E-jection fraction - Tracking how your website pumps out E-Mails

Ben Trovato*
Institute for Clarity in Documentation
1932 Wallamaloo Lane
Wallamaloo, New Zealand
trovato@corporation.com

G.K.M. Tobin†
Institute for Clarity in Documentation
P.O. Box 1212
Dublin, Ohio 43017-6221
webmaster@marysville-ohio.com

Lars Thørväld‡
The Thørväld Group
1 Thørväld Circle
Hekla, Iceland
larst@affiliation.org

Lawrence P. Leipuner
Brookhaven Laboratories
Brookhaven National Lab
P.O. Box 5000
lleipuner@researchlabs.org

Sean Fogarty
NASA Ames Research Center
Moffett Field
California 94035
fogartys@amesres.org

Charles Palmer
Palmer Research Laboratories
8600 Datapoint Drive
San Antonio, Texas 78229
cpalmer@prl.com

ABSTRACT

E-Mail header injection vulnerability is a class of vulnerability that can occur in web applications that use user input to construct e-mail messages. E-Mail injection is possible when the mailing script fails to check for the presence of e-mail headers in user input (either form fields or URL parameters). 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, allowing an attacker to completely alter the content of the e-mail.

This thesis develops a scalable mechanism to automatically detect E-Mail Header Injection vulnerability and uses this mechanism to quantify the prevalence of EMail Header Injection vulnerabilities on the Internet. Using a black-box testing approach, the system crawled 21,675,680 URLs to find URLs which contained form fields. 6,794,917 such forms were found by the system, of which 1,132,157 forms contained e-mail fields. The system used this data feed to discern the forms that could be fuzzed with malicious payloads. Amongst the 934,016 forms tested, 52,724 forms were found to be injectable with more malicious payloads. The system tested 46,156 of these, and was able to find 496 vulnerable URLs across 222 domains, which proves that the threat is widespread and deserves future research attention.

*Dr. Trovato insisted his name be first.

†The secretary disavows any knowledge of this author's actions.

‡This author is the one who did all the really hard work.

WOODSTOCK '97 El Paso, Texas USA

DOI: 10.475/123_4

CCS Concepts

•**Computer systems organization** → **Embedded systems;** *Redundancy;* Robotics; •**Networks** → Network reliability;

Keywords

ACM proceedings; LaTeX; text tagging

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's 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 having control over the e-mail communication sent by websites to users is of an enormous magnitude.

Since attackers typically masquerade themselves as 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 such as SQL injection or cross-site scripting (XSS) [24] 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 [14]. 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 — with the potential to alter the contents of the e-mail message — while still appearing to be from a legitimate source.

E-Mail Header Injection attacks have the potential to allow an attacker to perform e-mail spoofing, resulting in phishing attacks that can lead to identity theft. The objective of our research is to study the prevalence of this vulnerability on the World Wide Web, and identify whether further research is required in this area.

We performed an expansive crawl of the web, extracting forms with e-mail fields, and injecting them with different payloads to infer the existence of E-Mail Header Injection vulnerability. We then audited 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. The entire system works in a black-box manner, without looking at the web application's source code, and only analyzes the e-mails we receive based on the injected payloads.

We hope that our research sheds some light on this relatively less well-known 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 a crawl of the Web, including 21,675,680 URLs and 6,794,917 forms.

2. 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], [11], [31], Cross-Site Scripting (XSS) [16], [19] or even HTTP Header Injection [17], 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 script that constructs e-mails from user input fails to check for the presence of e-mail headers in the user input, a malicious user — using a well-crafted payload — can control the headers set for this particular e-mail. This can be leveraged to enable malicious attacks, including, but not limited to, spoofing, phishing, etc.

2.1 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 [36] 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 [20] and an accompanying wiki article [10] describe the attack vector and outline a few defense measures for the same.

As 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.

An example of the vulnerable code written in PHP is shown in Listing 1. This code takes in user input from the PHP superglobal `$_REQUEST['email']`, and stores it in the variable `$from`, which is later passed to the `mail()` function to construct and send the e-mail.

```
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 // example attack string to be injected as
      the value for
7 // $_REQUEST['email'] => 'sai@sai.com\nCC:
      spc@spc.com'
8 $retValue = mail($to, $subject, $message, "
      From: $from");
9 // E-Mail gets sent to both schand31@asu.
      edu AND spc@spc.com
```

Listing 1: PHP program with e-mail header injection vulnerability.

When this code is given the malicious input `sai@sai.com \nBCC:spc@spc.com` as the value of the `$_REQUEST['email']`, it generates the SMTP Headers shown in Listing 2. It can be seen that the `CC` (carbon copy) header that we injected appears as part of the resulting SMTP message. This will make the e-mail get sent to the e-mail address specified as part of the `CC` as well.

```
1 Received: from mail.ourdomain.com
      ([62.121.130.29])
2          by sai.com (Postfix) with ESMTP id
             5A08E52C0154
3          for <sai@sai.com>; Sun, 20 Mar 2016
              13:56:58 -0700 (MST)
4 To: sai@sai.com
5 Subject: Hello Sai Pc
6 CC: spc@spc.com
7 Date: Sun, 20 Mar 2016 13:56:58 -0700 (MST)
8
9 We need you to reset your password
```

Listing 2: SMTP headers generated by a PHP mailing script.

2.2 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** - 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 [38], PHP is used by 81.9% of all the websites in existence, thereby creating the possibility of this vulnerability to be widespread.

  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 after 15 years. However, it is specified in the PHP documentation [27] that 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 1.

