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 Doupé, 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 e-mail equivalent of HTTP Header Injection Vulnerability. Email injection is possible when the mailing script fails to check for the presence of e-mail headers in the form fields that take in e-mail addresses. The vulnerability exists in the reference implementation of the built-in âĂIJmailâĂİ 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.

This thesis serves to understand and quantify the prevalence of E-Mail Header Injection vulnerabilities. Using a black-box testing approach, we crawled 13,467,548 URLs in order to find the URLs which contained form fields. We found 3,725,106 such forms, out of which 581,435 forms contained e-mail fields. Our system used this data feed to classify which of these forms could be fuzzed with malicious payloads. Amongst the 412,623 forms tested, 31,666 forms were fuzzable, and our system was able to find 289 vulnerable URLs across 124 domains, which proves that the threat is 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 Doupé. 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

CHAPTER Page

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 an enormous 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) [27] 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 âĂIJmailâĂİ 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 e-mails 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 e-mails 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 on to 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. 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 shows clearly 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, which 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 an expansive crawl across the Web, including 'x' URLs and 'y' forms.

Chapter 2

E-MAIL HEADER INJECTION BACKGROUND

This chapter goes into the background of the problem at hand and gives a brief history of E-Mail Header Injection. It then describes the languages affected by this vulnerability and discusses the overall impact E-Mail Header Injection can have, and the attacks that can result from this vulnerability.

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 ([5], [14], [40]), cross-site scripting (XSS) ([18], [21]) or even HTTP Header Injection ([19]), 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 ([44]) accredited to user tobozo@phpsecure.info describing how this vulnerability existed in the reference implementation of the mail function in PHP, and how it can be exploited. More recently, a blog post by Damon

Kohler ([22]) and an accompanying wiki article ([12]) 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 the beginning of the 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 Flaws*" ([41]). 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 [45], 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 ([30]) that the *mail()*

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 2004	Next Major version of PHP - Version 5.0 releases
Dec 2004	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 2007	The vulnerability makes its way into a text by Stuttard and Pinto.
Dec 2008	Blog post and accompanying wiki about the header injection attack in detail with examples.
Apr 2009	Bug filed about email.header package to fix the issue on Python Bug Tracker
Jan 2011	Bug fix for Python 3.1, Python 3.2, Python 2.7 for email.header package, backport to older versions not available.
Sep 2011	The vulnerability is described with an example in the 2nd edition of the text by Stuttard and Pinto.
Aug 2013	Acunetix adds E-Mail Header Injection to the list of vulnerabilties they detect, as part of their Enterprise Web Vulnerability Scanner Software.
May 2014	Security Advisory for JavaMail SMTP Header Injection via method setSubject is written by Alexandre Herzog.
Dec 2015	PHP 7 releases, mail function still unpatched.

Table 2.1: A brief history of e-mail header injection.

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 Listing. 2.1.

```php
$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: PHP program with the vulnerability.

- 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 Listing. 2.2.

```python
from email.parser import Parser
import cgi
form = cgi.FieldStorage()
to = form["email"] # input() exhibits the same behavior
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: Python program with the vulnerability.

- 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 [15], complete with a proof of concept program that exploits the API to inject headers.

