wallarm

Securing Web Apps with NGINX

http://wallarm.com

Stephan Ilyin, si@wallarm.com

SOLD OUT!

nginx.conf 2014

How many of you have your websites hacked?

wallarm

Each application probably has vulnerabilities

… and someday it
can be hacked

How to harder/secure your application?

How deal with attacks to your application?

Chapter 1.

wallarm

Tip #1. mod_security can be a good choice

Mod_security rocks!

- Open-source. Finally available for NGINX
- It works! It can be quite efficient in detecting attacks
- Supports virtual patching
- It is incredible customisable

```
 server {
      listen       80;
      server_name  localhost;

         location / {
              ModSecurityEnabled on;
              ModSecurityConfig modsecurity.conf;
              ModSecurityPass @backend;
         }

         location @backend {
              proxy_pass http://localhost:8011;
              proxy_read_timeout 180s;
        }
}
```

but mod_security is not so good!

- Relies on regex
- It is expensive in performance prospective
- If you use default rulesets, you will get a huge number of false-positives
- Rules tuning is a hard job (difficult to maintain)
- Signatures never covers all the attacks
- REGEXs can be bypassed

What rules look like

# ShellShock virtual patch (Bash attack)

```
SecRule REQUEST_HEADERS
"^\(\s*\)\s+{" "phase:1,deny,id:
1000000,t:urlDecode,status:
400,log,msg:'CVE-2014-6271 - Bash
Attack'"
```

Good practice (imho)

- Use public ruleset — *for monitoring mode*
- Craft rules from scratch specifically for your application — *for blocking mode*

More rules =
More overhead!

Using phases is good idea

1. Request headers (REQUEST_HEADERS)
2. Request body (REQUEST_BODY)
3. Response headers (RESPONSE_HEADERS)
4. Response body (RESPONSE_BODY)
5. Logging (LOGGING)

SecRule phase 2

```
SecRule REQUEST_BODY "/+etc/+passwd"
"t:none,ctl:ResponseBodyAccess=On,msg:'-
IN- PASSWD path detected', phase:
2,pass,log,auditlog,id:'10001',t:urlDeco
de,t:lowercase,severity:1"
```

SecRule phase 4

```
SecRule RESPONSE_BODY "root\:x\:0\:0" "id:'20001',ctl:auditLogParts=+E, msg:'-OUT- Content of PASSWD detected!',phase:4,allow,log,auditlog,t:lowercase,severity:0"
```

HANDBOOK

The Complete Guide to the Popular
Open Source Web Application Firewall

Sample

Handbook by Ivan Ristic. Must read!

Tip #2. Give a chance to naxsi (another WAF for NGINX)

Why naxsi?

- NAXSI means Nginx Anti Xss & Sql Injection (but do more)
- Naxsi doesn't rely on a signature base (regex)!

https://github.com/nbs-system/naxsi

naxsi rules

- Reads a small subset of simple scoring rules (naxsi_core.rules) containing 99% of known patterns involved in websites vulnerabilities.

- For example, '<', '|' or 'drop' are not supposed to be part of a URI.

This rule triggers on *select* or other SQL operators

```
MainRule "rx:select|union|update|delete|
insert|table|from|ascii|hex|unhex|drop"
"msg:sql keywords" "mz:BODY|URL|ARGS|
$HEADERS_VAR:Cookie" "s:$SQL:4" id:1000;
```

