YXORP

The reverse proxy for HTTP

Design document

Version: 0.1
Date: November 23, 2003

Table of Contents

Introduction

Almost since the invention of HTTP, hacking web servers has been a popular activity. As recent history has shown[1], lots of commercial web servers are still vulnerable to all kinds of attacks.

Most people react to this statement with a remark like... 'maybe you should install a firewall'. But what does a firewall do? Most firewalls are basically just port filters, that allow you to have a rule set that passes traffic from the internet as long as the destined port is 80. That does not help you against buffer overflow problems on your web server.

So what could you do then? Apply security fixes. If they are there, already... And maybe you have a commercial application, or one you have developed yourself, and you cannot fix the problem easily.

Wouldn't it be nice if you could do something else. Wouldn't it be nice if you could have some kind of check that would only allow the traffic you really want.

Well, that's what YXORP does. And it can do a lot of other things as well, although checking traffic for validity is the most important thing.

YXORP will[2]:

- Check if the URL is reasonable (i.e. is a buffer overflow attempted)
- Check if the URL is in a table of valid URLs
- Check if all headers sent in a HTTP request are reasonable (i.e. is a buffer overflow is attempted)
- Check if the content looks like HTTP
- Check any header in the request for any value.
- Check the amount of inbound and outbound traffic.
- Check the return traffic for error messages.

YXORP is a reverse proxy by nature. This means it is in the right place to do a lot of things, besides just denying or forwarding traffic based on the rules above. Current planning is to provide at least some functions for:

- Load balancing and/or failover for web servers
- Request redirection (e.g. If you have a dedicated server for small gif images, it could redirect URLs starting with /images/ to another server or server group
- Maintenance pages: if you have to take a web server offline, you can use YXORP to serve up a page to tell your users you're doing maintenance

Isn't YXORP itself vulnerable to buffer overflows and similar nasties? Well, yes, it is. However, the design is specifically aimed at preventing this kind of software errors. But still they can occur, and given that YXORP is a complicated piece of programming, there will be errors.

You should also realize that having YXORP, exciting as operating such first rate software[3] might be, is second best to having a secure server.

1 Remember, for instance, the Code Red and Nimda viruses
2 Not everything on this list will be implemented in the first release, though.
3 The careful reader will of course note that this classification is not biased in any way

Major functions explained

This chapter will provide some background as to what the security functions are intended to do, and why they're there. Not all functions are described, the focus here is more on why we would go this far to protect things.

What does a HTTP request look like

As a kind of preparation for the next chapters, we will first take a look at a HTTP request. This one I sampled as an example (btw. The URL points to a rather interesting document on the SANS website)

```
GET /newlook/digests/unicode.htm HTTP/1.1
Host: www.sans.org
User-Agent: Mozilla/5.0 (X11; U; Linux i686; en-US; rv:0.9.2) Gecko/20010726 Netscape6/6.1
Accept: text/xml, application/xml, application/xhtml+xml, text/html;q=0.9, image/png, image/jpeg, image/gif;q=0.2, text/plain;q=0.8, text/css, */*;q=0.1
Accept-Language: en-us
Accept-Encoding: gzip,deflate,compress,identity
Accept-Charset: ISO-8859-1, utf-8;q=0.66, *;q=0.66
Keep-Alive: 300
Connection: keep-alive
Referer: http://www.sans.org/newlook/digests/ntarchives/103101.htm#3
If-Modified-Since: Thu, 08 Nov 2001 19:23:49 GMT
If-None-Match: "63d40-f65a-3beadbc5"
Cache-Control: max-age=0
```

The first line contains the request. The GET part is called the *method*, where GET is the most usual one; it is used for retrieving documents from a server. Another method, POST is also interesting, because it is intended for transmitting data to a server. Sometimes, the amount of data transmitted to a web server using POST is quite large (e.g. Most webmail systems use POST data for uploading attachments to the emails you're sending)

Every line in the request, besides the first one containing GET, POST, CONNECT, or some other method, MUST start with an ID part, followed by a colon and exactly one space character. Each ID identifies some aspect of the request. Regrettably, the IDs are not standardized very well, there is no such thing as a complete list of all IDs and the functions they have. The values are as bad; they might cause different behavior on different web servers, some fields are seldomly used, some fields you can set to whatever value you like. RFC2616 standardizes a basic list of headers and header values.