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

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

```

```
4  get '/hello' do
5  email = params[:email]
6
7  message = """
8  From: Sai <schand31@asu.edu>
9  Subject: SMTP e-mail test
10 To: #{email}
11
12 This is a test e-mail message.
13 """
14 # construct a post request with email set to attack_string
15 # attack_string => sai@sai.com%0abcc:spc@spc.com%0aSubject:Hello
16 Net::SMTP.start('localhost', 1025) do |smtp|
17 smtp.send_message message, 'schand31@asu.edu',
18 'to@todomain.com'
19 end
20 # Headers get added, and Subject field changes to what we set.
21 end
```

Listing 2.3: Ruby program with the vulnerability.

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 [45]. PHP, Java, Python and Ruby (combined) account for over 85% [1] of the websites in existence. The vulnerability can be exploited to do potentially any of the following:

- Phishing and Spoofing Attacks

  Phishing (a variation of spoofing) refers to an attack where the recipient of an E-

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

Mail is made to believe that the E-Mail is a legitimate one. The E-Mail usually redirects them to a malicious website, which then steals their credentials.

E-Mail Header Injection gives attackers the ability to inject arbitrary headers into an E-Mail sent by a website and control the output of the E-Mail. This adds credibility to the generated E-Mail, and can result in more successful phishing attacks.

- Spam Networks

  Spam networks can capitalize on the ability to send a large amount of E-Mail from servers that are trusted. By adding additional 'cc' or 'bcc' headers to the generated E-Mail, attackers can easily achieve this effect.

  Due to the E-Mails being from trusted domains, E-Mail clients might not flag them as 'spam'. If they do flag them as 'spam', then that can lead to the website getting blacklisted as a spam generator. Irrespective of the behavior of spam filters, this does not bode well for the website.

- Information Extraction of legitimate users

  E-Mails can contains sensitive data that is meant to be accessed only by the user. Due to E-Mail Header Injection, an attacker can easily add a 'bcc' header, and send the E-Mail to himself, thereby extracting important information. User Privacy can thus be compromised, and loss of private information can by itself lead to a host of other attacks.

- Denial of service by attacking the underlying mail server

  Denial of service attacks (DoS as they are popularly known), can also be aided by E-Mail Header Injection. The ability to send hundreds of thousands of E-Mails by just injecting one header field can result in overloading the mail server,

and cause crashes and/or instability.

It is evident 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 [45].

Chapter 3

SYSTEM DESIGN

This chapter discusses the System design, and explains the architecture and the components of the system in detail. It then proceeds to enumerate the issues faced, and the assumptions made during the building of the system.

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 [46]:

> "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. A high level logical overview of our system is presented in Figure 3.1.

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 Figure 3.2) is primarily responsible for

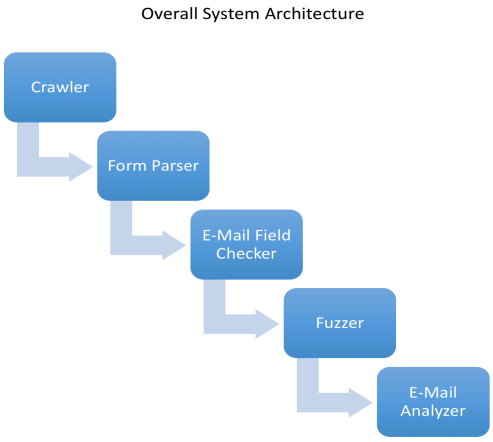

Figure 3.1: Overall system architecture - logical overview.

the following activities:

- Interface with the 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 Figure 3.3) 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), and generate an HTTP request to the corresponding URL (Section 3.3.5).

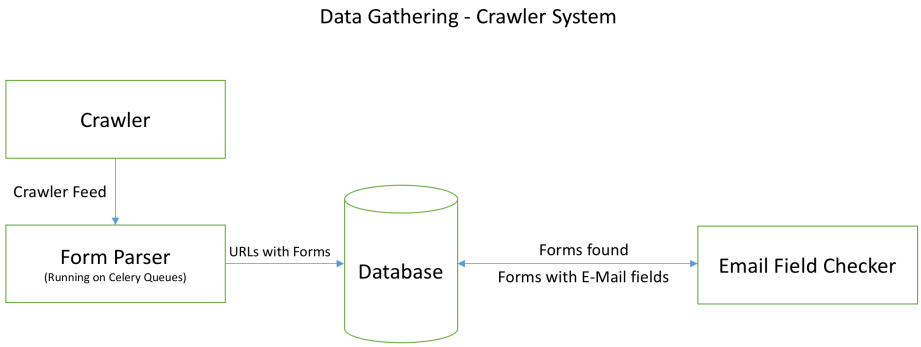

Figure 3.2: System architecture - crawler.

- 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, and generate an 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 a linear, but 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:

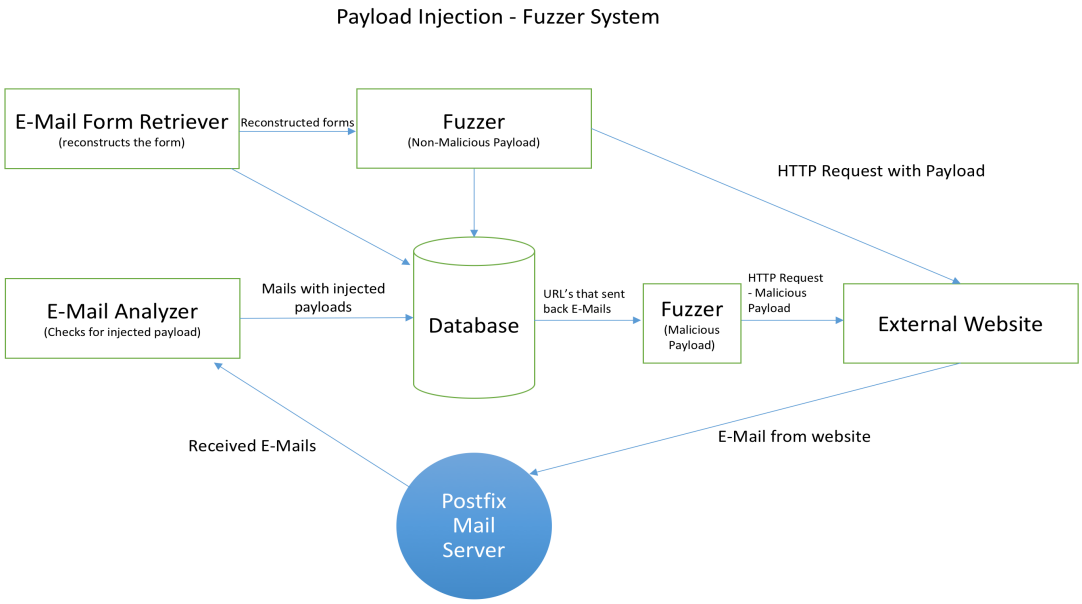

Figure 3.3: System architecture - fuzzer & e-mail analyzer.

3.3.1 Crawler

We used an open-source Apache Nutch based Crawler. 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 words 'e-mail', 'mail' or 'email' within the form, instead of an explicit e-mail 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 an id or name set to '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 words 'e-mail', 'mail' or 'email' in the form, not only are we assured zero false negatives — as our system is bound to find an E-Mail field if it is present — but the system is also substantially faster as we do not have to parse the individual form fields at this point in the pipeline. However, this might lead to a low false positive

rate. We discuss this possibility in Section 3.4 - 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 particular 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 making 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)@example.com', where 'xxxxx' is replaced by the `form_id', so as to create a one-to-one mapping of the payloads to the forms, and 'example.com' is replaced by the required domain. In our case, this domain is 'wackopicko.com'. This non-malicious payload allows us to check whether we can inject data into a form and whether we can 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.

A special case is to be considered here: the addition of the 'x-check:in' header field to the payloads. This is due to Python's exhibited behavior when attaching headers. Instead of overwriting a header if it is already 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.

The malicious payloads consist of 4 different payloads. Each of these payloads is crafted for a particular use case. The four 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` - As discussed above, the addition of the 'x-check- header is to inject Python based websites.

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, each payload covers a different platform/language.

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 standard 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.

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 an 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 is 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 E-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 3.4.

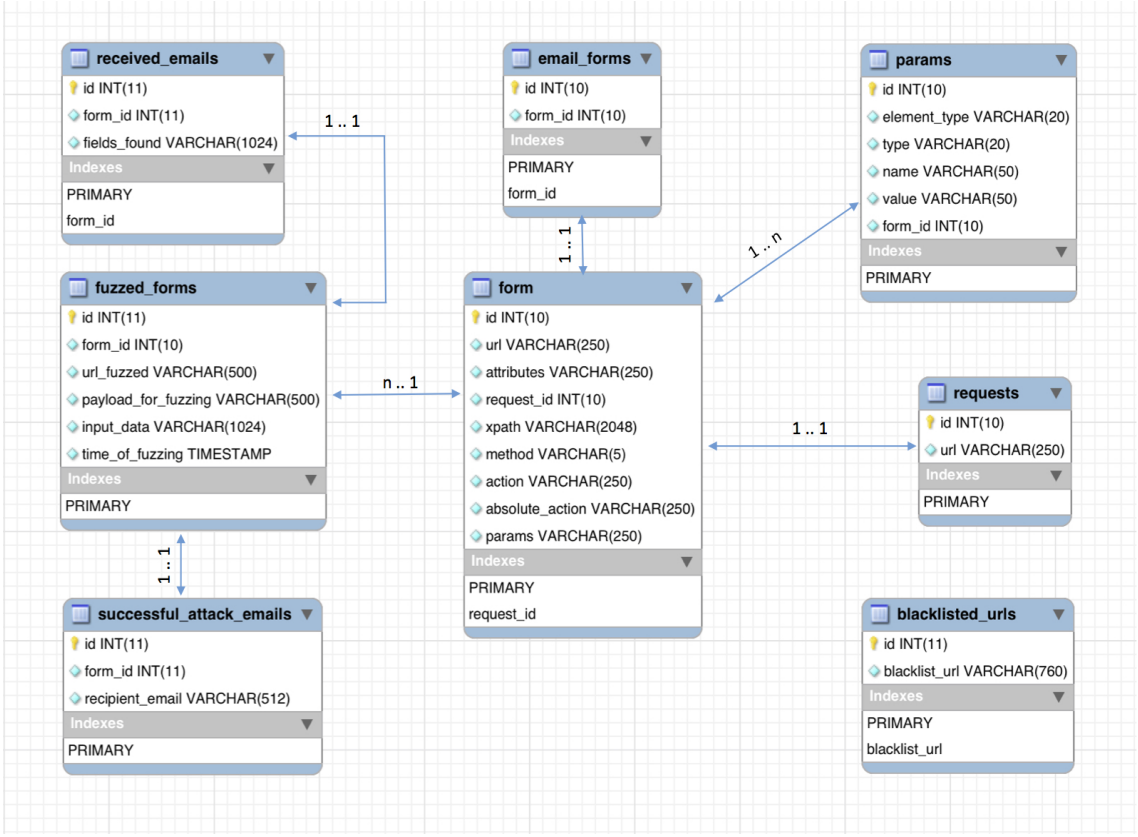

Figure 3.4: Database schema.

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: The different tables in our database.

3.4 Design Issues

This section will describe the issues we faced with the design decisions we made, and how we did our best to mitigate them, and their effect on the system.

- False Positive rate for the E-Mail Field Checker

  As discussed in Section 3.3.3, we only search for the words 'email', 'mail' or 'e-mail' (case insensitive) inside the forms to detect the presence of E-Mail fields, instead of searching for an `<input type = email>`. This might result in a false positive in certain forms, like the one shown in Listing. 3.1.

