with a proof of concept program that exploits the API to inject headers.

- Ruby - From our preliminary testing, Ruby's builtin Net::SMTP Library has this vulnerability. This is not documented in the Library's home page. A working code sample of the vulnerability, written in Ruby 2.0.0 (the latest stable version), is shown in 2.3

```
require 'sinatra'
require 'net/smtp'

post '/hello' do
greeting = params[:greeting]
```

```
email = params[:email]

message = """
From: Sai <schand31@asu.edu>
To: #{email}
Subject: SMTP e-mail test

This is a test e-mail message.
"""
# construct a post request with email set to attack_string
# attack_string => sai@sai.com%0abcc:spc@spc.com
Net::SMTP.start('localhost', 1025) do |smtp|
smtp.send_message message, 'schand31@asu.edu',
'to@todomain.com'
end
# E-Mail gets sent to both sai@sai.com AND spc@spc.com
end
```

Listing 2.3: E-Mail Header Injection - Ruby

2.4 Potential Impact

The impact of the vulnerability can be pretty far-reaching. Table 2.2 shows the current Server side language usage statistics on the Web, compiled from W3techs (2016). PHP, Java, Python and Ruby (combined) account for over 85% [1] of the websites in existence. The vulnerability can be exploited to potentially do any of the following:

- Phishing and Spoofing Attacks

[1] Note: a website may use more than one server-side programming language

- Denial of service by attacking the underlying mail server
- Spam Networks

It is clear that if proper validation for E-Mail is not performed by these sites, this can quickly escalate to a huge issue.

Server Side Language	% of Usage
PHP	81.9
ASP.NET	15.8
Java	3.1
Ruby	0.6
Perl	0.5
JavaScript	0.2
Python	0.2

Table 2.2: Language Usage Statistics compiled from W3Techs

The next chapter looks at the System Design, and goes in depth into the architecture of our system.

Chapter 3

SYSTEM DESIGN

3.1 Our Approach to the Problem

We took a black-box approach to find out the prevalence of this vulnerability on the World Wide Web. According to Wikipedia (2016a):

> Black-box testing is a method of software testing that examines the functionality of an application without peering into its internal structures or workings.

Since we did not have the source code for each of these websites (and even if we did, the sheer number of websites would have made it a *very* tall task), black-box testing was the ideal approach for this project. Black-box testing gave us the freedom to not worry about the underlying code for the website under test, letting us concentrate on the payload instead.

3.2 System Architecture

The black-box testing system can be divided broadly into two modules:

1. Data Gathering

   The Data Gathering module (shown in Fig. 3.1) is primarily responsible for the following activities:

   - Interface with the UCSB Crawler (Section 3.3.1) and receive the URLs.
   - Parse the HTML for the corresponding URL, and store the relevant form data (Section 3.3.2).

- Check for the presence of forms that allow the user to send/receive E-Mail, and store references to these forms (Section 3.3.3).

2. Payload Injection

   The Payload Injection module (shown in Fig. 3.2) is primarily responsible for the following activities:

   - Retrieve the forms that allow users of a website to send/receive E-Mail, and reconstruct these forms (Section 3.3.4).
   - Inject these forms with benign data (non-malicious payloads), generate a HTTP request to the corresponding URL (Section 3.3.5).
   - Analyze the E-Mails, extracting the header fields, and checking for the presence of the injected payloads (Section 3.3.6).
   - Inject the forms that sent us E-Mails with malicious payloads, generate a HTTP request to the corresponding URL to check if E-Mail Header Injection vulnerability exists in that form (Section 3.3.5).

   The functionality of each component is discussed further in the 'Components' section (3.3). It is to be noted that the Payload Injection pipeline is not linear, but is a cyclic process.