Buffer overflows

Recent viruses (Code Red, Nimda) used a buffer overflow in the URL part of the request. This basically means a a request like http://your.server.com/xxxxxxxxxxxxxxxxxxxx is issued, but with a lot more X-es. Somewhere after a lot of X-es, some (executable) code is inserted in the URL. This is aimed at a certain location after the buffer your web server has for storing the URL temporarily. If this location contains something the program execution of your web server depends on, it can thus be altered and cause mayhem.

Code Red and Nimda used the URL part to do the buffer overflow. However, ANY line in the HTTP request could theoretically be used for this, and your web server might contain errors that allow for buffer overflows for any of them.

All the serious vulnerabilities I remember while writing this have used the URL part. However, most web servers are adding functionality based on the other fields, so vulnerabilities could emerge even if they are not there yet.

In general, there are a number of ways to protect against buffer overflows.

1. Your system should separate executable code from data.

   In mainframe systems, way back in 1970, every page of memory had a couple of flags specifying what functions the CPU was allowed to do with it (like read, write, execute instructions). Since this

proved to be a very useful mechanism to prevent system crashes due to (relatively simple) programming mistakes, it found its way to almost any serious system you use today (Yes, your laptop can do this!). If every application would use these functions, buffer overflows would not be possible.

However, most current operating systems do not use this function on the application level (most only use it to protect data belonging to the operating system itself).

There is at least one system I know of where you can have some influence on this yourself (if you are a sysadmin, that is): Sun's Solaris (but only on Sun Sparc hardware). Read the security advisories on how to turn it on, it is off by default.

For other operating systems/hardware combinations, well, ehhrm, read the documentation, maybe?

2. Use sane programming

   The style of programming can greatly influence the chance that buffer overflows are possible. However programming is a creative effort by humans and thus susceptible to personal style, mistakes, errors, and the like; just the same as you can be sure any program over a certain size has errors in it, you can also be sure that buffer overflows will exist.

   Still, careful programming can help a lot. And especially in client-server programming, you should never assume the client is abiding by your rules; rather, on the Internet, it is a sport to take your rules and do everything that is not specified there.

   There is a lot of literature on programming styles to avoid buffer overflows, go out there and look for it.

3. Test your applications

   One of the most useful ideas to decrease the risk of vulnerabilities in your applications, is to test for them. Obviously, testing should never replace careful programming, but still it can help to identify areas that have been overlooked.

   Lots of tools, commercial as well as open source, exist to automate this kind of testing. As a general philosophy, I would recommend to use the scanning tools that a hacker would use, since that is what you are trying to protect against. Nessus does not quite fall in this group, but is quite complete in its checking.

What does YXORP do to prevent buffer overflows

YXORP checks every line of the request. If a line is unreasonably long, it will not forward the request to the web server. YXORP can also check fields for specific values, and only forward a request if the value in the field is known and correct.

All checks are configurable.

Checking the amount of inbound and outbound data

Say you have a web site to let the general public view your annual report, but it's not doing much else. The annual report is 20Mb. If you take a look at your server logs, you see POST requests that bring in 20Mb of data. What's happened? You've been hacked, and you are providing storage space to the general public!

Similar scenario, but worse: You have a transaction processing site, and your log says you have processed GET requests for 2.5Gigabytes. Your complete database has been copied to the highest bidder.

Inspired by the fear of this last example becoming real, YXORP has a feature to limit the total inbound or outbound data. If the inbound limit is exceeded, the session is terminated before any response is sent.[4] If the outbound limit is exceeded, the session is terminated (but a part of the output will already have been sent[5]

4 But, will YXORP have already sent the request on to the webserver, or is it still buffering even for a large request? Need to think on this.

5 Same applies here, do we buffer until we're sure? Might not be possible. Could also pick up the content-length from the web server, but this might not necessarily be correct (ie streaming)

Checking if an URL is in a table of allowed URLs

Say some nasty hacker discovers a new flaw in the MacroHard DisInformationServer[6]. It works as follows: Joe Hacker formats the URL to contain a number of backspaces larger than the normal part, and the web server translates this to a file above its document root. Oops, we've got the directory traversal problem again! But on www.yxorp.com, this does not work... because the all URLs are checked to a table before they are passed to the DisInformationServer!

Well, this is not exactly a new feature. There are many proxies that do this, mainly to keep corporate browsers away from p0rn sites etc. Basically there are two strategies: blacklisting and whitelisting. The way YXORP will do this is trough a whitelist, that is: every conceivable piece of data allowed through must be named.

