Transparent Web Application Bot Protection

Joe Pruitt, 2012-12-01

The Problem

HTML Web forms are a mechanism for someone to build a web page that allows a user to send data to a server for processing. Users can fill out the forms with elements such as checkboxes, radio buttons, or text fields. When a user clicks the associated “Submit” button, the data they entered into the form is sent to the server for processing.

This is where the issue arises. Most form processing components are not smart enough to know where or how the data was sent to them. They just look at the data and process away. This introduces a nice way for a malicious person to build an automated process to repeatedly submit form data to your server, thus flooding it with requests and ultimately either using it for unintended purposes or taking the server down completely. This is a form of a Denial of Service attack on the application server and the application code is helpless to protect itself.

An Example Scenario

Imagine you built a website and wanted to provide a way for your users to contact you. Since you don’t want your email address exposed to the world, you had one of your developers build you a simple form taking some contact information from your users as well as some form of comments they can send along to you. The backend server will take this form data and then formulate it into an email message to be sent right to your inbox. The form is put live on your website and you test it out and it works great. After a long weekend, you get into work only to find you can no longer login to your email because there are over 3 million emails in your inbox. What happened? Odds are someone took a trace of a valid request to the contact form and are replaying it with an automation tool.

Alternate Solutions

One solution that is commonly used is to implement a CAPTCHA control on your contact form. The CAPTCHA control is an image that contains a picture of a set of characters or words.   For the form to process, the user must type in what they see in the image into a text box and the server will compare it to what the correct value is. This may sound like a great solution, and most of the time it is. The problem is that as computers have evolved and become more powerful, so have the mechanisms to defeating a CAPTCHA control. Character recognition software and even cheap human labor can be used to beat out even the most sophisticated CAPTCHA implementations.

A Transparent Solution

I personally hate CAPTCHAs as they require me, as a user, to take an extra effort to try to copy something that I usually can’t read myself in the first place. There has to be a better way to do this without requiring the user to do the extra work to prove they are, in fact, a real user.   Lucky for you, you have a BIG-IP LTM sitting in front of your contact form! With a little bit of logic written into an iRule, you can

1. Uniquely identify users
2. Verify that requests follow the correct application flow (ie, the user can’t jump direction to a form submission without first going through the actual form first).
3. Incorporate rate limiting/blocking so that valid users (or bots that have worked their way around #2) can’t issue repetitive valid requests in an abusive manner.
4. Provide an administration console to be able to monitor the status of requests.

Components

My solution implements the following iRules.

- app_protection - This iRule will contain the logic for the requirements 1-3 above.
- app_protection_admin - This iRule implements the admin console allowing you to see who’s coming and going with your forms.

Internally, the data needs to go somewhere. I've split this up into 3 distinct locations:

- Flow Class - This is a data group containing the mapping of the form page and it's submission link defining the application flows you want to enforce.
- Validation Table - This is a session table that is used with a configurable timeout to store visits to the form page. This record is used when the submission page is requested to ensure the user when through the correct application flow.
- Retry Table - Once a valid form submission has occurred, the client info is stored in a retry table that will have a randomized timeout. If a request from the same client is requested in that window of time, the request will be rejected.

Logic Flow

Before we dig into the iRule itself, I'll attempt to describe how it works with the simple logic flow below. Each of the steps in this flow will have a corresponding code sample below in this article.

- * Record client and request characteristics
- If request is a valid form target (in form lookup table)
    - if request is a valid POST or GET form submission
        - if Referer header != form request page
            - *Block client request
        - else
            - If validation record exists in validation table
                - * Remove validation record
                - * Insert record into retry table
                - * Stop processing and allow request to continue
            - else
                - * Block client request
    - else
        - * Continue processing
- If valid form request (in form lookup table)
    - If no client entry in retry table
        - * Add current request to validation table
        - * Stop processsing and allow request to continue
    - else
        - * Block client request (bot attempt)

