



ORF
FUSION

CASE STUDY: HEADLESS BROWSERS IN WEB FORUM SPAM

Vamsoft e-Security Kft.

Peter Karsai (peter.karsai@vamsoft.com, [@peterkarsai](https://twitter.com/peterkarsai))
Ciprian Benyi (ciprian.benyi@vamsoft.com)

CONTENTS

Background	3
Investigation	4
About our JavaScript spambot challenge	4
Initial data	4
Profiling the browser.....	6
Enter the headless browser	7
Conclusions	10
Possible responses.....	10

BACKGROUND

We've been through a few episodes with comment spam on the Vamssoft Community Forums, so when in early May 2014 we started receiving a new wave, we just shrugged and attributed it to another mislead soul posting links in hope of earning a few dollars with a spam affiliate program.

Comment spam and web form abuse in general is a well-known phenomenon on the Internet. It primarily affects forums, blogs and other services that don't require registration prior to posting. Spammer robots—or "*spambots*" for short—frequently attack any form on a webpage that is slightly reminiscent of a comment form. For all the mess they do, conventional wisdom of the web developer community holds that spambots are relatively dumb programs, specially crafted for extracting, populating and posting forms retrieved from the HTML source code of web pages.



Figure 1: A typical post made by a spambot

The crude nature of spambots also makes defense against them very easy. Their limited understanding of a web page can be exploited to the defender's purposes by posing a challenge a glorified web page parser won't understand—for instance, requiring JavaScript code to be run prior to posting. Spam posted by humans will still get through, but the volume of affiliate marketing spam¹ (the primary driving force for such posts) is naturally limited by the number of posts an affiliate can make in a day.

Our forum is susceptible for comment spam attacks, because we don't require registration or completing CAPTCHAs before posting. We love our forum that way, because it respects the posters' privacy and there are no annoying hoops to jump through. We use a homebrew JavaScript solution as the spambot challenge, which does an excellent job separating humans (who use actual, JavaScript-enabled web browsers) from spambots.

All things considered, we were confident that our spammer is a human being who'll eventually give up and move on. Only they didn't. When we got bored of moderating their posts and started

¹ An affiliate marketing scheme where affiliates get paid for posting links to forums in order to drive traffic to the advertisers' website.

railing them into HTTP 500 errors, they kept on trying. This was remarkably unlike human behavior and naturally posed the question: *if they're not human, then what is it that defeats our JavaScript challenge?*

This case study summarizes the short story of our investigation and our findings about the sophisticated way of how our spammer simulates a true web browser for improved spam deliverability. We also take a brief glimpse at possible options of defeating the spammers.

INVESTIGATION

ABOUT OUR JAVASCRIPT SPAMBOT CHALLENGE

We use a fairly simple trick to separate humans from spambots: we insert an extra hidden field in our comment form using JavaScript at runtime, which is verified by our server when the comment is posted. The assumption is that humans use a “*true web browser*” like Google Chrome or Mozilla Firefox and true web browsers run JavaScript just fine, sending along our extra field with their post. Spambots, however, don't understand JavaScript and will fail miserably on this test.

In this particular case, we had a strong suspicion that the spambot has perfect understanding of JavaScript and so our assumption no longer holds.

INITIAL DATA

Investigating our spambot started with the requests it issued.

The bot itself operated in a fairly low-profile way, visiting 1-3 times a day, making only a handful of posts in quick succession. This was consistent with the behavior of a human visitor, although the posts were made slightly faster than expected from a human.

The format of the posts was the same all the time (*see Figure 1*), with links pointing to drug store websites selling erectile dysfunction drugs. We collected a total of 128 distinct URLs from 1,485 total target URLs found in 400 posts. A manual examination showed that all, but 3 URLs pointed to the website root path (/) and none of the URLs featured an Affiliate ID² usually seen in affiliate marketing spam URLs, which ruled out affiliate spam.

² In affiliate schemes, an Affiliate ID helps measuring the traffic driven by the affiliate, e.g. <http://example.org/?affiliate=58442> has an Affiliate ID “58442” embedded in the URL.

