The HTML Form Protocol Attack

Jochen Topf
<jochen@remote.org>
http://www.remote.org/jochen/

Version 1.0 – 2001-08-14

This paper is available at http://www.remote.org/jochen/sec/hfpa/.

Abstract

This paper describes how some HTML browsers can be tricked through the use of HTML forms into sending more or less arbitrary data to any TCP port. This can be used to send commands to servers using ASCII based protocols like SMTP, NNTP, POP3, IMAP, IRC, and others. By sending HTML email to unsuspecting users or using a trojan HTML page, an attacker might be able to send mail or post Usenet News through servers normally not accessible to him. In special cases an attacker might be able to do other harm, e.g. deleting mail from a POP3 mailbox.

1 The Problem

Using a specially crafted HTML page, an attacker can trick a browser displaying this HTML page into accessing SMTP, NNTP, POP3, IRC, or other servers, possibly behind a firewall.

A HTML page like this will typically be sent via email or put on a hacked web server where unsuspecting users might find it.

In most situations this attack would not be considered a big problem, but it is an interesting example on how the combination of several innocuous and seemingly totally unrelated protocol features can be used to mount an attack.

In special circumstances this attack can possibly do quite a lot of harm. At least one large ISP was vulnerable to this attack, which – in this case – could be used to delete users' mail from POP3 servers. This hole has been closed now.

Although all the details of the attack are explained in this paper, no complete code or HTML examples are given in order to make it a little bit more difficult to mount an actual attack.

2 How a HTML form works

2.1 A simple example

To understand the attack, one has to understand how a HTML form [1] works.

Here is an example of a simple HTML form:

```
<form method="post"
      action="http://www.example.com/cgi-bin/foo.pl">
  First name:
  <input type="text" name="firstname" value="Joe"><br>
  Last name:
  <input type="text" name="lastname" value="Sixpack"><br>
  <input type="submit" value="Send it!">
</form>
```

In a typical browser this form will be shown as two one-line text input boxes with the default values ”Joe” and ”Sixpack” in them and a ”Send it!” button below them to submit the form content.

If the user activates the ”Send it!” button, the browser will use a HTTP POST request [2] on the URL `http://www.example.com/cgi-bin/foo.pl` to send the form data to the server.

Looking at the HTTP request going from the browser to the server we will see something like this:

```
POST /cgi-bin/foo.pl HTTP/1.0
User-Agent: Mozilla/4.77
Accept: image/gif, image/jpeg, image/png, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
Content-type: application/x-www-form-urlencoded
Content-length: 29

firstname=Sixpack&lastname=Joe
```

The form data is at the end of the request.

2.2 Using the `enctype` attribute

As seen in the previous example, the form data will normally be URL encoded, but it is possible to use the `enctype` attribute of the `form` element to change this:

```
<form method="post"
      action="http://www.example.com/cgi-bin/foo.pl"
      enctype="multipart/form-data">
...
```

The browser will now send something like this to the server:

```
POST /cgi-bin/foo.pl HTTP/1.0
User-Agent: Mozilla/4.77
Accept: image/gif, image/jpeg, image/png, */*
Accept-Encoding: gzip
Accept-Language: en
Accept-Charset: iso-8859-1,*,utf-8
Content-type: multipart/form-data;
  boundary=-------------------18006185524819184861641129113
Content-Length: 299

---------------------18006185524819184861641129113
Content-Disposition: form-data; name="firstname"

Sixpack
---------------------18006185524819184861641129113
Content-Disposition: form-data; name="lastname"

Joe
---------------------18006185524819184861641129113--
```

This time the content of the form is send unencoded, but put inside a MIME `multipart/form-data` wrapper. This method was invented for the file upload feature of HTML forms but can also be used with any other form. It is documented in RFC1867 [3] and RFC2388 [4].

Note that the values of the input boxes are now sent exactly as they appear in the HTML form.