The iRules

In order to explain the iRule, I'll break it into parts described above

Initial Setup

> Record client and request characteristics

During the client request, I'll use the following variables

- lifetime - The duration (in seconds) between a valid form request and submission.
- repeattime - A random time (in seconds) that a person is not allowed to re-submit the same form.
- flowclass - A data group containing the form request/submit flows.
- validationtable - A session table containing valid form requests with the specified life time of the lifetime variable.
- retrytable - A session table containing valid form submissions with a lifetime of the repeattime variable.
- lpath - a lower case version of the path portion of the URI.
- lquery - A lower case version of the query string portion of the URI.
- luri - A lower case version of the whole URI.
- client_id - The client identifier consisting of the client connection info (ip and port) along with the VIP's local address and port.

```
set STOP_PROCESSING 0;

# Move to static variables later on
# Move to static variables later on
# form request timeout
set lifetime 60;
set repeattime [expr {int(rand()*5)} + 2]

# The class contiaining the form flows. The format is with the key
# being the form request uri and the value being the form submit uri.
# "uri1:uri2" := ""
set flowclass "flow_class";

# The table containing the validated requests.  The Key is the client flow id
# and the value is the form request uri.
# client_id:uri1
set validationtable "flow_validation";

# The table containing the timeout before a subsequent request can occur for
# a given URI from a client flow
# client_id:uri1
set retrytable "bot_retries";

set lpath [string tolower [HTTP::path]];
set lquery [string tolower [HTTP::query]];
set luri [string tolower [HTTP::uri]];

# Create unique client identifier.
set client_id "[IP::client_addr]:[TCP::client_port]:[IP::local_addr]:[TCP::local_port]";
```

Submit Form Testing

> If request is a valid form target (in form lookup table)

The form of the flow class entries are “form_request_uri:form_submit_uri”. The “class” command is used to search for entries with the requested URL to see if a corresponding “form_submit_uri” is found. If so, then we’ll process this request as a form submission.

```
set entry [class search -name $flowclass ends_with ":${lpath}"];
if { "" ne $entry } {

  set tokens [split $entry ":"];
  set uri1 [lindex $tokens 0];
  set uri2 [lindex $tokens 1];
```

> if request is a valid POST or GET form submission

If the URI is found to be a form submission, then we’ll check to make sure it really is a form submission by checking if it’s a HTTP POST command or a GET command with parameters. If so, we can continue validating it.

```
# Found current URI in the submit section of the flow class.
# Determine whether this is a form submission or a page request
if { ([HTTP::method] eq "POST") || ([HTTP::query] ne "") } {
```

> if Referer header != form request page

For one more sanity check, we’ll test to see if the client passed through the correct HTTP Referer header from the previous page it visited.

```
# First line of defense: check referer header
if { [HTTP::header "Referer"] ne "http://[HTTP::host]$uri1" } {
```

> *Block client request

If the Referer header doesn’t match the expected form request, then we’ll send them a little error message and send them on their way.

```
# No referer header.  Say hasta
HTTP::respond 200 Content "<html><head><title>NICE TRY</title></head><body>ERROR!</body></html>";
set STOP_PROCESSING 1;
return;
```

> If validation record exists in validation table

At this point, we believe it’s a form submission request and that the Referer is valid. So, we’ll check in the validation table to see if the same client has previously requested the form request page.

```
# Referer is valid.  Now lookup validation record.
set is_valid [table lookup -notouch -subtable $validationtable "$client_id:$uri1"];
if { $is_valid ne "" } {
```

> Remove validation record

Now this is a valid form submission. We will now remove the validation record from the validation table with the table command.

```
# valid request
# cleanup entry in validation table and allow through
table delete -subtable $validationtable "$client_id:$uri1";
```

> Insert record into retry table

And we will then insert a record into the retry table with the variable length lifetime defined in the “repeattime” variable.

```
# Insert record into bot table
table set -subtable $retrytable "$client_id:$uri1" 1 $repeattime indefinite;
```

> Stop processing and allow request to continue

The return statement is then used to exit the iRule and allow further processing to the application.

```
# Allow Through
return;
```

> Block client request

If no validation record is present above, then we’ll send an error back to the client and tell them to try again.

```
# invalid request
HTTP::respond 200 Content "<html<body>ERROR! <a href='[HTTP::uri]'>Try Again</a></body></html>";
set STOP_PROCESSING 1;
return;
```

Initial Form Request Processing

> If valid form request (in form lookup table)

Now that we have completed processing the form submission logic, we’ll process form requests. Again, we will lookup the URI in the flowclass data group, but this time we will look for the first part of the application flow by using the “starts_with” operator. If an entry exists, then we will further process this as a form request.

```
set entry [class search -name $flowclass starts_with "${lpath}:"];
if { "" ne $entry } {

  set tokens [split $entry ":"];
  set uri1 [lindex $tokens 0];
  set uri2 [lindex $tokens 1];
```