3.3 System Components

This section expands on the brief overview given in the previous section 3.2, describing in detail the functionality of each of the components:

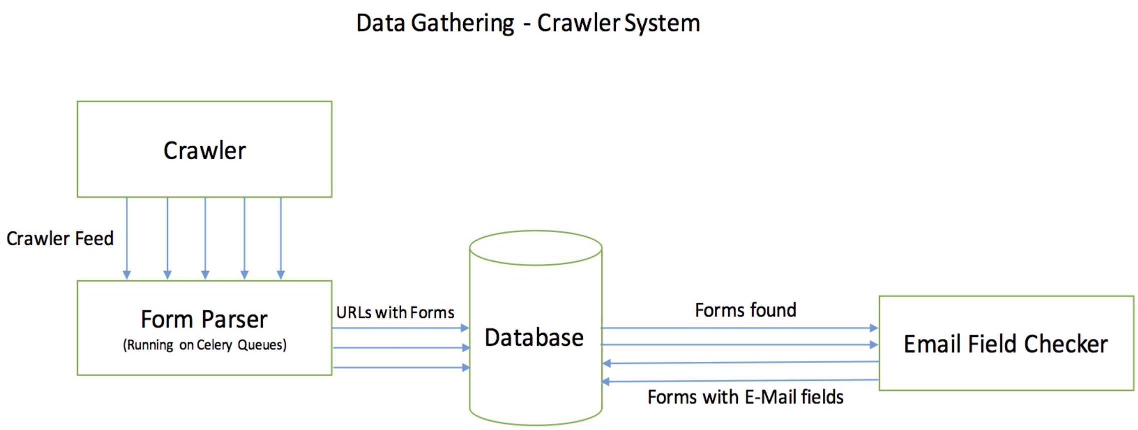

Figure 3.1: System Architecture - Crawler

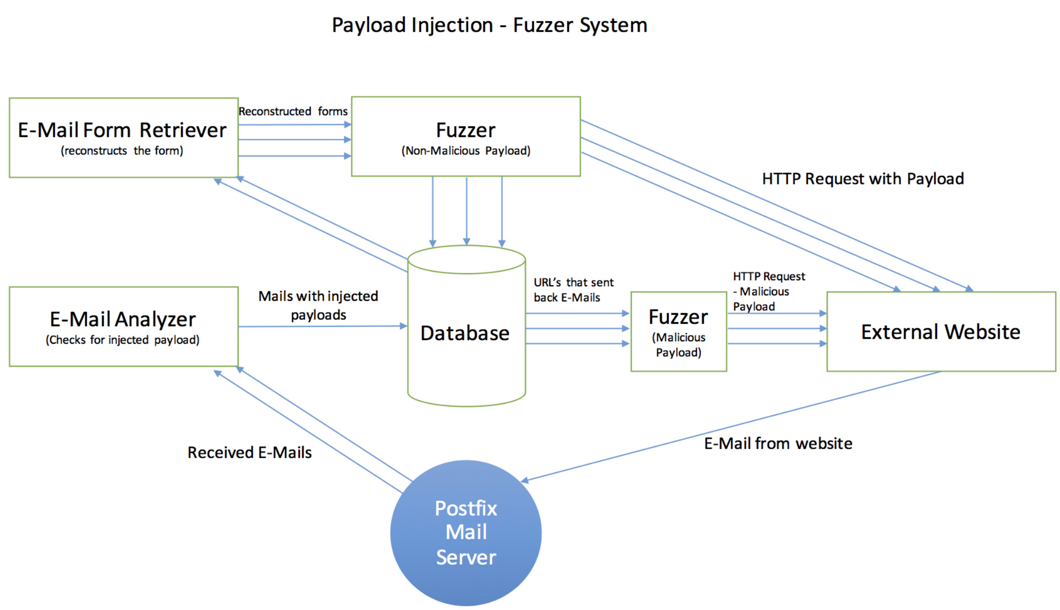

Figure 3.2: System Architecture - Fuzzer & E-Mail Analyzer

3.3.1 Crawler

We used an open-source Crawler built at University of California - Santa Barbara. The Crawler provides us with a continuous feed of URLs and the HTML contained in those pages. This feed is tunneled to our Form Parser over a Celery Queue.

3.3.2 Form Parser

The actual pipeline begins at the Form Parser. This module is responsible for parsing the HTML and retrieving data about the forms on the page, including the following:

- Form attributes, such as method, action, etc. These dictate where we send the HTTP Request, and what kind of request it is (GET or POST).