naxsi setup

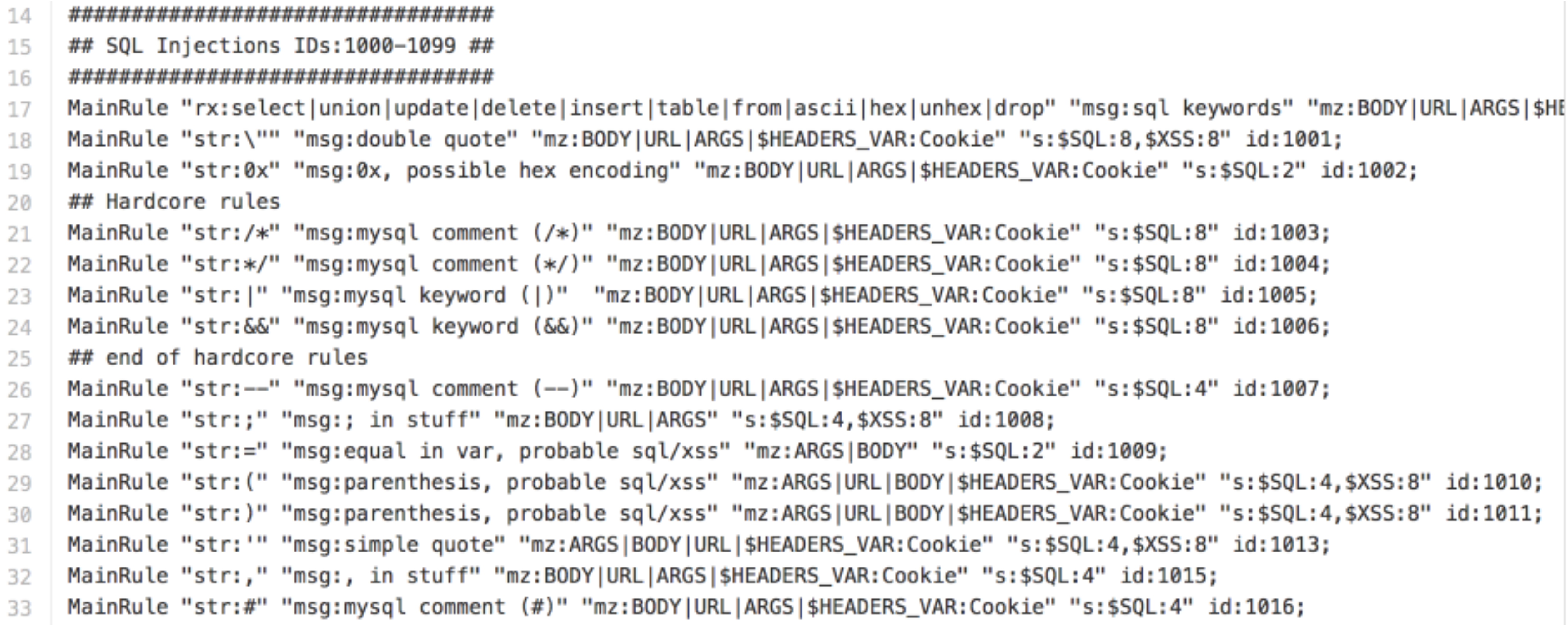
```
http {
  include /etc/nginx/naxsi_core.rules;
  include /etc/nginx/mime.types;

  [...]
}
```

But! Ruleset is not enough!

- Those patterns may match legitimate queries!
- Therefore, naxsi **relies on whitelists** to avoid false positives
- Nxutil tool helps the administrator to create the appropriate whitelist
- there are pre-generated whitelists for some CMS (e.g. WordPress)