2.3 The `textarea` element

It is very easy to put anything you like into this HTTP request. Just use the `<textarea>` element:

```
<textarea name="foo">
This is some
long text.
You can put anything
you want here.
</textarea>
```

This form will send something like this to the server:

```
POST /cgi-bin/foo.pl HTTP/1.0
...
---------------------1800618552481918486164112911 3
Content-Disposition: form-data; name="foo"

This is some
long text.
You can put anything
you want here.
---------------------18006185524819184861641129113--
```

3 Details of the attack

Knowing how a HTML form works, it is quite easy to mount the attack: Just point the `action` URL to an "interesting" TCP port and put some commands used in the protocol on that port into the text area. In this example a SMTP dialog is used:

```
<textarea name="foo">
HELO example.com
MAIL FROM:<somebody@example.com>
RCPT TO:<recipient@example.org>
DATA
Subject: Hi there!
From: somebody@example.com
To: recipient@example.org

Hello world!
.
QUIT
</textarea>
```

If you send this to `http://mail.example.org:25/` you get the following result:

```
220 mail.example.org ESMTP Hi there!
500 Command unrecognized
500 Command unrecognized
500 Command unrecognized
500 Command unrecognized
500 Command unrecognized
500 Command unrecognized
500 Command unrecognized
500 Command unrecognized
```

```
500 Command unrecognized
500 Command unrecognized
500 Command unrecognized
500 Command unrecognized
500 Command unrecognized
250 mail.example.org Hello example.com [10.11.12.13]
250 <somebody@example.com> is syntactically correct
250 <recipient@example.org> is syntactically correct
354 Enter message, ending with "." on a line by itself
250 OK id=15IYAS-00073G-00
221 mail.example.org closing connection
```

(Note that because the answer is not a valid HTTP answer, not all browsers will show the results.)

The `500 Command unrecognized` errors are due to the HTTP protocol overhead, the header lines, MIME wrapper, etc. They are ignored by most if not all servers.

From the other messages, especially the `250 OK` line, you can see that a mail has been sent successfully.

3.1 Why is this a problem?

You might ask why this is a problem, after all you could connect to the SMTP server yourself and send the mail. There are several reasons why this attack might be used to obtain access privileges not otherwise held by an attacker:

- The user looking at the trojan HTML page might be behind a firewall and has access to services that the attacker doesn't have access to.
- The SMTP or NNTP server might not allow the attacker to relay mail or post something if he accesses it from his IP number, but it allows access from the victim's IP number.
- Some Intranets might give users access to NNTP, POP3, or IMAP servers without authentication because they are known to come from an authenticated IP address. An attacker might be able to use this to delete the victim's mail etc.

Additionally, an attacker might use this attack to hide his tracks.

3.2 A real attack

In a real attack a one-line Javascript command can be used to automatically "push the Submit button" the moment the page is loaded. The user has no chance to see what is going on or to do something about it until it is too late.

Although I haven't looked into it, it is probably possible to hide the form and the result using frames, text in the background color, Javascript or other tricks so the user will not suspect anything.

A proof-of-concept test has verified that it is possible to write a simple worm using this attack that propagates through NNTP. Everybody with a HTML enabled News client such as Microsoft Outlook Express reading the specially crafted HTML posting with embedded Javascript commands will post it again... The same attack should be possible using SMTP, but as there is no way for the Javascript to get to lists of email addresses except from some that might be embedded in the worm, the damage that such a worm can do is very limited.

3.3 Vulnerable protocols

Generally, all ASCII command based protocols might be vulnerable to this attack. This includes, but is probably not limited to: FTP, SMTP, NNTP, POP3, IMAP, and IRC.

Servers using binary protocols like SSH or DNS will generally close the connection if they receive anything they don't understand. Because using this attack implies that there is always the HTTP protocol overhead at the beginning, there is no way to use this attack on "binary" protocols.

Some protocols like the one used for `rsh` require that the connection originates from a TCP port smaller than 1024, which is generally not the case for a request originating from a web browser.

3.4 A variant of the attack

In a mail to the BugTraq mailing list [12] on July 30, 2001, Michal Zalewski described an attack which uses fake IRC protocol data to open a TCP channel into a firewalled network through an IP router which uses masquerading (NAT) [13].

I have verified that the HTML Protocol Attack can be combined with this attack to connect into a masqueraded network through a router running Linux 2.2 (with the ip_masq_irc module enabled, tested with kernel 2.2.12).

4 Tests with common software

4.1 Server software

I have tested the mail transfer agents (MTAs) Sendmail [5] (8.9.3) and Exim [6] (3.16, fixed in 3.32) and the Usenet News Server INN [7] (nnrpd 2.2). All ignore "bad commands" and allow a mail to be sent or a news article to be posted, respectively.

