Measuring E-Mail Header Injections on the World Wide Web

Sai Prashanth Chandramouli[1], Pierre-Marie Bajan[2], Christopher Kruegel[3], Giovanni Vigna[3], Ziming Zhao[1], Adam Doupé[1], and Gail-Joon Ahn[1]

[1] Arizona State University
e-mail{saipc, zzhao30, doupe, gahn}@asu.edu
[2] IRT SystemX
e-mailpierre-marie.bajan@irt-systemx.fr
[3] University of California, Santa Barbara
e-mail{chris, vigna}@cs.ucsb.edu

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 script fails to check for the presence of e-mail headers in user input. We discovered that the vulnerability exists in the built-in e-mail functionality of the popular languages PHP, Java, Python, and Ruby. With the proper injection string, this vulnerability can be exploited to allow an attacker to inject additional headers, modify existing headers, and alter the content of the e-mail.

While E-mail Header Injection vulnerabilities are known by the community, and some commercial vulnerability scanners claim to discover E-mail Header Injection vulnerabilities, they have never been studied by the academic community. 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 23,553,796 URLs to identify web pages which contained form fields. 7,354,425 such forms were found by the system, of which 1,228,774 forms contained e-mail fields. We then fuzzed 1,012,530 forms to see if they would send us an e-mail, and 74,192 forms sent us an e-mail. Of these, 64,510 forms were tested with E-mail Header Injection payloads and, of these, we found 994 vulnerable URLs across 414 domains. Then, to measure if E-mail Header Injection vulnerabilities are actively being exploited to create a spamming platform, we found evidence that 157 IPs that were vulnerable to E-mail Header Injection are on spamming blacklists. We also performed a virus check on the received e-mails and found 265 domains to be sending malicious content. This work shows that E-mail Header Injection vulnerabilities are widespread and deserve future research attention.

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 made it possible for any-

one to access information and services anywhere and on multiple devices such as phones, laptops, TVs, and cars. This access has ushered in an era of web applications which depend on user input. While the rapid pace of development has improved the speed of information dissemination, it comes 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 steal lucrative data.

Many common and well-known web application vulnerabilities, such as SQL Injection and Cross-Site Scripting [38], are command injection vulnerabilities [51], where malicious user input is used to alter the structure of a command (SQL query and JavaScript code respectively). Developers of web applications must use the correct sanitization routine in all paths leading from user input to a command.

E-mail Header Injection vulnerabilities are a lesser-known command injection vulnerability. We verified that this vulnerability exists in the implementation of the built-in e-mail functionality in the popular languages PHP, Java, Python, and Ruby. The format of e-mail messages is defined by the Simple Mail Transfer Protocol (SMTP) [45]. 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 such as 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. In fact, the Acunetix vulnerability scanner contains an AcuMonitor component which claims to detect E-mail Header Injection vulnerabilities while scanning [10]. Unfortunately, as a commercial product, little is known about how AcuMonitor detects E-mail Header Injection vulnerabilities.

To shed light on this little-studied vulnerability class, we describe E-mail Header Injection vulnerabilities and measure E-mail Header Injection vulnerabilities. To perform this measurement, we crawled 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 payloads in the e-mails.

In summary, we make the following contributions:

– We develop a black-box approach to detect E-mail Header Injection vulnerabilities in a web application.

```php
$from = $_REQUEST['email'];
$subject = 'Hello XYZ';
$message = 'We need you to reset your password';
$to = 'xyz@example.com';
$retValue = mail($to, $subject, $message, "From: $from");
```

