Expose Documentation

Release 1.6

Chris Cornutt

Oct 04, 2017

Contents

Expose (pronounced *ex-pose-a*, pretend you're French) is an Intrusion Detection System for PHP loosely based on the PHPIDS project (and using its ruleset for detecting potential threats). You can find the latest version over on its github page.

ALL CREDIT for the rule set for Expose goes to the PHP IDS project. Expose literally uses the same JSON configuration for its execution. I am not claiming any kind of ownership or authorship of these rules. Please see the PHPIDS github README for names of those who have contributed.

CHAPTER 1

Requirements

Expose requires:

- PHP 5.3

Additionally, the default for queue and logging support is a MongoDB database (in the `\Expose\Log\Mongo` class, so you'd need Mongo support if you want to use that) but both the Queue handler and Logging can be overwritten with your choice of adapters. The Logger can also be replaced with any other logger class that follows the PSR-3 standard.

NOTE: Expose *requires* that you define a logger. Any good security tool that doesn't produce logs is pretty useless, so you're required to set one up. You can use the `\Expose\Log\Mongo` class already provided or create your own that matches the `\Expose\Log` abstract class structure.

CHAPTER 2

Sample Code

The code below is a simple example of using Expose to handle the incoming data (`$data`) and process it against the filter rules. It also shows some of the helper methods you can get to get the results of the filter run.

```
$data = array(
    'POST' => array(
        'test' => 'foo',
        'bar' => array(
            'baz' => 'quux',
            'testing' => '<script>test</script>'
        )
    )
);

$filters = new \Expose\FilterCollection();
$filters->load();

$logger = new \Expose\Log\Mongo();

$manager = new \Expose\Manager($filters, $logger);
$manager->run($data);

echo 'impact: '.$manager->getImpact()."\n"; // should return 8

// get all matching filter reports
$reports = $manager->getReports();
print_r($reports);

// export out the report in the given format ("text" is default)
echo $manager->export();
echo "\n\n";
```

CHAPTER 3

Real-time versus Queued Handling

Expose allows for two kinds of processing - real-time as the request comes in and delayed (queued). This can be controlled by setting the the `queueRequests` parameter on the `run` method in the `Manager`. If it is set to true, Expose will take the request data and insert it into the data store. By default, queuing is disabled.

If you choose to enable Queue support, you'll be required to define a Queue object to use. This can either be the included `\Expose\Queue\Mongo` or one of your own creation. See more about making a custom Queue object in the "Extending Expose - Custom Queue" section below.

Real-time reporting will process the impact scores of the matching rules and report back the results. These results can be fetched with the `getReports` method (as shown above). You're then free to do with the results as you wish.

Queued processing can be handled by something like a cron job using the command-line tool. When enabled, the request data is pushed into the data store with a `processed` value of `false`. The CLI then grabs the latest entries from this queue and processes them against the rules. The results are either directly outputted in a JSON format or can be written to an external file.

See the section on command line usage for more information.

CHAPTER 4

Exceptions

An exception basically allows you to say "evaluate everything except this value". For example, to bypass the POSTed value of "foo" you would use:

```
$manager->setException('POST.foo');
```

This bypasses the value for that field and doesn't execute the filters on it.

Additionally, you exception handling is regular expression aware. This means you can do more complex matching on the incoming parameters like:

```
// would match "POST.var1.baz", "POST.var2.baz", etc.
$manager->setException('POST.var[0-9]+.baz')
```

The string is treated like a normal regex, so be aware of the periods (as they still represent the "any character" match in the world of regex).

CHAPTER 5

Restrictions

A restriction lets you tell Expose to only evaluate certain values and ignore all others. For example, we might have more data than we care around coming in and only want to check the value of POST.foo.bar:

```
$data = array(
    'POST' => array(
        'foo' => array(
            'bar' => 'test one'
        ),
        'baz' => 'test two'
    )
);

$filters = new \Expose\FilterCollection();
$filters->load();

$logger = new \Expose\Log\Mongo();

$manager = new \Expose\Manager($filters, $logger);
$manager->setRestriction('POST.foo.bar');
$manager->run($data);
```

In this case, the filters would only run on `POST.foo.bar` and not on *POST.baz*.

CHAPTER 6

Notifications

Expose allows you to be notified of the results of its execution. You can configure the notifications by defining a *Notify* object and telling it to use it with the third parameter of the `run` method. For example, to send an email notification with the impact score and matching filters you could use:

```
$manager = new \Expose\Manager($filters);

$notify = new \Expose\Notify\Email();
$notify->setToAddress('sample@my-domain.com');
$notify->setFromAddress('notify@my-domain.com');
$manager->setNotify($notify);

$manager->run($data, false, true);
```

You can create your own custom notification methods by extending the `\Expose\Notify` abstract class and defining the `send` method.

CHAPTER 7

Thresholds

As the impact scores in Expose are numeric (0 through whatever, depending on the rules matched) you can easily set a threshold to prevent low-level, annoying notifications being delivered. Some applications know for a fact that they'll always be getting a certain amount of traffic that's in the 1-2 impact score range. Getting notifications for *every one* of these requests would get annoying pretty quickly, so you can set your *threshold* a bit higher:

```
$manager = new \Expose\Manager($filters);
$manage->setThreshold(8);
```

This example sets the impact threshold to *8*, meaning that it will only send notifications when the score is **greater than or equal to** *8*. There's no concept of "HIGH", "MEDIUM" or "LOW" in Expose as these vary greatly by environment and application.

