

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 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 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.

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 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

	Page
LIST OF TABLES	vi
LIST OF FIGURES	vii
CHAPTER	
1 INTRODUCTION	1
2 E-MAIL HEADER INJECTION BACKGROUND	4
2.1 Problem Background	4
2.2 History of E-Mail Injection	4
2.3 Languages Affected	5
2.4 Potential Impact	9
3 SYSTEM DESIGN	11
3.1 Approach	11
3.2 System Architecture	11
3.3 System Components	12
3.3.1 Crawler	14
3.3.2 Form Parser	14
3.3.3 E-Mail Field Checker	15
3.3.4 E-Mail Form Retriever	15
3.3.5 Fuzzer	16
3.3.6 E-Mail Analyzer	18
3.3.7 Database	20
3.4 Test Plan	20
3.5 Proof of Concept Attacks	24
3.6 Issues	26
3.7 Assumptions	32

CHAPTER	Page
4 EXPERIMENTAL SETUP	34
4.1 System Configuration	34
4.2 Platform	34
4.3 Languages used	35
4.4 Celery Queues	36
5 RESULTS	37
5.1 Data	37
5.1.1 URLS crawled	37
5.1.2 Forms collected	37
5.1.3 Forms with E-Mail Fields	37
5.1.4 E-Mail received from Forms	37
5.1.5 Fuzzed Forms	37
6 DISCUSSION	38
6.1 Lessons Learned	38
6.2 Limitations	38
6.3 Mitigation Strategy	39
7 RELATED WORK	42
8 CONCLUSION	43
REFERENCES	44
APPENDIX	
A	48

LIST OF TABLES

Table	Page
2.1 A brief history of E-Mail Header Injection	6
2.2 Language Usage Statistics compiled from W3Techs	10
3.1 Payload Coverage	18
3.2 Database - Tables	21
4.1 Platforms and Software used	35
4.2 Libraries and their functions	35
6.1 Mail Libraries to prevent E-Mail Header Injection	40

LIST OF FIGURES

Figure	Page
3.1 System Architecture - Crawler	13
3.2 System Architecture - Fuzzer & E-Mail Analyzer	13
3.3 Database Schema	22
3.4 Fuzzed Request for Ruby	26
3.5 E-Mail Header Injection Proof of Concept - Ruby	27
3.6 Fuzzed Request for PHP	27
3.7 E-Mail Header Injection Proof of Concept - PHP	28

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) 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 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, 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, 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.

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 in 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 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 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 vulnerabilities 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 Listing. 2.1.

```
1 $from = $_REQUEST[ 'email' ];
2 $subject = "Hello Sai Pc";
3 $message = "We need you to reset your password";
4 $to = "schand31@asu.edu";
5
6 // attack string => 'sai@sai.com\nBCC:spc@spc.com'
7 $returnValue = mail($to, $subject, $message, "From: $from");
8 // 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 Listing. 2.2

```
1 from email.parser import Parser
2 import cgi
3 form = cgi.FieldStorage()
4 to = form["email"] # input() exhibits the same behavior
5 msg = """To: """ + to + """\n
```

```

6 From: <user@example.com>\n
7 Subject: Test message\n\n
8 Body would go here\n"""
9
10 f = FeedParser() # Parser.parsestr() also
11 # contains the same vulnerability
12 f.feed(msg)
13 headers = FeedParser.close(f)
14
15 # attack string => 'sai@sai.com\nBCC:spc@spc.com'
16
17 # both to:sai@sai.com AND bcc:spc@spc.com
18 # are added to the headers
19 print 'To: %s' % headers['to']
20 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 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.

```

1 require 'sinatra'
2 require 'net/smtp'