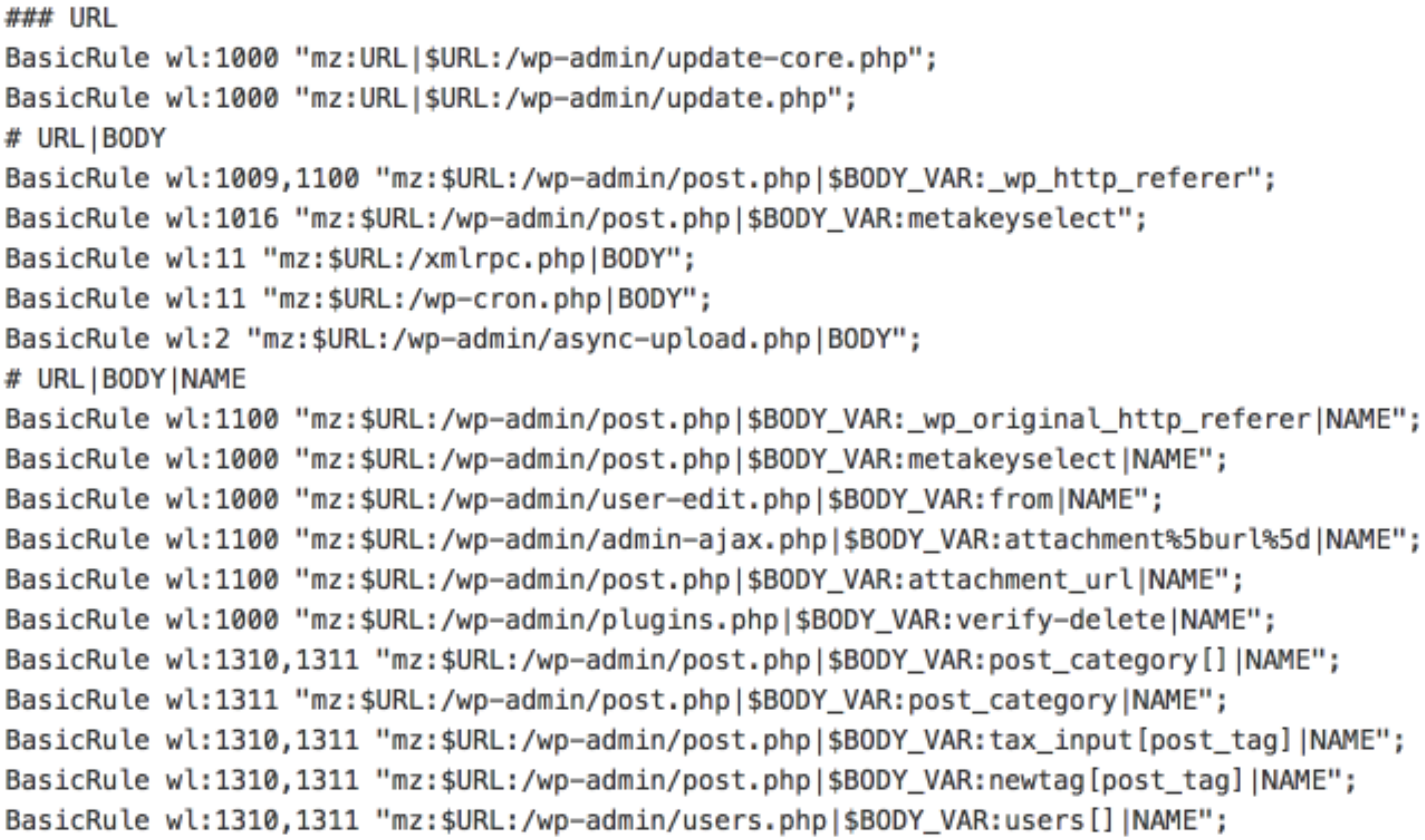
```
LearningMode; #Enables learning mode

SecRulesEnabled;
#SecRulesDisabled;

DeniedUrl "/RequestDenied";

## check rules
CheckRule "$SQL >= 8" BLOCK;
CheckRule "$RFI >= 8" BLOCK;
CheckRule "$TRAVERSAL >= 4" BLOCK;
CheckRule "$EVADE >= 4" BLOCK;
CheckRule "$XSS >= 8" BLOCK;
```