Listing 1.1: PHP program with e-mail header injection vulnerability.

```
Received: from mail.ourdomain.com ([62.121.130.29])
  by xyz.com (Postfix) with ESMTP id 5A08E52C0154
  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 1.2: SMTP headers generated by a PHP mail script.

– We develop an open-source system to crawl the web and automatically detect E-mail Header Injection vulnerabilities.
– We use our system to crawl 23,553,796 URLs, and we find 994 URLs vulnerable to E-mail Header Injection across 414 domains.

2 Background

E-mail Header Injection belongs to the class of command injection vulnerabilities. However, unlike its more popular siblings, SQL injection [17,25,47], Cross-Site Scripting [30,33], or HTTP Header Injection [31], relatively little research is available on E-mail Header Injection vulnerabilities.

As with other command injection vulnerabilities, E-mail Header Injection is caused by improper or nonexistent sanitization of user input. If the program constructs e-mails from user input and fails to check for the presence of e-mail headers, a malicious user can control the headers of this particular e-mail. E-mail Header Injection vulnerabilities can be leveraged to enable malicious attacks, including, but not limited to, spoofing or phishing.

2.1 History of E-mail Header Injection

We found the first E-mail Header Injection description in a late 2004 article on phpsecure.info [54] accredited to user `tobozo` 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 [34] and an accompanying wiki article [48] 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 1.1. 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.

CVE No.	Affected Software	Year
2002-1575	cgiemail	2004
2002-1771	FormMail 1.9	2005
2002-1917	Geeklog 1.35	2005
2005-0493	Biz Mail Form ¡=2.2	2005
2005-2854	thesitewizard.com	2005
2005-3883	PHP mb_send_mail	2005
2006-0631	Perl mailback.pl	2006
2006-0712	Squishdot 1.5.0	2006
2006-1225	Drupal 4.5.0-4.5.8 and 4.6.0-4.6.8	2006
2006-1305	Microsoft Outlook 2000, 2002-03	2006
2006-2159	Russcom Network	2006
2006-2943	CGI-RESCUE WebFORM 4.1	2006
2006-2944	CGI-RESCUE FORM2MAIL 1.21	2006
2006-3171	CS-Forum ¡=0.82	2006
2006-3473	Drupal Module ¡=1.8.2.2	2006
2006-4344	CGI-Rescue Mail	2006
2006-7020	phpwcms 1.2.5-DEV	2007
2006-7087	Dotdeb PHP	2007
2007-1718	PHP 4.0-4.4.6 and 5.0-5.2.1	2007
2007-1898	Jetbox CMS 2.1	2007
2007-1900	FILTER_VALIDATE_EMAIL PHP	2007
2007-2731	Jetbox CMS 2.1	2007
2008-2105	Bugzilla	2008
2009-1469	IceWarp	2009
2008-7281	OTRS - Open Ticket Request System	2011
2014-2957	Exim	2014
2015-8476	PHPMailer	2015
2016-4803	dotCMS	2016

Table 1: History of software found in Common Vulnerabilities and Exposures database affected by e-mail header injection vulnerability

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 1.2. 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.

We gathered reported Common Vulnerabilities and Exposures (CVE) [4] to get an idea of the distribution of reported E-mail Header Injection vulnerabilities over time. From the 28 reports we found (Table 1), it can be seen that even though many vulnerabilities were found in earlier years (2005-07), there have been recently discovered E-mail Header Injection vulnerabilities which suggests that it is still a very real and relevant threat to modern web security.

2.2 Languages Affected

PHP was the first language found vulnerable to E-mail Header Injection in its implementation of the `mail()` function at the time of release of PHP 4.0. According to w3techs [55], PHP is used by 81.9% of all 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, the PHP documentation [40] specifies that the `mail()` function does not protect against E-mail Header Injection. A working code sample with the vulnerability is shown in Listing 1.1.

```
Received: from mail.ourdomain.com ([62.121.130.29])
  by xyz.com (Postfix) with ESMTP id 5A08E52C0154
  for <abc@example.com>; Sun, 20 Mar 2016 13:56:58 -0700 (MST)
From: abc@example.com
CC: 1@example.com, 2@example.com, 3@example.com
Subject: My Subject
Content-Type: multipart/mixed; boundary=foobar;
--foobar
Content-Type: text/html

This is the attacker's body
--foobar
To: xyz@example.com
Subject: Hello XYZ
Date: Sun, 20 Mar 2016 13:56:58 -0700(MST)

