Using Libwhisker

Neil Desai 2004-08-24

As noted in the article "Penetration Testing of Web Applications" the use of web applications to conduct business is increasing. Companies often have custom sites built by in-house developers, and it is almost impossible to find all the vulnerabilities in a web site using automated tools. Simply looking for default installations of different software may turn up nothing, but it may still be vulnerable to many different programming errors in this custom-built site. Conducting an assessment of website can be a major undertaking and it is much more painful if the assessment is carried out with out the proper tools. A manual inspection of the site is almost always required, but when a particular vulnerability is found it can be very handy to have a set of tools to automate certain steps from there.

Why Libwhisker?

Since we are dealing with custom applications we need a set of custom utilities. There are many different tools out there that can be scripted, but we are going to focus on Libwhisker. Libwhisker is not a tool or application in itself; it is a PERL library which allows for the creation of custom HTTP packets.

Since Libwhisker is a PERL module and not an application, it is assumed that the reader has some knowledge of the HTTP protocol and is familiar with writing PERL scripts that use external modules.

First let's answer the question: why do we do we need to look at another PERL module to do what can all ready be done through existing PERL modules (i.e. LWP, HTTP, URI)? Libwhisker offers us many advantages over other PERL modules:

1. It is 100% PERL code. This means that it is completely portable to any system that runs PERL.
2. It combines the functionality of many different PERL modules normally used for interacting with web applications.
3. It does not need to be installed. Just have a copy of the "LW2.pm" in the same directory as the script that needs to use it.
4. It was created with a no rules approach. This means that it allows the user to create requests that don't conform to the standards. Other PERL modules may try and enforce standards on our requests.

Using Libwhisker

The main data structure in Libwhisker is the 'whisker' anonymous hash. A hash is a data structure in PERL that is comparable to associated arrays in other programming and scripting languages.

This 'whisker' anonymous hash can either define different aspects of a HTTP request or read different parts of the HTTP response. However, determining how to access this information can be a source of confusion.

Prior to using any of the Libwhisker functions, two PERL hashes first need to be defined, one for the HTTP request and one for the HTTP response. Some items will be defined in the 'whisker' hash and some will be either defined directly in the request hash or read directly from the response hash. To determine which portions of the HTTP request/response are part of the 'whisker' hash and which are part of the request/response hash, let's look at the possible options for the 'whisker' hash that relate to a HTTP packet.

Note that internal to the 'whisker' hash are the 'hin' and 'hout' hashes which are directly mapped to the request and response hash, respectively. For this article we will use the %request, % response and %jar PERL hashes to refer to the HTTP request hash, HTTP response hash and HTTP cookies hash, respectively.

1. `{'whisker'}->{'host'}` = This will be the remote host that we want to connect to. This value will also be reflected in the HTTP 'Host' header. This can be an IP address, DNS name or a NetBIOS name. If you want to specify another value in the HTTP 'Host" header then you will have to do it manually prior to using the 'LW2::http_do_request' function. The HTTP 'Host' header must be included in all HTTP 1.1 packets to be RFC compliant. By default this is 'localhost'.
2. `{'whisker'}->{'port'}` = This is the remote port that we want to connect to. The only valid values are numbers between 1 and 65535. By default this is port 80.
3. `{'whisker'}->{'proxy_host'}` = This is the proxy host that we want to use. By default this is not set.
4. `{'whisker'}->{'proxy_port'}` = This is the port of the proxy that we want to use. This must be set if the 'proxy_host" option is set.
5. `{'whisker'}->{'method'}` = This the HTTP verb that we want to use. By default this is GET.
6. `{'whisker'}->{'uri'}` = This is URI that we want to use. By default this is '/'.