- **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 such as *email.parser*, where both the classic *Parser()* and the newer *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 3.

```
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'
# for form["email"]

# 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 3: Python program with e-mail header injection vulnerability.

- **Java** - Java has a bug report about E-Mail Header Injection filed against its JavaMail API. A detailed write-up by Alexandre Herzog [12] is 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.

2.3 Potential Impact

The impact of the vulnerability can be pretty far-reaching. Table 1 shows the current server-side language usage statistics on the Web [38].

Server Side Language	% of Usage
PHP	81.9
Java	3.1
Ruby	0.6
Python	0.2

Table 1: Language usage statistics compiled from w3techs [38].

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

- Phishing and Spoofing Attacks
  Phishing [26] (a variation of spoofing [44]) refers to an attack where the recipient of an e-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, as it is sent right from the websites and people are more ready to trust e-mail that is received from the website directly and can thus result in more successful phishing attacks.

- Spam Networks
  Spam networks can use E-Mail Header Injection vulnerabilities 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, recipient e-mail clients might not flag them as spam. If they do flag them as spam, then that can lead to the website being blacklisted as a spam generator.

- 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 other attacks.

- Denial of service by attacking the underlying mail server
  Denial of service attacks (DoS), 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 E-Mail Header Injection is a critical vulnerability that web applications must address.

3. SYSTEM DESIGN

[1] A website may use more than one server-side programming language

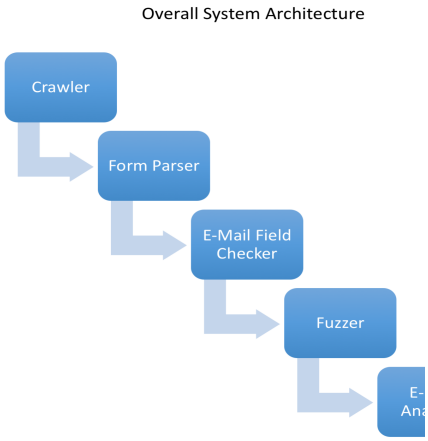

Figure 1: Overall system architecture - logical overview.

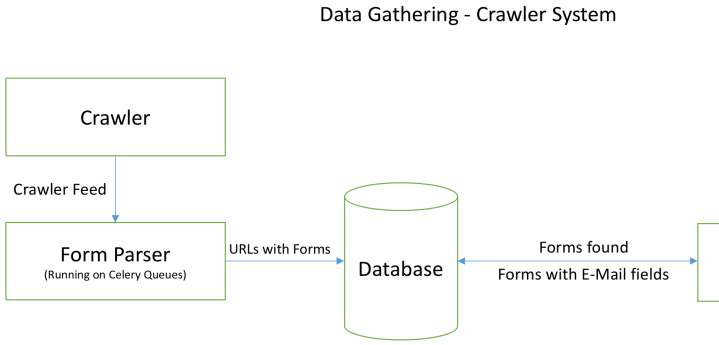

Figure 2: System architecture - crawler & form parser.

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 measure prevalence of E-Mail Header Injection

We took a black-box approach to measure the prevalence of E-Mail Header Injection vulnerability on the World Wide Web. Black-box testing [39] is a way to examine the functionality of an application without looking at its source code.

As we did not have the source code for each of these websites, black-box testing was the ideal approach for this project. Black-box testing allows our system to detect E-Mail Header Injection vulnerabilities in any server-side language (not simply those we identified in Section 2.2). A high level logical overview of our system is presented in Figure 1. The components shown in Figure 1 can be more broadly categorized into different modules, as discussed in the following section.

3.2 System Architecture

The black-box testing system can be divided broadly into two modules; Data Gathering and Payload Injection.

1. Data Gathering
   The Data Gathering module (shown in Figure 2) is primarily responsible for 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) 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 (Section 3.3). The Payload Injection pipeline is not a linear, but cyclic process, as we inject different payloads and analyze the received e-mails.

3.3 System Components

The Data Gathering module and Payload Injection module are made up of a number of smaller components. This section describes in detail the functionality of each of the components.

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 sent 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 and action. These dictate where we send the HTTP request and what kind of request it is (GET or POST).

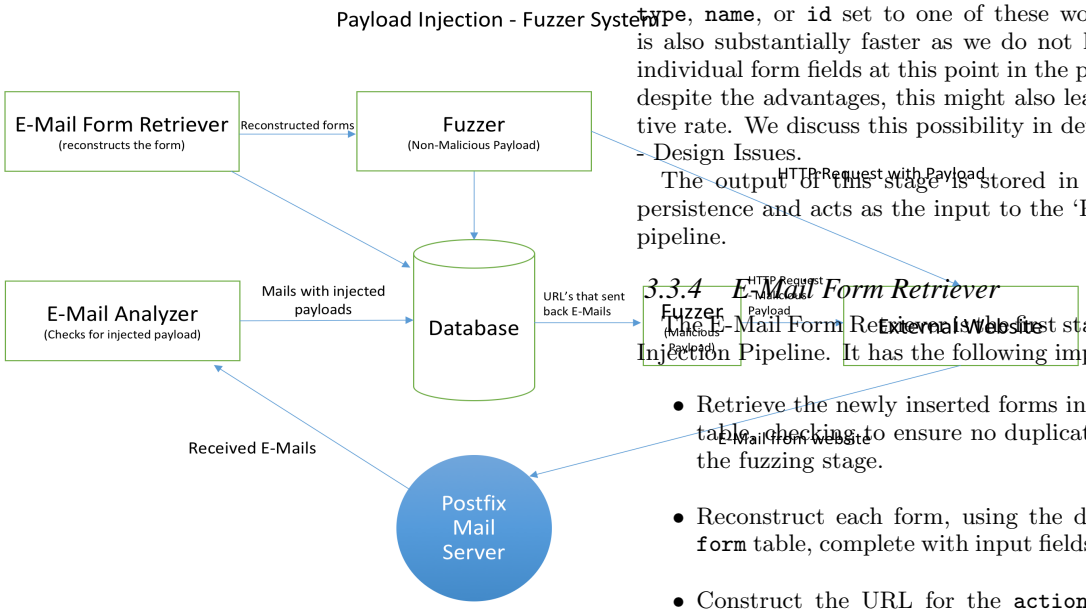

Figure 3: System architecture - fuzzer & e-mail analyzer.

- 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 in the HTML, 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 database, so as to allow us to reconstruct the forms later for fuzzing, as required.

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 separates the forms that are potentially vulnerable from the forms that are not.