A cursory look at the web server logs revealed nothing extraordinary about our visitor. The requests followed the same pattern as with any regular browser—fetching the HTML page was followed by scripts, images, and other resources.

```
GET /App_Themes/Public/css/forum.css v=7 80 - 95.211.192.231 HTTP/1.1 Mozilla/4.0+(Windows+NT+6.2)+AppleWebKit/537.17+(KHTML+like+Gecko)+Chrome/24.0.1312.70+Safari/537.17
GET /Scripts/Public/vs-global.js v=6 80 - 95.211.192.231 HTTP/1.1 Mozilla/4.0+(Windows+NT+6.2)+AppleWebKit/537.17+(KHTML+like+Gecko)+Chrome/24.0.1312.70+Safari/537.17
GET /Scripts/Public/vs-forum.js v=6 80 - 95.211.192.231 HTTP/1.1 Mozilla/4.0+(Windows+NT+6.2)+AppleWebKit/537.17+(KHTML+like+Gecko)+Chrome/24.0.1312.70+Safari/537.17
GET /Scripts/Public/jquery.merged-plugins.js v=6 80 - 95.211.192.231 HTTP/1.1 Mozilla/4.0+(Windows+NT+6.2)+AppleWebKit/537.17+(KHTML+like+Gecko)+Chrome/24.0.1312.70+Safari/537.17
GET /Scripts/jquery.dd.js v=6 80 - 95.211.192.231 HTTP/1.1 Mozilla/4.0+(Windows+NT+6.2)+AppleWebKit/537.17+(KHTML+like+Gecko)+Chrome/24.0.1312.70+Safari/537.17 VSSessionID=
GET /Scripts/Public/jquery.tools.min.js v=6 80 - 95.211.192.231 HTTP/1.1 Mozilla/4.0+(Windows+NT+6.2)+AppleWebKit/537.17+(KHTML+like+Gecko)+Chrome/24.0.1312.70+Safari/537.17
```

Figure 2: Requests in the web server logs

The *User-Agent* string was more interesting, though.

```
Mozilla/4.0 (Windows NT 6.2) AppleWebKit/537.17 (KHTML, like Gecko)
Chrome/24.0.1312.70 Safari/537.17
```

The string suggests that the browser is *Google Chrome*, running on either *Microsoft® Windows® 8* or *Microsoft® Windows Server® 2012*. However, the build number *24.0.1312.70* reveals that something is off, because this build number belongs to a Linux-only release of Chrome³.

Indeed, when we looked up this *User-Agent* on the internet, we found a slightly different version:

```
Mozilla/5.0 (X11; Linux i686) AppleWebKit/537.17 (KHTML, like Gecko)
Chrome/24.0.1312.70 Safari/537.17
```

which indicates a 64-bit Linux version of Chrome/Chromium.

This suggests that the *User-Agent* string was modified intentionally to cover up the actual browser being used.

Our logs also had the spambot IP addresses recorded. To our surprise, we found that the hundreds of spam comments came from just two IPs:

- 5.79.73.142
- 95.211.192.231

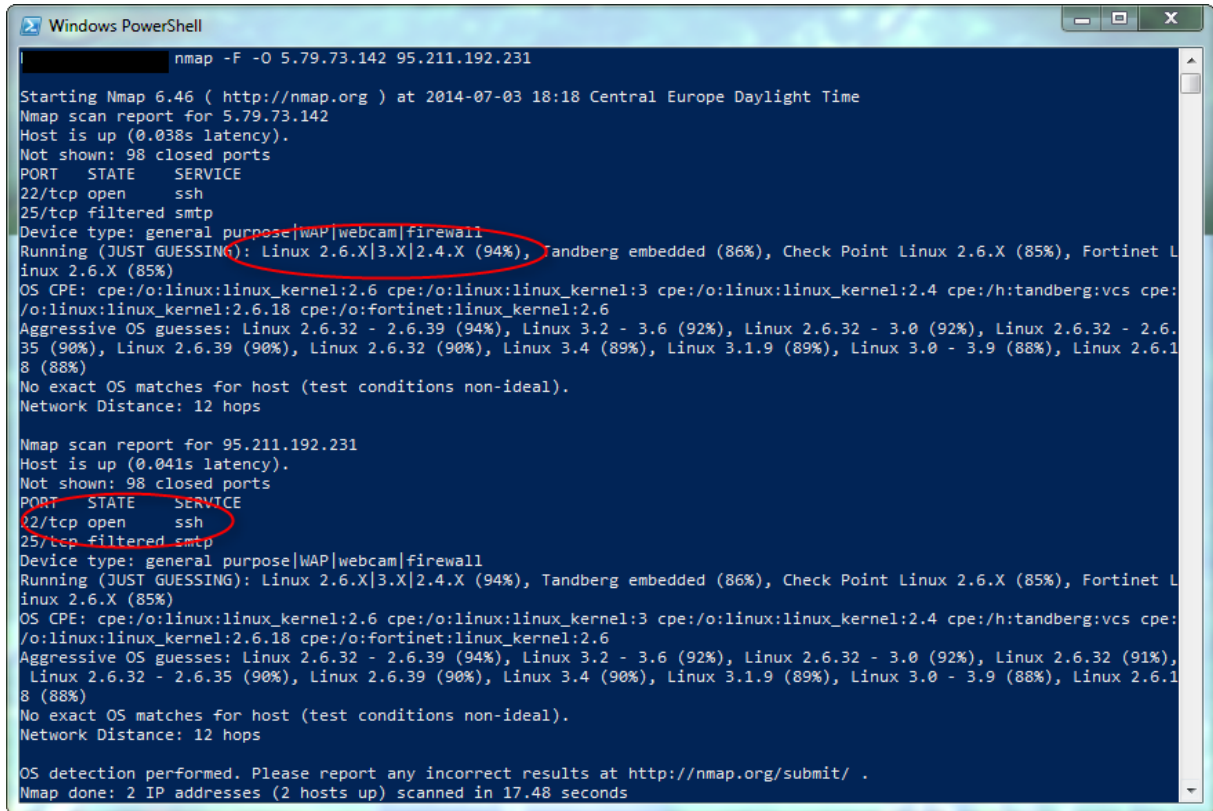
Neither of the IP addresses has reverse DNS information. They both belong to the network of LeaseWeb B.V.⁴, a Netherlands-based hosting provider (we reported the hosts to abuse@leaseweb.com).

