Adding a field-level vulnerability

Adding a field-level vulnerability is almost identical to adding a context-level vulnerability. In this tutorial we will add the IntegerOverflow vulnerability. To learn the basics of adding vulns please see [Adding a complex context-level vulnerability]() tutorial. Here we will expect you to know the basics.

IntegerOverflow vulnerability uses limitations of variable size. We need to control this behaviour.
First of all, when vuln is off, we can pass any values that a client sends. If vuln is on, we want to control the behavior of the app when value is either valid, or invalid, and also when the value is not a number at all.

You can see the implementaton of IntegerOvwerflow vuln in `{hackazon_dir}\modules\vulninjection\classes\VulnModule\Vulnerability\IntegerOverflow.php`.

Now that we know the vulnerability, let's use it.

1. Disabled state.

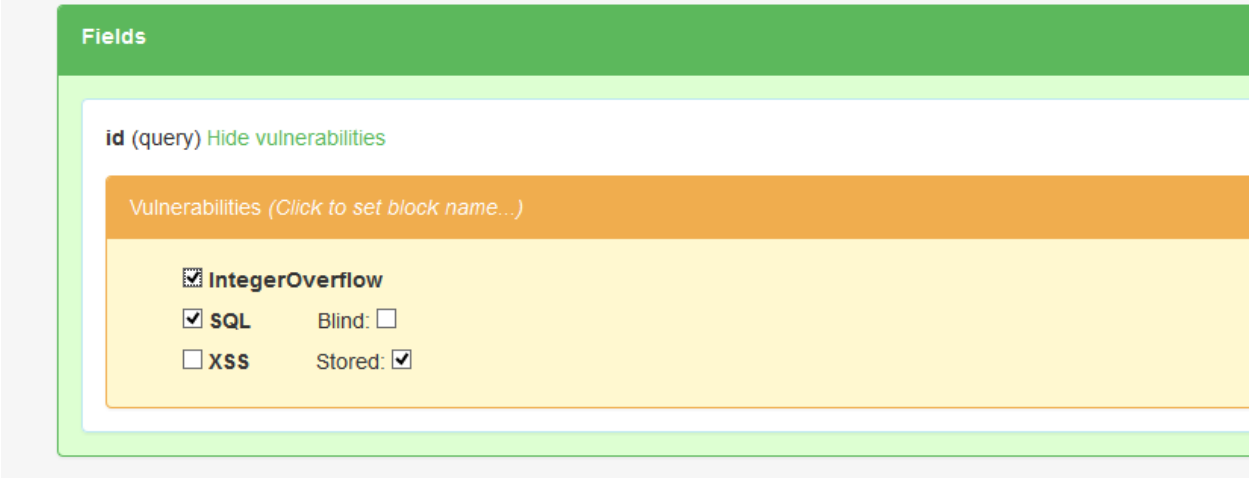
By default the state of any vulnerability is disabled. IntegerOverflow has additional options for this state.

a. Value transformation.
    i. cast to integer (default). Casts a value by `(int)` operator.
    ii. Set zero
    iii. Set NULL
    iv. Set custom value
b. Action performed when the value is not a number. It is needed to be able to inject other vulnerabilities on this fields, e.g. SQL Injection. Values are:
    i. filter (forces the filters from **A** item)
    ii. bypass

If no IntegerOverflow vulnerability is set on field, value is cast to integer if it's numeric, and passed by otherwise.

2. Enabled state.

To enable the vulnerability, just add it to the field:



Enabled state just passes through all input values.

3. Injecting the vulnerability into a controller.

Controller has some helpers for getting values from request. For example `getWrap()` fetches a value from $_GET array. Other methods are: postWrap, cookieWrap and so on.
Values are wrapped by the `VulnModule\VulnerableField` class, which containst a field descriptor,
a vulnerability set, and the raw value.