The E-Mail Field Checker searches for the words `e-mail`, `mail` or `email` within the form, instead of an explicit e-mail field (e.g., <input type="email">). This is by design, taking into account a very common design pattern used by web developers, 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, we are assured very few false negatives. This is because our system is bound to find e-mail fields with their `type`, `name`, or `id` set to one of these words. The system is also substantially faster as we do not have to parse the individual form fields at this point in the pipeline. However, despite the advantages, this might also lead to a false positive rate. We discuss this possibility in detail 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 has the following 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 correct URL.

3.3.5 Fuzzer

The Fuzzer is the heart of the system and is the only component that interacts directly with the external websites. The Fuzzer is split into smaller modules, each of which is responsible for a particular type of fuzzing. We inject payloads in two different stages, to improve the efficiency, and reduce the total number of HTTP requests we generate. This is because making HTTP requests is an expensive process [21], and can be a cause of bottlenecks in a Crawler-Fuzzer system [32]. 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(xxxx)@example.com`, where `xxxx` is replaced by our internal `form_id`, 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 was `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, without attempting to fuzz the 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 a copy of the e-mail to the e-mail address we added as part of the `bcc` field.

We consider a special case: the addition of a `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. To overcome this, we need to inject a new header that is not likely to be generated by the web application. Hence,

we inject our own `x-check:in` 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 'new-line' 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: `\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:in` header is to inject Python based websites.

4. `nuser(xxxx)@wackopicko.com\r\nbcc:maluser(xxxx)@wackopicko.com\r\nx−check:in`

   - Same as the previous payload, but containing the additional `\r` for Windows compatibility.

The `xxxx` in all of the payloads is replaced by our internal `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 Table 2.

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

Table 2: Payload coverage, each payload covers a different platform/language.

Along with the payload, the Fuzzer also injects data into the other fields of the form. This data must 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`, and `submit`. The default values for these are generated on-the-fly for each form, based on generally followed guidelines for such fields. For example, 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.

Analyzing regular e-mail.

'Regular e-mail' refers to the e-mails received by the reguser(xxxx)@wac — where `xxxx` is our internal `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 identify if 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.

Analyzing e-mail with payloads.

The 'e-mails with payloads' refer to e-mails received by either the nuser(xxxx)@wackopicko.com or maluser(xxxx)@wackopicko.co 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 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 exploit of the vulnerability, 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 along with the data it is designed to hold. A schema of the database is shown in Figure 4.

S.No	Table Name	Purpose
1	form	To hold data about all the forms that we receive 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 of the form, including their default values.
3	params	Holds the actual input
4	fuzzed_forms	Holds the data of the
5	received_emails	Contains data about
6	successful_attack_emails	Contains data about
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: The different tables in our database.

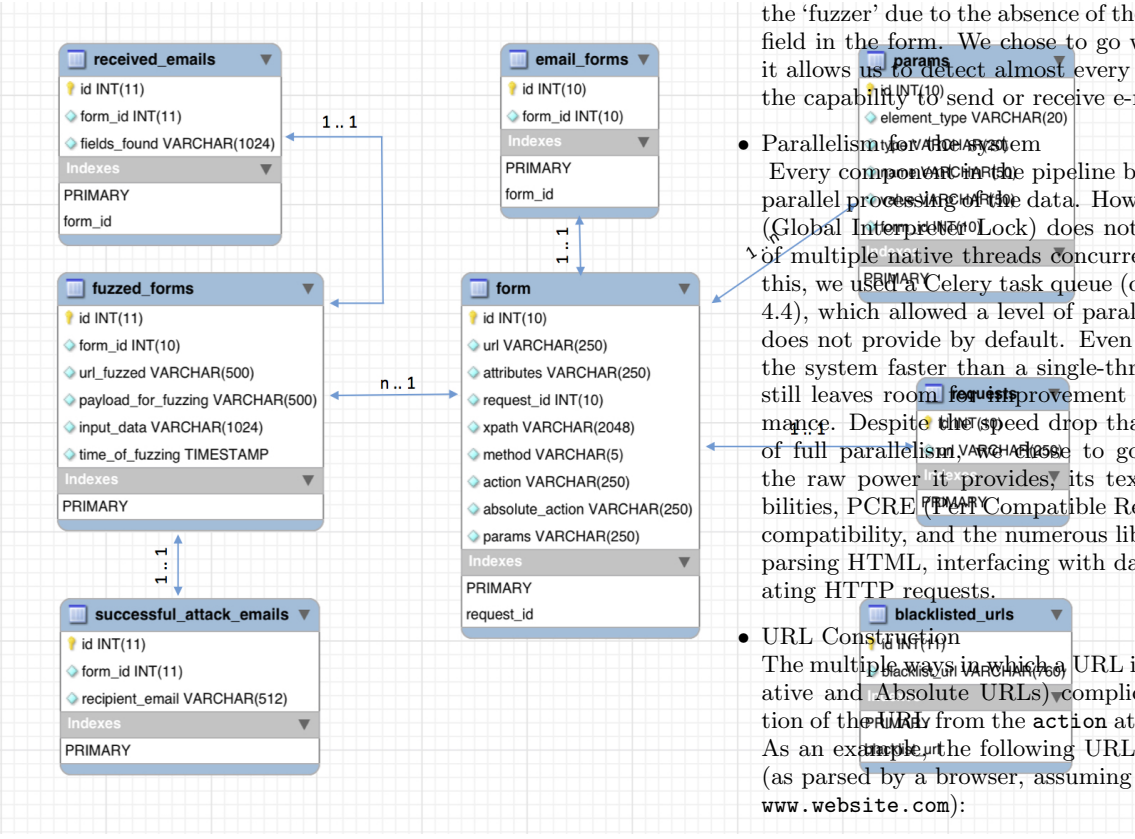

Figure 4: Database schema.

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 false positives in certain forms, like the one shown in Listing. 4.