```



```

3
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: 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 do potentially any of the following:

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

- Phishing and Spoofing Attacks
- Denial of service by attacking the underlying mail server
- Spam Networks

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

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), and generate an 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, 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:

Data Gathering - Crawler System

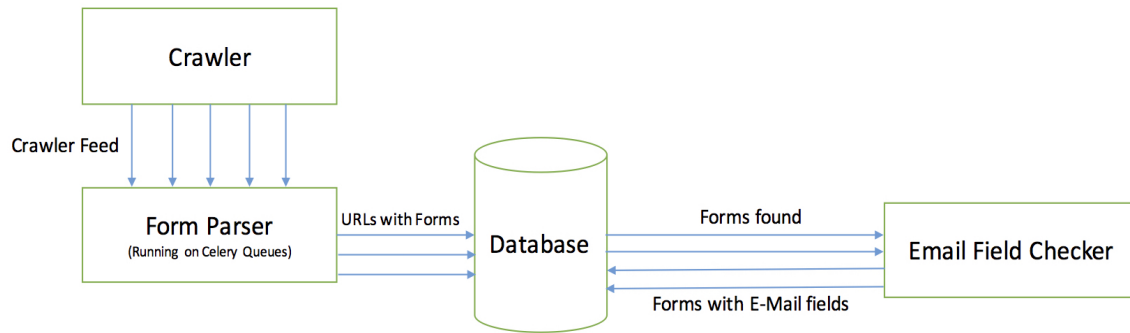


Figure 3.1: System Architecture - Crawler

Payload Injection - Fuzzer System

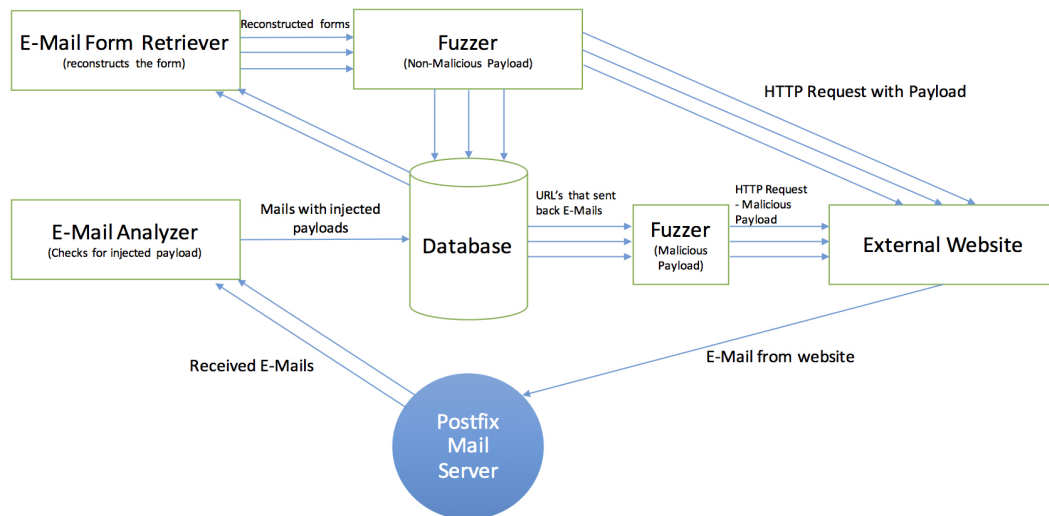


Figure 3.2: System Architecture - Fuzzer & E-Mail Analyzer

3.3.1 Crawler

We used an open-source Apache Nutch based Crawler built at the 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 words ‘e-mail’, ‘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 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 3.6 - 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)@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 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. 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` - 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.

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

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

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. 3.3.

3.4 Test Plan

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. Further, we have unit tests for each function in the modules.

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

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

- `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.

3.5 Proof of Concept Attacks

The previous section (Section. 3.4) 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. 3.1. 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 Fig. 3.4. The E-Mail sent due to injecting this payload (as captured by MailCatcher) is shown in Fig. 3.5. 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 Fig. 3.6 and the corresponding E-Mail caught by MailCatcher is shown in Fig. 3.7. 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 (Berners-Lee *et al.* (2008)), 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.

```
1 <!doctype html>
2 <html lang="en">
3 <head>
4 <meta charset="utf-8">
5 <meta http-equiv="x-ua-compatible" content="ie=edge">
6 <title>Mock Email</title>
7 <meta name="author" content="Sai Pc">
8 <meta name="viewport" content="width=device-width, initial-scale=1">
9 </head>
10 <body>
11 <div id="container" class="container">
```