7. `{'whisker'}->{'version'}` = This is the version of HTTP that we want to use. We can specify either 0.9, 1.0 or 1.1. If this is set to 0.9 then Libwhisker, by default, will follow only what is supported by 0.9. By default this is set to 1.1.
8. `{'whisker'}->{'error'}` = This is only in the 'hout' hash if it is defined. This has nothing do with the HTTP code returned by the server, but with error's received when reading the data from the server. By default this is empty.
9. `{'whisker'}->{'data'}` = If used in the request hash, this would be used for any POST or PUT data. If the 'http_fixup_request' function is used after the {'whisker'}->{'data'} option is defined and then the HTTP 'Content-Length' will be set. If this is used in the response hash then it would refer to the HTML data returned from the server.
10. `{'whisker'}->{'code'}` = This is only available in the response hash. This will be the numeric HTTP code returned by the remote server.
11. `{'whisker'}->{'message'}` = This will be the textual message associated with {'whisker'}->{'code'}.
12. `{'whisker'}->{'header_order'}` = This is an anonymous array that holds all the HTTP headers that were returned from the server, and the order that they were received.
13. `{'whisker'}->{'http_space1'}` = This is the value of the space that is used between the HTTP method/verb and the URI. By default this is ' ' (single space).
14. `{"whisker'}->{'uri_prefix'}` = This is the value of data that is placed before the URI. By default this is blank.
15. `{'whisker'}->{'uri_postfix'}` = This is the value of the data that is placed after the URI. By default this is blank.
16. `{'whisker'}->{'uri_param_sep'}` = This is the value of the character that is used to separate parameters. By default this is '?'.
17. `{'whisker'}->{'http_space2'}` = This is the value of that is placed between the URI and the HTTP version. By default this is ' ' (space).
18. `{'whisker'}->{'http_eol'}` = This is the value of the header line terminator.
19. `{'whisker'}->{'cookies'}` = This is available in the response hash only. This is usually not used directly as there are functions to read and write the cookies. If a hash has been specified for the cookies then they can be stored in there.

Below is a diagram of an HTTP request packet that shows where the parts of the 'whisker' hash relate to and where the parts of the 'request' hash relate to.

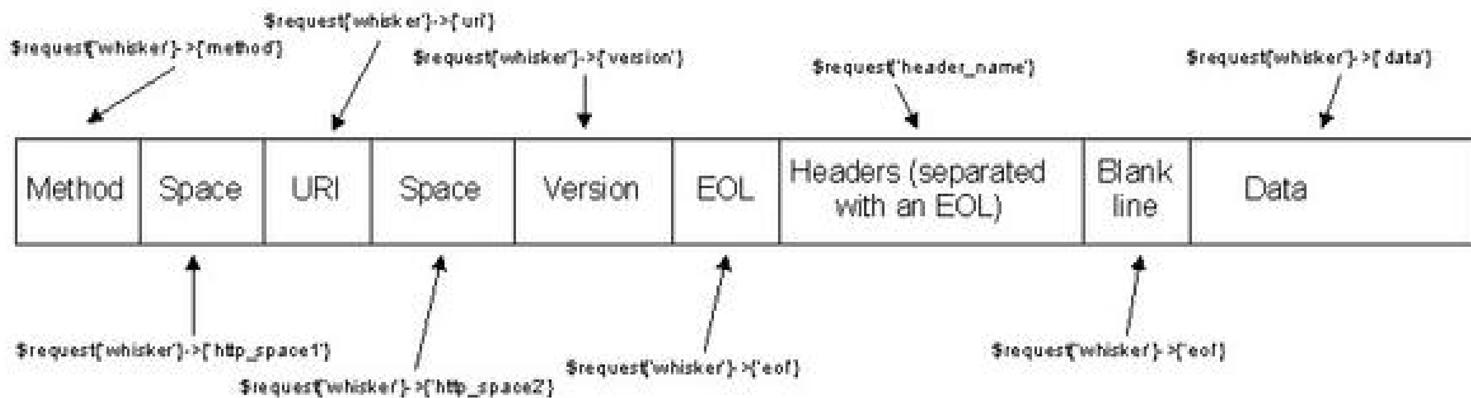

Figure 1: HTTP request packet & 'whisker' hask

Below is a diagram of an HTTP response packet that shows how to access it with the 'whisker' hash.