- Data about the input fields, such as their attributes, names, and default values. The default values are essential for fields like `<input type="hidden">` as these fields are usually used to check for the submission of forms by bots.

- Presence of the `<base>` element, as this affects the final URL to which the form is to be submitted.

- Headers associated with the page, such as *referrer*. Once again, these were required to avoid the website from ignoring our system as a bot.

The Form Parser stores all this data in our Databases, so as to allow us to reconstruct the forms later for fuzzing, as needed.

3.3.3 E-Mail Field Checker

The E-Mail Field Checker script is the final stage in the 'Data Gathering' pipeline. It receives the output of the previous stage - Form Data from the queue, and checks for the presence of E-Mail fields in those forms. If any E-Mail fields are found, it stores references to these forms in a separate table. This allows us to separate the forms that are potentially vulnerable from the forms that are not.

The E-Mail Field Checker particularly searches for the word 'e-mail' or 'email' within the form, instead of an explicit email field (ie) `<input type="email">`. This is by design, taking into account a very common design pattern used by web designers, where they may have a text field with a label called email, instead of an actual E-Mail field, for purposes of backward compatibility with older browsers. Compared to searching for explicit E-Mail fields, by searching for the presence of the word 'e-mail' or 'email' in the form, we are assured zero false negatives, as our system is bound to find an E-Mail field if it is present. However, this might lead to a low false positive rate. We discuss this possibility in Section 3.5 - Design Issues.

The output of this stage is stored in the Database for persistence, and acts as the input to the 'Payload Injection' pipeline.

3.3.4 E-Mail Form Retriever

The E-Mail Form Retriever is the first stage in the Payload Injection Pipeline. It takes care of the following three important functions:

- Retrieve the newly inserted forms in the 'email_forms' table, checking to ensure no duplication occurs before the fuzzing stage.
- Reconstruct each form, using the data stored in the 'form' table, complete with input fields and their values.
- Construct the URL for the 'action' attribute of the form, so that we can send the HTTP Request to the right URL.

3.3.5 Fuzzer

The Fuzzer forms the heart of the system, and is the only component that interacts directly with the external websites. The Fuzzer is not just one monolithic fuzzing system, but is split into smaller modules each of which is responsible for a certain type of fuzzing. We inject payloads in two different stages, so as to improve the efficiency, and reduce the total number of HTTP Requests we generate. This is because generating HTTP Requests is a very expensive process and is usually the cause of bottlenecks in a Crawler-Fuzzer system. The two different types of payloads we use for fuzzing are,

Non-Malicious Payload The regular or non-malicious payload is a straight forward E-Mail address of the format – 'reguser(xxxxx)@wackopicko.com', where 'xxxxx' is replaced by the 'form_id', so as to create a one-to-one mapping of the payloads to the forms. This non-malicious payload allows us to check whether we are able to inject data into a form, and whether we are able to overcome the 'anti-bot' measures on the given website.

Malicious Payload In the malicious payload scenario, we inject the fields with the 'bcc' - blind carbon copy element. If the vulnerability is present, this will cause the server to send us a copy of the E-Mail to the E-Mail address we added as part of the 'bcc' field. The malicious payloads consist of 4 different payloads. Each of these payloads are crafted for a particular use case. The 4 payloads are:

1. `nuser(xxxx)@wackopicko.com\nbcc:maluser(xxxx)@wackopicko.com` - This is the most minimal payload, it injects a 'newline' character followed by the 'bcc' field.

2. `nuser(xxxx)@wackopicko.com\r\nbcc:maluser(xxxx)@wackopicko.com` - This payload is

added for purposes of cross-platform fuzzing (ie) '\r\n' is the 'Carriage Return - New Line (CRLF)' used on Windows systems.

3. `nuser(xxxx)@wackopicko.com\nbcc:maluser(xxxx)@wackopicko.com\nx−check:in` - The addition of the 'x-check:in' header field to the payload is due to Python's exhibited behavior when attaching headers. Instead of overwriting a header if it is present, it ignores duplicate headers. So, in case the 'bcc' field is already present as part of the headers, our injected 'bcc' header would be ignored. In order to overcome this, we need to inject a new header that has not been seen before. Hence, we inject our own 'x-dummy-header' to ensure we can get results if the injection was successful.