We need you to reset your password
```

Listing 1.3: Exploiting the E-mail Header Injection vulnerability in Listing 1.1 to control the recipients, subject, and body of the SMTP message.

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 2011.

Unfortunately, the bug fix was only for the `email.header` package, and not for 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.

Java has a bug report about E-mail Header Injection filed against its `JavaMail` API. A detailed write-up by Alexandre Herzog [26] contains a proof-of-concept program that exploits the API to inject headers.

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

2.3 Exploitation

Successful exploitation of an E-mail Header Injection vulnerability depends on where injection occurs in the SMTP message. The attacker cannot alter parts of the SMTP message that precede the injection location, but the attacker has complete control over everything that follows. However, similar to other command injection vulnerabilities, the remaining parts of the SMTP message will always be appended to the attacker's injection, so the attacker must contend with this. By exploiting an E-mail Header Injection vulnerability, an attacker can control who receives the message (and can include multiple `CC` and `BCC` recipients), the body, and possibly the subject (depending on if the subject header occurs before/after the injection point and the language used).

The main vector for exploiting E-mail Header Injection vulnerabilities follows the template of command injection vulnerability exploitation: first inject the attacker's desired commands, then comment out the rest of the message. In E-mail Header Injection vulnerabilities, the attacker first includes all SMTP headers she desires. These will typically be the `Subject` header to control the subject of the e-mail[4], `CC` or `BCC` headers to control the recipients of the e-mail.

To handle the extra content after the injection point, one technique is to use a `Content-type` header to specify that the SMTP message is a multi-part email and that the sections are separated by an attacker-specified boundary. The boundary delineates different parts of the message so that the attacker's body is the only valid part of the message, and the attacker can choose a random value for the boundary that is not present in the developer-controlled part of the SMTP message.

Using this technique, the attacker can completely control the e-mail. For instance, injecting the following attack payload: `abc@example.com\nCC:1@example.com, 2@example.com, 3@example.com\nSubject: My Subject\nContent-Type:multipart/mixed; boundary=foobar;\n--foobar\nContent-Type: text/html \n\nThis is the attacker's body\n--foobar` into the `email` parameter of the PHP program in Listing 1.1 results in the SMTP message shown in Listing 1.3.

By expanding on this technique, the attacker can include links in the e-mail, or even attachments, by adding additional multipart messages with different content types.

A shorter technique, in case the injection point is limited in input size, is to use an HTML comment to ignore the developer-controlled part of the SMTP message, using a payload such as: `abc@example.com\nCC:1@example.com, 2@example.com, 3@example.com\nSubject: My Subject\n Content-Type: text/html\n\nThis is the attacker's body<!--`. However, this technique will only work if the developer-controlled part of the SMTP message does not contain a closing HTML comment tag `-->`.

2.4 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% 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 [29] (a variation of spoofing [22]) 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).

[4] The SMTP protocol specifies that there should only be one `Subject` header, so the attacker may not be able to alter the subject if the header is already defined. This behavior would be MUA-dependent.

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 e-mail to the attacker, thereby exposing 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 to send excessive e-mails resulting in overloading the mail server and cause crashes or instability.

3 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.

3.1 Approach

We took a black-box approach to measure the prevalence of E-mail Header Injection vulnerability on the web. Black-box testing [14] 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 2.2). The overall architecture of our system is presented in Figure 1.

3.2 System Components

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

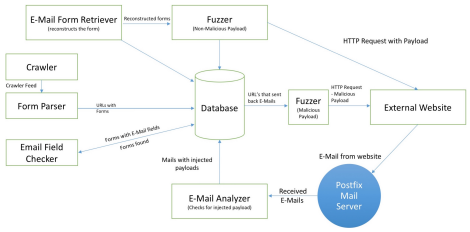

Fig. 1: Overall system architecture.