naxsi ruleset



```
####################################
## SQL Injections IDs:1000-1099 ##
####################################
MainRule "rx:select|union|update|delete|insert|table|from|ascii|hex|unhex|drop" "msg:sql keywords" "mz:BODY|URL|ARGS|$HE
MainRule "str:\"" "msg:double quote" "mz:BODY|URL|ARGS|$HEADERS_VAR:Cookie" "s:$SQL:8,$XSS:8" id:1001;
MainRule "str:0x" "msg:0x, possible hex encoding" "mz:BODY|URL|ARGS|$HEADERS_VAR:Cookie" "s:$SQL:2" id:1002;
## Hardcore rules
MainRule "str:/*" "msg:mysql comment (/*)" "mz:BODY|URL|ARGS|$HEADERS_VAR:Cookie" "s:$SQL:8" id:1003;
MainRule "str:*/" "msg:mysql comment (*/)" "mz:BODY|URL|ARGS|$HEADERS_VAR:Cookie" "s:$SQL:8" id:1004;
MainRule "str:|" "msg:mysql keyword (|)"  "mz:BODY|URL|ARGS|$HEADERS_VAR:Cookie" "s:$SQL:8" id:1005;
MainRule "str:&&" "msg:mysql keyword (&&)" "mz:BODY|URL|ARGS|$HEADERS_VAR:Cookie" "s:$SQL:8" id:1006;
## end of hardcore rules
MainRule "str:--" "msg:mysql comment (--)" "mz:BODY|URL|ARGS|$HEADERS_VAR:Cookie" "s:$SQL:4" id:1007;
MainRule "str:;" "msg:; in stuff" "mz:BODY|URL|ARGS" "s:$SQL:4,$XSS:8" id:1008;
MainRule "str:=" "msg:equal in var, probable sql/xss" "mz:ARGS|BODY" "s:$SQL:2" id:1009;
MainRule "str:(" "msg:parenthesis, probable sql/xss" "mz:ARGS|URL|BODY|$HEADERS_VAR:Cookie" "s:$SQL:4,$XSS:8" id:1010;
MainRule "str:)" "msg:parenthesis, probable sql/xss" "mz:ARGS|URL|BODY|$HEADERS_VAR:Cookie" "s:$SQL:4,$XSS:8" id:1011;
MainRule "str:'" "msg:simple quote" "mz:ARGS|BODY|URL|$HEADERS_VAR:Cookie" "s:$SQL:4,$XSS:8" id:1013;
MainRule "str:," "msg:, in stuff" "mz:BODY|URL|ARGS|$HEADERS_VAR:Cookie" "s:$SQL:4" id:1015;
MainRule "str:#" "msg:mysql comment (#)" "mz:BODY|URL|ARGS|$HEADERS_VAR:Cookie" "s:$SQL:4" id:1016;
```

naxsi whitelist



```
### URL
BasicRule wl:1000 "mz:URL|$URL:/wp-admin/update-core.php";
BasicRule wl:1000 "mz:URL|$URL:/wp-admin/update.php";
# URL|BODY
BasicRule wl:1009,1100 "mz:$URL:/wp-admin/post.php|$BODY_VAR:_wp_http_referer";
BasicRule wl:1016 "mz:$URL:/wp-admin/post.php|$BODY_VAR:metakeyselect";
BasicRule wl:11 "mz:$URL:/xmlrpc.php|BODY";
BasicRule wl:11 "mz:$URL:/wp-cron.php|BODY";
BasicRule wl:2 "mz:$URL:/wp-admin/async-upload.php|BODY";
# URL|BODY|NAME
BasicRule wl:1100 "mz:$URL:/wp-admin/post.php|$BODY_VAR:_wp_original_http_referer|NAME";
BasicRule wl:1000 "mz:$URL:/wp-admin/post.php|$BODY_VAR:metakeyselect|NAME";
BasicRule wl:1000 "mz:$URL:/wp-admin/user-edit.php|$BODY_VAR:from|NAME";
BasicRule wl:1100 "mz:$URL:/wp-admin/admin-ajax.php|$BODY_VAR:attachment%5burl%5d|NAME";
BasicRule wl:1100 "mz:$URL:/wp-admin/post.php|$BODY_VAR:attachment_url|NAME";
BasicRule wl:1000 "mz:$URL:/wp-admin/plugins.php|$BODY_VAR:verify-delete|NAME";
BasicRule wl:1310,1311 "mz:$URL:/wp-admin/post.php|$BODY_VAR:post_category[]|NAME";
BasicRule wl:1311 "mz:$URL:/wp-admin/post.php|$BODY_VAR:post_category|NAME";
BasicRule wl:1310,1311 "mz:$URL:/wp-admin/post.php|$BODY_VAR:tax_input[post_tag]|NAME";
BasicRule wl:1310,1311 "mz:$URL:/wp-admin/post.php|$BODY_VAR:newtag[post_tag]|NAME";
BasicRule wl:1310,1311 "mz:$URL:/wp-admin/users.php|$BODY_VAR:users[]|NAME";
```

Naxsi pros and cons

Pros:

- Pretty fast!
- Update independent
- Resistant to many waf-bypass techniques

Cons:

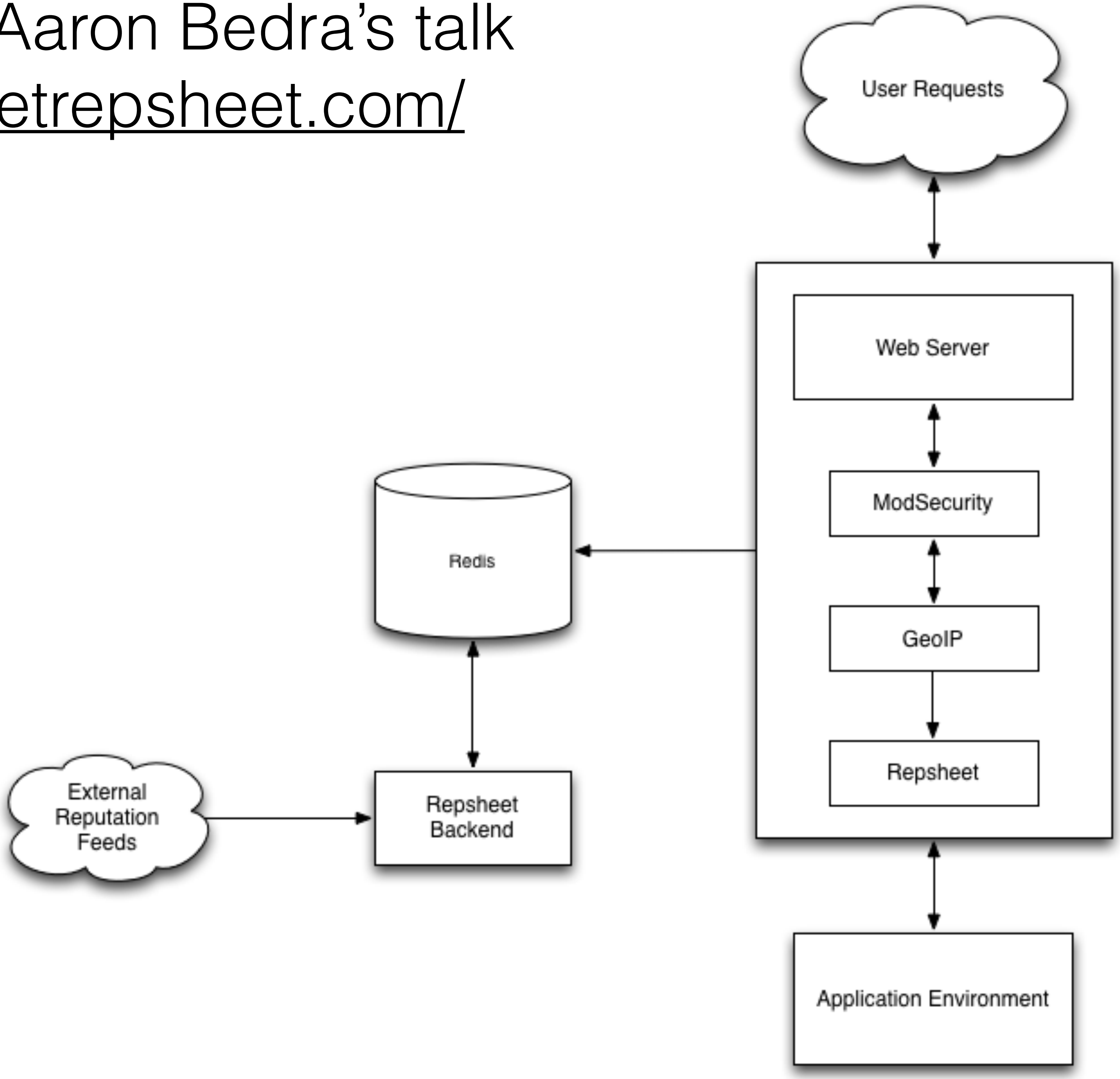
- You need to use LearningMode with each significant code deployment

Tip #3.

