The Prevalence of E-Mail Header Injections on the World Wide Web

***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 Header 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 such as PHP, Java, Python, and Ruby. With the proper injection string, this vulnerability can be exploited to inject additional headers, modify existing headers, and alter the content of the e-mail.**

This paper presents a scalable mechanism to automatically detect E-mail Header Injection vulnerabilities and uses this mechanism to quantify the prevalence of E-mail Header Injection vulnerabilities on the web. Using a black-box testing approach, the system crawled 21,675,680 URLs to identify web pages which contained form fields. 6,794,917 such forms were found by the system, of which 1,132,157 forms contained e-mail fields. We then tested 934,016 forms to see if they would send us an e-mail, and 52,724 forms sent us an e-mail. Of these, 46,156 forms were tested with E-mail Header Injection payloads and, of these, we found 673 vulnerable URLs across 296 domains. Then, to demonstrate that E-mail Header Injection vulnerabilities are actively being exploited to create a spamming platform, we found 106 IPs that were vulnerable on spamming blacklists. This work shows that E-mail Header Injection vulnerabilities are widespread and deserve future research attention.

I. 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 anyone to access information and services anywhere and on multiple devices such as phones, laptops, personal digital assistants, TVs, and cars. This access has ushered in an era of 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. As users move more and more of their personal and financial information to web applications, attackers are responding by using web application vulnerabilities to access this lucrative data.

Many common and well-known web application vulnerabilities, such as SQL Injection and Cross-Site Scripting [**?**], are command injection vulnerabilities [**?**], where malicious user input is used to alter the structure of a command (a SQL query in the case of SQL Injection and JavaScript code in the case of Cross-Site Scripting). Developers of web applications must have proper sanitization routines in place to use user input as part of these commands.

E-mail Header Injection vulnerabilities are a lesser-known command injection vulnerability. E-mail Header Injection can be considered as the e-mail equivalent of HTTP Header Injection [**?**]. This vulnerability exists in the implementation of the built-in `mail` functionality in popular languages such as PHP, Java, Python, and Ruby. The format of e-mail messages is defined by the Simple Mail Transfer Protocol (SMTP) [**?**]. Each e-mail message is represented by a series of headers separated by newlines, followed by the body content (separated from the headers by two newlines). Some of these headers are mandatory (From, To, Date), but the headers could also include other information like the Subject, CC, BCC, etc.

With the proper injection string, E-mail Header Injection vulnerabilities can be exploited by an attacker to inject additional headers, modify existing headers, or alter the contents of the e-mail—while still appearing to be from a legitimate source. E-mail Header Injection exploits allow an attacker to perform e-mail spoofing, resulting in phishing attacks *that are sent from the actual e-mail server.*

While some command injection vulnerabilities have received extensive attention from the research community, E-mail Header Injection vulnerabilities have received little focus. Therefore, we study the prevalence of E-mail Header Injection vulnerabilities on the web. We performed a crawl of the web, extracted forms with e-mail fields, and injected them with different payloads to infer the existence of an 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. Our automated 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.

In summary, we make the following contributions:

- We develop a black-box approach to detect E-mail Header Injection vulnerabilities in a web application.
- We develop a system to crawl the web and automatically detect E-mail Header Injection vulnerabilities.
- We use our system to crawl 21,675,680 URLs, and we find 673 URLs vulnerable to E-mail Header Injection across 296 domains.

NDSS '16, 21-24 February 2016, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2016.23xxx

```php
$from = $_REQUEST['email'];
$subject = 'Hello XYZ';
$message = 'We need you to reset your
    password';
$to = 'xyz@example.com';

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

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

```
Received: from mail.ourdomain.com
    ([62.121.130.29])
  by xyz.com (Postfix) with ESMTP id 5
     A08E52C0154
  for <abc@example.com>; Sun, 20 Mar 2016
     13:56:58 -0700 (MST)
From: abc@example.com
To: xyz@example.com
Subject: Hello XYZ
CC: spc@example.com
Date: Sun, 20 Mar 2016 13:56:58 -0700 (MST
    )