Data Gathering We used an open-source Apache Nutch based Crawler [1]. The `Crawler` provides the system with a continuous feed of URLs and the HTML contained in those pages. The `Form Parser` is responsible for parsing the HTML and retrieving data about the HTML forms on the page, including the following: (1) Form attributes, such as `method` and `action` (URL) for the HTTP request, (2) Data about form inputs, 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, and (3) 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 `E-mail Field Checker` is the final stage in the Data Gathering module. It receives the 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 and acts as the input to the Payload Injection module.

E-mail Form Retriever The E-mail Form Retriever is the first stage in the Payload Injection module. It does the following: (1) Retrieve forms and remove any duplicates, (2) reconstruct each form's input fields and values with the stored form data, and (3) construct the target URL to create an HTTP request for fuzzing.

Fuzzer The Fuzzer interacts directly with the external web applications. 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 [16], and can cause bottlenecks in a Crawler-Fuzzer system [49]. The two different types of payloads used for fuzzing are:

Non-Malicious Payload. The 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#@example.com`, where `#` is replaced by an internal id that uniquely maps the payload to the form, and `example.com` is replaced by our domain.
Malicious Payload. After receiving an e-mail from a specific form, we use the malicious payload to try to exploit an E-mail Header Injection vulnerability. We inject the form fields 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#@example.com\nbcc:-maluser#@example.com, (2) nuser#@example.com\nbcc:maluser#@example.com\n-x-check:in, (3) nuser#@example.com\r\nbcc:maluser#@example.com, and (4) nuser#@example.com\r\nbcc:maluser#@example.com\r\nx-check:in.

Payload 1 is the most minimal payload: it injects a newline character followed by the `bcc` field. Payload 2 contains the additional `x-check` header to inject Python-based web applications. Payloads 3 and 4 are added for purposes of cross-platform fuzzing: `\r\n` is the "Carriage Return - New Line (CRLF)" used on Windows systems [23]. The # are replaced by an internal id, for mapping to the forms.

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 [44], 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 3.2).

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.

Analyzing regular e-mail. 'Regular e-mail' refers to the e-mails received by account `reguser#@example.com` that were sent due to injecting the regular, non-malicious, payload (discussed in Section 3.2). 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 in either the headers or body of the e-mail. These could be fields such as name, username, age, etc. 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, where these fields are *also* fuzzed along with the e-mail fields to check for the presence of E-mail Header Injection.

Analyzing e-mail with payloads The "e-mails with payloads" refers to e-mails received by either the `nuser#@example.com` or `maluser#@example.com` accounts. These e-mails could only be received as a result of injecting the malicious payloads that were discussed in Section 3.2.

Detecting injected `bcc` headers As discussed in the payloads Section (3.2), the payloads were crafted such that the e-mails received by the `maluser` account directly indicate the presence of the injected `bcc` field.

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.

4 Evaluation

We ran our system on the web at large, attempting to discover E-mail Header Injection vulnerabilities in web applications. From our extensive crawl of the web, we were able to gather the data shown in Table 2. We ran the system for 76 days, during which our system crawled 23,553,796 unique URLs, and found a total of 7,354,425 forms from 1,085,365 unique domains. Out of these forms, our system found 1,228,774 forms that contained an e-mail field, from 198,306 unique domains. Table 3 shows the quantity of e-mails we received for the benign and malicious payloads.

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

Type of Data	Quantity
URLs Crawled	23,553,796
Total Forms found	7,354,425
Forms with E-Mail Fields	1,228,774

Table 2: The data collected for our project.

Type of fuzzing	Forms fuzzed	E-Mails received
Regular payload	1,012,530	74,192
Malicious payload	64,510	994

Table 3: The data we fuzzed and the e-mails we received.

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 contain an E-mail Header Injection vulnerability.

4.1 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 three categories:

(1) E-mails with the `bcc` header successfully injected. This form of injection was our initial objective, and we found 583 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 229 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 3.2, 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. 493 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 4. 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 54% of the received `bcc` header injected e-mails are also susceptible to being injected with additional headers.