```
1  <form method="post">
2  E-Mail us if you have any questions!!
3  <input type="text" name="query"><br>
4  <input type="submit" value="Search">
5  </form>
```

Listing 3.1: E-Mail field checker - false positives, the system incorrectly classifies this as an e-mail form.

  The word 'E-Mail' on Line 2 will result in our system classifying this form as a potential E-Mail form, while it clearly is not. However, as we will see, this is not really a big issue, as despite being added to the `email_forms' table, this form will never be injected in the 'fuzzer' due to the absence of the appropriate input field in the form. We chose to go with this design, as it allows us to detect *every* form that provides the capability to send or receive E-Mail, and it lets us do so in a very fast and inexpensive way.

- Parallelism for the system

  Every component in the pipeline benefits hugely from parallel processing of the data. However, Python's GIL (Global Interpreter Lock) does not allow the

running of multiple native threads concurrently. To overcome this, we used a Celery task queue (discussed in Section 4.4), which allowed a fair level of parallelism that vanilla Python does not provide by default. Even though this makes the system faster than a single-threaded approach, it still leaves room for improvement in terms of speed. Despite the obvious speed drop, we chose to go with Python, for the raw power it provides, its text processing capabilities, PCRE (Perl Compatible Regular Expressions) compatibility, and the numerous libraries for HTML Parsing, HTTP request generation, etc.

- URL Construction

  The multiple ways in which a URL is specified (i.e. Relative and Absolute URLs) complicates the construction of the URL from the 'action' attribute of the form. As an example, the following URLs are all equivalent (as parsed by a browser, assuming we are in the path 'www.website.com'):

  - `action=mail.php`
  - `action=./mail.php`
  - `action=http://website.com/mail.php`
  - `action=www.website.com/mail.php`

  Add to this, if the form is a self-referencing form [1] , and is present in mail.php, the following are equivalent to the above URLs as well:

  - `action=""`
  - `action=#`
  - 'action' is completely omitted

[1] A self-referencing form is one which submits the form data to itself. It includes logic to both display the form and process it. It is a *very* common feature in PHP based scripts.

Also, relative URLs pose another problem. If the URL of the form page ends with '/' and the 'action' specifies a path starting with '/' (illustrated in Listing 3.2), we would need to strip one of the two slashes. This increases the overall complexity of our URL generator, as we have to account for all these possibilities.

```
1        Current URL = www.website.com/
2        <form action=/mail.php>
```

Listing 3.2: URL construction, the resulting url needs to be www.website.com/mail.php and not www.website.com//mail.php

We chose to go with a best-effort approach to this problem, where our system covers all these possibilities, however, we cannot know for certain whether this works for other unforeseen ways of specifying a URL.

- Black-box Testing

  The approach that we have selected — Black-box testing — is highly beneficial as explained in Section 3.1. However, it also has a drawback in that we cannot verify whether the reported vulnerability exists in the source code, or is a feature of the website. We have to manually E-Mail the developers to get this kind of feedback.

- Mapping responses to requests

  Since we are generating multiple payloads for each form, and the received E-Mail may not contain the name of the domain from which we received the E-Mail, it is difficult to map the response E-Mails to the right requests. We instead use the `form_id' as part of the payload to reduce the difficulty in mapping responses to requests.

- Bot Blockers

  Since our system is fully automated, it is also susceptible to being stopped by 'bot-blockers' i.e. mechanisms built-in to a website to prevent automated crawls or form submissions. Measures like CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart) and hidden form fields are often used to detect bots. We have made sure that we do not affect hidden fields in the form, however, we do not have an anti-CAPTCHA functionality built into our system, and thus our system does not inject such websites.

- Handling Malformed HTML

  The parser that we use for HTML parsing — Beautiful Soup — does not try to parse malformed HTML, and throws an exception on encountering malformed content. Thus, we have designed the system to exit gracefully on such occasions. A side-effect of this is that our system is unable to parse websites with bad markup [2]

- Crawling WordPress and other CMS based websites

  In contrast to bot blockers that try to prevent the automated systems from attacking them, WordPress and other CMS based websites use a blacklisting approach to prevent bot attacks. Unfortunately, since we also generate multiple requests to each website, this results in our IPs getting blacklisted. To overcome this, we did two things:

  1. Used an IP range of 60 different IP addresses.
  2. Used a blacklist of our own to prevent our Fuzzer from fuzzing websites that are known to blacklist automated crawlers.

[2] We do not have any data about whether bad markup indicates an overall lower quality of the website, and thus cannot comment on whether such websites are more likely to have vulnerabilities, although the author strongly suspects that that might be the case.

3.5 Assumptions

We made certain assumptions while building the system. This section describes the assumptions and explores to what extent these hold true:

1. **Crawler is not blocked by firewalls**

   This is a requisite for our system to work. If the Crawler is blocked for any reason, we do not get the data feed for our system, and without this input, it is almost impossible to set our system up.

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

   This is a reasonable expectation, albeit an unrealistic one.

   It is unrealistic because Crawlers work on the concept of proximity. They detect for the presence of In-Links and Out-Links from a particular URL, and hence the returned URLs are usually related to each other (at least the ones that are returned adjacent to each other).

   However, this assumption is reasonable due to the 'Law of averages' ([47]), the 'Law of big numbers' ([48]), and the concept of 'Regression to the mean' ([51]). Simply stated, a crawl of this large magnitude should give us a very distributed sample of the overall Web, eventually converging to the average of all websites in existence.

3. **Injection of 'bcc' indicates the existence of E-Mail Header Injection Vulnerability**

   We assume that the ability to inject a 'bcc' header field is proof that the E-Mail Header Injection vulnerability exists in the application. We do not inject any additional payloads that can modify the subject, message body, etc. as this analysis is to be as benign as possible. We believe that this is a reasonable

assumption, as in theory, it points to the existence of the vulnerability.

That concludes our discussion about the design of the system. To recap, we discussed our approach, the system architecture and how the components fit into our architecture. We also discussed the issues faced, and the assumptions that we made while building the system.

Chapter 4

EVALUATION

This chapter describes the experimental setup for our project including the servers used, the software and the platforms involved, the languages used, and the task queue system that was used for parallelism. We follow this up with our evaluation of the system, with a test suite, and proof of concept examples.

4.1 System Configuration

We used two systems for the project, and their configurations are as follows:

- **Dell PowerEdge T110 II Server**

```
CPU: Intel(R) Xeon(R) CPU E3-1220 V2 @ 3.10GHz
Cache size : 8192 KB
No. of Cores : 4
Total Memory (RAM) : 16 GB
Disk Space : 2 TB
```

- **MacBook Pro**