4. `nuser(xxxx)@wackopicko.com\r\nbcc:maluser(xxxx)@wackopicko.com\r\nx−check:in` - Same as previous payload, but containing the additional '\r' for Windows compatibility.

The 'xxxx' in the above payloads is replaced by the 'form_id', so as to create a one-to-one mapping of the payloads to the forms. The coverage provided by each Payload is shown in Fig. 3.1.

Payload	Languages covered	Platforms covered
1	PHP, Java, Ruby, etc.	Unix
2	PHP, Java, Ruby, etc.	Windows
3	Python	Unix
4	Python	Windows

Table 3.1: Payload Coverage

Along with the payload, the Fuzzer also has to inject data into the other fields of the form. This data also has to pass validation constraints on the individual input fields e.g. for a name field, numbers might not be allowed. It is essential that the data we inject into the input fields adhere to the constraints. Our Fuzzer does this by making use of a 'Data Dictionary' which has predefined 'keys' and 'values' for common input fields such as name, date, username, password, text, etc. The default values for these are generated on-the-fly for each form, based on generally followed guidelines for such fields. e.g. password fields should consist of at least one uppercase letter, one lowercase letter, and a special character.

Once the data (including the payload) for the form is ready, the Fuzzer constructs the appropriate HTTP Request (GET or POST), and sends the HTTP Request to the URL that was generated by the E-Mail Form Retriever (Section. 3.3.4).

3.3.6 E-Mail Analyzer

The E-Mail Analyzer checks for the presence of injected data in the received E-Mails. This module works on the E-Mails received and stored by our Postfix server, and depending on the user who received the E-Mail, it performs different functions. This is outlined below:

Analyzing Regular E-Mail 'Regular E-Mail' refers to the E-Mails received by the `reguser(xxxx)@wackopicko.com` — where xxxx is the form_id — that were sent due to injecting the 'regular or non-malicious' payload (discussed in Section. 3.3.5). The objective of the analysis on this E-Mail is to figure out whether the input fields that we injected with data appear on the resulting E-Mail, and if so, which fields appear where.

In order to find this, we read through each received E-Mail, and check whether *any* of the fields we injected with data appear as part of either the headers or the body of the E-Mail. If they do, we add them to the list of fields that can potentially result in a E-Mail Header Injection for the given E-Mail. We then pass on this information back to the Fuzzer pipeline, along with the form_id, so that the Fuzzer can now inject the malicious payloads into the same form, completing the pipeline.

Analyzing E-Mail with payloads 'E-Mail with payloads' refer to E-Mails received by either the `nuser(xxxx)@wackopicko.com` or `maluser(xxxx)@wackopicko.com` accounts. These E-Mails were received due to injecting the malicious payloads that were discussed in Section. 3.3.5. Analysis of these E-Mails are considerably simpler than that of the regular E-Mails. This is due to the fact that this involves lesser processing of the contents of the E-Mail compared to the previous section.

Detecting injected bcc headers As discussed in the payloads section (3.3.5), the payloads were crafted in such a way that the mails received by 'maluser' account directly indicate the presence of the injected 'bcc' field. Thus, we simply parse the E-Mails and store them in the Database.

Detecting injected x-check headers E-Mails not received by the 'maluser' account but by the 'nuser' account constitute a special category of E-Mails. These E-Mails could have been generated due to two reasons:

1. The websites performed some sanitization routines and stripped out the 'bcc' part of the payload, thereby sending E-Mails only to the 'nuser' account. These E-Mails then act as proof that the vulnerability was not found on the given website.

2. A more conducive scenario for us is when the 'bcc' header was ignored for some

reason, e.g. Python's default behavior when it encounters duplicate headers. In this case, we check whether the E-Mail contains the custom header 'x-check'. If it does, then this is a successful attack as well, and we store it in the Database.