Type of Injection	**No. of e-mails received**
E-Mail Header Injections with `bcc` header	583
E-Mail Header Injections with `x-check` header	493
`To header` injections alone	229
Injections with both `bcc` and `x-check` headers	310
Both `To header` injections and x-check headers	15
`x-check` headers found in `nuser` e-mails	239
Unique `x-check` headers found in `nuser` e-mails	182
Total successful injections (1 + 3 + 7)	994

Table 4: Classification of the e-mails that we received into broad categories of the vulnerability.

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 239 e-mails in the `nuser` account. Out of these, 182 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 (3), and Unique `x-check` headers found in `nuser` e-mails (7). A total of 994 vulnerabilities were found by our system.

4.2 Understanding the Data Pipeline

Table 5 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 23,553,796 URLs we crawled, only 7,354,425 forms (31.22%) were found. Out of these, only 1,228,774 forms (16.71%) contained e-mail fields.

In our fuzzing attempts, the same behavior is observed. We fuzzed 1,012,530 forms with the regular payload, which resulted in a total of 74,192 e-mails (7.33%). After analysis of the received e-mails, we further fuzzed 64,510 forms, which resulted in 994 e-mails (1.54%) which contain the vulnerability across 604 IP addresses from 414 domains.

We attribute the difference in the number of forms found to the number of forms fuzzed (a difference of 9,682 forms) to the presence of bot-blocking

Pipeline Stage	Quantity	Differential
Crawled URLs	23,553,796	Δ d2/d1 * 100
Forms found	7,354,425	31.22%
E-Mail Forms found	1,228,774	16.71%
Fuzzed with regular payload	1,012,530	82.40%
Received e-mails	74,192	7.33%
Fuzzed with malicious payload	64,510	86.95%
Successful attacks	994	1.54%

Table 5: Data gathered by our pipeline at each stage

Alexa Ranking Ranges	Vulnerable domains
0-5K	1
5-50k	7
50k-100k	6
100k-250k	23
250k-500k	30
500k-1m	68

Table 6: Distribution of vulnerable domains based on Alexa Rankings

mechanisms on a website (discussed in Section 5.2), though we do not know what percentage was caused by the individual bot-blocking mechanisms discussed in Section 5.2.

We would like to remark that over 1% of the forms that were not fuzzed (100 out of 9,682) were also tested manually using PostMan to generate HTTP requests with payloads to verify that our system was working as intended.

4.3 Analysis of Vulnerable domains

We performed the following analyses on the vulnerable domains:

- Checking Alexa rankings of the vulnerable domains.
- Investigating the back-end languages used by the vulnerable domains.
- Analyzing the presence of DKIM (DomainKeys Identified Mail), SPF (Sender Policy Framework), and DMARC (Domain-based Message Authentication, Reporting & Conformance) mechanisms on these domains.

Alexa rankings of vulnerable domains. We searched through the Alexa rankings data[7] for the domains that were found to be vulnerable, and found 135 of these domains in the top 1 million ranked websites. A detailed distribution of the rankings is shown in Table 6 / Figure 2 / Figure 3.

Back-end technologies of vulnerable domains. We investigated the top 100 vulnerable domains on our list (based on the alexa rankings) to find the back-end technologies used by these vulnerable domains to see if there was any recurring pattern. Using BuiltWith [8] and wappalyzer [9], we found that a majority of the vulnerable domains (67%) used PHP as one of the back-end technologies, while 15% of domains used WordPress and another 12% used ASP.Net. Other technologies used include Java (4%), Ruby on Rails (1%), and Perl (1%). We also found a combination of other technologies like Magento, CakePHP, CodeIgniter,

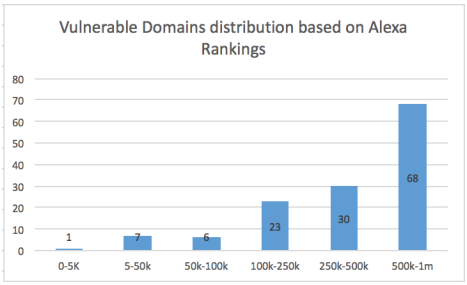

Fig. 2: Distribution of vulnerable domains based on Alexa Rankings

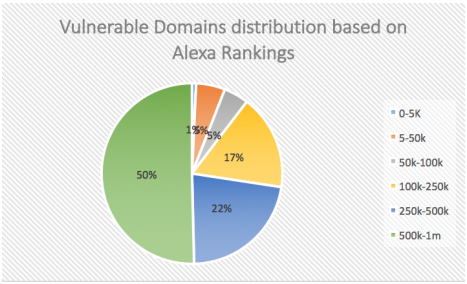

Fig. 3: Distribution of vulnerable domains based on Alexa Rankings

Joomla, mail.ru and Drupal being used in conjunction with one of the above languages. A point to be noted is that 4% of the websites also used Contact Form 7 [2], which is supposed to protect against such attacks.

Presence of e-mail spoofing protection on vulnerable domains.

4.4 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 [19]: `security@domain.com`, `admin@domain.com`, and `webmaster@domain.com`

Out of the 414 vulnerable domains found, only 113 websites had the mailboxes able to receive e-mails. For the remaining domains, we used the `whois` [5] data to find the contact details of the owner and then e-mailed them. We received 21 developer responses, confirming 15 discovered vulnerabilities. Four 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 1.54% 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.