```
CPU: Intel Core i7 @ 2.8 GHz
Cache size : 6144 KB
No. of Cores : 4
Total Memory (RAM) : 16 GB
Disk Space : 500 GB
```

4.2 Platforms and Software

We enumerate the platforms and the software used for our project in Table 4.1.

Operating system	Ubuntu 14.04
Server	Apache - 2.4.17
Database	MariaDB - 10.1.9
Mail Server	Postfix - 2.11.0
Other software used	Mailcatcher, PostMan, HTTPRequester, RabbitMQ

Table 4.1: Platforms and software used for our project.

4.3 Languages used

We used Python 2 to build the system. The following factors influenced our choice of language: text processing capabilities, PCRE (Perl Compatible Regular Expressions) compatibility, and the numerous libraries for HTML Parsing, HTTP request generation, mail processing etc. We made use of the following major libraries (shown in table 4.2) for our system.

Library	**Functionality**
Requests	HTTP Request Generation
Beautiful Soup	HTML Parsing
Mailbox	Mail Processing
Celery	Task Queues

Table 4.2: Libraries that we used and their functions.

Despite the many benefits that Python 2 provides, we had certain issues with the language — discussed in Section 3.4 — like Python's GIL (Global Interpreter

Lock) which does not allow the running of multiple native threads concurrently. The following section (Section 4.4) describes in detail the task queue system (Celery) that we used to overcome this limitation of Python.

4.4 Celery Queues

We used a Celery task queue running on top of RabbitMQ to overcome the GIL. According to Celery Project Homepage ([7]):

> "Celery is an asynchronous task queue/job queue based on distributed message passing."

Simply put, Celery allows us to process multiple tasks in parallel by making use of what is known as a task queue. Celery instantiates multiple workers that listen to these queues and processes each task individually. This simulates pseudo-parallel processing to a certain degree, by allowing us to run multiple instances of the same program. It does this by using a message broker called RabbitMQ. According to RabbitMQ's Wikipedia page ([50]),

> "RabbitMQ is an open source message broker software that implements the Advanced Message Queuing Protocol (AMQP)"

RabbitMQ facilitates the storage and transport of messages on queuing systems. It is also cross-platform and open source, providing us with clients and servers for many different languages, thereby being the ideal fit for Celery. Thus, by using Celery and RabbitMQ together, we were able to achieve a certain degree of parallelism that would not have been possible with vanilla Python.

4.5 Test Suite

The test plan for our system includes a set of unit tests for each module in the pipeline. Further, we have unit tests for every individual function in the modules. The functions are tested separately, using mocks and stubs, so as to ensure isolated testing. This section outlines the test plan in the following manner. We list the modules that are tested, and then describe what each unit test tests for.

- Form Parser
  - `test_url_exception` - Tests whether the system handles incorrect or malformed URLs properly and terminates cleanly.
  - `test_db_connection` - Tests whether the Database Connection is set up and queries can be executed.
  - `test_form_parser` - Tests for the proper parsing of HTML, and if the system exits cleanly in case parsing isn't possible.
- E-Mail Field Checker
  - `test_check_for_email` - Tests whether the system finds E-Mail fields in the form when the words 'e-mail' or 'email' are present in the form (case insensitive).
  - `test_check_for_no_email` - Tests whether the system finds no E-Mail fields when the words 'e-mail' or 'email' are *not* present in the form (case insensitive).
- E-Mail Form Retriever
  - `test_reconstruct_form` - Tests for the proper reconstruction of the form stored in the Database.

  - `test_construct_url` - Tests whether the URL for submission was constructed properly, includes checks for relative URLs, absolute URLs, and presence of `base' tags.
  - `test_email_form_retriever_already_fuzzed` - Tests for duplicate fuzz requests, and whether the system rejects these requests.
  - `test_email_form_retriever_calls_fuzzer_for_new_fuzz` - Tests whether the E-Mail Form Retriever calls the Fuzzer module with the proper data when it gets a new fuzz request.

- Fuzzer
  - `test_send_get_request` - Tests for the proper handling of GET requests.
  - `test_send_post_request` - Tests for the proper handling of POST requests.
  - `test_correct_fuzzer_data` - Tests whether the payload generated for the given form data is correct and consistent. Also tests whether the payload was part of the resulting HTTP request.
  - `test_incorrect_fuzzer_data` - Tests for incorrect form data, and ensures that a payload does not end up in the wrong input field in the resulting HTTP request.

- E-Mail Analyzer
  - `test_load_mail` - Tests whether the E-Mails are loaded and parsed correctly by the E-Mail Analyzer.
  - `test_parse_headers` - Tests for the proper parsing of headers present in the E-Mail.

- `test_analyze_regular_mail` - Tests whether the E-Mail Analyzer parses the regular E-Mail properly and extracts the injected input fields that are present in the E-Mail.
- `test_analyze_malicious_mail` - Tests whether the E-Mail Analyzer parses the E-Mails received due to the malicious payloads properly, is able to extract the 'bcc' headers, and is able to link them to the proper fuzzing request and payload.
- `test_analyze_x_check_header` - Tests whether the 'x-check' header is read by the E-Mail Analyzer.

The unit tests were written using Python's built-in 'Unittest' module, mocking was done using the built-in 'MagicMock' module. The tests allow us to be reasonably certain that our system works as expected.

4.6 Proof of Concept Attacks

The previous section (Section. 4.5) discussed in detail how we verified that our system functions according to our expectations. This section describes how we validated our expectations. In order to do this, we constructed three sets of web applications in PHP, Python, and Ruby. Each of these applications was a simple web app that took in user input to construct and send an E-Mail.

The front-end for each of the three applications is shown in Listing. 4.1. The back-end for the three languages are shown in Listings 2.1, 2.2, and 2.3.

We tested for the headers being injected in real-time by running an instance of MailCatcher, set to listen on all SMTP messages. A sample screenshot of a fuzzed request for the Ruby backend (generated in PostMan) is shown in Figure 4.1. The E-Mail sent due to injecting this payload (as captured by MailCatcher) is shown in

Figure 4.2. It can be seen that the Headers have been added to the resulting E-Mail, and we have successfully managed to overwrite the 'Subject' field with our message, 'hello'.

A similar injection example for PHP is shown in Figure 4.3 and the corresponding E-Mail caught by MailCatcher is shown in Figure 4.4. The astute reader might have noticed that in the given examples we have used '%0a' to separate the headers, while in Section. 3.3.5, we had used '\n'. This is due to URL Encoding ([3]), wherein special characters are 'encoded' or 'escaped' with their ASCII equivalent. The reason why we do not have to do this with the payloads our system injects is due to the fact that the 'Requests' library that we use to generate the HTTP requests automatically does this encoding for us.