4.2 Web browsers

Netscape (tested with version 4.77 on Linux) uses a list of "dangerous" ports compiled in. Access to ports 25 (SMTP), 110 (POP3), 119 (NNTP) and others failed with the message: "Sorry, access to the port number given has been disabled for security reasons." The restriction is only enforced if no proxy is used, otherwise the request is sent to the proxy.

Opera [8] (tested with version 5 on Linux): No restrictions.

Internet Explorer (tested with version 5.50.4522.1800 on Windows 2000): No restrictions.

4.3 Proxies

Squid [9] (version 2.4): Configurable. Recommended configuration has a list of safe ports, all others are not reachable.

Junkbuster [10] (tested with version 2.0.2 on Linux): No restrictions.

5 What to do about it

5.1 The user

Basically the only thing a user can do to prevent this attack is to disable Javascript in the browser. The attack is still possible, but instead of being fully automatic, the attacker has to get the user to click on the submit button.

5.2 Web browser and proxy authors

Web browser authors should make sure that they don't allow well known ports of potentially vulnerable applications as port numbers in a HTTP URLs. This includes, but is probably not limited to ports 21 (FTP), 25 (SMTP), 110 (POP3), 119 (NNTP), 143 (IMAP), and 6667 (IRC).

The port should be checked even if the request is not sent directly but through a web proxy.

5.3 Authors of server software

It should be easy to defend against this attack on the server side: Just close the connection if an unknown command is received (possibly after sending an error response). There will always be several lines of HTTP request "rubbish" in front of the commands due to the way HTML forms are send to the server.

Unfortunately it is not as easy in the real world: Most protocols have evolved over time and the set of commands supported by client and server have changed several times. It is very common for client and server software to support slightly different sets of commands. Thus, the server cannot close the connection when it receives an unknown command without risking incompatibility with a variety of clients.

I propose two different solutions:

(a) Look for a something that looks like a HTTP POST request. This is easy to do but if something changes in the HTTP protocol or if an attack using a different HTTP method (such as PUT) is found the server has to be changed again. Care has to be taken when dealing with protocols that have a valid POST command (like NNTP).

(b) Count the number of bad commands and kill the connection after a few unkown commands. This is very easy to implement and has the additional benefit that it might protect the server against some kinds of denial-of-service attacks. Depending on the protocol, it might make sense to reset the counter after every valid command and/or disable the counter after successful authentication. All known valid but unimplemented commands should probably not be counted.

Note that some standards more or less explicitly forbid closing the connection if a bad command is received. (For SMTP see section 3.9 in RFC2821 [11].)

6 Thanks

Thanks to Sven Dickert and Sven Paulus for some ideas and tests on how to exploit this attack.

References

[1] HTML 4.01 Specification: Forms
http://www.w3.org/TR/html4/interact/forms.html

[2] RFC 2616: Hypertext Transfer Protocol – HTTP/1.1
ftp://ftp.isi.edu/in-notes/rfc2616.txt

[3] RFC 1867: Form-based File Upload in HTML.
ftp://ftp.isi.edu/in-notes/rfc1867.txt

[4] RFC 2388: Returning Values from Forms: multipart/form-data.
ftp://ftp.isi.edu/in-notes/rfc2388.txt

[5] http://www.sendmail.com/

[6] http://www.exim.org/

[7] INN: InterNetNews http://www.isc.org/products/INN/

[8] http://www.opera.com/

[9] http://www.squid-cache.org/

[10] http://www.junkbuster.com/

[11] RFC 2821: Simple Mail Transfer Protocol
ftp://ftp.isi.edu/in-notes/rfc2821.txt

[12] http://www.securityfocus.com/frames/?content=/forums/bugtraq/intro.html

[13] http://www.securityfocus.com/templates/archive.pike?list=1&mid=200361&_ref=1661949996

Subject: [RAZOR] Linux kernel IP masquerading vulnerability
Date: Mon Jul 30 2001 12:49:51
Author: Michal Zalewski <lcamtuf@razor.bindview.com>
Message-ID: <Pine.LNX.4.21.0107301249390.747-100000@nimue.bos.bindview.com>