```

12 <form action="{Replace with path to back-end}" method="post">
13 <input type="text" placeholder="Email" name="email" id="email"><br>
14 <textarea name="msg" rows="20" cols="120"></textarea>
15 <input type="submit" value="Email Me!">
16 </form>
17 </div>
18 </body>
19 </html>

```

Listing 3.1: E-Mail Header Injection - Front-End

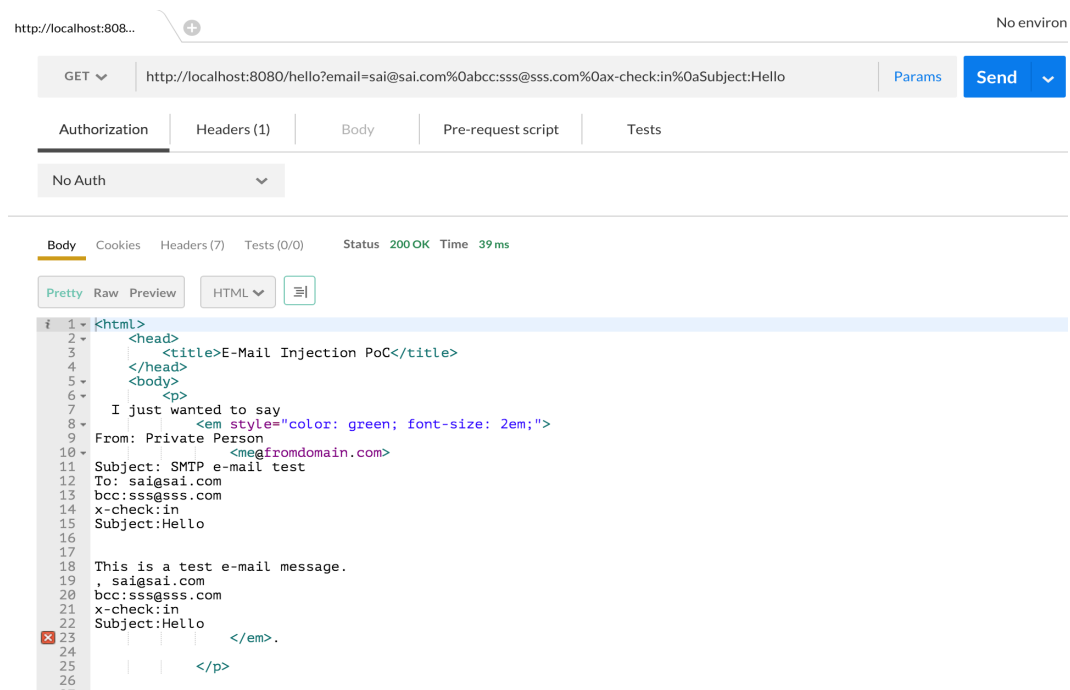


Figure 3.4: Fuzzed Request for Ruby