```
<!doctype html>
<html lang="en">
<head>
<meta charset="utf-8">
<meta name="author" content="Sai Pc">
<title>Mock Email</title>
</head>
<body>
<form action="{Replace with path to back-end}" method="post">
<input type="text" placeholder="Email" name="email" id="e-mail"><br>
<textarea name="msg" rows="20" cols="120"></textarea>
<input type="submit" value="Email Me!">
</form>
</body>
</html>
```

Listing 4.1: HTML page for showcasing e-mail header injection, a simple front-end for our examples.

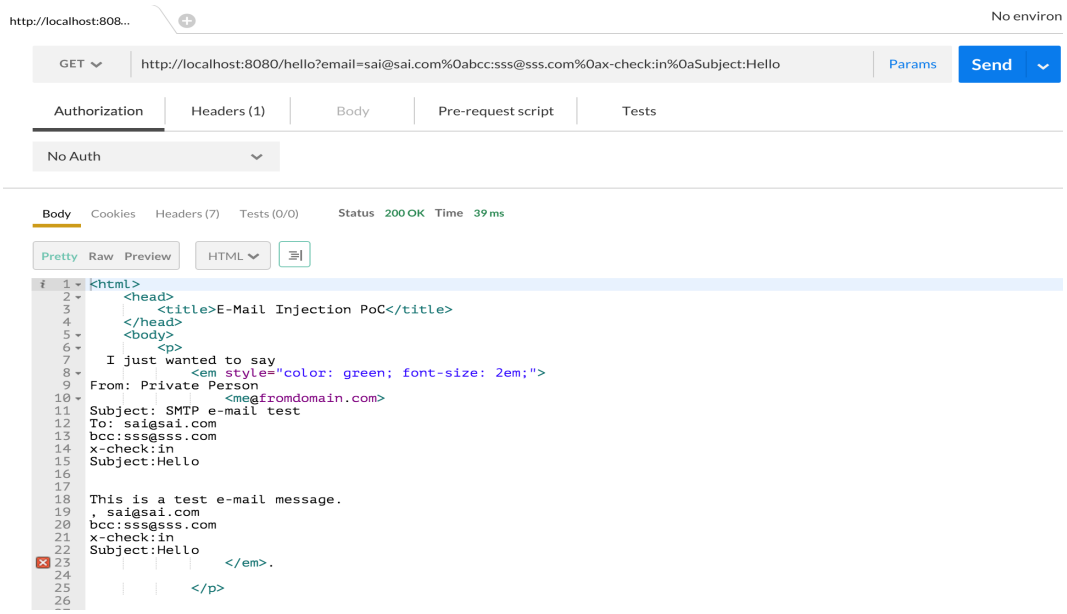

Figure 4.1: Fuzzing a request for the ruby backend, the payload can be seen inside the address bar on top.

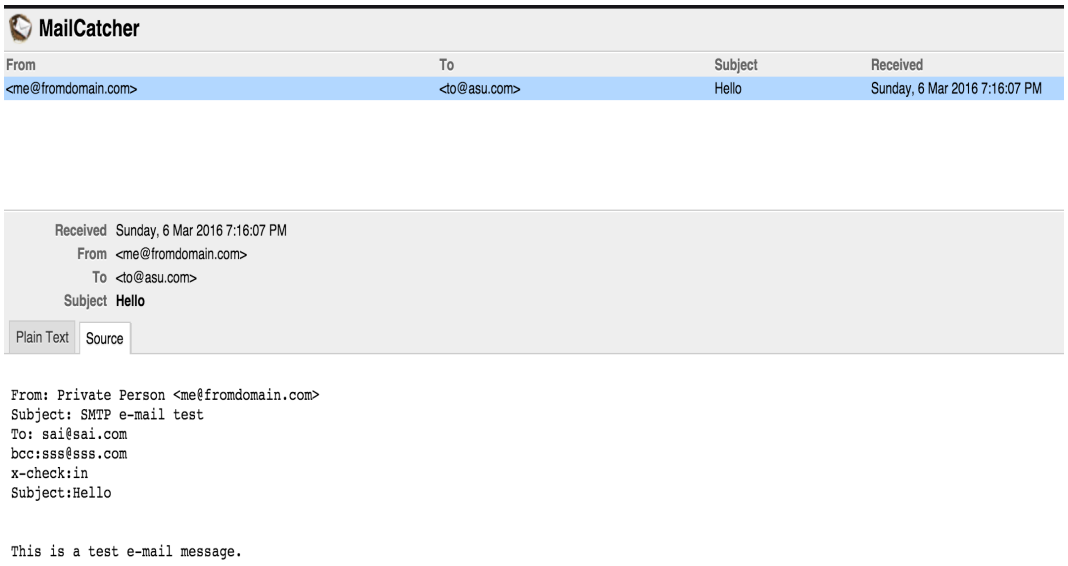

Figure 4.2: E-Mail header injection proof of concept - ruby, we can see that multiple headers (bcc, x-check, Subject) have been inserted into the resulting e-mail.

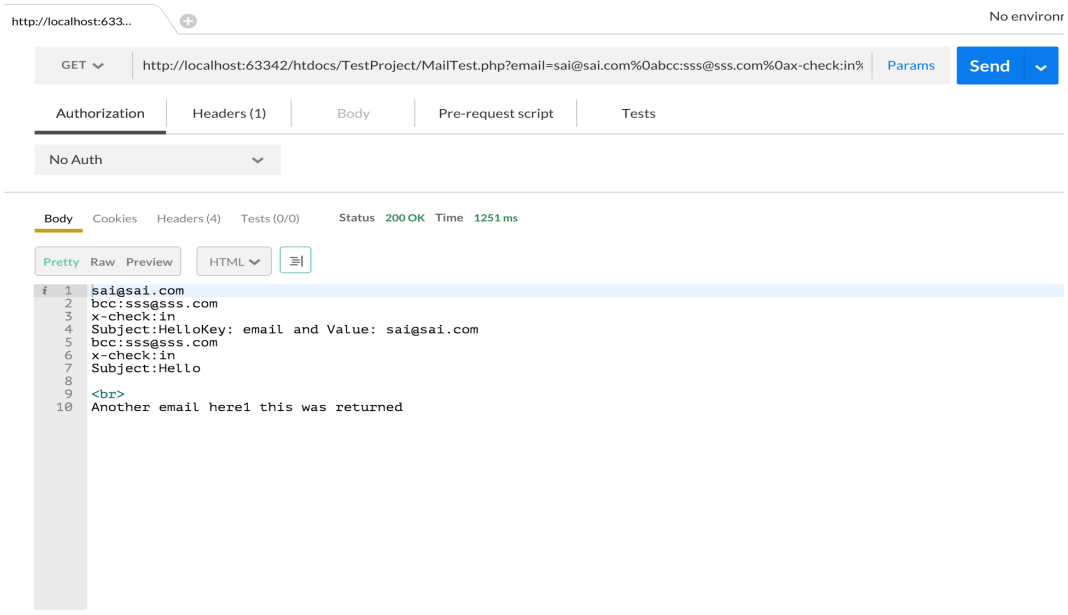

Figure 4.3: Fuzzing a request for the php backend, the payload can be seen inside the address bar on top.

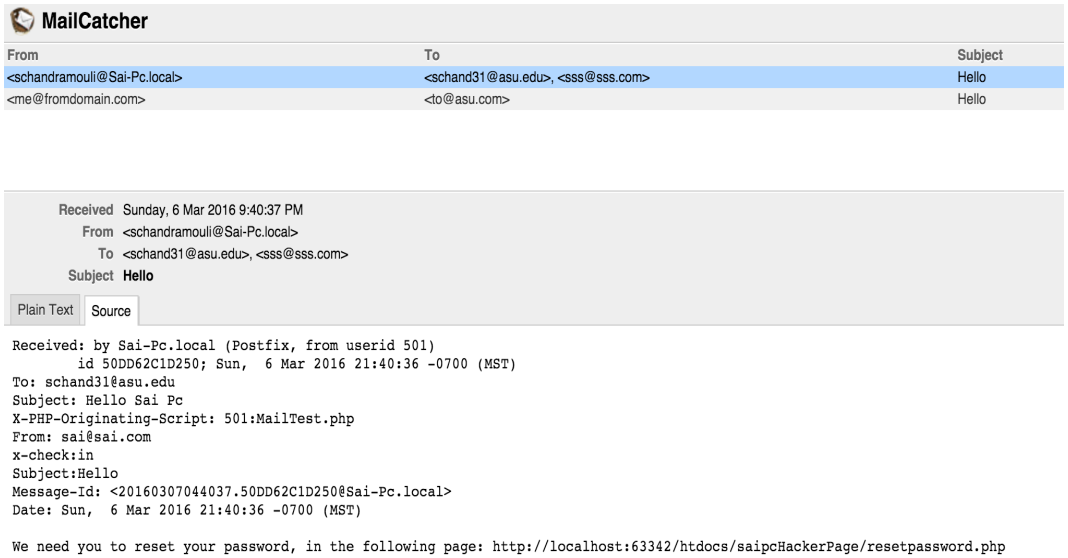

Figure 4.4: E-Mail header injection proof of concept - php, we can see that multiple headers (bcc, x-check, Subject) have been inserted into the resulting e-mail.

Chapter 5

DATA ANALYSIS AND RESULTS

This chapter serves to present our findings: the data that we gathered from our crawl, the data generated due to the fuzzing attempts and our analysis on this data.

5.1 Collected Data

From our extensive crawl of the web, we were able to gather the data shown in Table 5.1. The following paragraphs describe in detail what each kind of data represents, and what they signify.

S.No	**Type of Data**	**Quantity**
1	URLs Crawled	13,464,300
2	Total Forms found	3,725,106
3	Forms with E-Mail Fields	581,435

Table 5.1: The data collected for our project.

URL's crawled This represents the total number of unique URLs that we crawled on the World Wide Web. It is to be noted that this quantity refers to the URL's, and not websites.

Total Forms found This represents the total number of forms that were found on the URL's crawled. We found a total of 3,725,106 forms from 512,480 unique domains.

Forms with E-Mail Fields This represents the total number of forms that were found to contain e-mail fields. We found 581,435 such forms from 113,663 unique domains.

5.2 Fuzzed data

We performed our fuzzing attempts on the gathered data with both the regular payload and malicious payload and received a set of e-mails. Table 5.2 shows the quantity of e-mails that we received for each payload. We explain in detail what each piece of data shown in the table represents, in the following section.

S.No	Type of fuzzing	Forms Fuzzed	E-Mails received
1	Regular Payload	412,623	31,666
2	Malicious Payload	31,666	289

Table 5.2: The data that we fuzzed and the e-mails that we received.

E-Mail received from Forms The e-mails that we received can be broadly categorized into two categories:

1. E-Mails due to regular payload

   This represents the total number of websites that sent back e-mails to us. This indicates that we were able to successfully submit the forms on these sites, without any bot-prevention etc.

2. E-Mails due to malicious payload

   Once we receive an e-mail from a website due to the regular payload, we go back and fuzz those forms with more malicious payloads. This field, in essence,

represents the total number of websites that are vulnerable to e-mail header injection.

The following section showcases the different categories of e-mails we received and presents our analysis on these e-mails.

5.3 Our Analysis of the received E-Mail Data

During our analysis of the received e-mails, we found that the e-mails that we received belonged to one of three broad categories:

1. E-Mails with the 'bcc' header successfully injected

   This form of injection was our initial objective and we found 180 such e-mails in our received e-mails. This indicates that the websites that sent out these e-mails are vulnerable to e-mail header injection, where we could inject and manipulate any header.

2. E-Mails with the 'to' header successfully injected

   We discovered an unintended vulnerability which we would like to christen 'TO header injection'. These injections reflect the ability to inject any number of e-mail addresses into the 'to' field while being unable to inject any other header into the e-mails. We attribute this behavior to inconsistent sanitization by the application. The vulnerability is further aided by the leniency shown by mail servers, wherein they parsed malformed e-mail addresses and delivered it to the right mail server, and on the receiving end, the mail got delivered to the right mailbox.

   While not allowing us complete control over the e-mails sent, 'TO header injection' makes it possible to append any number of e-mail addresses, thereby enabling us to generate freely any amount of spam.

3. E-Mails with the 'x-check' header successfully injected

   The third category of e-mails received were e-mails with the x-check header injected. As discussed in Section. 3.3.6, these let us differentiate between unsuccessful attempts and successful attempts by injecting the additional header.

We list each category and the number of e-mails received by the category in Table 5.3.

S.No	**Type of Injection**	**No. of e-mails received**
1	E-Mail Header Injections with 'bcc' header	180
2	E-Mail Header Injections with 'x-check' header	153
3	'To header' injections alone	83
4	E-Mail Header Injections with 'bcc' and 'x-check' headers	136
5	Both 'To header' injections and x-check headers	9
6	'x-check' headers found in 'nuser' e-mails	40
7	Unique 'x-check' headers found in 'nuser' e-mails	16

Table 5.3: Classification of the e-mails that we received into broad categories of the vulnerability.

We explain the combination of these header injections (4-7) as follows:

- E-Mail Header Injections with 'bcc' and 'x-check' headers

  These represent the perfect attack scenario where we are able to inject multiple headers into the e-mails. We can see that over 75% of the received 'bcc' header injected e-mails are also susceptible to other header injections.

- Both 'To header' injections and x-check headers

  This combination shows us that in addition to being able to inject into the 'To'

fields, we are able to inject additional headers into the e-mail. It is not clear what causes this behavior; however, these can be exploited to achieve the same result as a regular E-Mail Header Injection.

- 'x-check' headers found in 'nuser' e-mails

  In addition to analyzing the 'maluser' account, we also analyze emails received by the 'nuser' account. We explain the presence of these headers in the below paragraph.

- Unique 'x-check' headers found in 'nuser' e-mails

  These represent the e-mails with form_ids that were *not* already found in the 'maluser' account. We attribute these e-mails to (probably) having a backend that was built with Python or another language having a similar behavior with respect to constructing headers.

5.4 Understanding the pipeline

This section serves to represent our pipeline quantitatively and graphically. Table 5.4 showcases the data gathered by our pipeline, with the differential changes at each stage of the pipeline.

S.No	Pipeline Stage	Quantity	Differential Δ d2/d1 * 100
1	Crawled URLs	13,467,548	—
2	Forms found	3,725,106	27.66%
3	E-Mail Forms found	581,435	15.61%