We need you to reset your password
```

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

II. Background

E-mail Header Injection belongs to a broad class of vulnerabilities known as command injection vulnerabilities. However, unlike its more popular siblings, SQL injection [**?** **?** **?**], Cross-Site Scripting [**?** **?**], or HTTP Header Injection [**?**], relatively little research is available on E-mail Header Injection vulnerabilities.

As with other vulnerabilities in this class, E-mail Header Injection is caused due to improper or nonexistent sanitization of user input. If the program 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. E-mail Header Injection vulnerabilities can be leveraged to enable malicious attacks, including, but not limited to, spoofing or phishing.

A. History of E-mail Header Injection

We found the first E-mail Header Injection description in a late 2004 article on phpsecure.info [**?**] accredited to user `tobozo@phpsecure.info` describing how an E-mail Header Injection vulnerability existed in the implementation of the `mail()` function in PHP and how it can be exploited. More recently, a blog post by Damon Kohler [**?**] and an accompanying wiki article [**?**] describe the attack vector and outline few defense measures for E-mail Header Injection vulnerabilities.

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.

When this code is given the malicious input `abc@example.com\nCC:spc@example.com` as the value of the `$_REQUEST['email']`, it generates the equivalent 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.

B. Languages Affected

This section describes the popular programming languages that contain E-mail Header Injection vulnerabilities in their standard email libraries. 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 vulnerable.

PHP was one of the first languages found to be vulnerable to E-mail Header Injection in its implementation of the `mail()` function at the time of release of PHP 4.0. 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 [**?**], PHP is used by 81.9% of all the websites.

After 13 further iterations of PHP 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 [**?**] that the `mail()` function does not protect against E-mail Header Injection. 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 an E-mail Header Injection vulnerability in Python's implementation of the `email.header` library and the header parsing functions allowing newlines in early 2009, which was followed by a partial patch in early 2011.

Unfortunately, the bug fix was only for the `email.header` package, and thus exists in other frequently used packages such as `email.parser`, where both the classic `Parser()` and the newer `FeedParser()` contain E-mail Header Injection vulnerabilities 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. Contrary to PHP's behavior of overwriting existing headers, Python only recognizes the first occurrence of a header, and ignores duplicate headers. A working code sample of the vulnerability, written in Python 2.7.11, is shown in Listing 3.

Java has a bug report about E-mail Header Injection filed against its `JavaMail` API. A detailed write-up by Alexandre

```
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 =>
# 'abc@example.com\nBCC:spc@example.com'
# for form["email"]

# to:abc@example.com AND bcc:spc@example.
    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.

Herzog [**?**] contains a proof-of-concept program that exploits the API to inject headers.

Ruby From our preliminary testing Ruby's built-in `Net::SMTP` library also has an E-mail Header Injection library. This is not documented on the library's homepage.

C. Impact of E-mail Header Injection

The impact of an E-mail Header Injection vulnerability can be far-reaching. According to w3tech, PHP, Java, Python, and Ruby (combined) account for over 85%[1] of the server-side programming languages in websites measured, and the default implementation of the e-mail functionality of these languages is vulnerable to E-mail Header Injection.

An E-mail Header Injection vulnerability can be exploited to do potentially any of the following:

Phishing and Spoofing Attacks Phishing [**?**] (a variation of spoofing [**?**]) refers to an attack where the recipient of an e-mail is made to believe that the e-mail is legitimate when it was really created by a malicious party. The e-mail usually redirects the victim to a malicious website, which then steals their credentials or infects their computer with malware (via a drive-by-download).

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 from the website's mail server and users (and anti-spam defenses) are more likely to trust an e-mail that is received from the proper mail server. Therefore, attackers could leverage E-mail Header Injection vulnerabilities to perform enhanced phishing attacks.

Spam Networks Spam networks can use E-mail Header Injection vulnerabilities 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 choose the recipient of the spam email.

Due to the e-mail being from trusted domains, recipient e-mail clients and anti-spam systems 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 (which would cause a Denial of Service on the vulnerable web application).

Information Extraction E-mails can contain sensitive data that is meant to be accessed only by the user. Due to an E-mail Header Injection vulnerability, an attacker can add a `bcc` header, and the e-mail server will send a copy of the private e-mail to the attacker, 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 Denial of service attacks (DoS), can also be caused by exploiting an E-mail Header Injection vulnerability. The ability to send many e-mails by injecting one header field can result in overloading the mail server and cause crashes or instability.