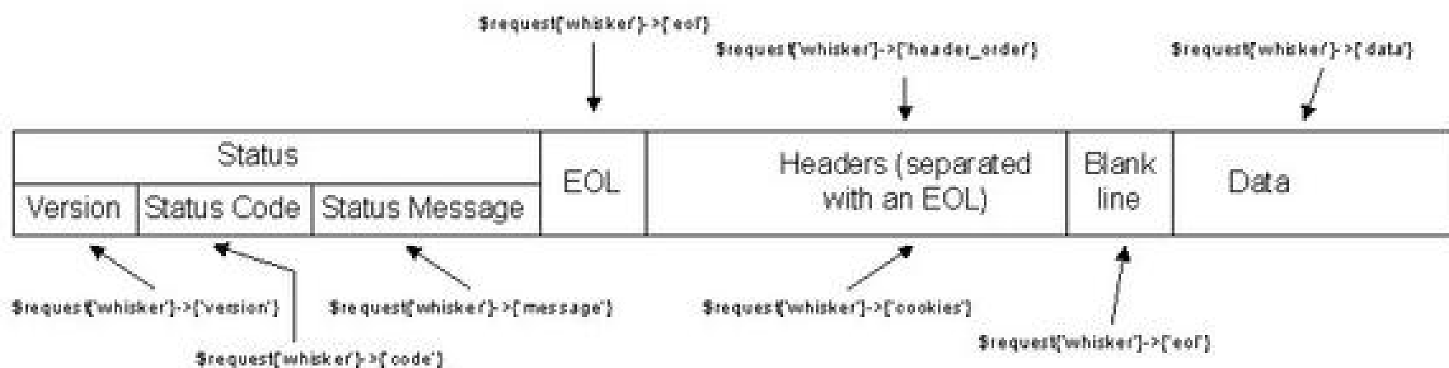

Figure 1: HTTP request packet & 'whisker' hask

Getting Started

To get used to using the Libwhisker module we will write a command line tool that allows us to follow the first five steps in "Penetration Testing of Web Applications". This will provide us a good example to base our scripts on. For each step we will add to the script. To briefly summarize, the information gathering steps we are going to script are:

1. Investigate the output from HEAD and OPTIONS http requests
2. Investigate the format and wording of 404/other error pages
3. Test for recognized file types/extensions/directories
4. Examine source of available pages
5. Manipulate inputs in order to elicit a scripting error

There are a couple of different ways to initialize the request and response hash. One is to

initialize the hashes manually [ref1] and the other is to use the Libwhisker functions 'LW2::http_new_request' and 'LW2::http_new_response' [ref2].

I. use LW2;

```
%request = ();
%response = ();
LW2::http_init_request(\%request);
$request{'whisker'}->{'host'} = "www.victim.com";
```

II. use LW2;

```
$request = LW2::http_new_request(host=>'www.victim.com', uri =>'/');
$response = LW2::http_new_response();
```

To accomplish the first of our five steps, we need to be able to send a HEAD or OPTIONS request to the server and see what type of information is sent back. This means that we are going to have to alter the value in {'whisker'}->{'method'} and print out all the headers that are returned from the server. Instead of hard coding the method we will allow it to be specified via the command line using the '-m' option. We will only allow the GET, HEAD and OPTIONS methods.

```
#Define the modules that we intend to use.
use strict;
use LW2;
use Getopt::Std;

#Define hashes for our command line options, request
#information and response information.
my (%opts, %request, %response, $headers_array, $header);

getopts('h:m:', \%opts);

#Initialize all the request variables. Some of these we will overwrite.
LW2::http_init_request(\%request);

if (!(defined($opts{h}))) {
```

```
        die "You must specify a host to scan.\n";

    }

    if (defined($opts{m})) {

        if ($opts{m} =~ /OPTIONS|HEAD|GET/) {

            $request{'whisker'}->{'method'} = $opts{m};
        } else {

            die "You can only use OPTIONS, HEAD or GET for the method.\n";

        }
    }

    ##start making requests

    #Set the host that we want to scan
    $request{'whisker'}->{'host'} = $opts{h};

    #Make RFC compliant
    LW2::http_fixup_request(\%request);

    #Do the actual scan.
    if(LW2::http_do_request(\%request,\%response)){

    ##error handling
        print 'ERROR: ', $response{'whisker'}->{'error'}, "\n";
        print $response{'whisker'}->{'data'}, "\n";

    } else {

    ##show results

        #Get the information out of the anonymous array.
```

```
            #'$headers_array' is a reference.
        $headers_array = $response{'whisker'}->{'header_order'};
        print "HTTP " ,$response{'whisker'}->{'version'}, "\t";
        print $response{'whisker'}->{'code'} , "\n";

        foreach $header (@$headers_array) {

            print "$header";
            print "\t$response{$header}\n";

        }
    }
```