This might seem a major task for a webserver with more than just an index page. It is. Nevertheless, I know of serveral servers that are protected by these techniques, and happily so... They were vulnerable to Nimda, Code Red, etc; they were attacked all right, but not infected...

Still this is a feature you should very well think about before activating. It will impact the change procedures on the web server; programmers and content managers will be forced to provide the firewall people with documentation before the new content can go live (depending on your procedures and organization, this might not be a bad thing). Also, an error in the URL table means the site does not work (at least in part).

To make this a bit easier, YXORP will have some features to make table building easier. There might be all of the following:

- Learning mode: YXORP will remember all URLs it passes, and generate a table from it. Then someone issues a magik comand and YXORP will switch to deny mode; anything not in the table will be denied from then on[7].
- Warn mode: Same idea as above, but YXORP will nag you like: Did you know I just processed this URL you forget to tell me about?
- Expressions in the table (like *.html, *.gif or maybe even more specific like [Ii]ndex.php)

Checking if the traffic is actually HTML

What if your server has been hacked, and someone is accessing cmd.exe trough it? Answer: the traffic would use port 80 (and flash trough any firewall i.e. packet filter just fine) but it would not be HTML. YXORP can check this.

Well, ehhrm, YXORP can do some basic simple checks. Checking for all possible tricks you can currently do in HTML goes a bit too far[8].

Web servers carry other traffic besides HTML (remember most pages insert loads of gifs, they are not HTML). Some of these can be easily recognised (eg. GIF89), and can be compared to MIME types. Clearly this goes a bit too far for a first release, but it can be done!

Checking all request headers

Besides the obvious buffer overflow story, there's more. Say you have an application running on your server, but it does not work right with the browser your users started using yesterday. What can you do? You could do some Javascript tricks to solve this, but : you've bought the application and the company went bankrupt, the programmer who built it went to Turkey for an extended stay in a zero-

6 If you think I'm making fun of a certain company, I'll found MacroHard, and write the server just to prove you wrong.

7 In retrospect, I'm not all that sure I like this functionality. It might be taken out of the release planning.

8 But the <html> and </html> tags should be there, and no problem to check. Going a bit further might be possible. Another trick, very easy to implement, would be statistical analysis of the data, and demand a tag for every X bytes out of a tag. But this would not work for all content.

star hotel operated by the government and isn't expected back this decade, the disks in the server are crashing and you dare not approach for fear they will stop spinning. So one thing remains: have YXORP redirect the traffic based on the User-Agent field in the request header to some other page, fix the problem in there, and all other users will still take the original page.

A better example could be based on the accept-language header. Newer browsers use this to tell the server what language version they are running. You might want to speak the same language to them, but maybe your web server has no support for this yet.

Checking the return traffic for error messages

In the response a web server sends out, there are sometimes pieces of information you do not want to go out. To explain this, let's take a look at a normal response from a web server.

```
HTTP/1.1 304 Not Modified
Date: Wed, 06 Mar 2002 00:20:31 GMT
Server: Apache
Connection: Keep-Alive
Keep-Alive: timeout=5, max=100
```

This server (it's the SANS server again, actually) is not telling us very much besides that it might be running Apache (some servers, and Apache certainly, allow you to change this). Some servers report the version they are running; this is a clear example of information you do not want to post on your front door. If hackers need to know, let them work a bit harder to find out.

Much worse is if something really goes wrong e.g. HTTP/1.1 500 Server Failed, especially if this includes a very detailed description of exactly what line in some piece of Java code caused traps in what modules, what the environment was at the time, what JRE was running, etc. Yes, this happens.

Not all of these situations can be 'improved' from YXORP's point of view. What might be useful, is checking the server response codes, and decide if you want to send them to your clients or if you would rather send them nothing (and let them wait on a timeout). A maintenance page would be friendlier, probably.

Maintenance pages

For some reason, your server is offline. So your clients see nothing but the familiar popup, after a couple of minutes. Maybe it would be better to have something that sends them a page saying you're doing maintenance...

Since YXORP is normally sitting in the traffic flow, it could send this for the server that is not responding. Or it could redirect the request to a web server that is online, and serves up this page[9].

9 Big philosophical difference between these two options. There is the argument that a firewalling device should not run services itself. Besides that, having content (and content managers, corporate style guide enforcers, web programmers, etc) on your YXORP would not be the most secure implementation ever. Still, the other option would require another server. Maybe it's best to allow for both, and leave the choise to someone else.