III. System Design

To quantify the existence of E-mail Header Injection vulnerabilities on the web at large, we developed a system to automatically detect E-mail Header Injection vulnerabilities in a black-box manner.

A. Approach

We took a black-box approach to measure the prevalence of E-mail Header Injection vulnerability on the web. Black-box testing [**?**] is a way to examine the functionality of an application without analyzing its source code.

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 II-B).

The overall architecture of our system is presented in Figure 1. The components shown in Figure 1 are discussed in the following section.

B. System Architecture

Our system can be broadly divided into two modules: Data Gathering and Payload Injection.

The Data Gathering module is responsible for the following activities:

- Interface with the Crawler (Section III-C1) and receive the URLs.
- Parse the HTML for the corresponding URL and store the relevant form data (Section III-C2).

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

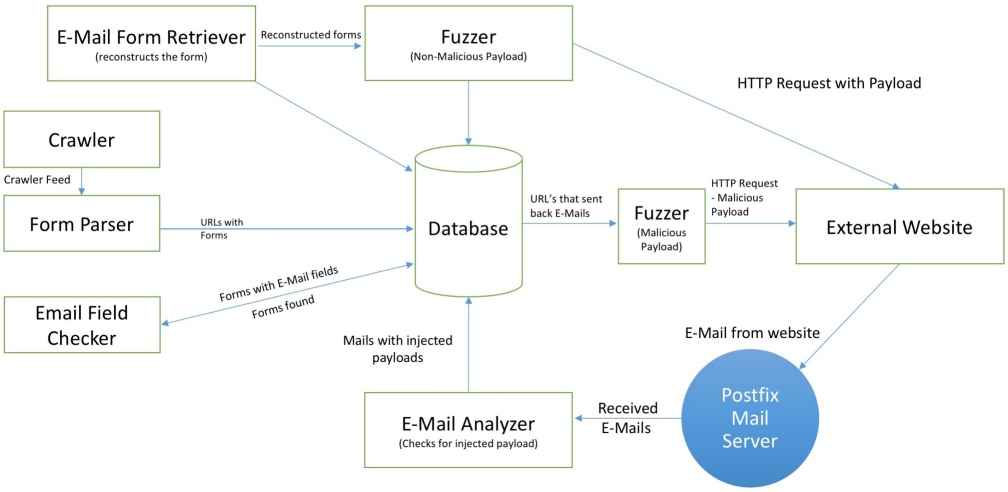

Fig. 1: Overall system architecture.

- Check for the presence of HTML forms that could allow a user to send/receive e-mail and store references to these forms (Section III-C3).

The Payload Injection module is responsible for the following activities:

- Retrieve the HTML forms that could allow a user to send/receive e-mail and reconstruct these forms (Section III-C4).
- Inject these forms with benign data (non-malicious payloads) and generate an HTTP request to the corresponding URL (Section III-C5a).
- Analyze the received e-mails, extracting the e-mail header fields and checking for the presence of the injected payloads (Section III-C6).
- Inject the HTTP requests that sent us e-mails with malicious payloads, and generate an HTTP request to the corresponding URL to check if an E-mail Header Injection vulnerability exists in that web application (Section III-C5b).

C. System Components

The Data Gathering module and Payload Injection modules are composed of smaller components. This section describes the functionality of each of the components.

1) Crawler: We used an open-source Apache Nutch based Crawler [**?**]. The Crawler continuously crawls the web. The Crawler provides the system with a continuous feed of URLs and the HTML contained in those pages.

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

- Form attributes, such as `method` and `action`. These dictate the resulting URL for the HTTP request and the method of the HTTP request.
- Data about the form 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 (if the `action` attribute is a relative URL).

The Form Parser stores all this data in our database, so as to allow the system to reconstruct all data necessary for fuzzing the web application.

3) E-mail Field Checker: The E-mail Field Checker is the final stage in the Data Gathering module. It receives the output of the previous stage—HTML form data—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. The intuition here is that we do not want to try to fuzz all HTML forms on the web to look for E-mail Header Injection vulnerabilities, rather just those HTML forms that are likely to invoke server-side email functionality.