4.5 Exploitation Evidence

We compared the 604 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 157 of these IPs were blacklisted on at least one of the above blacklists for sending out spam, and 46 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).

4.6 Emails with Malicious Attachments

As additional analysis to find domains on the internet that send out malicious attachments, we checked the e-mails received by the 'reguser' account (which are injected with regular e-mail addresses and not malicious data) for the presence of attachments that may contain malicious software, which will indicate that the server itself is not benign.

To do this, we passed the 2,950 attachments we received to VirusTotal [6] – an online malware scanner that checks for the presence of malware by running 40+ virus scanners on the uploaded files. We found that out of the 2,950 attachments, 443 were malicious, out of which 265 were from unique domains. This data is just a by-product of our research and could very well lead to future research.

5 Discussion

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

5.1 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 [28] as of early 2016—and calculate 0.038% 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.

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 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 4, information leakage using `To header injection` was possible on 229 forms, while spoofing using `To header injection` was possible on 15 forms.

5.2 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. 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) [11] pose a very difficult problem for our system to exploit E-mail Header Injection, even if it is present. Other measures such as hidden form fields and CSRF (Cross-site Request Forgery [35]) tokens are also often used to detect bots [43,11].

We made sure that we do not fuzz hidden fields in the form, and because our system does not depend on authenticated sessions, CSRF tokens do not pose an issue. However, despite considerable active research in breaking CAPTCHAs [11,56], 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.

A comparison of the running times between the different approaches is shown in Table 7. We chose not to use Selenium as it results in a slowdown of 31.58%.

Method	Running time	Slowdown
Using our pipeline	629.043	-
Our pipeline with Selenium	919.372	31.58%
Parsing e-mail fields instead of 'grep'ing	707.154	11.05%

Table 7: Running times in seconds for crawling, parsing, and detecting presence of e-mail fields in 1000 random wikipedia pages.

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 by using the French word for `e-mail`—courrier électronique— our system is unable to find the presence of the e-mail field.

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, and (2) Used a blacklist of our own to prevent our Fuzzer from fuzzing applications that are known to blacklist automated crawlers. This restricted us from gathering data about these 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.

The parser that we use for HTML parsing—Beautiful Soup—does not parse heavily malformed HTML and throws an exception on encountering such 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 very bad HTML markup.

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

Language	Mail Libraries
PHP	PEAR Mail[24], PHPMailer[41], Swiftmailer[52]
Python	SMTPLib with email.header.Header
Java	Apache Commons E-Mail[12]
Ruby	Ruby Mail >= 2.6[46]

Table 8: Mail libraries that prevent e-mail header injection.

users to send bulk e-mail, adding many `cc` or `bcc` headers). We must manually notify the developers to get this feedback.

5.3 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 informational website at the IP that we crawled from 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.

5.4 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.

Using a safe, well tested e-mail library is the preferred way of preventing E-mail Header Injection vulnerabilities (removing the burden of input sanitization from the developer). A list of known secure libraries for each language and framework discussed is shown in Table 8.

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 plugin is shown in Table 8.

If neither of the two options are feasible (in-house production, or lack of support infrastructure), developers can choose to perform proper input sanitization. Sanitization should be done with RFC5322 [45] in mind to ensure that all edge cases are covered.

6 Related Work