```
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 4: 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 significant 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 almost every form that provides the capability to send or receive e-mail.

- 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 level of parallelism that 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 performance. Despite the speed drop that results from lack of full parallelism, 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 available for parsing HTML, interfacing with databases and generating HTTP requests.

- 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 [2], 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 5), 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 5: URL construction, the resulting url needs to be www.website.com/mail.php and not www.website.com//mail.php

As using a browser engine to reconstruct these URLs and connecting it to the fuzzer pipeline would have added unnecessary bulk to the project, we chose to go with a best-effort approach to this problem, where our system covers all these possibilities with a lightweight URL Generator, 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 (e.g., the website allows users to send bulk e-mail, adding as many `cc` or `bcc` headers). We have to manually e-mail the developers to get this feedback.

- Mapping responses to requests
  As 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 map responses to requests accurately.

- Bot Blockers
  Because 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 [29], [37].

  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 will not test 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 [3].

- 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, because we 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.

3.5 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' [40], the 'Law of big numbers' [41], and the concept of 'Regression to the mean' [43]. 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

[2] 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.

[3] 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.

exists in the application. We do not inject any additional payloads that can modify the subject, message body, etc. as this analysis is designed to be as benign as possible. We believe that this is a reasonable assumption, as altering e-mail headers is a goal of exploiting E-Mail Header Injection 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.

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.

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: 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 5) for our system.

Despite the many benefits that Python 2 provides, we had certain issues with the language — discussed in Section 3.4 — such as 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.

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

Table 5: Libraries that we used and their functions.

4.4 Celery Queues

We used a Celery task queue running on 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 [42],

> "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 traditional 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 is not 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 accepted user input to construct and send an E-Mail.

The front-end for each of the three applications is shown in Listing 6. The server-side code for the three languages are shown in Listings 1, 3, and **??**.

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 5. The e-mail sent due to injecting this payload (as captured by MailCatcher) is shown in Figure 6. 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 7 and the corresponding e-mail caught by MailCatcher is shown in Figure 8. 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 in th URL 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 Python 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 6: HTML page for showcasing e-mail header injection, a simple front-end for our examples.

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 6. The following paragraphs describe in detail what each kind of data represents, and what they signify.

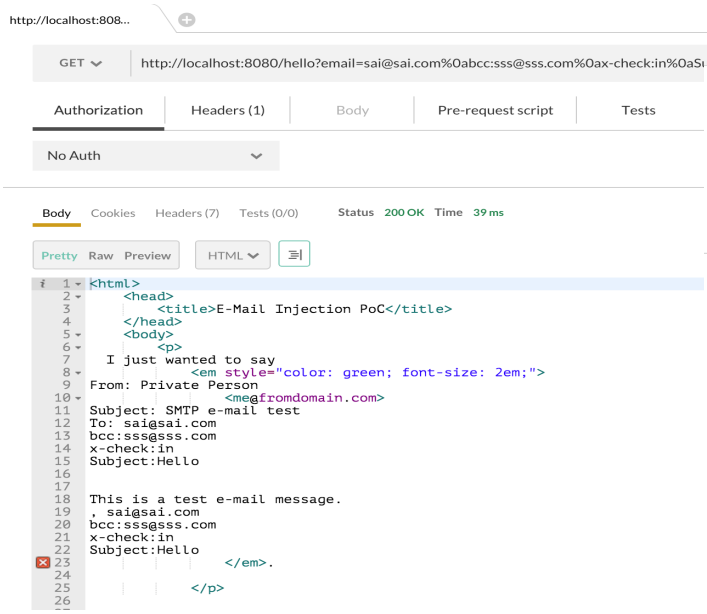

Figure 5: Fuzzing a request for the Ruby backend, the payload can be seen inside the address bar.