3.6 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.

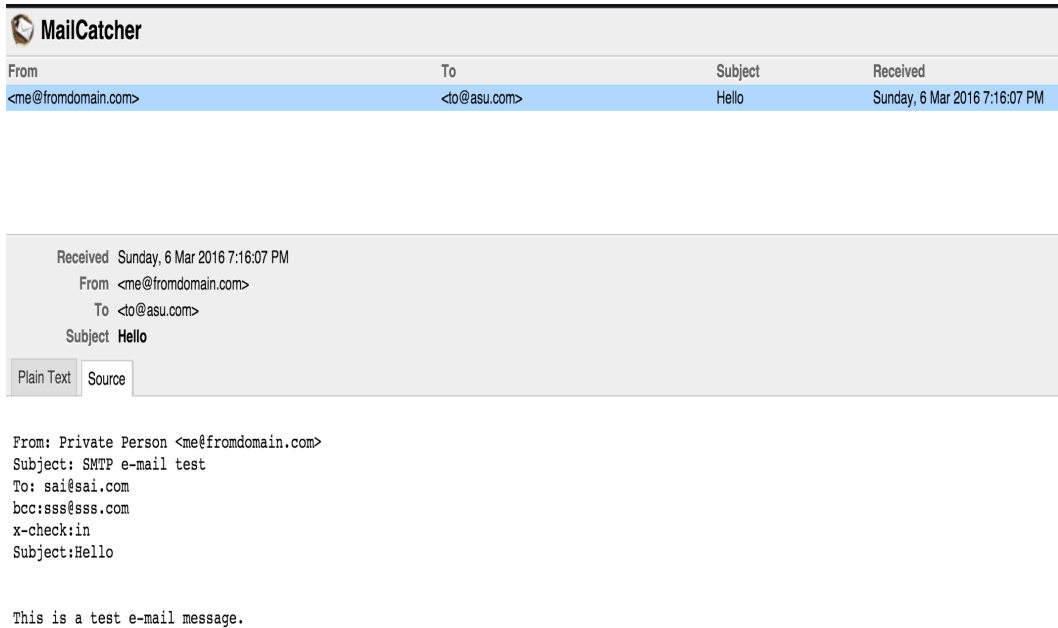


Figure 3.5: E-Mail Header Injection Proof of Concept - Ruby

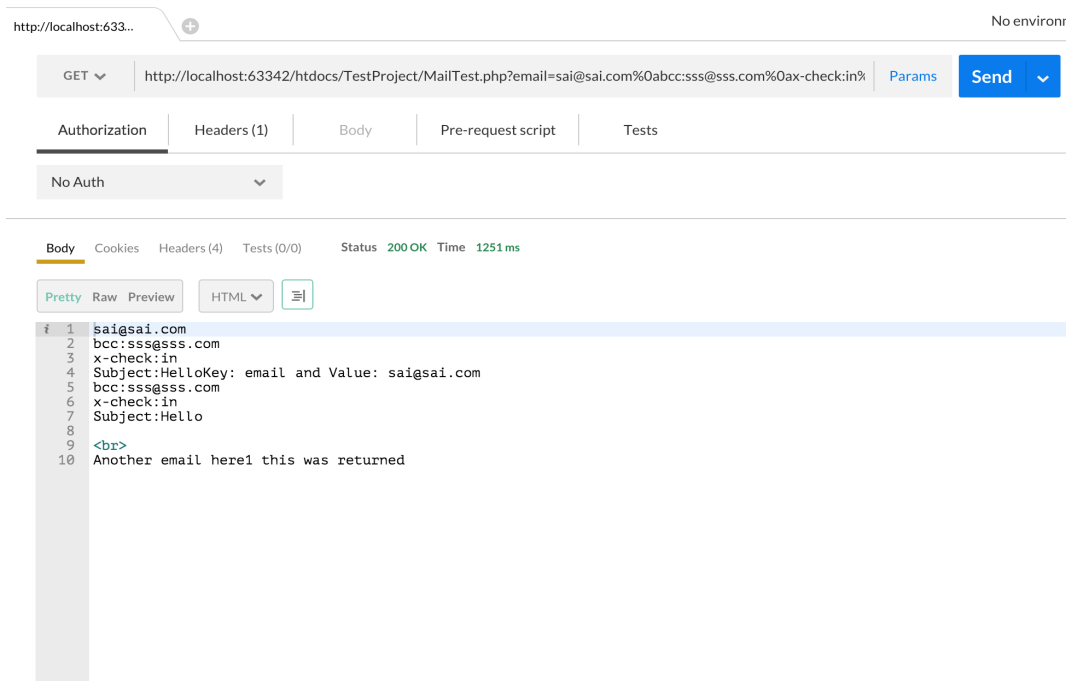


Figure 3.6: Fuzzed Request for PHP

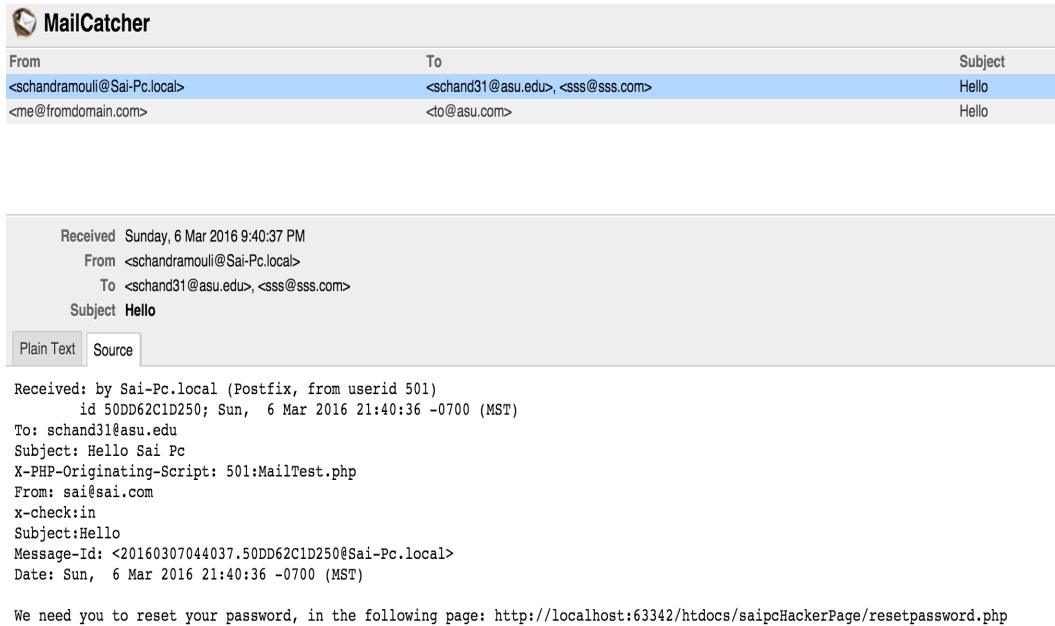


Figure 3.7: E-Mail Header Injection Proof of Concept - PHP

- 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. 3.2.