There are different approaches to finding vulnerabilities in web applications, and most approaches use 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 [14,27,32,39,57]. There has been significant discussion on both the benefits of such an approach [13] and its shortcomings [20,21].

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 [17,25,47], Cross-Site Scripting [30,33], HTTP Header Injection [31], and the related Simple Mail Transfer Protocol (SMTP) Injection [53].

The attack described by Terada [53] 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. Terada's paper also 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 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 [54] accredited to user `tobozo` 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 [18,34,36,37,42], each describing how to exploit the vulnerability by using newlines to camouflage headers inside user input. A wiki entry [48] 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` [3], recounts an actual attack against a WordPress plugin, `Contact Form`, with a proof of concept[5]. It also showcases the vulnerable code in the plugin that causes the vulnerability. 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. Note that this plugin is used actively on 300,000 websites (according to [15]), but is yet to be fixed. E-mail Header Injection vulnerability was described briefly by Stuttard and Pinto in their book, "*The Web Application Hacker's Handbook*" [50]. 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 3.2).

To the best of our knowledge, no other research has been conducted to determine the prevalence of this vulnerability on the World Wide Web. We have managed to, on a large scale, crawl and inject web applications with comparatively benign payloads (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 can't cause any unintended consequences. Also, as we are only injecting 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 its prevalence.

7 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 23,553,796 web pages finding 7,354,425 forms, out of which 1,228,774 forms were fuzzable. We fuzzed 1,012,530 forms and found 74,192 forms that allowed us to send/receive e-mails. Out of these, we were able to inject malicious payloads into 64,510 forms, identifying 994 vulnerable forms (1.54% success rate). This indicates that the vulnerability is widespread, and needs attention from both web application 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.

References

1. Apache Nutch. `http://nutch.apache.org/`
2. ContactForm7. `https://wordpress.org/plugins/contact-form-7`
3. Vexatious Tendencies. `https://vexatioustendencies.com/wordpress-plugin-vulnerability-dump-part-2/` (2014)
4. CVE - Common Vulnerabilities and Exposures (CVE) (2016), `http://cve.mitre.org/`
5. ICANN WHOIS Data. `https://whois.icann.org/en` (2016)
6. VirusTotal - Free Online Virus, Malware and URL Scanner (2016), `https://www.virustotal.com/`
7. Alexa Rankings. `data.alexa.com/data` (2017)
8. BuiltWith Website Data. `https://builtwith.com` (2017)
9. Wappalyzer. `https://wappalyzer.com/` (2017)
10. Acunetix: AcuMonitor: For detecting Email Header Injection, Blind XSS and SSRF - Acunetix. `http://www.acunetix.com/vulnerability-scanner/acumonitor-blind-xss-detection/`

11. von Ahn, L., Blum, M., Langford, J.: Telling humans and computers apart automatically. Commun. ACM 47(2) (2004), http://doi.acm.org/10.1145/966389.966390
12. Apache Commons Email: (2016), https://commons.apache.org/proper/commons-email
13. Bau, J., Bursztein, E., Gupta, D., Mitchell, J.: State of the art: Automated black-box web application vulnerability testing. In: Security and Privacy (SP), 2010 IEEE Symposium on. pp. 332–345 (May 2010)
14. Beizer, B.: Black-box Testing: Techniques for Functional Testing of Software and Systems. John Wiley & Sons, Inc., New York, NY, USA (1995)
15. BestWebSoft: Contact Form by BestWebSoft WordPress Plugins. https://wordpress.org/plugins/contact-form-plugin/ (2016)
16. Bhide, C.W., Singh, J., Oestreicher, D.: Performance optimizations for computer networks utilizing http (Dec 22 1998), uS Patent 5,852,717
17. Boyd, S.W., Keromytis, A.D.: Sqlrand: Preventing sql injection attacks. In: Applied Cryptography and Network Security. pp. 292–302. Springer (2004)
18. Calin, B.: Email Header Injection Web Vulnerability - Acunetix. https://www.acunetix.com/blog/articles/email-header-injection-web-vulnerability-detection/ (2013)
19. Crocker, D.: Internet Message Format - RFC 2142 (1997), https://www.ietf.org/rfc/rfc2142
20. Doupé, A., Cavedon, L., Kruegel, C., Vigna, G.: Enemy of the state: A state-aware black-box web vulnerability scanner. In: Presented as part of the 21st USENIX Security Symposium (USENIX Security 12). pp. 523–538. USENIX, Bellevue, WA (2012), https://www.usenix.org/system/files/conference/usenixsecurity12/sec12-final225.pdf