Try repsheet

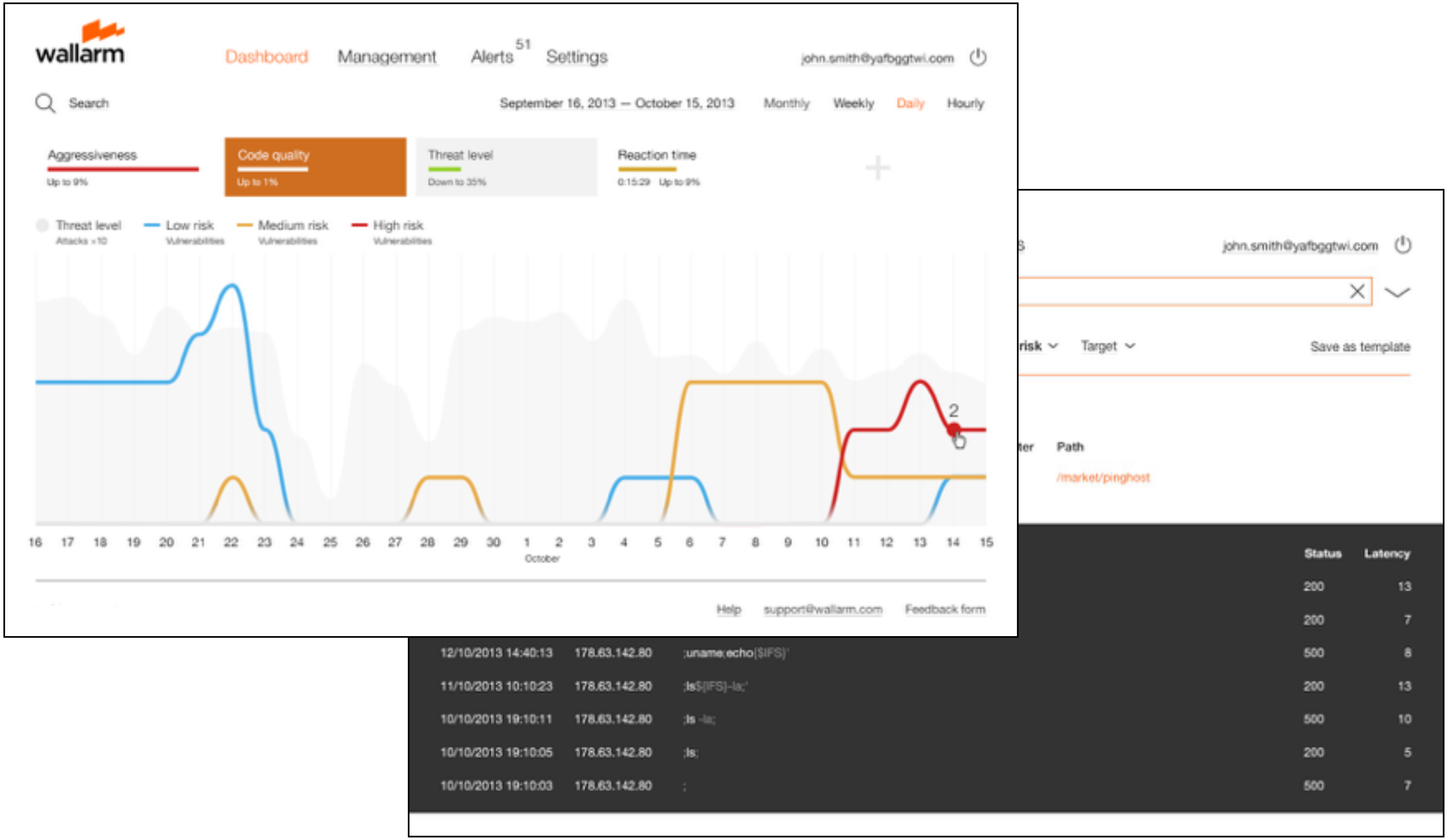
(behaviour based security)

Watch Aaron Bedra's talk
http://getrepsheet.com/



Tip #4.

And there is also
Wallarm WAF based on NGINX



			Status	Latency
			200	13
			200	7
12/10/2013 14:40:13	178.63.142.80	;uname;echo${IFS}'	500	8
11/10/2013 10:10:23	178.63.142.80	;ls${IFS}-la;'	200	13
10/10/2013 19:10:11	178.63.142.80	;ls -la;	500	10
10/10/2013 19:10:05	178.63.142.80	;ls;	200	5
10/10/2013 19:10:03	178.63.142.80	;	500	7

http://wallarm.com

wallarm

How deal with DDoS?

Chapter 2.

wallarm

How to deal with DDoS?