The second, third, fourth, and fifth steps in our example can be combined, as they either deal with specifying a URI and/or looking at the actual HTML data returned from the server. To modify the URI from it's default '/' we need to change {'whisker'}->{'uri'}. We are also going to want to add an option to print out the HTML data. This way we can see if there is anything interesting in the HTML data that is returned like 404 messages (step2), file and directories (step3), source code (step4), and errors generated by manipulating the GET request through the URI (step 5). For the URI we will use the '-u' switch and for the printing of the HTML code we will use the '-d' switch.

Let's take a look at our new code snippet, which takes the previous one and builds on it:

```
    #Define the modules that we intend to use.
    use strict;
    use LW2;
    use Getopt::Std;

    #Define hashes for our command line options,
    #request information and response information.
    my (%opts, %request, %response, $headers_array, $header);

    ##note the additions of 'd' for data and 'u' as
    ##the option for the URI, below
    getopts('dh:m:u:', \%opts);
```

```
#Initialize all the request variables. Some of these we will overwrite.
LW2::http_init_request(\%request);

if (!(defined($opts{h}))) {

    die "You must specify a host to scan.\n";

}

if (defined($opts{m})) {

    if ($opts{m} =~ /OPTIONS|HEAD|GET/) {

        $request{'whisker'}->{'method'} = $opts{m};
    } else {

        die "You can only use OPTIONS, HEAD or GET for the method.\n";

    }
}

##now set URI if passed on command line
if (defined($opts{u})) {

    $request{'whisker'}->{'uri'} = $opts{u};

}

#Set the host that we want to scan
$request{'whisker'}->{'host'} = $opts{h};

#Make RFC compliant
LW2::http_fixup_request(\%request);

#Do the actual scan.
if(LW2::http_do_request(\%request,\%response)){
```

```
        print 'ERROR: ', $response{'whisker'}->{'error'}, "\n";
        print $response{'whisker'}->{'data'}, "\n";

    } else {

        #Get the information out of the anonymous array.
            #'$headers_array' is a reference.
        $headers_array = $response{'whisker'}->{'header_order'};
        print "\n\n";
        print "HTTP " ,$response{'whisker'}->{'version'}, "\t";
        print $response{'whisker'}->{'code'} , "\n";

        foreach $header (@$headers_array) {

            print "$header";
            print "\t$response{$header}\n";

        }

    ##if 'd' is passed, print some data
        if (defined($opts{d})) {

            print "\n\n-----------------------------
                            -----------------------------\n\n";
            print $response{'whisker'}->{'data'} , "\n";

        }
    }
```

Now we will add support for changing the 'User-Agent' header. Developers use this to determine if the client is running a browser that they support. By default the 'User-Agent' header used by Libwhisker is 'Mozilla (libwhisker/2.0)'. We will add support for three different browsers, Netscape 7.1, Microsoft IE 6 and Mozilla Firefox 0.9.

To change which browser will be spoofed, the command line option '-U' will be added to our script and it will take either N (Netscape), I (Internet Explorer) or F (Mozilla Firefox). There is more to

spoofing a browser than just changing the 'User-Agent' header, however. While this will most likely do the trick most of the time, each browser also uses other headers and values. If you want to fully spoof a browser you will have to first determine which headers are used and what their associated values are, and then set them in your script accordingly.

We also want to add support for the POST method. Some applications require the user to have a session established before allowing them to post data, so we will need to grab the cookies from the response we get and set them for our POST request. One caveat to this is the URI that we will be specifying using the '-u' option should only be used in the POST request. Initially the script will do a simple GET request and get the cookie, then set the cookie in the POST request prior to calling 'LW2::http_do_request'. Below is a script that will allow us to change the 'User-Agent' header and also do POST requests. The '-D' option will be used to specify the data that should be in the POST request.