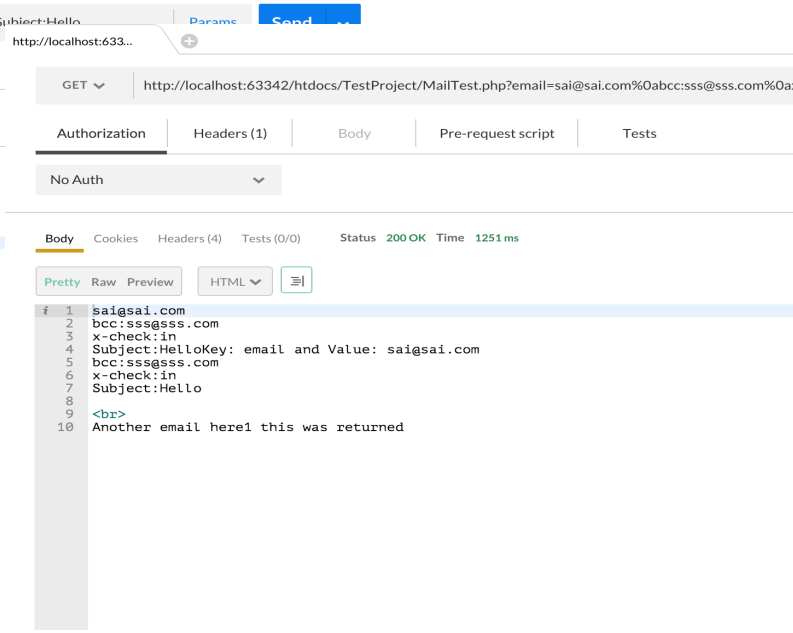

Figure 7: Fuzzing a request for the PHP backend, the payload can be seen inside the address bar on top.

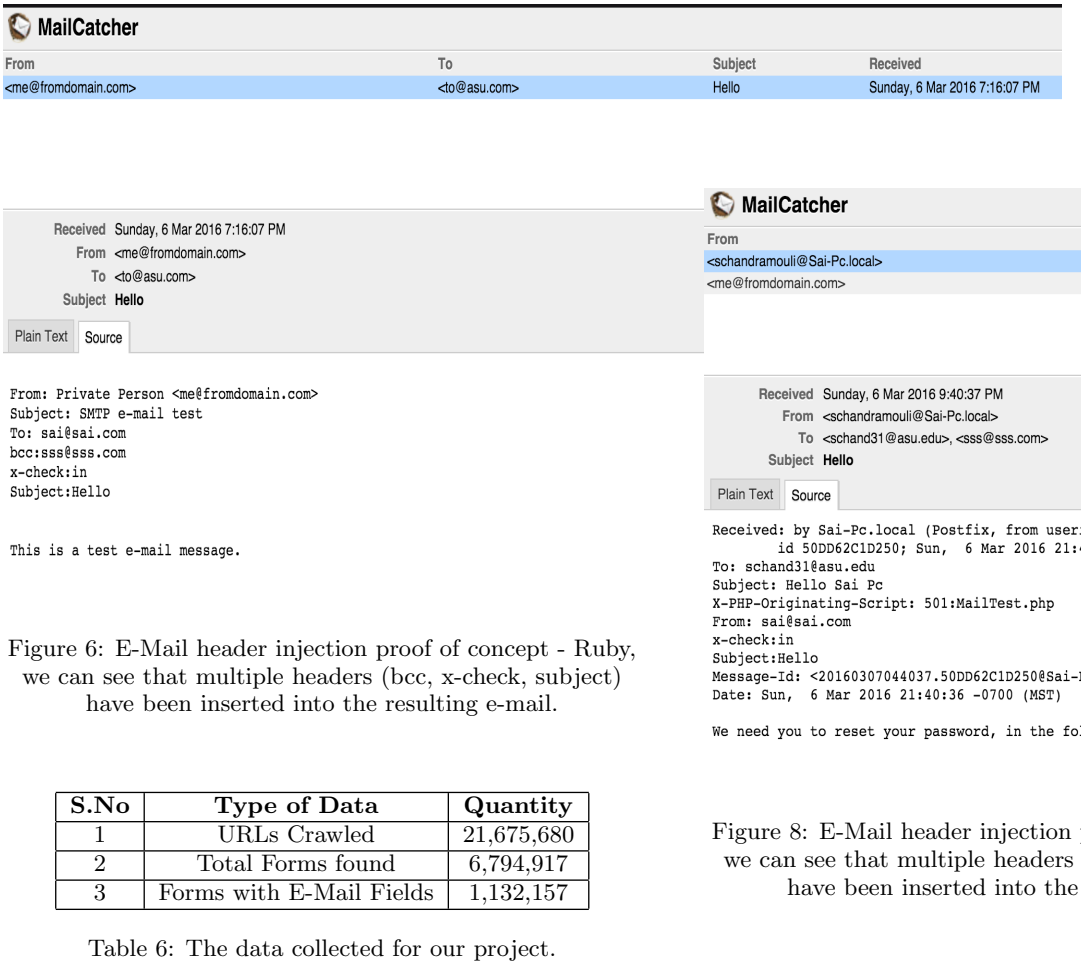

Figure 6: E-Mail header injection proof of concept - Ruby, we can see that multiple headers (bcc, x-check, subject) have been inserted into the resulting e-mail.

S.No	Type of Data	Quantity
1	URLs Crawled	21,675,680
2	Total Forms found	6,794,917
3	Forms with E-Mail Fields	1,132,157

Table 6: The data collected for our project.

