A New Open Source Tool: OWASP ESAPI for PHP

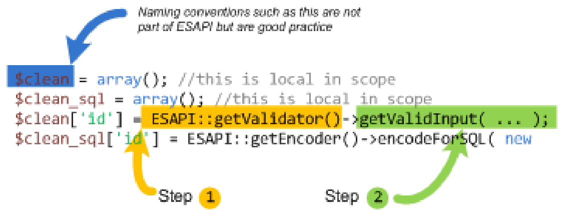
Mike Boberski, *OWASP ESAPI for PHP Project Manager*

***Abstract*—Don't write your own security controls! Reinventing the wheel when it comes to developing security controls for every PHP web application leads to wasted time and massive security holes. OWASP Enterprise Security API (ESAPI) for PHP helps software developers guard against security-related design and implementation flaws. ESAPI for PHP is designed to make it easy to retrofit security into existing applications, as well as providing a solid foundation for new development.**

***Index Terms*—OWASP, ESAPI, PHP, XSS, SQLi**

I. Introduction

OWASP ESAPI for PHP is designed to ensure that strong simple security controls are available to PHP developers. All OWASP ESAPI for PHP security controls are called in the same basic way, as depicted in the figure below.


Naming conventions such as this are not part of ESAPI but are good practice
Step 1
Step 2

```
$clean = array(); //this is local in scope
$clean_sql = array(); //this is local in scope
$clean['id'] = ESAPI::getValidator()->getValidInput( ... );
$clean_sql['id'] = ESAPI::getEncoder()->encodeForSQL( new
```

Fig. 1. How ESAPI for PHP works out of the box.

The basic OWASP ESAPI for PHP design:

- There is a set of security control interfaces. There is no application logic contained in these interfaces. They define for example types of parameters that are passed to types of security controls. There is no proprietary information or logic contained in these interfaces.
- There is a reference implementation for each security control. There is application logic contained in these classes, i.e. contained in these interface implementations. However, the logic is not organization-specific and the logic is not application-specific. There is no proprietary information or logic contained in these reference implementation classes. An example: string-based input validation.
- There are optionally your own implementations for each security control. There may be application logic contained in these classes which may be developed by or for your organization. The logic may be organization-specific and/or application-specific. There may be proprietary information or logic contained in these classes which may be developed by or for your organization. An example: enterprise authentication.

Paper submitted March 24, 2010. Mike Boberski is the OWASP ESAPI for PHP project manager, McLean, VA 22101 USA (e-mail: mike.boberski@owasp.org).

There are three common ways to write your own implementations for each security control: using a "built-in" singleton pattern, using an "extended" singleton pattern, or using an "extended" factory pattern. The remainder of this paper explores these three design patterns, including situations where taking more than one approach may be appropriate.

II. The Built-In Singleton Pattern

The ESAPI security control interfaces include an "ESAPI" class that is commonly referred to as a "locator" class. The ESAPI locator class is called in order to retrieve singleton instances of individual security controls, which are then called in order to perform security checks (such as performing an access control check) or that result in security effects (such as generating an audit record).

The "built-in" singleton pattern refers to the replacement of security control reference implementations with your own implementations. ESAPI interfaces are otherwise left intact.

A. *For example:*

```
...
require_once dirname(__FILE__) . '/../Authenticator.php';
...
//your implementation
class MyAuthenticator implements Authenticator {
...
```

B. *Developers would call ESAPI in this example as follows:*

```
...
$ESAPI = new ESAPI();
$myauthenticator = new MyAuthenticator();

//register with locator class
ESAPI::setAuthenticator($myauthenticator);
$authenticator = ESAPI::getAuthenticator();
$authenticator->login(...); //use your implementation
...
```

The UML for the above example is in the figure below.