NOTE: Currently *notifications* are the only thing that setting a threshold changes. Logging and other processing is unchanged.

CHAPTER 8

Caching

Expose also allows for caching of the results for a request (based on the data given in the request). It does not have this enabled by default, so you'll need to add it to the Manager. For example, to add a file-based caching mechanism:

```
$cache = new \Expose\Cache\File();
$cache->setPath('/foo/bar/cache');

$manager = new \Expose\Manager($filters);
$manager->setCache($cache);
```

In this example we're also settng the path for the caching mechanism to save the files to. You can integrate your own custom caching tool by extending the `\Expose\Cache` class.

CHAPTER 9

Command Line

Expose comes with a command-line tool to help make using the system simpler. You'll find it in the `bin/` directory inside of your installation. The CLI script includes a few different commands:

- `filter`
- `process-queue`

Below are examples of how to use these commands.

CHAPTER 10

Command Line - Filters

The `filter` command gives you information about the filters loaded into the system. By default, it will give you a list of the filters and their descriptions:

```
bin/expose filter
```

The result is a list of IDs and the summaries from the filters, for example:

```
1: finds html breaking injections including whitespace attacks
2: finds attribute breaking injections including whitespace attacks
3: finds unquoted attribute breaking injections
4: Detects url-, name-, JSON, and referrer-contained payload attacks
5: Detects hash-contained xss payload attacks, setter usage and property overloading
6: Detects self contained xss via with(), common loops and regex to string conversion
7: Detects JavaScript with(), ternary operators and XML predicate attacks
```

To get more information about a filter, use the `id` option:

```
bin/expose filter --id=2
```

You'll be given the details about that filter:

```
bin/expose --id=2

[2] finds unquoted attribute breaking injections
    Rule: (?:^>[\w\s]*<\/?\w{2,}>)
    Tags: xss, csrf
    Impact: 2
```

Or, if you'd like information on more than one filter at a time, you can append them with a comma:

```
bin/expose --id=2,3

[2] finds unquoted attribute breaking injections
    Rule: (?:^>[\w\s]*<\/?\w{2,}>)
    Tags: xss, csrf
```

```
    Impact: 2

[3] Detects url-, name-, JSON, and referrer-contained payload attacks
        Rule: (?:[+\/]\s*name[\W\d]*[)+])|(?:;\W*url\s*=)|(?:[^\w\s\/?:>]\s*(?
↪:location|referrer|name)\s*[^\/\w\s-])
        Tags: xss, csrf
        Impact: 5
```

CHAPTER 11

Command Line - Queue

The `process-queue` command lets you work with the queued request data. To use the queue processing, you need to enable it with the `queue_requests` configuration option.

To process the current items in the queue, you can execute it without any command line options:

```
bin/expose process-queue
```

This will provide you some messaging about how many items it will be processing (the default is 10 records at a time) and output the resulting filter matches as JSON data.

If you'd like to output these results to a file instead, you can use the `export-file` option:

```
bin/expose process-queue --export-file=/tmp/output.txt
```

This will apprend to the file if it already exists.

Custom Queue Settings

By default, the queue system the CLI uses will look for a Mongo server running on the localhost with an `expose` database it can access. You can change this, however, to work with your own Mongo server (or MySQL). When using the CLI, you can add two parameters to define the type and the connect string to use - `queue-type` and `queue-connect`:

```
bin/expose --queue-type=mongo --queue-connect=mongoUser:testing123@db.myhost.int
```

Using the combination of these two parameters, Expose will try to connect to the Mongo database living on the `db.myhost.int` server and use the `expose` database there.

You can also use a MySQL database in the same way, just using a type of "mysql" rather than "mongo".

CHAPTER 12

Extending Expose - Custom Queue

By default, Expose assumes a local Mongo instance to handle the queue processing. You can, however, override this with a custom queue object of your own. It only needs to do a few things:

- extend the `\Expose\Queue` abstract class
- define the `getPending`, `markProcessed` and `add` methods
- Pass in an adapter to use

So, if we wanted to use a Mongo instance on another machine, we could redefine our object like:

```
class MyQueue extends \Expose\Queue
{
    public function add($data)
    {
        /* add a new record */
    }
    public function markProcessed($id)
    {
        /* update the record */
    }
    public function getPending($limit)
    {
        /* return the pending records */
    }
}
```

then, to use it:

```
$filters = new \Expose\FilterCollection();
$filters->load();

$adapter = new MongoClient('mongodb://myserver1.example.com');
$myQueue = new MyQueue($adapter);
```

```
$manager = new \Expose\Manager($filters);
$manager->setQueue($myQueue);
```

If no queue is set with `setQueue` Expose will default to the Mongo version (configured for local connection).

CHAPTER 13

Extending Expose - Custom Filters

Expose lets you inect your own custom filters with your logic to be executed right along with the built in filters. The default filters use regular expressions to try to match attacks in the given data. Your custom filters can execute whatever login you want. All you have to do is add them to the `FilterCollection`:

```
class CustomFilter extends \Expose\Filter
{
    public function execute()
    {
        echo "Custom filter!\n";
        return true;
    }
}
$custom = new CustomFilter();
$filters->addFilter($custom);
```

You just define the `execute` method in your filter and Expose with run it. The `execute` method should return `true` if there's a match and `false` if there's none.