4	Fuzzed with regular payload	412,623	70.97%
5	Received e-mails	31,666	7.67%
6	Fuzzed with malicious payload	31,666	100.00%
7	Successful attacks	289	0.91%

Table 5.4: Data gathered by our pipeline at each stage, with the differential between the stages.

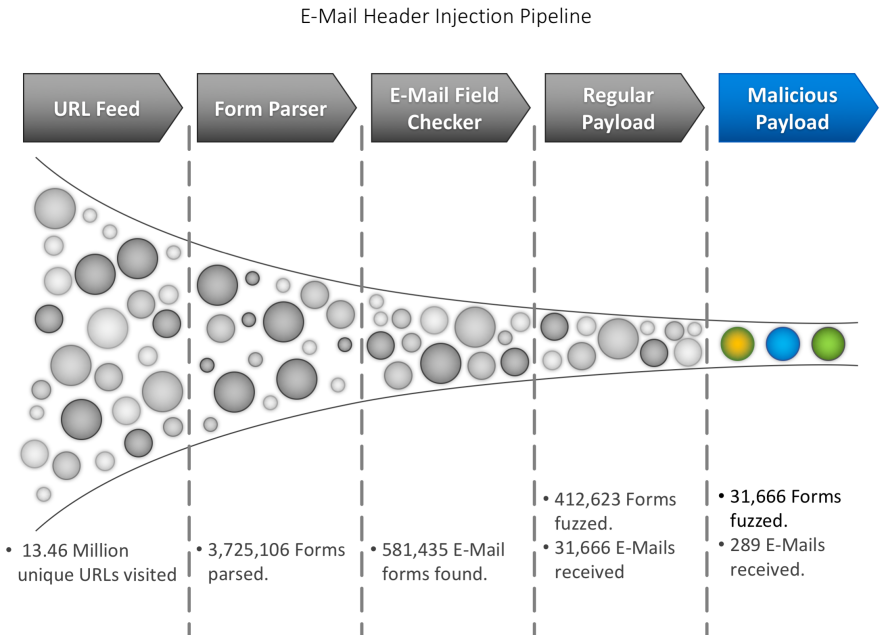

Figure 5.1: Pipeline - shows the quantity of data gathered at each stage of the pipeline.

Chapter 6

DISCUSSION

In the previous chapter, we discussed our results and presented our analysis of the data gathered. In this chapter, we discuss the things we learned, and the limitations of our system. We conclude this chapter with a few ways to mitigate this vulnerability.

6.1 Lessons Learned

From our results, it is evident that the vulnerability exists in the wild. Despite its relatively low occurrence rate compared to the more popular SQL Injection and XSS (Cross-Site Scripting), when we take the total number of websites on the World Wide Web and calculate the percentage of that number, we end up with a pretty significant number. We agree that an extrapolation of that kind might not be a good measure of the prevalence of the vulnerability. However, even with as few as a thousand websites affected by this vulnerability, it can still have a disastrous impact on the global Web.

After analyzing the results, we would like to make a few more observations. We believe that one of the reasons for the small percentage of occurrence (compared to SQL Injection or XSS), can be attributed to what we like to call the 'car parking analogy'. The car parking analogy is something like this: Imagine that we are parking a car on a road that is prone to attacks by thieves. Now, if all the cars were unlocked, the car that is most likely to get stolen is quite unsurprisingly the most expensive one or the one that is easiest to get away with.

Now imagine the same thing on the World Wide Web, we have websites that can each have multiple vulnerabilities. Now, it makes sense for an attacker to try and attack the website with more widespread and glamorous attacks such as SQL

Injection or XSS, rather than attempt to exploit E-Mail Header Injection, seeing as this requires a more concentrated effort, with carefully crafted payloads and a waiting time for the E-Mail to be delivered.

This also gives more incentive for the website developer to add protection against attacks such as SQL injection and XSS. What this does, however, is to (unknowingly) help prevent E-Mail Header Injection. This is explained as follows: E-Mail Header Injection thrives on the ability to inject a newline into the input fields. XSS and SQL Injection Prevention serves to remove special characters like newlines, and backslashes (encoded or otherwise).

Thus, indirectly, it affects the attempts to perform E-Mail Header Injection. However, this does not completely negate the attempts if the checks are only on the client-side. Also, even with server-side validation, often, the only input fields that are validated are ones that are either inserted into the database (SQL Injection) and the ones that are displayed back to the user as part of the web site (XSS).

A second and a far more common reason for our fuzzing attempts to fail is the bot-blocking mechanisms built into the websites. CAPTCHAs (as explained in Section 3.4 and in the following section) pose a very difficult problem for our system to exploit E-Mail Header Injection, even if it is present.

This does not mean that the vulnerability is not a big threat. In fact, as mentioned many times previously, this vulnerability too can have some major consequences, the least of which can be spamming and phishing attacks. In today's digital world, identity theft has become all the more common. E-Mail Header Injection provides attackers with the ability to extract easily information about users, not just from a server, but from the user himself, by sending him fake messages that look extremely authentic, since these messages are sent by the website itself.

From our research, we found two different forms of the E-Mail Header Injection

Vulnerability: the first one is the traditional one, where we are able to inject any header into the forms, allowing us complete control over the contents of the E-Mail. We identify this with the presence of both the 'bcc' header and the 'x-check' header. This is the most potent form of the vulnerability and is found on quite a few websites. This is also the vulnerability that is documented and discussed on many websites.

The second attack is an interesting one, as this has not yet been documented, and provides the ability to inject multiple E-Mail addresses into only the 'TO' field. In this form of the vulnerability, we are able to simply add addresses to the 'To' field of the form with newlines separating the E-Mail addresses. Whether this particular form of the vulnerability is found due to the websites in question, or whether this is an implementation issue with a particular language or framework, is unclear. However, from our preliminary analysis, it is evident that these websites do not share much with respect to the languages and frameworks used. Even in this form of the attack, we are still able to extract information that should be private to a given user, and in most of these cases, able to inject enough data to spoof the first few lines of the E-Mail message.

While not being as lethal as the primary vulnerability, this second form of the vulnerability does still provide the ability to send the E-Mails to multiple recipients, and can easily result in information leakage, and/or spam generation.

6.2 Limitations of the Project

This section complements Section 3.4, and discusses the limitations of our project. The following list, although not exhaustive, goes into the limitations of our project in detail:

- CAPTCHAs - As noted in section 3.4, CAPTCHAs pose a significant problem to our automated system. Since CAPTCHAs are robust, there is no easy way

to break them. There has been considerable research in this area ([24], [52] to name a few) and although not impossible to break, it remains out of the scope of this project, and thus, we chose just to ignore the websites which require CAPTCHA verification.

- JavaScript Apps - Due to the emergence of Node.js as a server-side language, and the growing emphasis on responsive web applications, more and more applications are being built purely with JavaScript. Even conventional applications are now making use of JavaScript to dynamically insert content and update the pages. It might not be immediately clear, but what this means for the web application is that these dynamically injected components are not a part of the source code that is sent by the web server.

  Thus, our system never receives dynamically injected forms from the web server and hence is unable to detect whether these vulnerabilities are present in such forms. The only workaround would be, ironically, to use JavaScript to query for the document.getElementsByTagName('html')[0].innerHTML (from inside web browser automation tools like Selenium, etc.), and then use that as the source code for our URL.

  Since this would add unnecessary bulk and complexity to our application, we chose not to do it, and thus, we consider this to be a limitation.

- Blogs powered by WordPress/Drupal