Below is the final incarnation of our example script, which can now achieve all five of the tasks that we initially set out to automate and solve.

```
#Define the modules that we intend to use.
use strict;
use LW2;
use Getopt::Std;

#Define hashes for our command line options, request
#information, response information and cookies.
my (%opts, %request, %response, , %jar, $headers_array, $header);

##note the addition of 'U' and 'D' as options
getopts('dh:m:u:U:D:', \%opts);

#Initialize all the request variables. Some of these we will overwrite.
LW2::http_init_request(\%request);

if (!(defined($opts{h}))) {

    die "You must specify a host to scan.\n";
```

```
}

if (defined($opts{m})) {
##defaults to GET if we need to POST
    if ($opts{m} =~ /OPTIONS|HEAD|GET/) {

        $request{'whisker'}->{'method'} = $opts{m};
    }
}

if (defined($opts{u})) {
##don't set URI if method is POST
    $request{'whisker'}->{'uri'} = $opts{u} unless ($opts{m} eq "POST");

}

##now set user-agent based on 'U' option
##'F', 'I' or 'N' for Firefox/IE/Netscape
##as explained in the text
if (defined($opts{U})) {

    if ($opts{U} eq "F") {

        $request{'User-Agent'} =
                "Mozilla/5.0 (Windows; U; Windows NT 5.1;
                en-US; rv:1.7) Gecko/20040614 Firefox/0.9";

    } elsif ($opts{U} eq "I") {

        $request{'User-Agent'} =
                "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)";

    } elsif ($opts{U} eq "N") {

        $request{'User-Agent'} =
                "Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.4)
```

```
                    Gecko/20030624 Netscape/7.1 (ax)";

    } else {

        die "You did not specify a supported \'User-Agent\'.\n";

    }
}

#Set the host that we want to scan
$request{'whisker'}->{'host'} = $opts{h};

#Make RFC compliant
LW2::http_fixup_request(\%request);

#Do the actual scan.
H_REQUEST:
if(LW2::http_do_request(\%request,\%response)){

    print 'ERROR: ', $response{'whisker'}->{'error'}, "\n";
    print $response{'whisker'}->{'data'}, "\n";

} else {
##else preserve cookie info for our next request
    LW2::cookie_read(\%jar, \%response);
    #Get the information out of the anonymous array.
        #'$headers_array' is a reference.
    $headers_array = $response{'whisker'}->{'header_order'};
    print "\n\n";
    print "HTTP " ,$response{'whisker'}->{'version'}, "\t";
    print $response{'whisker'}->{'code'} , "\n";

    foreach $header (@$headers_array) {

        print "$header";
        print "\t$response{$header}\n";
```

```
        }

        if (defined($opts{d})) {

            print "\n\n----------------------------------
                                ------------------------\n\n";
            print $response{'whisker'}->{'data'} , "\n";

        }

        if ($opts{m} eq "POST") {

            LW2::cookie_write(\%jar, \%request);
            $request{'whisker'}->{'method'} = "POST";
            $request{'whisker'}->{'uri'} = $opts{u};
            $request{'whisker'}->{'data'} = $opts{d  };
            LW2::http_fixup_request(\%request);
            $opts{d} = undef;
            $opts{m} = undef;
            goto H_REQUEST;

        }
    }
```

Conclusion

Having your own suite of custom scripts and tools can be very handy when it comes to web application assessments. One of the benefits to this approach is that you know how the tools work and can add new functionality to fit your needs. This also helps in learning how an application works.

Many tools out there will tell a user that an application is vulnerable by doing some basic testing, but sometimes you need more than that. Libwhisker can be used to go beyond the information gathering phase that was shown here. Just like many security tools that use libnet and libpcap to create and read specially crafted packets, Libwhisker allows us the same type of functionality for creating and parsing HTTP packets and can very useful to a penetration tester.

Comments

Comments or reprint requests can be sent to the editor.

View more articles by Neil Desai on SecurityFocus.

Privacy Statement

Copyright 2006, SecurityFocus