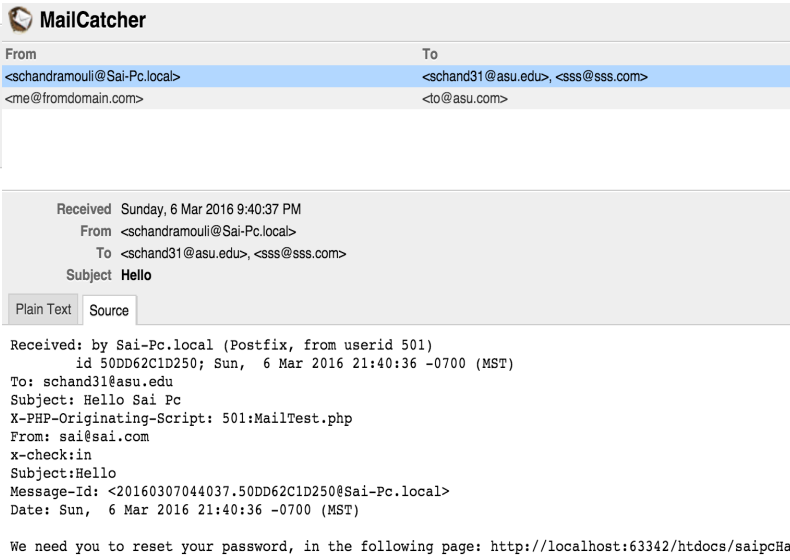

Figure 8: E-Mail header injection proof of concept - PHP, we can see that multiple headers (bcc, x-check, subject) have been inserted into the resulting e-mail.

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 unique 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 6,794,917 forms from 1,019,921 unique domains.

Forms with E-Mail Fields.
This represents the total number of forms that were found to contain e-mail fields. We found 1,132,157 such forms from 197,570 unique domains.

5.2 Fuzzed data

We performed our fuzzing attempts on the gathered data with both the regular payload and malicious payload. Table 7 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 sections.

S.No	Type of fuzzing	Forms fuzzed	E-Mails received
1	Regular payload	934,016	52,724
2	Malicious payload	46,156	496

Table 7: 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 e-mails to us. This indicates that we were able to successfully submit the forms on these sites.

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 unique URLs that contain E-Mail Header Injection vulnerability.

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 265 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 was 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 leak information, and/or perform DoS (Denial of Service) attacks.

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, allowing us to check whether headers other than the `bcc` header can be injected into the generated e-mail.

We list each category and the number of e-mails received by the category in Table 8.

S.No	Type of Injection	No
1	E-Mail Header Injections with `bcc` header	
2	E-Mail Header Injections with `x-check` header	
3	`To header` injections alone	
4	E-Mail Header Injections with `bcc` and `x-check` headers	
5	Both `To header` injections and x-check headers	
6	`x-check` headers found in `nuser` e-mails	
7	Unique `x-check` headers found in `nuser` e-mails	
8	Total successful injections (1 + 3 + 7)	

Table 8: 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 following 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.

- Total successful injections
  This represents the total number of successful injections our system made. This includes the E-Mail Header Injections with `bcc` header (1), `To` header injections alone (3), and Unique `x-check` headers found in `nuser` e-mails (7). This is the total number of vulnerabilities that were found by our system.

5.4 Understanding the pipeline

This section serves to represent our pipeline quantitatively and graphically. Table 9 showcases the data gathered by our pipeline, with the differential changes at each stage of the pipeline.

At each stage of the pipeline, the amount of data trickles down, for instance, out of the 21,675,680 URLs we crawled, only 6,794,917 forms (31.35%) were found. Out of these, only 1,132,157 forms (16.66%) contained e-mail fields.

In our fuzzing attempts, the same behavior is repeated. We fuzzed 934,016 forms with the regular payload, which resulted in a total of 52,724 e-mails (5.64%). After analysis of the received e-mails, we further fuzzed 934,016 forms, which resulted in 496 e-mails (1.07%) which contain the vulnerability.

S.No	Pipeline Stage	Quantity	Differential $\Delta \frac{d2}{d1} * 100$
1	Crawled URLs	21,675,680	—
2	Forms found	6,794,917	31.35%
3	E-Mail Forms found	1,132,157	16.66%
4	Fuzzed with regular payload	934,016	82.50%
5	Received e-mails	52,724	5.64%
6	Fuzzed with malicious payload	46,156	87.54%
7	Successful attacks	496	1.07%

Table 9: Data gathered by our pipeline at each stage, with the differential between the stages.

Thus, our pipeline can be visualized as a funnel (shown in Figure 9), where the quantity decreases until it reaches the end product.

From our research, it is clear that E-Mail Header Injection is quite widespread as a vulnerability, appearing on 1.07% of forms that we were able to perform automated attacks on. This value acts as a lower bound for E-Mail Header Injection vulnerability, and can quite easily be much more if the attacks were of a more concentrated nature, crafted for the individual websites and less automated. We discuss this possibility, and other concepts such as limitations in the following 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.

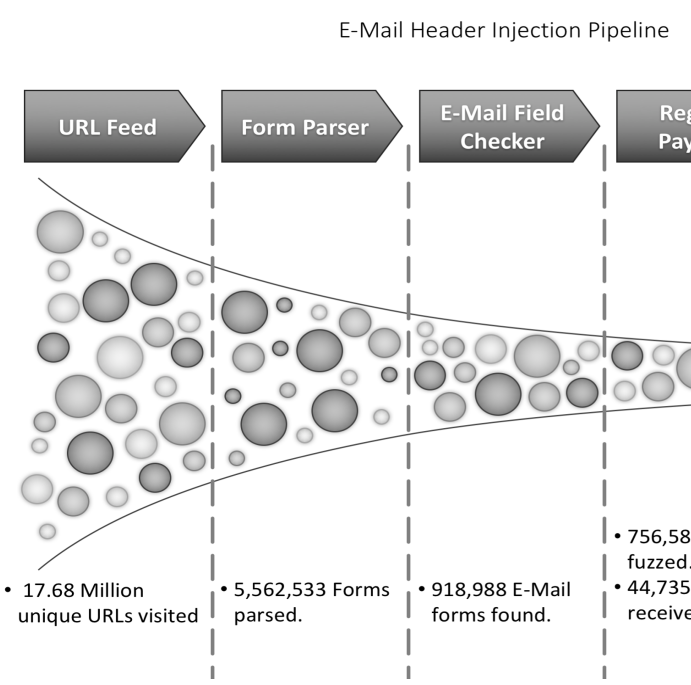

Figure 9: Pipeline - shows the quantity of data gathered at each stage of the pipeline.