> If no client entry in retry table

Next, we’ll check the retry table to make sure that this client hasn’t successfully submitted this form within the retry window.

```
# Check bot table
set entry [table lookup -notouch -subtable $retrytable "$client_id:$uri1"];
if { "" eq $entry } {
```

> Add current request to validation table

No entry was found, meaning that the client hasn’t tried to slam the service with a subsequent request. So we can now add an entry into the validation table so that when they submit a form submission, the request will be honored.

```
# no bot entry, allow through
# Found current request in flowclass.  Assume this is a form submission
table set -subtable $validationtable "$client_id:$uri1" [crc32 "$client_id:$uri1"] $lifetime indefinite;

return;
```

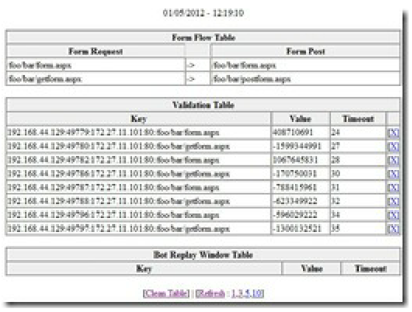
> Block client request (bot attempt)

If the requested form was in the retry table, then we can assume a malicious entity is attempting to flood the service. An error message is sent back to the client and further processing is halted.

```
# user retried within request window.  Reject.
HTTP::respond 200 Content "<html><body>ERROR!<br/> <a href='[HTTP::uri]'>Try Again</a></body></html>";
set STOP_PROCESSING 1;
return;
```

Admin Console

The admin console could be a tech tip in itself. It is implemented as a standalone iRule that, ideally, would sit on an internal vip that is not accessed by the external network (you don’t want your users peeking in on what’s going on do you?).



01/05/2012 - 12:19:10

Form Flow Table		
Form Request		Form Post
/foo/bar/form.aspx	->	/foo/bar/form.aspx
/foo/bar/getform.aspx	->	/foo/bar/postform.aspx

Validation Table			
Key	Value	Timeout	
192.168.44.129:49779:172.27.11.101:80:/foo/bar/form.aspx	408710691	24	[X]
192.168.44.129:49780:172.27.11.101:80:/foo/bar/getform.aspx	-1599344991	27	[X]
192.168.44.129:49782:172.27.11.101:80:/foo/bar/form.aspx	1067645831	28	[X]
192.168.44.129:49786:172.27.11.101:80:/foo/bar/getform.aspx	-170750031	30	[X]
192.168.44.129:49787:172.27.11.101:80:/foo/bar/form.aspx	-788415961	31	[X]
192.168.44.129:49788:172.27.11.101:80:/foo/bar/getform.aspx	-623349922	32	[X]
192.168.44.129:49796:172.27.11.101:80:/foo/bar/form.aspx	-596029222	34	[X]
192.168.44.129:49797:172.27.11.101:80:/foo/bar/getform.aspx	-1300132521	35	[X]

Bot Replay Window Table		
Key	Value	Timeout

[Clean Table] | [Refresh : 1,3,5,10]

In the video walkthrough below, I've illustrated it's use and I'll leave it to the reader to dissect the code in the corresponding CodeShare contribution below.

Demo Walkthrough

Conclusion

Just as CAPTCHA's are not attack proof, neither is this solution. If some one was crafty enough they could develop a system to bypass this solution. It's really meant to protect against the 90% of the generic web scrapers out there that aren't smart enough to figure out how the filtering works - or who can't read this article! It would be fairly trivial to enhance this solution to store hit-counts for successful form submissions in a separate session table. For unique clients that submit the form more than a "normal" number of times, you could put them into a blacklist and reject all future requests from them.   Of course, there are workarounds to that as well.

This article, and accompanying iRules, are not meant to replace a full-blown web application firewall. But it has hopefully given you some ideas on the kinds of things you can do with iRules to help protect your applications if you don't have a firewall in place - or if your firewall doesn't fit your specific protection requirements for your application.

Source Code

The source for this article can be found in the iRules CodeShare under the TransparentBotProtection topic.

F5 Networks, Inc. | 401 Elliot Avenue West, Seattle, WA 98119 | 888-882-4447 | f5.com

F5 Networks, Inc.
Corporate Headquarters
info@f5.com

F5 Networks
Asia-Pacific
apacinfo@f5.com

F5 Networks Ltd.
Europe/Middle-East/Africa
emeainfo@f5.com

F5 Networks
Japan K.K.
f5j-info@f5.com