We used the [nmap](#) tool to profile the spambot hosts. Operating system detection reported *Linux* with high confidence. A partial port scan also reported the SSH port TCP/22 open, which further supported Linux as the spambot host OS.

³ Source: http://googlechromereleases.blogspot.com/2013/02/stable-channel-update_12.html

⁴ RIPE WHOIS data retrieved via <http://who.is> on July 7, 2014.

It should be noted that these hosts are might have been hijacked HTTP proxies or virtual machine hosts, so *nmap* results don't necessarily indicate the OS used by the spammer.



```

Windows PowerShell
nmap -F -O 5.79.73.142 95.211.192.231

Starting Nmap 6.46 ( http://nmap.org ) at 2014-07-03 18:18 Central Europe Daylight Time
Nmap scan report for 5.79.73.142
Host is up (0.038s latency).
Not shown: 98 closed ports
PORT      STATE SERVICE
22/tcp    open  ssh
25/tcp    filtered smtp
Device type: general purpose|WAP|webcam|firewall
Running (JUST GUESSING): Linux 2.6.X|3.X|2.4.X (94%), Tandberg embedded (86%), Check Point Linux 2.6.X (85%), Fortinet L
inux 2.6.X (85%)
OS CPE: cpe:/o:linux:linux_kernel:2.6 cpe:/o:linux:linux_kernel:3 cpe:/o:linux:linux_kernel:2.4 cpe:/h:tandberg:vcs cpe:
/o:linux:linux_kernel:2.6.18 cpe:/o:fortinet:linux_kernel:2.6
Aggressive OS guesses: Linux 2.6.32 - 2.6.39 (94%), Linux 3.2 - 3.6 (92%), Linux 2.6.32 - 3.0 (92%), Linux 2.6.32 - 2.6.
35 (90%), Linux 2.6.39 (90%), Linux 2.6.32 (90%), Linux 3.4 (89%), Linux 3.1.9 (89%), Linux 3.0 - 3.9 (88%), Linux 2.6.1
8 (88%)
No exact OS matches for host (test conditions non-ideal).
Network Distance: 12 hops

Nmap scan report for 95.211.192.231
Host is up (0.041s latency).
Not shown: 98 closed ports
PORT      STATE SERVICE
22/tcp    open  ssh
25/tcp    filtered smtp
Device type: general purpose|WAP|webcam|firewall
Running (JUST GUESSING): Linux 2.6.X|3.X|2.4.X (94%), Tandberg embedded (86%), Check Point Linux 2.6.X (85%), Fortinet L
inux 2.6.X (85%)
OS CPE: cpe:/o:linux:linux_kernel:2.6 cpe:/o:linux:linux_kernel:3 cpe:/o:linux:linux_kernel:2.4 cpe:/h:tandberg:vcs cpe:
/o:linux:linux_kernel:2.6.18 cpe:/o:fortinet:linux_kernel:2.6
Aggressive OS guesses: Linux 2.6.32 - 2.6.39 (94%), Linux 3.2 - 3.6 (92%), Linux 2.6.32 - 3.0 (92%), Linux 2.6.32 (91%),
Linux 2.6.32 - 2.6.35 (90%), Linux 2.6.39 (90%), Linux 3.4 (90%), Linux 3.1.9 (89%), Linux 3.0 - 3.9 (88%), Linux 2.6.1
8 (88%)
No exact OS matches for host (test conditions non-ideal).
Network Distance: 12 hops

OS detection performed. Please report any incorrect results at http://nmap.org/submit/ .
Nmap done: 2 IP addresses (2 hosts up) scanned in 17.48 seconds
  