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 — 1,018,863,952 according to Internet Live Stats [15] as of early 2016 — and calculate 1.07% percent (the occurrence rate of E-Mail Header Injection vulnerability as found by our system) of that number, we end up with 10,709,390 websites - a pretty significant number. We agree that an extrapolation of that kind might not be an accurate 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 these websites, and also on overall World Wide Web due to the traffic caused by the sheer number of generated e-mails.

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 websites with more widespread vulnerabilities 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. SQL Injection attacks and XSS

attacks are also better documented, with well-known attack vectors, and automated tools to help detect the presence of these vulnerabilities on websites.

This also gives more incentive for the website developer to add protection against attacks such as SQL injection and XSS. The developer might then (possibly with the help of a sanitization library) sanitize the user input and remove *all* special characters, including the newline characters (\n, \r), which adversely affects E-Mail Header Injection attacks.

We come to this conclusion because of our discovery of the `To header injection`. Clearly, this is possible due to incomplete sanitization performed by the application. We suspect that this incomplete sanitization is actually sanitization that is performed for some other vulnerability, and not specifically for E-Mail Header Injection attacks. We would also like to remark that `To header injection` is not complete E-Mail Header Injection, but only a special subset.

Thus, indirectly, this kind of protection against other attacks 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 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 large threat. In fact, this vulnerability can also 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 easily extract 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 mail server of 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. We christened this as `To header injection`. 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 some of these cases, able to inject enough data to spoof the first few lines of the e-mail message. From Table 8, information leakage using `To header injection` was possible on 142 forms, while spoofing using `To header injection` was possible on 11 forms.

While not being as impactful as the primary vulnerability, this second form of the vulnerability does still provide the ability to send e-mails to multiple recipients, and can easily result in information leakage or spam generation on a large scale.

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. As CAPTCHAs are designed to be robust, there is no easy way to break them. There has been considerable research in this area [37], [45] to name a few and although not impossible to break, it remains out of the scope of this project, and thus, we chose to ignore websites which require CAPTCHA verification.

- JavaScript Apps - Due to the growing emphasis on responsive web applications, more and more single-page web applications are being built purely with JavaScript. Even conventional applications are now making use of JavaScript to dynamically insert content and update the pages. This means that these dynamically injected components are not a part of the initial source code that is sent to the client 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, to use a JavaScript engine 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, our system was blacklisted by a few websites (mostly WordPress ones), and Internet Service Providers (ISPs). We then created 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 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. A few other libraries for each language are discussed in the following section.

- Websites that are not in English
  Because we are only searching for the words `e-mail`, `mail` or `email` within the form, if the website does not use English names for its forms, our system will not be able to find the presence of an e-mail field. An example is shown in Listing 7. Here, the French word for `e-mail` — courrier Ã'lectronique — is used, and our system is unable to find the presence of the e-mail form.

```
<!doctype html>
<html lang="fr">
<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="courrier é
    lectronique"
        name="courrier_électronique"><br>
<textarea name="msg" rows="20" cols="120"><
    /textarea>
<input type="submit" value="courrier é
    lectronique!">
</form></body>
</html>
```

Listing 7: HTML page with e-mail form, written in a different language - French.

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 10.

  Using libraries such as PEAR Mail, PHPMailer, Apache Commons E-Mail, Contact Form 7, and Swiftmailer can significantly reduce the occurrence of E-Mail Header Injection vulnerability.

Language	Mail Libraries
PHP	PEAR Mail[4], PHPMailer[5], Swiftmailer[6]
Python	SMTPLib with email.header.Header[7]
Java	Apache Commons E-Mail[8]
Ruby	Ruby Mail >= 2.6[9]
WordPress	Contact Form 7[10]

Table 10: 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 10.

- 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 [30], 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 mitigate 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.

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], [13], [18], [25], [46]. There has been significant discussion on both the benefits of such an approach [1] and its shortcomings [8], [9].

Our work does not intend to act as a vulnerability scanner, but as a means to identify the presence of E-Mail Header Injection vulnerabilities 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], [11], [31], Cross-Site Scripting — XSS — [16], [19], HTTP Header Injection [17], and is very closely related to Simple Mail Transfer Protocol (SMTP) Injection [35].

The attack described by Terada [35] 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, e.g. `To` e-mail header has a corresponding `To` SMTP header) to exploit the SMTP server's pipelining mechanism. The paper also describes

[4]PEAR Mail Website: https://pear.php.net/package/Mail
[5]PHPMailer Website: https://github.com/PHPMailer/PHPMailer
[6]Swiftmailer Website: http://swiftmailer.org/
[7]instead of using email.parser.Parser to parse the header
[8]Apache Commons E-Mail: https://commons.apache.org/proper/commons-email/
[9]Ruby Mail Website: https://rubygems.org/gems/mail
[10]Contact Form 7 Download: https://wordpress.org/plugins/contact-form-7/

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 few articles on the Internet describing the attack.