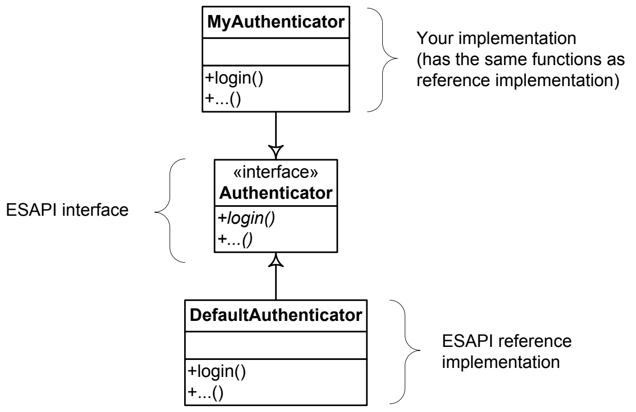

MyAuthenticator
+login()
+...()
Your implementation (has the same functions as reference implementation)
«interface» Authenticator
+login()
+...()
ESAPI interface
DefaultAuthenticator
+login()
+...()
ESAPI reference implementation

Fig. 2. Built-In Singleton Pattern Example.

Pros of taking this approach include loose coupling between ESAPI and your own implementations.

Cons include the need for developers to understand how to call ESAPI functions with the parameters required by your organization and/or application.

III. The Extended Singleton Pattern

While ESAPI security control reference implementations may perform the security checks and result in the security effects required by your organization and/or application, there may be a need to minimize the need for developers to understand how to call ESAPI functions with the parameters required by your organization and/or application. Availability of training may be an issue, for example. Another example would be to facilitate enforcing a coding standard.

The "extended" singleton pattern refers to the replacement of security control reference implementations with your own implementations and the addition/modification/subtraction of corresponding security control interfaces.

A. For example:

```
...
require_once dirname(__FILE__) . '/../Validator.php';
...
//reference implementation
class DefaultValidator implements Validator {
...
//not defined in Validator interface
function isValidEmployeeID($eid) {
...
```

B. Developers would call ESAPI in this example as follows:

```
...
$ESAPI = new ESAPI();
$validator = ESAPI::getValidator();
$validator->isValidEmployeeID(1234);
...
```

The UML for the above example is in the figure below.

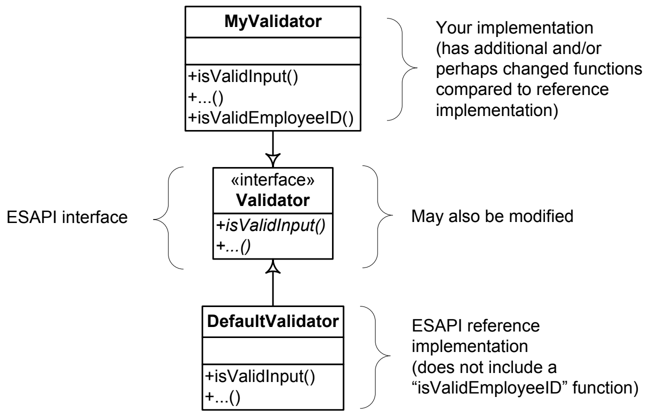

MyValidator
+isValidInput()
+...()
+isValidEmployeeID()
Your implementation (has additional and/or perhaps changed functions compared to reference implementation)
«interface» Validator
+isValidInput()
+...()
ESAPI interface
May also be modified
DefaultValidator
+isValidInput()
+...()
ESAPI reference implementation (does not include a "isValidEmployeeID" function)

Fig. 3. Extended Singleton Pattern Example.

Pros of taking this approach are the lessening of the need for developers to understand how to call ESAPI functions with the specific parameters required by your organization and/or application. Pros also include minimizing or eliminating the ability for developers to call ESAPI functions that deviate from your organization's and/or application's policies.

Cons result from the tight coupling between ESAPI and your own implementations: you will need to maintain both the modified security control reference implementations and the modified security control interfaces (as new versions of ESAPI are released over time).

IV. The Extended Factory Pattern

While ESAPI security control reference implementations may perform the security checks and result in the security effects required by your organization and/or application, there may be a need to eliminate the ability of developers to deviate from your organization's and/or application's policies. High developer turnover may be an issue, for example. Another example would be to strongly enforce a coding standard.

