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 [16], are command injection vulnerabilities [23], 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) [19]. 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 [3]. 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 [6,9,20], Cross-Site Scripting [12,14], or HTTP Header Injection [13], 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 [24] 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 [15] and an accompanying wiki article [21] 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) [2] 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 [25], 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 [17] 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 [10] 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 [11] (a variation of spoofing [7]) 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 [4] 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.

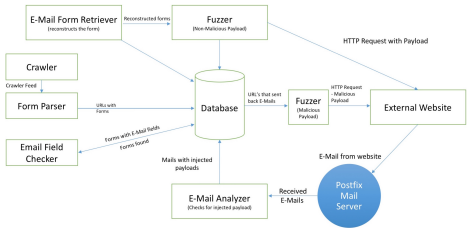

Fig. 1. Overall system architecture.

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 [5], and can cause bottlenecks in a Crawler-Fuzzer system [22]. 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\nx-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 [8]. 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 [18], 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 Other Sections

Other sections are here.

5 Conclusion

Conclusions are here.

6 Footnotes

Footnotes within the text should be coded:

`\footnote{Text}`

Sample Input

Text with a footnote`\footnote{The footnote is automatically numbered.}` and text continues ...

Sample Output

Text with a footnote[5] and text continues ...

7 Tables

Table captions should be treated in the same way as figure legends, except that the table captions appear *above* the tables. The tables will be numbered automatically.

7.1 Tables Coded with LaTeX

Sample Output

Table 2. Critical N values

$M_\odot$	β_0	T_{c6}	γ	N_{crit}^{L}	N_{crit}^{Te}
30	0.82	38.4	35.7	154	320
60	0.67	42.1	34.7	138	340
120	0.52	45.1	34.0	124	370

References

1. Apache Nutch. `http://nutch.apache.org/`
2. CVE - Common Vulnerabilities and Exposures (CVE) (2016), `http://cve.mitre.org/`
3. Acunetix: AcuMonitor: For detecting Email Header Injection, Blind XSS and SSRF - Acunetix. `http://www.acunetix.com/vulnerability-scanner/acumonitor-blind-xss-detection/`

[5] The footnote is automatically numbered.

4. Beizer, B.: Black-box Testing: Techniques for Functional Testing of Software and Systems. John Wiley & Sons, Inc., New York, NY, USA (1995)
5. Bhide, C.W., Singh, J., Oestreicher, D.: Performance optimizations for computer networks utilizing http (Dec 22 1998), uS Patent 5,852,717
6. Boyd, S.W., Keromytis, A.D.: Sqlrand: Preventing sql injection attacks. In: Applied Cryptography and Network Security. pp. 292–302. Springer (2004)
7. Felten, E.W., Balfanz, D., Dean, D., Wallach, D.S.: Web spoofing: An internet con game. Software World 28(2) (1997)
8. 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
9. 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)
10. Herzog, A.: Full Disclosure: JavaMail SMTP Header Injection via method setSubject [CSNC-2014-001] (2014), http://seclists.org/fulldisclosure/2014/May/81
11. Jakobsson, M., Myers, S.: Phishing and countermeasures: understanding the increasing problem of electronic identity theft. John Wiley & Sons (2006)
12. 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
13. Johns, M., Winter, J.: Requestrodeo: Client side protection against session riding. In: Proceedings of the OWASP (2006)
14. 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
15. Kohler, D.: damonkohler: Email Injection. http://www.damonkohler.com/2008/12/email-injection.html (2008)
16. OWASP: https://www.owasp.org/index.php/OWASP_Top_10
17. PHP-Manual: PHP mail - Send mail. http://php.net/manual/en/function.mail.php (2016)
18. Raghavan, S., Garcia-Molina, H.: Crawling the hidden web. Technical Report 2000-36, Stanford InfoLab (2000), http://ilpubs.stanford.edu:8090/456/
19. Resnick, P.W.: Internet Message Format - RFC 5322 (2008), https://tools.ietf.org/html/rfc5322
20. 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)
21. Email Injection - Secure PHP Wiki. http://securephpwiki.com/index.php/EmailInjection (2010)
22. Shkapenyuk, V., Suel, T.: Design and implementation of a high-performance distributed web crawler pp. 357–368 (2002)
23. Su, Z., Wassermann, G.: The essence of command injection attacks in web applications. In: ACM SIGPLAN Notices. vol. 41, pp. 372–382. ACM (2006)
24. Tobozo: Mail headers injections with PHP. http://www.phpsecure.info/v2/article/MailHeadersInject.en.php (2004)
25. W3techs: Usage Statistics and Market Share of PHP for Websites, February 2016. http://w3techs.com/technologies/details/pl-php/all/all (2016)