```

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 3.2: E-Mail Field Checker - False Positives

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

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 3.3), 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>  
3      Resulting URL = www.website.com/mail.php and NOT www.  
        website.com//mail.php
```

Listing 3.3: Issues - URL Construction

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

¹A 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.

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 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 — BeautifulSoup — 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

²We 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.

- 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.

3.7 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’ (Wikipedia (2016b)), the ‘Law of big numbers’ (Wikipedia (2016c)), and the concept of

‘Regression to the mean’ (Wikipedia (2016f)). 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 in, the test cases for our modules, followed by proof of concepts of our attack in multiple languages. We also discussed the issues faced, and the assumptions that we made while building the system. The next chapter describes, in brief, the experimental setup we used for our system.

Chapter 4

EXPERIMENTAL SETUP

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.

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

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 and their functions

Despite the many benefits that Python 2 provides, we had certain issues with the language — discussed in Section 3.6 — 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 (Celery (2016)):

“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 (Wikipedia (2016e)),

“RabbitMQ is an open source message broker software (sometimes called message-oriented middleware) 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 the vanilla Python.

The next chapter presents our results and showcases our analysis of the said results.

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

This section complements Section 3.6, 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 3.6, 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 (Yan and Ahmad (2007), Mori and Malik (2003) 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 3.6, we found that certain WordPress plugins also prevent the E-Mail Header Injection attack by sanitizing user input on Contact Forms. Some of these are discussed in the following section.

- Blacklisting by websites and ISPs

During the actual crawl, we ended up getting blacklisted by a few websites, and Internet Service Providers (ISPs)

- Mail libraries

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 to prevent E-Mail Header Injection

- Use a Content Management System (CMS)

Content management systems like WordPress and Drupal include certain libraries 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.

¹PEAR Mail Website: <https://pear.php.net/package/Mail>

²PHPMailer Website: <https://github.com/PHPMailer/PHPMailer>

³Swiftmailer Website: <http://swiftmailer.org/>

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

⁵Apache Commons E-Mail: <https://commons.apache.org/proper/commons-email/>

⁶Ruby Mail Website: <https://rubygems.org/gems/mail>

⁷Contact Form 7 Download: <https://wordpress.org/plugins/contact-form-7/>

- Input Validation

If neither of the two options above is feasible, due to reasons such as the website 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 (Resnick (2008)), 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

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).
- Berners-Lee, T., L. Masinter and M. McCahill, “Internet Message Format - RFC 1738”, URL <https://www.ietf.org/rfc/rfc1738.txt> (2008).
- 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).
- Celery, “Celery Homepage”, URL <http://www.celeryproject.org/> (2016).
- 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).
- 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).
- Murray, D., “Email.header.Header too lax with embedded newlines”, URL <http://bugs.python.org/issue5871> (2009).
- Nicol, J., “Securing PHP Contact Forms”, URL <http://jonathannicol.com/blog/2006/12/09/securing-php-contact-forms/> (2006).
- 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).
- Python, “Python Global Interpreter Lock”, URL <https://wiki.python.org/moin/GlobalInterpreterLock> (2016c).
- 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).
- RubySec, “Mail Gem for Ruby vulnerable to SMTP Injection via recipient email addresses”, URL <http://rubysec.com/advisories/OSVDB-131677/> (2015).
- 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).
- Toboza, “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, “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] (2016b).
- 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] (2016c).
- Wikipedia, “Newline — Wikipedia, the free encyclopedia”, <http://en.wikipedia.org/w/index.php?title=Newline&oldid=704759213>, [Online; accessed 05-March-2016] (2016d).
- Wikipedia, “RabbitMQ — Wikipedia, the free encyclopedia”, <http://en.wikipedia.org/w/index.php?title=RabbitMQ&oldid=708777848>, [Online; accessed 09-March-2016] (2016e).
- 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] (2016f).
- 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).

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

APPENDIX A