  In addition to what was discussed in Section 3.4, we found that certain WordPress 'plugins' also prevent the E-Mail Header Injection attack by sanitizing user input on Contact Forms. Some of these plugins are discussed in the following section. Although not all websites built with WordPress are secure from the attack, between the presence of the plugins on some websites, and getting

tagged as 'spambots' by others, we were able to do vulnerability analysis on very few sites powered by WordPress.

- Blacklisting by websites and ISPs

  During the actual crawl, we ended up getting blacklisted by a few websites (mostly WordPress ones), and Internet Service Providers (ISPs). We then had to create up a blacklist of our own to ensure that we did not inject these websites. The result was that we could not gather any data about these websites.

- E-Mail libraries

  E-Mail libraries like the PHP Extension and Application Repository's (PEAR) Mail Library provide inbuilt sanitization checks for user input. While this is technically not a limitation of our project, it still makes it such that we are not able to inject these sites successfully, and that is enough to justify its inclusion in the limitations section. A few other libraries for each language are discussed in the following section.

6.3 How to prevent this attack

This section describes the most common measures that can be taken to prevent the occurrence of this vulnerability, or at least reduce the impact.

- Use Mail Libraries

  This is the preferred way of combating this vulnerability. Using a library that is well tested can remove the burden of input sanitization from the developer. Also, since most of these libraries are open-source, bugs are identified quicker and fixes are readily available. A list of known secure libraries for each popular language and framework is shown in Table 6.1.

  Using libraries such as PEAR Mail, PHPMailer, Apache Commons E-Mail,

Contact Form 7, etc. can significantly reduce the occurrence of this type of attack.

Language	**Mail Libraries**
PHP	PEAR Mail[1], PHPMailer[2], Swiftmailer[3]
Python	SMTPLib with email.header.Header[4]
Java	Apache Commons E-Mail[5]
Ruby	Ruby Mail >= 2.6[6]
WordPress	Contact Form 7[7]

Table 6.1: Mail libraries that prevent e-mail header injection.

- Use a Content Management System (CMS)

  Content management systems like WordPress and Drupal include certain libraries and plugins to prevent E-Mail Header Injection. Thus, websites built with such CMS' are usually resistant to these attacks. However, it is advised to use the right E-Mail plugins when using such CMS', as not all plugins might be secure. An example of a secure plug-in is included as part of Table 6.1.

- Input Validation

  If neither of the two options above is feasible, due to reasons such as the website

[1]PEAR Mail Website: https://pear.php.net/package/Mail

[2]PHPMailer Website: https://github.com/PHPMailer/PHPMailer

[3]Swiftmailer Website: http://swiftmailer.org/

[4]instead of using email.parser.Parser to parse the header

[5]Apache Commons E-Mail: https://commons.apache.org/proper/commons-email/

[6]Ruby Mail Website: https://rubygems.org/gems/mail

[7]Contact Form 7 Download: https://wordpress.org/plugins/contact-form-7/

being an in-house production, or due to lack of support infrastructure, developers can choose to perform proper input sanitization. Sanitization should be done keeping in mind RFC5322 ([37]), and care must be taken to ensure that all edge cases are taken into account.

Client Side validation alone is not sufficient, and must be supplemented by server-side validation to be reasonably confident of mitigating the attack. Constant updates to validation methods are required so that new attack vectors do not harm the website in any way. Test driven development for such validation methods is also encouraged so that we can be reasonably sure of our defense mechanisms.

The next chapter goes into a detailed discussion of related work and how our work differs from existing work in this area.

Chapter 7

RELATED WORK

There are different approaches to finding vulnerabilities in web applications, two of them being Black-Box testing and White-Box testing. Our work is based on the black-box testing approach to finding vulnerabilities on websites, and there has been plenty of research that has made use of this methodology ([2], [17], [20], [28], [53] etc.). There has been significant discussion on both the benefits of such an approach ([1]) and its shortcomings ([10], [11]).

Our work does not intend to act as an 'everything and the kitchen sink' vulnerability scanner, but as a means to identify the presence of E-Mail Header Injection Vulnerability in a given web application. In this sense, since we are injecting payloads into the web application, our work is related to other injection based attacks, such as SQL Injection ([5], [14], [40]), Cross-Site Scripting — XSS — ([18], [21]), HTTP Header Injection ([19]), and is very closely related to Simple Mail Transfer Protocol (SMTP) Injection ([43]).

The attack described by Terada [43] is one that attacks the underlying SMTP mail servers by injecting SMTP commands (which are closely related to E-Mail Headers and usually have a one-to-one mapping) to exploit the SMTP server's pipelining mechanism. The paper also describes proof-of-concept attacks against certain Mailing libraries like Ruby Mail and JavaMail. This attack, although trying to achieve a similar result, is distinctly different from ours. The paper by itself makes this observation and discusses why it is different from E-Mail Header Injection.

In comparison, our work tries to exploit application-level flaws in user input sanitization, which allow us to perform this attack. Our work does not intend to exploit

the pipelining mechanism, but to exploit the implementation of the mail function in most popular programming languages, which leaves them with no way to distinguish between user supplied headers and headers that are legitimately added by the application.

As specified before, although this vulnerability has been present for over a decade, there has not been much written about it in the literature, and we only find a bunch of articles on the internet describing the attack.

The first documented article dates to over a decade ago, in a late 2004 Article on phpsecure.info ([44]) accredited to user tobozo@phpsecure.info describing how this vulnerability existed in the reference implementation of the mail function in PHP, and how it can be exploited. Following this, we have a plethora of other blog posts ([6], [22], [23], [26], [32]), each describing how to exploit the vulnerability by using newlines to camouflage headers inside user input. A Wiki entry ([12]) also describes the ways to prevent such an attack. However, none of these articles have performed these attacks against real-life websites.

Another blog post written by user Voxel@Night on Vexatious Tendencies ([42]), recounts an actual attack against a WordPress plugin, 'Contact Form', with a proof of concept [1] . It also showcases the vulnerable code in the plugin that causes this vulnerability to be present. However, this article targets just one plugin and does not aim to find the prevalence of said plugin usage. Neither does it inform the creators of the plug-in to fix the discovered vulnerability.

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 Flaws*" ([41]). The book, however, does not go