The E-mail Field Checker searches for the words `e-mail`, `mail` or `email` within the form, instead of an explicit HTML5 e-mail field (e.g., `<input type="email">`). This is by design, taking into account a common design pattern used by web developers, where they may have a text field with an `id` or `name` attribute set to `email`, instead of an actual e-mail type attribute, for purposes of backward compatibility with older browsers.

The output of this stage is stored in the database for persistence and acts as the input to the Payload Injection module.

4) E-mail Form Retriever: The E-mail Form Retriever is the first stage in the Payload Injection module. It has the following functions:

- Retrieve new forms and ensure no duplication occurs before the fuzzing stage.
- Reconstruct each form, using the stored form data, specifically the input fields and their values.
- Construct the target URL for the `action` attribute of the form to create an HTTP request to the correct URL for fuzzing.

5) Fuzzer: The Fuzzer is the only component that interacts directly with the external web applications. The Fuzzer is split into smaller parts, each of which is responsible for a particular type of fuzzing. The system injects payloads in two stages: the goal is to reduce the total number of HTTP requests the system generates to detect an E-mail Header Injection vulnerability. Making HTTP requests is an expensive process [**?**], and can cause bottlenecks in a Crawler-Fuzzer system [**?**]. The two different types of payloads used for fuzzing are:

a) Non-Malicious Payload: The regular or non-malicious payload is simply an e-mail address. The goal is to see if the web application will send an e-mail message based on our input. The specific format of the e-mail is `reguser(xxxx)@example.com`, where `xxxx` is replaced by an internal id that uniquely maps the payload to the form, and `example.com` is replaced by our domain.

b) Malicious Payload: After receiving an e-mail from a specific form, we then use the malicious payload to try to exploit an E-mail Header Injection vulnerability. We inject the form parameters with the `bcc` (blind carbon copy) header. 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 the `bcc` field.

We consider a special case: the addition of an `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, Python will ignore duplicate headers. So, if the `bcc` field is already present as part of the headers, our injected `bcc` header would be ignored. To overcome this, we inject a new header that is not likely to be generated by the web application.

We created four different malicious payloads. Each of these payloads is crafted for a particular use case. The four payloads are:

1) `nuser(xxxx)@example.com\n bcc:maluser(xxxx)@example.com`
   This payload is the most minimal payload: it injects a newline character followed by the `bcc` field.
2) `nuser(xxxx)@example.com\r\n bcc:maluser(xxxx)@example.com`
   This payload is added for purposes of cross-platform fuzzing: `\r\n` is the "Carriage Return - New Line (CRLF)" used on Windows systems. [**?**]

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

TABLE I: Payload coverage, each payload covers a different platform/language.

3) `nuser(xxxx)@example.com\n bcc:maluser(xxxx)@example.com\nx-check:in`
   As discussed previously, the addition of the `x-check:in` header is to inject Python-based web applications.
4) `nuser(xxxx)@example.com\r\n bcc:maluser(xxxx)@example.com\r\nx-check:in`
   Same as the previous payload, but containing the additional `\r` for Windows compatibility.

The `(xxxx)` in each payload is replaced by an internal unique id to create a one-to-one mapping of the payloads to the forms. The coverage provided by each payload is shown in Table I.

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). As crawling and fuzzing input fields on the web is an open problem [**?**], we chose to go with a best-effort approach. To maximize the amount of vulnerabilities the system discovers, the data injected into the input fields should adhere to the constraints. The Fuzzer uses a "Data Dictionary" which has predefined "keys" and "values" for standard input fields such as `name`, `date`, `username`, `password`, `text`, and `submit`. The values for these are generated from the Data dictionary 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 special characters.

When the fuzzed data 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 III-C4).

6) Injection Verification: The Injection Verification module 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 account that received the e-mail, it performs different functions.

a) Analyzing regular e-mail: 'Regular e-mail' refers to the e-mails received by account `reguser(xxxx)@example.com` that were sent due to injecting the regular, non-malicious, payload (discussed in Section III-C5a). 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 parse 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 vulnerability for the given e-mail. We then pass on this information back to the Fuzzer pipeline, along with the vulnerable form.

b) Analyzing e-mail with payloads: The "e-mails with payloads" refers to e-mails received by either the `nuser(xxxx)@example.com` or `maluser(xxxx)@example.com` accounts. These e-mails could only be received as a result of injecting the malicious payloads that were discussed in Section III-C5b.

c) Detecting injected `bcc` headers: As discussed in the payloads section (III-C5b), the payloads were crafted such that the e-mails received by the `maluser` account directly indicate the presence of the injected `bcc` field.

d) 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 web application 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 URL.
2) The `bcc` header can be ignored for other reasons (e.g., Python's default behavior when it encounters duplicate headers). In this case, we check if the e-mail contains the custom header `x-check`. If it does, then this is a successful exploit of the vulnerability.