```

Figure 3: *nmap* results for the spambot hosts (Linux with SSH port open)

PROFILING THE BROWSER

Initial data suggested that some kind of browser running on a Linux host was responsible for our comment spam. To gather more intelligence on the browser, we have extended our comment form with a piece of JavaScript code that captured runtime browser information and sent it back to our server. JavaScript provides a broad range of browser information for the inquisitive developer, as demonstrated by <http://browserspy.dk/>.

In our first experiment we opted to capture certain basic properties of the [window.navigator](#) object, the [window](#) object and the browser plugins. The spammer visited a short while after our script was published and the data we captured showed some odd properties.

```
{
  "userAgent": "Mozilla/4.0 (Windows NT 6.2) AppleWebKit/537.17 (KHTML, like Gecko) Chrome/24.0.1312.70 Safari/537.17",
  "appName": "Mozilla",
  "appVersion": "4.0 (Windows NT 6.2) AppleWebKit/537.17 (KHTML, like Gecko) Chrome/24.0.1312.70 Safari/537.17",
  "onLine": false,
  "platform": "Linux x86_64",
  "product": "Gecko",
  "language": "en-US"
  "plugins": {
  },
  "window": {
    "innerWidth": 1680,
    "innerHeight": 15000,
    "outerWidth": 0,
    "outerHeight": 0,
    "screenWidth": 1024,
    "screenHeight": 768,
    "colorDepth": 32
  }
}
```

Figure 4: Data captured about the spambot browser

We made the following observations:

- The `window.navigator.onLine` property is *false*. This property is supposed to report if Internet connectivity is available. Chances are slim that the browser is offline while posting to our pages online, so this was not the expected value.
- The `windows.navigator.platform` property is *Linux x86_64*, indicating that the browser actually runs on a 64-bit version of *Linux*. This again contradicts the Windows 8 operating system reported by the *User-Agent* string.
- The list of plugins is empty. In a way, this is not surprising—as per our tests, *Internet Explorer 11* and *Chrome for Android* don't report any plugins. However, the desktop *Chrome* browser the spambot claims to be should normally report at least a few plugins.
- Perhaps most tellingly, the `window.outerWidth` and `window.outerHeight` properties are both *0*. These properties specify the browser window dimensions in pixels, so a zero value suggests that there is no browser window to speak of.

ENTER THE HEADLESS BROWSER

At this point, we started suspecting that we are facing a custom version of *Chrome / WebKit*: something that is not quite a true browser, but something very similar and something that was built for automation.

Our research brought us to a special category of browsers called *headless browsers*. These are barebone browser implementations, typically based on *WebKit*⁵ or *Gecko*⁶, used for a number of purposes like automated website testing and taking web page screenshots. They are called „*headless*”, because they don't have a user interface like regular web browsers do. From all other aspects, they are full-featured web browsers with the same understanding of HTML, JavaScript and CSS as browser engine they are based on.

To see if a *headless browser* exhibits similar properties as our spambot, we ran an experiment with one of the most popular headless browser called [PhantomJS](#). In a couple of minutes, we have managed to sketch up a proof-of-concept attack on our forums.

```
var page = require('webpage').create();

page.settings.userAgent = 'Mozilla/4.0 (Windows NT 6.2) AppleWebKit/537.17 (KHTML, like Gecko) Chrome/24.0.1312.70 Safari/537.17';
page.open('http://[redacted]/forum/topic/539/spam-test', function()
{
  page.evaluate(function ()
  {
    // this script is run within the sandbox
    document.querySelector('input[name=NickName]').value = 'DrEvil';
    document.querySelector('input[name=Email]').value = 'test@example.org';
    document.querySelector('#newCommentTextarea').value = 'This is my spam.';

    document.querySelector('#frmComment').submit();
  });

  // capture screenshot and exit after 5 seconds
  window.setTimeout(function()
  {
    page.render('final-status.png');
    phantom.exit();
  }, 5000);
});
```

Figure 5: Our PhantomJS attack script, complete with *User-Agent* forgery and screenshot taking

The data we captured from PhantomJS was tellingly similar to that of our spambot: the browser said it's offline, no plugins were reported and both *outerWidth* and *outerHeight* were 0.