- The traditional technique for self-defense is to read the HTTP server’s log file, write a pattern for grep (to catch bot requests), and ban anyone who falls under it.
- That’s not easy!
- The following are tips on where to place pillows in advance so it won’t hurt so much when you fall.

Tip #5.

Use test_cookie module

Use test_cookie module

- Usually HTTP-flooding bots are pretty stupid
- Lack HTTP cookie and redirect mechanisms
- Testcookie-nginx works as a quick filter between the bots and the backend during L7 DDoS attacks, allowing you to screen out junk requests

Use test_cookie module

Straightforward checks:

- Whether the client can perform HTTP Redirect
- Whether it supports JavaScript
- Whether it supports Flash

Use test_cookie module

In addition to its merits, test_cookies also has its drawbacks:

- Cuts out all bots (including Googlebot)
- Creates problems for users with Links and w3m browsers
- Does not protect against bots with full-browser-stack

https://github.com/kyprizel/testcookie-nginx-module

Tip #6. Code 444

Code 444

- The goal of DDoSers is often the most resource-intensive part of the site.
- A typical example is a search engine. Naturally, it can be exploited by charging tens of thousands of queries
- So what can we do?

Code 444

- Temporarily disable this search function
- Nginx supports custom code 444, which allows you to simply close the connection and give nothing in response

Code 444

```
location /search {
    return 444;
}
```

Tip #7. Use ipset

Ban bots’ IPs with ipset

- If you’re sure that location/search requests are coming only from bots

- Ban bots (getting 444) with a simple shell script
ipset -N ban iphash

```
tail -f access.log | while read LINE; do
echo “$LINE” | cut -d’”’ -f3 | cut -d’ ‘
-f2 | grep -q 444 && ipset -A ban “${L%%
*}”; done
```

Tip #8. Banning based on geographic indicators

wallarm

Tip #8. Banning based on geographic indicators

- You can strictly limit certain countries that make you feel uneasy

- But. It is a bad practice! GeoIP data isn’t completely accurate!

Tip #8. Banning based on geographic indicators

- Connect to the nginx GeoIP module
- Display the geographic indicator information on the access log
- grep the nginx access log and add clients by geographic indicators to the ban list.

Tip #9. You can use neural network!

Tip #9. You can use neural network

- Bad request:

```
0.0.0.0 - - [20/Dec/2011:20:00:08 +0400] "POST
/forum/index.php HTTP/1.1" 503 107 "http://
www.mozilla-europe.org/" “-"
```

- Good request:

```
0.0.0.0 - - [20/Dec/2011:15:00:03 +0400]
"GET /forum/rss.php?topic=347425 HTTP/1.0" 200
1685 "-" "Mozilla/5.0 (Windows; U; Windows NT
5.1; pl; rv:1.9) Gecko/2008052906 Firefox/3.0"
```

Tip #9. You can use neural network

Use Machine Learning (ML) to detect bots:

- use neural network (e.g. PyBrain)
- stuffed logs inside
- analyse the requests for classification between "bad" and "good" clients under DDoS

A good proof-of-concept:
https://github.com/SaveTheRbtz/junk/tree/master/neural_networks_vs_ddos

Tip #9. You can use neural network

- Useful to have the access.log before a DDoS attack, because it lists virtually 100% of your legitimate clients

- It is an excellent dataset for neural network training

Tip #10.

Keep track of the number of requests per second

Tip #10. Keep track of the number of requests per second

- You can estimate this value with the following shell command

```
echo $(($(fgrep -c "$(env LC_ALL=C date
--date=@$(($(date +%s)-60)) +%d/%b/%Y:
%H:%M)" “$ACCESS_LOG”)/60))
```

Tuning the web server

- Of course, you put nginx on silent and hope that everything will be OK.
- However, things are not always OK.
- So the administrator of any server should devote a lot of time to tweaking and tuning nginx.

Tip #11.

Limit buffer sizes and timeouts in NGINX

wallarm

Every resource has a limit

- Every resource has a limit. In particular, this applies to memory.
- the size of the header and all buffers need to be limited to adequate values on the client and on the server as a whole

Limit buffers

- client_header_buffer_size
- large_client_header_buffers
- client_body_buffer_size
- client_max_body_size

And time_outs

- reset_timeout_connection
- client_header_timeout
- client_body_timeout
- keepalive_timeout
- send_timeout

Question: what are the correct parameters for the buffers and timeouts?