IV. Implementation

Our system was run on a server with the following configuration: Dell PowerEdge T110 II Server, CPU: Intel(R) Xeon(R) CPU E3-1220 V2 @ 3.10GHz, Cache Size: 8,192 KB, No. of Cores : 4, Total Memory (RAM) : 16 GB.

A. Validation

To validate our system, we constructed three sets of web applications in PHP, Python, and Ruby. Each of these applications was a simple web application that accepted user input to construct and send an E-mail.

The front-end for each of the three applications is shown in Listing 4. The server-side code for PHP, Python, and Ruby is shown in Listings 1, 3, and 5 respectively.

Before performing a wide scan of the web, we verified that our system was able to detect the E-mail Header Injection vulnerabilities present in all the sample web applications.

V. Evaluation

We ran our system on the web at large, attempting to discover E-mail Header Injection vulnerabilities in web applications.

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

Listing 4: HTML page for showcasing E-mail Header Injection, a simple front-end for our examples.

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

get '/hello' do
email = params[:email]

message = """
From: Sch <sch@example.com>
Subject: SMTP e-mail test
To: #{email}

This is a test e-mail message.
"""
# construct a post request with email set
    to attack_string
# attack_string => s@example.com%0abcc:
    spc@example.com%0aSubject:Hello
Net::SMTP.start('localhost', 1025) do |
    smtp|
smtp.send_message message, 'sch@example.
    com',
'to@todomain.com'
end
# Headers get added, and Subject field
    changes to what we set.
end
```

Listing 5: Ruby program with e-mail header injection vulnerability.

Type of Data	Quantity
URLs Crawled	21,675,680
Total Forms found	6,794,917
Forms with E-Mail Fields	1,132,157

TABLE II: The data collected for our project.

Type of fuzzing	Forms fuzzed	E-Mails received
Regular payload	934,016	52,724
Malicious payload	46,156	673

TABLE III: The data that we fuzzed and the e-mails that we received.

A. Collected Data

From our extensive crawl of the web, we were able to gather the data shown in Table II. We ran the system for 76 days, during which our system crawled 21,675,680 unique URLs, and found a total of 6,794,917 forms from 1,019,921 unique domains. Out of these forms, our system found 1,132,157 forms that contained an e-mail field, from 197,570 unique domains.

B. Fuzzed Data and Received E-mails

Table III shows the quantity of e-mails that we received for the benign and malicious payloads.

a) E-mail received from forms: The e-mails that we received can be categorized into two categories:

1) E-mails due to regular payload
   This represents the total number of web applications that sent e-mails to us. This indicates that we were able to successfully submit the forms on these sites to trigger the web application to send an e-mail.
2) E-mails due to malicious payload
   Once we receive an e-mail from a web application due to the regular payload, we fuzz those forms with the malicious payloads. This field represents the total number of unique URLs that are contain an E-mail Header Injection vulnerability.

C. 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 categories:

1) E-mails with the `bcc` header successfully injected
   This form of injection was our initial objective, and we found 329 such e-mails in our received e-mails. This validates that the web applications that sent these e-mails are vulnerable to E-mail Header Injection.
2) E-mails with the `to` header successfully injected
   We discovered an unintended vulnerability class during our analysis, which we call `To header injection`. These injections reflect the ability to inject any number of e-mail addresses into the `to` field of the SMTP message while being unable to inject any other header into the e-mails. We found 163 such e-mails in our received e-mails. We attribute this behavior to inconsistent sanitization by the application.
   While not allowing us complete control over content of the e-mails sent, `To header injection` makes it possible to append any number of e-mail addresses, thereby enabling us to leak information or perform DoS (Denial of Service) attacks against the web application.
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 III-C6c, we can differentiate between unsuccessful attempts and successful attempts by injecting the additional header and checking whether headers other than the `bcc` header can be injected into the generated e-mail. 396 e-mails were received with the `x-check` header injected.

We list each category and the number of e-mails received by that category in Table IV. We explain the combination of these header injections (4-7) as follows:

- E-mail Header Injections with both `bcc` and `x-check` headers
  These represent the scenario where an attacker can inject multiple headers into the e-mails. We can see that 65.65% of the received `bcc` header injected e-mails are also susceptible to being injected with additional headers.
- 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 injected 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.