The first documented article dates to over a decade ago; a late 2004 article on phpsecure.info [36] 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 found other blog posts [6], [20], [22], [23], [28], each describing how to exploit the vulnerability by using newlines to camouflage headers inside user input. A wiki entry [10] 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 [34], recounts an actual attack against a WordPress plugin, `Contact Form`, with a proof of concept[11]. 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 described briefly by Stuttard and Pinto in their book, "*The Web Application Hacker's Handbook: Discovering and Exploiting Security Flaws*" [33]. The book, however, does not go into detail on either the attack or the ways to mitigate such an attack. Our work, on the other hand discusses 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 large scale, crawl and inject websites with comparatively benign payloads (such as the BCC header) to identify the existence of this vulnerability without causing any ostensible harm to the website. Our injected payloads do not contain any special characters other than the newline characters and thus cannot cause any unintended consequences. Also, since we are only injecting a payload with the `bcc` header, the underlying mail servers should not be affected by the additional load. Our work serves to not only prove the existence of the vulnerability on the World Wide Web but to quantify it.

8. CONCLUSIONS

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 21,675,680 web pages finding 6,794,917 forms, out of which 1,132,157 forms were fuzzable. We fuzzed 934,016 forms and found 52,724 forms that allowed us to send/receive e-mails. Out of these, we were able to inject malicious payloads into 46,156 forms, identifying 496 vulnerable forms (1.07% success rate). 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.

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

References

[1] J. Bau, 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*, pages 332–345, May 2010.

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

[3] T. Berners-Lee, L. Masinter, and M. McCahill. Internet Message Format - RFC 1738. 2008.

[4] BestWebSoft. Contact Form by BestWebSoft âĂŤ WordPress Plugins, 2016.

[5] S. W. Boyd and A. D. Keromytis. Sqlrand: Preventing sql injection attacks. In *Applied Cryptography and Network Security*, pages 292–302. Springer, 2004.

[6] B. Calin. Email Header Injection Web Vulnerability - Acunetix, 2013.

[7] Celery. Celery Homepage, 2016.

[8] A. Doupé, 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)*, pages 523–538, Bellevue, WA, 2012. USENIX.

[9] A. Doupé, M. Cova, and G. Vigna. Why johnny can't pentest: An analysis of black-box web vulnerability scanners. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 111–131. Springer, 2010.

[10] 2010.

[11] W. G. Halfond, J. Viegas, and A. Orso. A classification of sql-injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, volume 1, pages 13–15. IEEE, 2006.

[12] A. Herzog. Full Disclosure: JavaMail SMTP Header Injection via method setSubject [CSNC-2014-001], 2014.

[13] Y.-W. Huang, 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, pages 148–159, New York, NY, USA, 2003. ACM.

[14] H. H. Injection. HTTP header injection — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=HTTP%20header%20injection&oldid=713295668, 2016. [Online; accessed 18-April-2016].

[15] Internet Live Stats. Total Number of Websites, 2016.

[16] T. Jim, 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, pages 601–610, New York, NY, USA, 2007. ACM.

[17] M. Johns and J. Winter. Requestrodeo: Client side protection against session riding. In *Proceedings of the OWASP Europe 2006 Conference*, 2006.

[18] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic. Secubat: a web vulnerability scanner. In *Proceedings of the 15th international conference on World Wide Web*, pages 247–256. ACM, 2006.

[19] A. Klein. [DOM Based Cross Site Scripting or XSS of the Third Kind] Web Security Articles - Web Application Security Consortium, 2005.

[20] D. Kohler. damonkohler.com: Email Injection, 2008.

[21] D. McGrath. HTTP requests optimization, 2009.

[22] A. Mohamed. PHP Email Injection Example - InfoSec Resources, 2013.

[23] J. Nicol. Securing PHP Contact Forms, 2006.

[24] OWASP. OWASP Top Ten Project, 2013.

[25] P. Payet, 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, March 2013.

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

[27] PHP-Manual. PHP mail - Send mail, 2016.

[28] A. Pope. Prevent Contact Form Spam Email Header Injection | Storm Consultancy âĂŞ Web Design Bath, 2008.

[29] C. Pope and K. Kaur. Is it human or computer? defending e-commerce with captchas. *IT Professional*, 7(2):43–49, Mar 2005.

[30] P. W. Resnick. Internet Message Format - RFC 5322. 2008.

[31] A. Sadeghian, 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*, pages 53–56. IEEE, 2013.

[32] V. Shkapenyuk Torsten Suel. Design and Implementation of a High-Performance Distributed Web Crawler. 2001.

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

[34] V. Tendencies. WordPress Plugin Vulnerability Dump âĂŞ Part 2 | Vexatious Tendencies, 2014.

[35] T. Terada. SMTP Injection via recipient email addresses. *MBSD White Paper*, December 2015.

[36] Tobozo. Mail headers injections with PHP, 2004.

[37] L. von Ahn, M. Blum, and J. Langford. Telling humans and computers apart automatically. *Commun. ACM*, 47(2):56–60, Feb. 2004.

[38] W3techs. Usage Statistics and Market Share of PHP for Websites, February 2016, 2016.

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

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

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

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

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

[44] Wikipedia. Spoofing attack — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Spoofing%20attack&oldid=711407565, 2016. [Online; accessed 23-April-2016].

[45] J. Yan and A. S. E. Ahmad. Breaking visual captchas with naive pattern recognition algorithms. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 279–291, Dec 2007.

[46] S. Zanero, L. Carettoni, and M. Zanchetta. Automatic detection of web application security flaws. *Black Hat Briefings*, 2005.