- There’s no universal recipe here

- But there is a proven approach you can try

How to limit buffers and timeout?

1. Mathematically arrange the minimum parameter value.

2. Launch site test runs.

3. If the site’s full functionality works without a problem, the parameter is set.

4. If not, increase the parameter value and go to step 2.

Tip #12.

Limit connections in NGINX (limit_conn and limit_req)

Ideally you need to test application to see *how many requests it can handle* and set that value in the NGINX configuration

```
http {
  limit_conn_zone $binary_remote_addr zone=download_c:10m;
  limit_req_zone $binary_remote_addr zone=search_r:10m
rate=1r/s;

  server {
    location /download/ {
      limit_conn download_c 1;
      ..
   }
   location /search/ {
      limit_req zone=search_r burst=5;
      ..
   }

  }
}
```

What to limit?

- It makes sense to set limits for limit_conn and limit_req for locations where it’s costly to implement scripts

- You can also fail2ban utility here: http://www.fail2ban.org

Bad practices / How not to configure NGINX

Chapter 3.

Bad practices

- NGINX has secure-enough defaults
- Sometimes administrators can make mistakes cooking it

Tip #13.
Be careful with
rewrite with $uri

wallarm

rewrite with $uri

- Everyone knows $uri /
("normalized" URI of the request)

- normalization is decoding the text encoded in the '%XX' form, resolving references to the relative path components '.' and '..', and possible compression of two or more adjacent slashes into a single slash

rewrite with $uri

Typical HTTP -> HTTPS redirect snippet:

```
location / {
    rewrite ^ https://$host/$uri;
}

location / {
    return 302 https://$host$uri;
}
```

What can go wrong? CRLF (%0d%0a) comes to play

rewrite with $uri

- Request

```
GET /test%0d%0aSet-Cookie:%20malicious%3d1 HTTP/1.0
Host: yourserver.com
```

- Respond

```
HTTP/1.1 302 Moved Temporarily
Server: nginx
Date: Mon, 02 Jun 2014 13:08:09 GMT
Content-Type: text/html
Content-Length: 154
Connection: close
Location: https://yourserver.com/test
Set-Cookie: malicious=1
```

Use $request_uri instead of $uri

Tip #14. Pay attention to try_files

try_files

- try_files checks the existence of files in the specified order and uses the first found file for request processing
- if none of the files were found, an internal redirect to the URI specified in the last parameter is made

try_files

There is a Django project

```
$ tree /your/django/project/root
+-- media
+---- some_static.css
+-- djangoproject
+---- __init__.py
+---- settings.py
+---- urls.py
+---- wsgi.py
+-- manage.py
```

try_files

Administrators decide to serve static files with nginx and use this configuration

```
root /your/django/project/root;

location / {
    try_files $uri @django;
}

location @django {
    proxy_pass http://django_backend;
}
```

try_files: what’s wrong?

- NGINX will first try to serve static file from root, and only if it does not exists pass the request to @django location

- Therefore, anyone can access manage.py and all of the project sources (including djangoproject/settings.py)

Tip #15. Use disable_symlinks if_not_owner

Hosters usually do this

```
location /static/ {
  root /home/someuser/www_root/static;
}
```

What’s the problem?

User can create symlink to any file available to nginx worker (including files of another users)!

```
[root@server4 www]# ls -alh
total 144K
drwxr-x--- 6 usertest nobody 4.0K Apr 10 20:09 .
drwx--x--x 13 usertest usertest 4.0K Apr 7 02:16 ..
-rw-r--r-- 1 usertest usertest 184 Apr 6 21:29 .htaccess
lrwxrwxrwx 1 usertest usertest 38 Apr 6 22:48 im1.txt -> /home/
another_user/public_html/config.php
-rw-r--r-- 1 usertest usertest 3 May 3 2011 index.html
```

What you can do

1. Turn off symlinks (and users will suffer)

2. Use option *disable_symlinks if_not_owner* (best choice)

wallarm

Slides:

bit.ly/nginx_secure_webapps

http://wallarm.com

Stephan Ilyin, si@wallarm.com