The "extended" factory patterns refers to the addition of a new security control interface and corresponding implementation, which in turn calls ESAPI security control reference implementations and/or security control reference implementations that were replaced with your own implementations. The ESAPI locator class would be called in order to retrieve a singleton instance of your new security control, which in turn would call ESAPI security control reference implementations and/or security control reference implementations that were replaced with your own implementations.

A. For example:

In the ESAPI locator class:

```
...
class ESAPI {
...
//not defined in ESAPI locator class
private static $adapter = null;
...
//new function
public static function getAdapter() {
    if ( is_null(self::$adapter) ) {
      require_once
dirname(__FILE__).'/adapters/MyAdapter.php';
      self::$adapter = new MyAdapter();
    }

    return self::$adapter;
  }

  //new function
  public static function setAdapter($adapter) {
    self::$adapter = $adapter;
  }
```

In the new security control class' interface:

```
  ...
  //new interface
  interface Adapter {

    function getValidEmployeeID($eid);
    function isValidEmployeeID($eid);

  }
```

In the new security control class:

```
  ...
  require_once dirname ( __FILE__ ) . '/../Adapter.php';

  //new class with your implementation
  class MyAdapter implements Adapter {

  //for your new interface
  function getValidEmployeeID($eid) {
    //calls reference implementation
    $val = ESAPI::getValidator();
    //calls using hardcoded parameters
    $val->getValidInput(
      "My Organization's Employee ID",
      $eid,
      "EmployeeID", //regex defined in ESAPI config
      4,
      false
      );
  }


  //for your new interface
  function isValidEmployeeID($eid) {
    try {
      $this->getValidEmployeeID($eid);
      return true;
    } catch ( Exception $e ) {
      return false;
    }

  }
```

B. Developers would call ESAPI in this example as follows:

```
  ...
  $ESAPI = new ESAPI();
  $adapter = ESAPI::getAdapter();
  $adapter->isValidEmployeeID(1234);
  ... //no other ESAPI controls called directly
```

The UML for the above example is in the figure below.

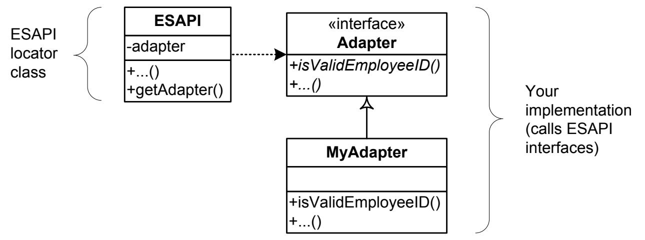


Fig. 4. Extended Factory Pattern Example.

Pros of taking this approach are the same as for the extended singleton pattern, and additionally include loose coupling between ESAPI and your own implementations, compared to the extended singleton pattern.

Cons include the need to maintain the modified ESAPI locator class (as new versions of ESAPI are released over time).

V. Conclusion

OWASP is the premier site for Web application security. The OWASP site hosts many projects, forums, blogs, presentations, tools, and papers. Additionally, OWASP hosts two major Web application security conferences per year, and has over 80 local chapters. The OWASP ESAPI project page can be found here http://www.owasp.org/index.php/ESAPI

The following OWASP projects are most likely to be useful to users/adopters of ESAPI:

- OWASP Application Security Verification Standard (ASVS) Project - http://www.owasp.org/index.php/ASVS
- OWASP Top Ten Project - http://www.owasp.org/index.php/Top_10
- OWASP Code Review Guide - http://www.owasp.org/index.php/Category:OWASP_Code_Review_Project
- OWASP Testing Guide - http://www.owasp.org/index.php/Testing_Guide
- OWASP Legal Project - http://www.owasp.org/index.php/Category:OWASP_Legal_Project

Similarly, the following Web sites are most likely to be useful to users/adopters of ESAPI:

- OWASP - http://www.owasp.org
- MITRE - Common Weakness Enumeration – Vulnerability Trends, http://cwe.mitre.org/documents/vuln-trends.html
- PCI Security Standards Council - publishers of the PCI standards, relevant to all organizations processing or holding credit card data, https://www.pcisecuritystandards.org
- PCI Data Security Standard (DSS) v1.1 - https://www.pcisecuritystandards.org/pdfs/pci_dss_v1-1.pdf