- Total `x-check` headers and unique `x-check` headers found in `nuser` e-mails
  We found a total of 219 e-mails in the `nuser` account. Out of these, 181 had unique form ids that were *not* already found in the `maluser` account. We attribute these e-mails to (probably) being sent by a web application that was built with Python or another language having a similar behavior with respect to ignoring duplicate headers while constructing an e-mail, thus appending the `x-check` header and *not* the `bcc` header.
- Total successful injections
  This represents the total number of successful injections. This includes the E-mail Header Injection with `bcc` header (1), `To` header injections alone (3), and Unique `x-check` headers found in `nuser` e-mails (7). A total of 673 vulnerabilities were found by our system.

D. Understanding the Data Pipeline

Table V 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 decreases, for instance, out of the 21,675,680 URLs we crawled, only

Type of Injection	No. of e-mails received
E-Mail Header Injections with `bcc` header	329
E-Mail Header Injections with `x-check` header	396
`To header` injections alone	163
E-Mail Header Injections with `bcc` and `x-check` headers	216
Both `To header` injections and x-check headers	13
`x-check` headers found in `nuser` e-mails	219
Unique `x-check` headers found in `nuser` e-mails	181
Total successful injections (1 + 3 + 7)	673

TABLE IV: Classification of the e-mails that we received into broad categories of the vulnerability.

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 673 e-mails (1.46%) which contain the vulnerability across 413 IP Addresses corresponding to 296 domains.

E. Responsible Disclosure of Discovered Vulnerabilities

After we discovered an E-mail Header Injection vulnerability on a particular website, we attempted to notify the developers of the vulnerable web application, along with a brief description of the vulnerability. We chose to e-mail the following mailboxes, following the rules specified in RFC 2142 [**?**]:

- `security@domain.com` - Used for Security bulletins or queries.
- `admin@domain.com` - Used to contact the administrator of a website.
- `webmaster@domain.com` - Synonym for administrator, same functionality as admin.

Out of the 296 vulnerable domains found, only 111 websites had the mailboxes able to receive e-mails. For the remaining domains, we used the `whois` [**?**] data to find the contact details of the owner and then e-mailed them. The number of emails we sent and the number of developer responses we received is shown in Table VI.

We received 15 developer responses, confirming 10 discovered vulnerabilities. Three of the developers fixed the vulnerability on their website. From our research, it is clear that E-mail Header Injection is quite widespread as a vulnerability, appearing on 1.46% of forms that we were able to perform automated attacks on. This value acts as a *lower bound* for prevalence of E-mail Header Injection vulnerability, and can quite easily be larger if the attacks were broader, crafted for the individual web application, and less automated.

F. Exploitation Evidence

We compared the 413 IPs that our system found to be vulnerable to E-mail Header Injection against 13 well-known IP blacklists, to see if these IPs were being exploited by attackers to send spam. The blacklists that we used were: `zen.spamhaus.org`, `spam.abuse.ch`, `cbl.abuseat.org`, `virbl.dnsbl.bit.nl`, `dnsbl.inps.de`, `ix.dnsbl.manitu.net`, `dnsbl.sorbs.net`, `bl.spamcannibal.org`, `bl.spamcop.net`, `dnsbl-1.uceprotect.net`, `dnsbl-2.uceprotect.net`, `dnsbl-3.uceprotect.net`, `db.wpbl.info`

We found that 106 of these IPs were blacklisted on at least one of the above blacklists for sending out spam, and 33 of them were found on multiple blacklists. We do not have enough data to make an observation about whether these attackers are exploiting E-mail Header Injection to send out the spam, as an alternative hypothesis is that these IPs are on the blacklists because the server has different vulnerabilities that attackers exploit to cause the server to send spam (assuming that the server is normally benign).

VI. Discussion

In this section, we discuss the lessons learned, the limitations of our system, and how to mitigate E-mail Header Injection vulnerabilities.

A. Lessons Learned

From our results, it is evident that E-mail Header Injection vulnerabilities exist in the wild. Despite its relatively low occurrence rate compared to the more popular SQL Injection and XSS (Cross-Site Scripting), when we consider total number of domains on the World Wide Web— 1,018,863,952 according to Internet Live Stats [**?**] as of early 2016—and calculate 0.029% percent (the occurrence rate of E-mail Header Injection vulnerability calculated from vulnerable domains as found by our system to total number of domains crawled) of that number, this yields 295,693 domains. Of course, extrapolation in this way is not an accurate measure of the prevalence of E-mail Header Injection vulnerabilities. However, even with as few as a thousand domains affected by this vulnerability, it can still have a disastrous impact on these domains, and also on overall World Wide Web due to the traffic caused by the sheer number of generated e-mails.