We started seeking out a way to detect if our script is being run in a sandbox of a headless browser. Our initial idea was to capture and compare the browser features using a feature detector like [Modernizr](#) to see if they exhibit a specific pattern distinguishable from desktop browsers.

We did not get there eventually, because we found a much simpler way for detection. We worked with embedded *Internet Explorer* before in [ORF](#) (Vamsoft's email anti-spam product) and used objects and functions exposed from the browser host application to the facilitate communication between the host and the embedded browser. It was a long shot, but a quick test could be done

⁵ WebKit is the engine behind Google Chrome and Apple Safari.

⁶ Gecko is the engine of Mozilla Firefox.

in a minute to check if *PhantomJS* does the same. Sure enough, when our test enumerated all functions on the *JavaScript* global object, we spotted one named *callPhantom()*. This function was confirmed by *PhantomJS* documentation as a method specifically injected by the headless browser. Another, broader search for objects found the *_phantom* object, which also belongs to *PhantomJS*.

```
"addEventListener",  
"alert",  
"atob",  
"blur",  
"btoa",  
"callPhantom",  
"captureEvents",  
"clearInterval",  
"clearTimeout",  
"close",
```

Figure 6: The *callPhantom()* function spotted by our test

This has provided us with a way to detect *PhantomJS* specifically, but we couldn't possibly know what kind of headless browser (if any!) is used by the spambot, so we further extended our browser intel script in the forum with capturing all global functions and objects.

Half an hour after the extended script was published, the spambot visited again and the captured data confirmed not only that our spambot uses a headless browser, but that it specifically uses *PhantomJS*.

```
"Worker",  
"XMLDocument",  
"XMLHttpRequest",  
"XMLHttpRequestException",  
"XMLHttpRequestUpload",  
"XMLSerializer",  
"XPathEvaluator",  
"XPathException",  
"XPathResult",  
"_events",  
"_gaq",  
"_gat",  
"_phantom",  
"applicationCache",
```

Figure 7: The *_phantom* object of *PhantomJS* in the data captured from the spambot

CONCLUSIONS

Our investigation concluded that our attacker uses a Linux build of *PhantomJS* to defeat our JavaScript spam challenge. We looked for reports of headless browser usage in comment spam and to our slight disappointment, we're not the first to report this type of spam: the references are sporadic, but the oldest we found is dates back to [late 2013](#).

Digging further also found that *PhantomJS* was used in a [massive botnet-based DDoS attack](#) in Q3 2013. This suggests that large-scale abuse of headless browsers for malicious purposes is a quite recent phenomenon and with room for further development. In fact, we expect that it to be discovered by more criminals for more purposes, like click fraud.

The Q3 2013 DDoS attack also suggests that the technology has already been scaled to botnets, so the spammers have access to nearly unlimited CPU and memory, which would normally limit headless browser use. With the floodgates open, researchers are likely to experience a higher-than-usual level of sophistication in future web attacks.

POSSIBLE RESPONSES

The few websites that rely exclusively on JavaScript-based *"true browser vs. impostor"* detection need to either introduce additional layers of defense like CAPTCHAs, or need to be extended with headless browser detection.

We contribute a very simple JavaScript-based approach for detecting PhantomJS:

```
function isPhantomJS() {  
    return  
        !!window._phantom || // PhantomJS extends window object with _phantom  
        !!window.callPhantom; // Function injected by PhantomJS  
}
```

During our research, we came across a [post](#) where *StackOverflow* user *hexalys* published exposed objects for various headless browsers, which can be used to further extend the above script.

While this simple detection might work for now, it is easy to see how it can be defeated using a custom build of PhantomJS with different function/object names, or by using another headless

browser. Spammers never lacked innovative power, so we have no doubt that any trivial detection method will have limited lifetime and efficiency.

More sophisticated detection approaches might involve keyboard/mouse/touch interaction monitoring, timing-based heuristics or capability/feature fingerprinting of the browser.

A special thanks go to Martijn Grooten, editor at Virus Bulletin for advising.

About Vamsoft

Vamsoft e-Security Kft. is the Budapest, Hungary-based vendor of ORF, an email spam filtering solution available for Microsoft® Exchange and the IIS SMTP server. ORF provides a vast range of tools for spam identification and comes with intuitive and detailed reporting features which enable system administrators to overview and manage spam threats with unprecedented ease. ORF is used in 112 countries around the world by SMBs, enterprises and governmental institutions.