3.3.7 Database

We collect and store as much data as possible at each stage of the pipeline. This is due to the two following reasons:

1. The data is used to validate our findings.

2. The data collected can be used for other research projects in this area.

Each table in our database is listed in Table. 3.2 along with the data it is designed to hold. A schema of the database is shown in Figure. 3.3.

3.4 Test Plan

This section will describe the test plan for the project, and will explain what was tested, and how our system conforms to the requirements.

3.5 Design Issues

This section will describe the issues we might face with the approach that we have chosen, and the design decisions.

- False Positive rate for the E-Mail Field Checker

3.6 Assumptions

This discusses the assumptions that we have made while building the system, examples include:

S.No	Table Name	Purpose
1	form	To hold data about all the forms that we get from the Form Parser.
2	email_forms	Holds the output of the E-Mail Field Checker, i.e. references to the ID's of the forms that contain E-Mail fields.
3	params	Holds the actual input fields of the forms, including their default values.
4	fuzzed_forms	Holds the data of the forms that were injected, including the payload used to inject and the URL to which the HTTP Request was delivered.
5	received_emails	Contains data about the E-Mails received for the regular payload, including which injected data fields were present in the E-Mail.
6	successful_attack_emails	Contains data about the E-Mails received for the malicious payload. This contains the end result of the payload injection pipeline.
7	requests	Contains data about the requests generated for each URL.
8	blacklisted_urls	Used for skipping certain websites that may blacklist our Crawler-Fuzzer.