A common reason for our fuzzing attempts to fail is the bot-blocking mechanisms built into the web applications. CAPTCHAs (Completely Automated Public Turing test to tell Computers and Humans Apart) [**?**] pose a very difficult problem for our system to exploit E-mail Header Injection, even if it is present.

We found two different forms of E-mail Header Injection: the first one is the traditional one, injecting any header into the e-mail that allows the attacker complete control over the contents of the e-mail. The second attack has not yet been documented and provides the ability to inject multiple e-mail addresses into the `To` field. We call this a `To`

Pipeline Stage	Quantity	Differential Δ d2/d1 * 100
Crawled URLs	21,675,680	—
Forms found	6,794,917	31.35%
E-Mail Forms found	1,132,157	16.66%
Fuzzed with regular payload	934,016	82.50%
Received e-mails	52,724	5.64%
Fuzzed with malicious payload	46,156	87.54%
Successful attacks	673	1.46%

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

Notified websites	Developer Responses	Confirmed discoveries
296	15	10

TABLE VI: Responsible disclosure of the discovered vulnerabilities to developers and the number of received responses.

`header injection`. In this vulnerability, an attacker can add addresses to the `To` field of the email with newlines separating the e-mail addresses. We could not determine if this vulnerability is due to unique flaws in each web application or if this vulnerability is due to an implementation issue with a particular language or framework. However, from our preliminary analysis, it is evident that the vulnerable web applications do not share much in common.

`To header injection` allows an attacker to extract information that should be private, and in some of these cases, able to inject enough data to spoof other headers of the e-mail message. From Table IV, information leakage using `To header injection` was possible on 163 forms, while spoofing using `To header injection` was possible on 13 forms.

B. Limitations

Because our system is fully automated, it is also susceptible to being stopped by mechanisms in web applications that prevent automated crawls or form submissions. Measures such as CAPTCHA and hidden form fields are often used to detect bots [**? ?**].

We made sure that we do not fuzz hidden fields in the form, however, despite considerable active research in breaking CAPTCHAs [**? ?**], breaking CAPTCHAs remains out of the scope of this project.

Due to the growing emphasis on responsive web applications, more and more web applications are being built with only client-side JavaScript. Even conventional web applications use JavaScript to dynamically insert content and update the pages. This trend means that these dynamically injected HTML components are not part of the initial HTML that is sent to the client by the server.

Thus, our system will not see dynamically injected forms and hence is unable to detect whether E-mail Header Injection vulnerabilities are present in these forms. The workaround would be to use a JavaScript engine to query for the `document.getElementsByTagName('html')[0].innerHTML` (from inside web browser automation tools such as Selenium), then use that as the source HTML. We chose not to do this for performance reasons.

During the crawl, our system was blacklisted by a few web applications (mostly WordPress ones), and Internet Service Providers (ISPs). 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 web application that are known to blacklist automated crawlers. However, we could not gather any data about these web applications.

We found that certain WordPress plugins prevent the E-mail Header Injection attack by sanitizing user input on contact forms. Although not all WordPress web applications are secure, between the presence of the plugins on some websites, and getting tagged as "spambots" by others, we found few vulnerabilities on WordPress web applications.

E-Mail libraries such as the PHP Extension and Application Repository's (PEAR) mail library provide sanitization for user input. While this is not strictly a limitation of our project, it still means that we are not able to inject sites that used these libraries successfully.

Because we search for the words `e-mail`, `mail`, or `email` within the HTML 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 6. Here, the French word for `e-mail`—courrier Ãl'lectronique—is used, and our system is unable to find the presence of the e-mail form.

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

Black-box testing is highly beneficial as explained in Section III-A, 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 must manually notify the developers to get this feedback.

C. Assumptions

In addition to the limitations that were already discussed, we made certain assumptions while building the system. This section describes the assumptions and explores to what extent these hold true:

[2]We do not have any data about whether bad markup indicates an overall lower quality of the web application, and thus cannot comment on whether such websites are more likely to have vulnerabilities.