[1]It is to be noted that this plugin is used actively on 300,000 websites (according to [4]), but is yet to be fixed.

into detail on either the attack or the ways to mitigate such an attack. Our work, on the other hand, dedicates an entire section (Section. 6.3) on the means to mitigate the attack. We also describe, in detail, the payloads that can be used and the need for varying the payloads (Section. 3.3.5).

To the best of our knowledge, no other research has been conducted to determine the prevalence of this vulnerability across the World Wide Web. We have managed to, on a vast scale, crawl and inject websites with comparatively benign payloads (such as the BCC header) to identify the existence of this vulnerability without costing any ostensible harm to the website. Our work serves to not only prove the existence of the vulnerability on the World Wide Web but to quantify it.

The next and final chapter starts by outlining our contributions and closes with concluding remarks about the project.

Chapter 8

CONCLUSION

We have showcased a novel approach involving black-box testing to identify the presence of E-Mail Header Injection in a web application. Using this approach, we have demonstrated that our system was able to crawl x web pages finding y forms that were fuzzable. We fuzzed z forms and found k vulnerable forms. This indicates that the vulnerability is widespread, and needs attention from both web application developers and library makers.

We hope that our work sheds light on the prevalence of this vulnerability and that it ensures that the implementation of the 'mail' function in popular languages is fixed to differentiate between User-supplied headers, and headers that are legitimately added by the application, and that the RFC's are updated to be more stringent and make it less ambiguous for future implementations.

REFERENCES

[1] Bau, J., E. Bursztein, D. Gupta and J. Mitchell, "State of the art: Automated black-box web application vulnerability testing", in "Security and Privacy (SP), 2010 IEEE Symposium on", pp. 332–345 (2010).

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

[3] Berners-Lee, T., L. Masinter and M. McCahill, "Internet Message Format - RFC 1738", URL `https://www.ietf.org/rfc/rfc1738.txt` (2008).

[4] BestWebSoft, "Contact Form by BestWebSoft âĂŤ WordPress Plugins", URL `https://wordpress.org/plugins/contact-form-plugin/` (2016).

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

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

[7] Celery, "Celery Homepage", URL `http://www.celeryproject.org/` (2016).

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

[9] 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).

[10] Doupé, A., L. Cavedon, C. Kruegel and G. Vigna, "Enemy of the state: A state-aware black-box web vulnerability scanner", in "Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)", pp. 523–538 (USENIX, Bellevue, WA, 2012), URL `https://www.usenix.org/system/files/conference/usenixsecurity12/sec12-final225.pdf`.

[11] Doupé, A., M. Cova and G. Vigna, "Why johnny canâĂŹt pentest: An analysis of black-box web vulnerability scanners", in "Detection of Intrusions and Malware, and Vulnerability Assessment", pp. 111–131 (Springer, 2010).

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

[13] 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).

[14] 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).

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

[16] 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).

[17] Huang, Y.-W., S.-K. Huang, T.-P. Lin and C.-H. Tsai, "Web application security assessment by fault injection and behavior monitoring", in "Proceedings of the 12th International Conference on World Wide Web", WWW '03, pp. 148–159 (ACM, New York, NY, USA, 2003), URL http://doi.acm.org/10.1145/775152.775174.

[18] 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.

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

[20] Kals, S., E. Kirda, C. Kruegel and N. Jovanovic, "Secubat: a web vulnerability scanner", in "Proceedings of the 15th international conference on World Wide Web", pp. 247–256 (ACM, 2006).

[21] 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).

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

[23] Mohamed, A., "PHP Email Injection Example - InfoSec Resources", URL http://resources.infosecinstitute.com/email-injection/ (2013).

[24] Mori, G. and J. Malik, "Recognizing objects in adversarial clutter: breaking a visual captcha", in "Computer Vision and Pattern Recognition, 2003. Proceedings. 2003 IEEE Computer Society Conference on", vol. 1, pp. I–134–I–141 vol.1 (2003).

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

[26] Nicol, J., "Securing PHP Contact Forms", URL http://jonathannicol.com/blog/2006/12/09/securing-php-contact-forms/ (2006).

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

[28] 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).

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

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

[31] 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).

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

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

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

[35] Python, "Python Global Interpreter Lock", URL https://wiki.python.org/moin/GlobalInterpreterLock (2016).

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

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

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

[39] RubySec, "Mail Gem for Ruby vulnerable to SMTP Injection via recipient email addresses", URL http://rubysec.com/advisories/OSVDB-131677/ (2015).

[40] 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).

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

[42] Tendencies, V., "WordPress Plugin Vulnerability Dump âĂŞ Part 2 | Vexatious Tendencies", URL https://vexatioustendencies.com/wordpress-plugin-vulnerability-dump-part-2/ (2014).

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

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

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

[46] 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] (2016).

[47] Wikipedia, "Law of averages — Wikipedia, the free encyclopedia", http://en.wikipedia.org/w/index.php?title=Law%20of%20averages&oldid=706716293, [Online; accessed 08-March-2016] (2016).

[48] Wikipedia, "Law of large numbers — Wikipedia, the free encyclopedia", http://en.wikipedia.org/w/index.php?title=Law%20of%20large%20numbers&oldid=706596753, [Online; accessed 08-March-2016] (2016).

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

[50] Wikipedia, "RabbitMQ — Wikipedia, the free encyclopedia", http://en.wikipedia.org/w/index.php?title=RabbitMQ&oldid=708777848, [Online; accessed 09-March-2016] (2016).

[51] Wikipedia, "Regression toward the mean — Wikipedia, the free encyclopedia", http://en.wikipedia.org/w/index.php?title=Regression%20toward%20the%20mean&oldid=703369877, [Online; accessed 08-March-2016] (2016).

[52] Yan, J. and A. S. E. Ahmad, "Breaking visual captchas with naive pattern recognition algorithms", in "Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual", pp. 279–291 (2007).

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

APPENDIX A