Table 3.2: Database - Tables

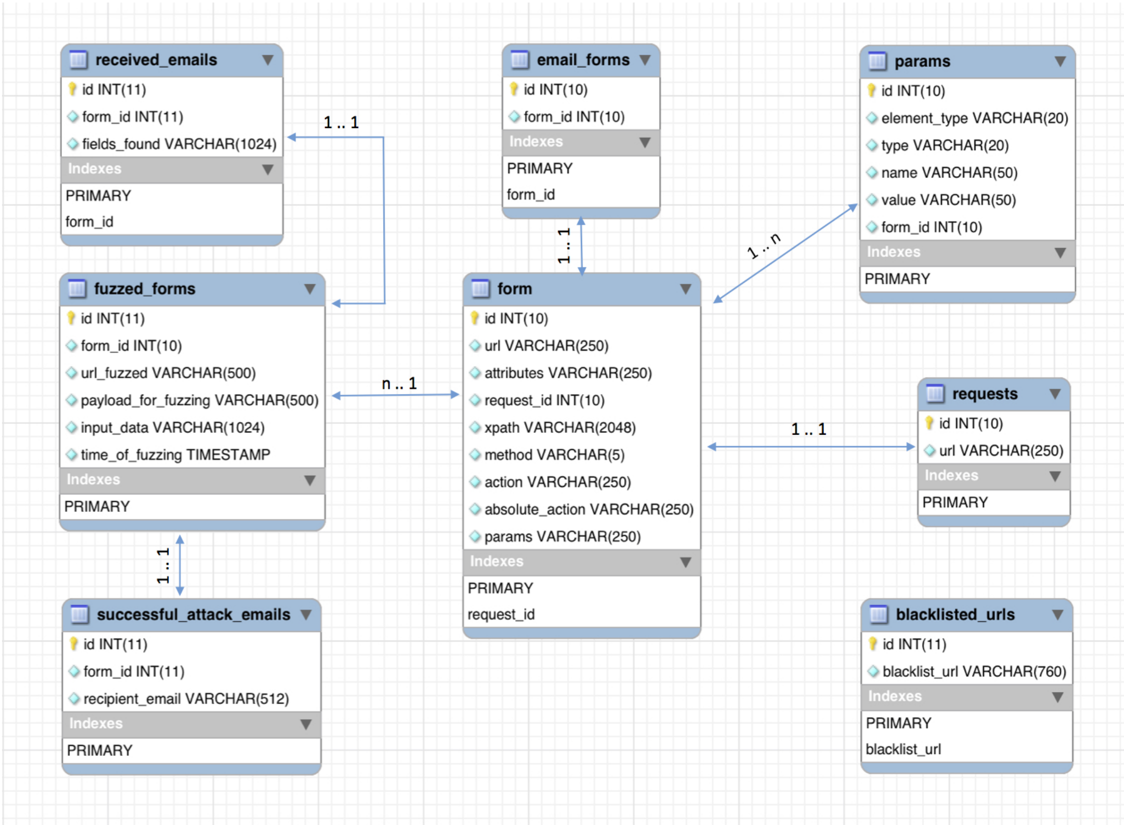

Figure 3.3: Database Schema

1. Crawler is not blocked by the firewalls.

2. The Crawler feed is an ideal representation of the World Wide Web.

Chapter 4

EXPERIMENTAL SETUP

4.1 System Configuration

Will briefly describe the servers used for the experiments.

4.2 Platforms and Software

Will briefly describe the platform, (ie) Ubuntu 14.04, and the softwares that were used for the experiments. (eg) Postfix, Apache, MySQL, etc.

4.3 Languages used

Will very briefly (maybe one paragraph) describe what we used to create the system. (Python 2) Will also describe the limitation of Python, (GIL basically), and point to next section.

4.4 Celery Queues

Will briefly describe how Celery and rabbitMQ help us to overcome the GIL, and do tasks in parallel.

Chapter 5

DATA ANALYSIS AND RESULTS

This section will have tables, images and charts.

5.1 Data

Will display a table/graph with the data, then go on to explain what the fields/-graphs mean.

5.1.1 URLS crawled

5.1.2 Forms collected

5.1.3 Forms with E-Mail Fields

5.1.4 E-Mail received from Forms

5.1.5 Fuzzed Forms

Chapter 6

DISCUSSION

6.1 Lessons Learned

Describes what we learned from this particular project.

6.2 Limitations of the Project

Describes what limitations were present, stuff like:

- CAPTCHAs
- JavaScript Apps
- Blogs powered by WordPress/Drupal
- Mail libraries

6.3 How to prevent this attack

Describes how to prevent this attack, stuff like:

- Use Mail Libraries
- CMS
- Input Validation

Chapter 7

RELATED WORK

This will be a detailed section on the papers that are related to our work, *but* important thing is to show why our work is different from prior work in this area. Also, can/will add references to the blogs and books that describe this attack :)

Chapter 8

CONCLUSION

Conclude with what the results were, whether the vulnerability was widespread or not, and how (if needed) this can be alleviated.

REFERENCES

Beizer, B., *Black-box Testing: Techniques for Functional Testing of Software and Systems* (John Wiley & Sons, Inc., New York, NY, USA, 1995).

Boyd, S. W. and A. D. Keromytis, "Sqlrand: Preventing sql injection attacks", in "Applied Cryptography and Network Security", pp. 292–302 (Springer, 2004).

Calin, B., "Email Header Injection Web Vulnerability - Acunetix", URL `https://www.acunetix.com/blog/articles/email-header-injection-web-vulnerability-detection/` (2013).

Christey, S. and R. A. Martin, "Vulnerability type distributions in cve", Mitre report, May (2007).

Doupé, A., B. Boe, C. Kruegel and G. Vigna, "Fear the EAR : Discovering and Mitigating Execution After Redirect Vulnerabilities", in "Computer and Communications Security (CCS)", CCS '11, pp. 251–261 (ACM Press, 2011).

Email Injection - Secure PHP Wiki, URL `http://securephpwiki.com/index.php/EmailInjection` (2010).

Franklin, J., A. Perrig, V. Paxson and S. Savage, "An inquiry into the nature and causes of the wealth of internet miscreants.", in "ACM conference on Computer and communications security", pp. 375–388 (2007).

Halfond, W. G., J. Viegas and A. Orso, "A classification of sql-injection attacks and countermeasures", in "Proceedings of the IEEE International Symposium on Secure Software Engineering", vol. 1, pp. 13–15 (IEEE, 2006).

Herzog, A., "Full Disclosure: JavaMail SMTP Header Injection via method setSubject [CSNC-2014-001]", URL `http://seclists.org/fulldisclosure/2014/May/81` (2014).

Hodges, J., C. Jackson and A. Barth, "Http strict transport security (hsts)", URL: http://tools. ietf. org/html/draft-ietf-websec-strict-transport-sec-04 (2012).