```
<!doctype html>
<html lang="fr">
<head>
<meta charset="utf-8">
<meta name="author" content="XYZ">
<title>Mock Email</title></head>
<body>
<form action="{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 6: HTML page with e-mail form, written in a different language - French.

1) **The crawler is not blocked**
   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 impossible to discover vulnerabilities.
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" [**?**], the "Law of big numbers" [**?**], and the concept of "Regression to the mean" [**?**]. Simply stated, a crawl of this magnitude should provide a distributed sample of the overall Web, eventually converging to the average of all web applications in existence.
3) **Injection of `bcc` indicates the existence of an 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 our 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.

D. Ethics

To make sure that our system did not cause any harm to the web applications that we crawled, we made sure that we did not inject any special characters other than the newline character. We also had an information website at the IP that we crawled from[3] that described what E-mail Header Injection was, and contained our contact details in case the developers of the web applications we crawled wanted to contact us. We maintained a separate blacklist of domain names that the owners did not want us to crawl, and ensured that our system did not crawl their domains.

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 VII: Mail libraries that prevent e-mail header injection.

E. Mitigation Strategy

After demonstrating that E-mail Header Injection vulnerabilities exist on the web at large, we now describe the most common measures that can be taken to prevent the occurrence of this vulnerability, or at least reduce the impact.

- Use Mail Libraries
  Using a safe e-mail library is the preferred way of preventing E-mail Header Injection vulnerabilities. Using a library that is well tested can remove the burden of input sanitization from the developer. A list of known secure libraries for each language and framework discussed previously is shown in Table VII. 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.
- Use a Content Management System (CMS)
  Content management systems such as WordPress and Drupal include 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 correct e-mail plugin, as not all plugins might be secure. An example of a secure plug-in is included as part of Table VII.
- Input Validation
  If neither of the two options are 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 [**?**], and care must be taken to ensure that all edge cases are taken into account.

[3]Removed for the sake of anonymity
[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/

VII. Related Work

There are different approaches to finding vulnerabilities in web applications, and most approaches will be either Black-Box testing or White-Box testing. Our work is based on the black-box testing approach to finding vulnerabilities on websites, and research has made use of this methodology to find vulnerabilities in web applications [**? ? ? ? ?**]. There has been significant discussion on both the benefits of such an approach [**?**] and its shortcomings [**? ?**].

Our work does not intend to act as a vulnerability scanner, but as a means to identify an E-mail Header Injection vulnerability in a given web application. In this sense, because we are injecting payloads into the web application, our work is related to other injection based attacks, such as SQL Injection [**? ? ?**], Cross-Site Scripting [**? ?**], HTTP Header Injection [**?**], and the related Simple Mail Transfer Protocol (SMTP) Injection [**?**].

The attack described by Terada [**?**] 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. Terada also describes proof-of-concept attacks against certain mailing libraries such as `Ruby Mail` and `JavaMail`. This attack, although trying to achieve a similar result, is distinctly different from ours. The paper 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.

Although E-mail Header Injection vulnerabilities have been present for over a decade, there has not been much written about it in the literature, and we find only 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 [**?**] 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 [**? ? ? ? ?**], each describing how to exploit the vulnerability by using newlines to camouflage headers inside user input. A wiki entry [**?**] 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 [**?**], 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 plugin 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*" [**?**]. 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 III-C5b).

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 web applications 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 character* and thus cannot cause any unintended consequences. Also, because 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 the prevalence.

VIII. 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 673 vulnerable forms (1.46% success rate). This indicates that the vulnerability is widespread, and needs attention from both web application developers and library developers.

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.

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