21. Doupé, A., Cova, M., Vigna, G.: Why johnny can't pentest: An analysis of black-box web vulnerability scanners. In: Detection of Intrusions and Malware, and Vulnerability Assessment, pp. 111–131. Springer (2010)
22. Felten, E.W., Balfanz, D., Dean, D., Wallach, D.S.: Web spoofing: An internet con game. Software World 28(2) (1997)
23. Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Berners-Lee, T., Masinter, L., Leach, P.: RFC 2616 (1999), https://www.ietf.org/rfc/rfc2616.txt
24. Hagenbuch, C., Heyes, R., Machniak, A.: PEARMail (2016), https://pear.php.net/package/Mail
25. Halfond, W.G., Viegas, J., Orso, A.: A classification of sql-injection attacks and countermeasures. In: Proceedings of the IEEE Symposium on Secure Software Engineering (2006)
26. Herzog, A.: Full Disclosure: JavaMail SMTP Header Injection via method setSubject [CSNC-2014-001] (2014), http://seclists.org/fulldisclosure/2014/May/81
27. Huang, Y.W., Huang, S.K., Lin, T.P., Tsai, C.H.: Web application security assessment by fault injection and behavior monitoring. In: Proceedings of the 12th International Conference on World Wide Web. WWW '03, ACM (2003), http://doi.acm.org/10.1145/775152.775174
28. Internet Live Stats: www.internetlivestats.com (2016)
29. Jakobsson, M., Myers, S.: Phishing and countermeasures: understanding the increasing problem of electronic identity theft. John Wiley & Sons (2006)
30. Jim, T., Swamy, N., Hicks, M.: Defeating script injection attacks with browser-enforced embedded policies. In: Proceedings of the 16th International Conference

on World Wide Web. pp. 601–610. WWW '07, ACM, New York, NY, USA (2007), http://doi.acm.org/10.1145/1242572.1242654
31. Johns, M., Winter, J.: Requestrodeo: Client side protection against session riding. In: Proceedings of the OWASP (2006)
32. Kals, S., Kirda, E., Kruegel, C., Jovanovic, N.: Secubat: a web vulnerability scanner. In: Proceedings of the 15th international conference on World Wide Web. pp. 247–256. ACM (2006)
33. Klein, A.: [DOM Based Cross Site Scripting or XSS of the Third Kind] Web Security Articles - WebApp Sec (2005), http://www.webappsec.org/projects/articles/071105.shtml
34. Kohler, D.: damonkohler: Email Injection. http://www.damonkohler.com/2008/12/email-injection.html (2008)
35. Lin, X., Zavarsky, P., Ruhl, R., Lindskog, D.: Threat modeling for csrf attacks. In: CSE (3). pp. 486–491 (2009)
36. Mohamed, A.: PHP Email Injection Example - InfoSec Resources. http://resources.infosecinstitute.com/email-injection/ (2013)
37. Nicol, J.: Securing PHP Contact Forms. http://jonathannicol.com/blog/2006/12/09/securing-php-contact-forms/ (2006)
38. OWASP: https://www.owasp.org/index.php/OWASP_Top_10
39. Payet, P., Doupé, A., Kruegel, C., Vigna, G.: 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)
40. PHP-Manual: PHP mail - Send mail. http://php.net/manual/en/function.mail.php (2016)
41. PHPMailer: https://github.com/PHPMailer/PHPMailer
42. Pope, A.: Prevent Contact Form Spam Email Header Injection — Storm Consultancy Web Design Bath (2008), https://www.stormconsultancy.co.uk/blog/development/dev-tutorials/secure-your-contact-form-against-spam-email-header-injection/
43. Pope, C., Kaur, K.: Is it human or computer? defending e-commerce with captchas. IT Professional 7(2) (Mar 2005)
44. Raghavan, S., Garcia-Molina, H.: Crawling the hidden web. Technical Report 2000-36, Stanford InfoLab (2000), http://ilpubs.stanford.edu:8090/456/
45. Resnick, P.W.: Internet Message Format - RFC 5322 (2008), https://tools.ietf.org/html/rfc5322
46. Ruby Mail Gem: https://rubygems.org/gems/mail
47. Sadeghian, A., Zamani, M., Manaf, A.A.: A taxonomy of sql injection detection and prevention techniques. In: Informatics and Creative Multimedia (ICICM), 2013 International Conference on. pp. 53–56. IEEE (2013)
48. Email Injection - Secure PHP Wiki. http://securephpwiki.com/index.php/EmailInjection (2010)
49. Shkapenyuk, V., Suel, T.: Design and implementation of a high-performance distributed web crawler pp. 357–368 (2002)
50. Stuttard, D., Pinto, M.: The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws. John Wiley & Sons (2011)
51. Su, Z., Wassermann, G.: The essence of command injection attacks in web applications. In: ACM SIGPLAN Notices. vol. 41, pp. 372–382. ACM (2006)
52. SwiftMailer: http://swiftmailer.org/
53. Terada, T.: SMTP Injection via recipient email addresses. MBSD White Paper (December 2015)

54. Tobozo: Mail headers injections with PHP. http://www.phpsecure.info/v2/article/MailHeadersInject.en.php (2004)
55. W3techs: Usage Statistics and Market Share of PHP for Websites, February 2016. http://w3techs.com/technologies/details/pl-php/all/all (2016)
56. Yan, J., Ahmad, A.S.E.: Breaking visual captchas with naive pattern recognition algorithms. In: Computer Security Applications Conference, ACSAC 2007 (Dec 2007)
57. Zanero, S., Carettoni, L., Zanchetta, M.: Automatic detection of web application security flaws. Black Hat Briefings (2005)