Jim, T., N. Swamy and M. Hicks, "Defeating script injection attacks with browser-enforced embedded policies", in "Proceedings of the 16th International Conference on World Wide Web", WWW '07, pp. 601–610 (ACM, New York, NY, USA, 2007), URL `http://doi.acm.org/10.1145/1242572.1242654`.

Johns, M. and J. Winter, "Requestrodeo: Client side protection against session riding", in "Proceedings of the OWASP Europe 2006 Conference", (2006).

Klein, A., "[DOM Based Cross Site Scripting or XSS of the Third Kind] Web Security Articles - Web Application Security Consortium", URL `http://www.webappsec.org/projects/articles/071105.shtml` (2005).

Kohler, D., "damonkohler.com: Email Injection", URL http://www.damonkohler.com/2008/12/email-injection.html (2008).

Murray, D., "Email.header.Header too lax with embedded newlines", URL http://bugs.python.org/issue5871 (2009).

OWASP, "OWASP Top Ten Project", URL https://www.owasp.org/index.php/OWASP_Top_10 (2013).

Payet, P., A. Doupé, C. Kruegel and G. Vigna, "EARs in the Wild: Large-Scale Analysis of Execution After Redirect Vulnerabilities", in "Proceedings of the ACM Symposium on Applied Computing (SAC)", (Coimbra, Portugal, 2013).

Phishing, "Phishing — Wikipedia, the free encyclopedia", http://en.wikipedia.org/w/index.php?title=Phishing&oldid=706223617, [Online; accessed 27-February-2016] (2016).

PHP-Manual, "PHP mail - Send mail", URL http://php.net/manual/en/function.mail.php (2016).

Pietraszek, T. and C. V. Berghe, "Defending against injection attacks through context-sensitive string evaluation", in "Recent Advances in Intrusion Detection", pp. 124–145 (Springer, 2005).

Pope, A., "Prevent Contact Form Spam Email Header Injection — Storm Consultancy Web Design Bath", URL https://www.stormconsultancy.co.uk/blog/development/dev-tutorials/secure-your-contact-form-against-spam-email-header-injection/ (2008).

Python, "email - an email and mime handling package", URL https://docs.python.org/2/library/email-examples.html (2016a).

Python, "email.parser: Parsing email messages", URL https://docs.python.org/2/library/email.parser.html (2016b).

Resnick, P. W., "Internet Message Format - RFC 822", URL https://tools.ietf.org/html/rfc2822 (2001).

Resnick, P. W., "Internet Message Format - RFC 5322", URL https://tools.ietf.org/html/rfc5322 (2008).

Ruby, "Ruby mail - Net::SMTP", URL http://ruby-doc.org/stdlib-2.0.0/libdoc/net/smtp/rdoc/Net/SMTP.html (2016).

Sadeghian, A., M. Zamani and A. A. Manaf, "A taxonomy of sql injection detection and prevention techniques", in "Informatics and Creative Multimedia (ICICM), 2013 International Conference on", pp. 53–56 (IEEE, 2013).

Stuttard, D. and M. Pinto, *The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws* (John Wiley & Sons, 2011).

Terada, T., "SMTP Injection via recipient email addresses", MBSD White Paper (2015).

Tobozo, "Mail headers injections with PHP", URL http://www.phpsecure.info/v2/article/MailHeadersInject.en.php (2004).

W3techs, "Usage Statistics and Market Share of PHP for Websites, February 2016", URL http://w3techs.com/technologies/details/pl-php/all/all (2016).

Wikipedia, "Black-box testing — Wikipedia, the free encyclopedia", http://en.wikipedia.org/w/index.php?title=Black-box%20testing&oldid=702083755, [Online; accessed 02-March-2016] (2016a).

Wikipedia, "Newline — Wikipedia, the free encyclopedia", http://en.wikipedia.org/w/index.php?title=Newline&oldid=704759213, [Online; accessed 05-March-2016] (2016b).

Zanero, S., L. Carettoni and M. Zanchetta, "Automatic detection of web application security flaws", Black Hat Briefings (2005).

APPENDIX A