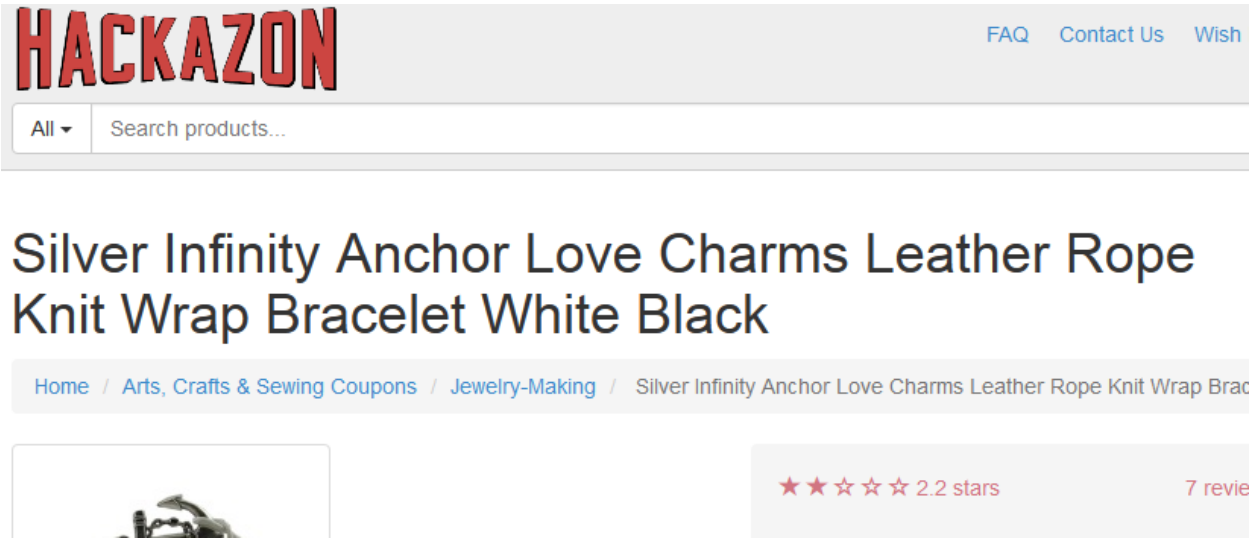
Let's getour wrapped value from query string:

```
class Product extends Page
{
   public function action_view()
   {
       // Get wrapped value
       $productID = $this->request->getWrap('id');

       // Each applied vulnerability can filter raw value, so we can get a
       // filtered value.
       if (!$productID->getFilteredValue()) {
           throw new NotFoundException("Missing product id.");
       }

       // DB subsystem understand
       $product = $this->model->where('productID', '=', $productID)->find();
```

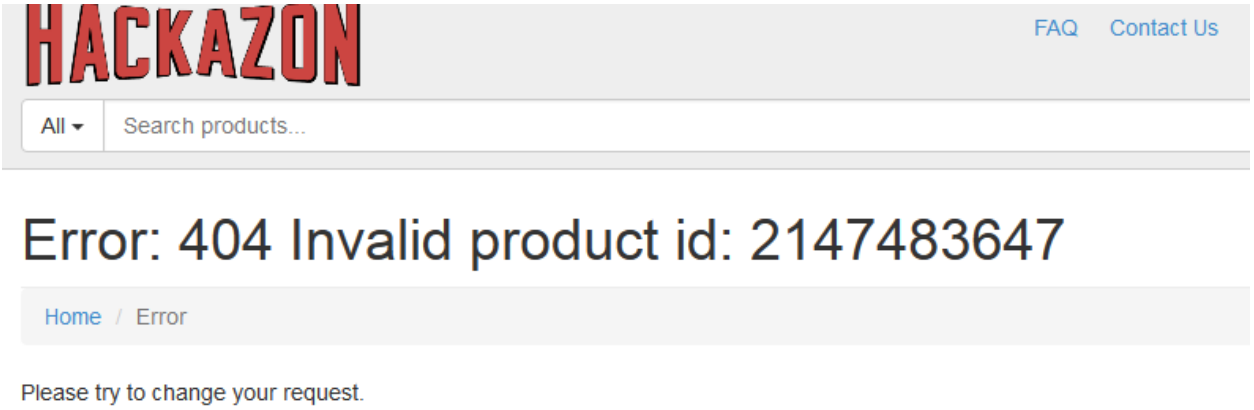
```
        if (!$product || !$product->loaded()) {
           throw new NotFoundException("Invalid product id"); //: "
                . $productID->escapeXSS());
        }

        // ……..

    }
   // …..
}
```

Let’s see how the id value influences the behavior of the app.

First we use id=10:

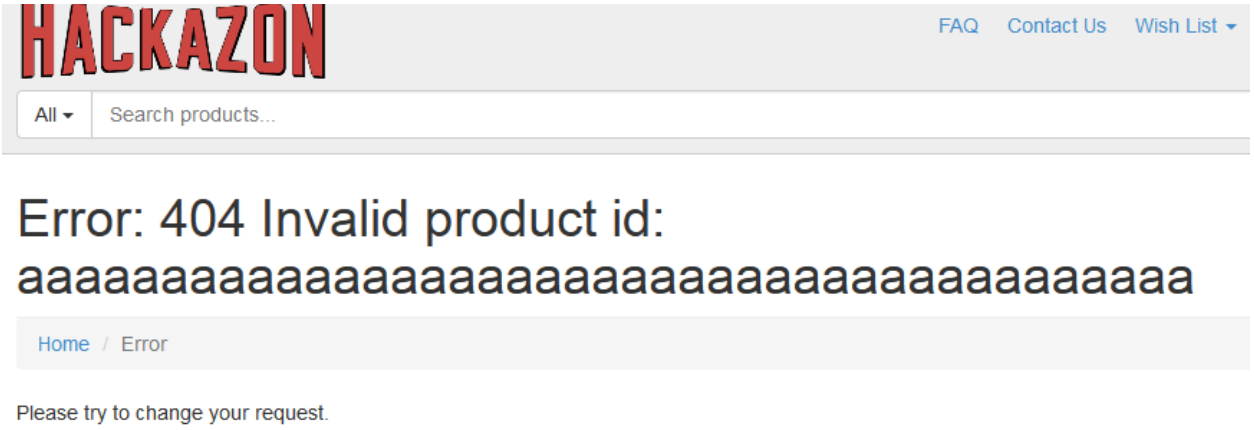


Everything is OK. Now we use id=1000000000000000000000000000000000:

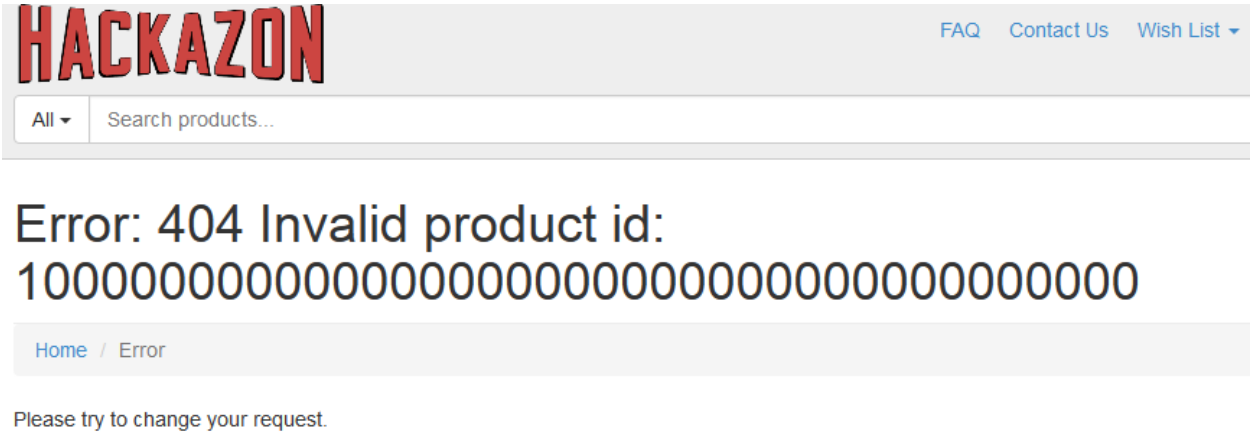


Value is cast to integer and truncated to max integer value in PHP.

If id is not a number and vulnerability bypasses it, then we will see the following picture:



And finally, if vulnerability is enabled, the following will be in our browser:



You see how flexible can be a vulnerability. The important takeaway is that any requested variable that we want to control must be wrapped by a VulnerableField object, which provides a list of vulnerabilities for this variable. And subsequent app logic may be anything we want.

4. Other operations with VulnerableField and Context.

```
$productID = $this->request->getWrap('id');

// Check whether the variable is vulnerable to certain vulnerability
$isSQLInjected = $productID->isVulnerableTo('SQL');

// Get the raw value of the variable
$raw = $productID->raw();

// Get the value passed through filters from all its vulnerabilities.
$filtered = $productID->getFilteredValue();

// Creates a new wrapped variable with the same field descriptor and vulnerability set.
$newValue = $productID->copy('The id is: ' . $productID);

// Different helper methods
$escapedForOutputInHTML = $productID->escapeXSS();

// Get current context for action
$context = $this->pixie->vulnService->getCurrentContext();

$context->isVulnerableTo('PHPSessionIdOverflow');
```

```php
// Direct access to Field element of the context if field exists
$context->getMatchingFields('id', FieldDescriptor::SOURCE_BODY);

// Returns a field element in any case
$context->getOrCreateMatchingField(new FieldDescriptor('id', FieldDescriptor::SOURCE_BODY));

// Get subcontext if exists
$context->getChildByName('index');

// Get subcontext in any case (created context is empty and vulnerability-free)
$context->getOrCreateChildByName('index');
```