



OWASP

The Open Web Application Security Project

OWASP ESAPI

Design Patterns

alpha



This page is intentionally blank

Foreword

This document explores three common OWASP Enterprise Security API (ESAPI) design patterns. OWASP ESAPI toolkits help software developers guard against security-related design and implementation flaws. Just as web applications and web services can be Public Key Infrastructure (PKI) enabled (PK-enabled) to perform for example certificate-based authentication, applications and services can be OWASP ESAPI-enabled (ES-enabled) to enable applications and services to protect themselves from attackers.

We'd Like to Hear from You

Further development of ESAPI occurs through mailing list discussions and occasional workshops, and suggestions for improvement are welcome. Please address comments and questions concerning the API and this document to the ESAPI mail list, owasp-esapi@lists.owasp.org

Copyright and License

Copyright © 2009 The OWASP Foundation.



This document is released under the Creative Commons Attribution ShareAlike 3.0 license. For any reuse or distribution, you must make clear to others the license terms of this work.

This page is intentionally blank

Table of Contents

1	About ESAPI.....	1
2	The Built-In Singleton Pattern.....	2
3	The Extended Singleton Pattern	4
4	The Extended Factory Pattern.....	6
5	Where to Go From Here	8

Figures

Figure 1: How ESAPI works out of the box from a programmer's perspective.....	1
Figure 2: Built-In Singleton Pattern Example	2
Figure 3: Extended Singleton Pattern Example	4
Figure 4: Extended Factory Pattern Example	7

This page is intentionally blank

1 About ESAPI

OWASP ESAPI Toolkits are designed to ensure that strong simple security controls are available to every developer in every environment. All OWASP ESAPI versions work in the same basic way, as depicted in the figure below.

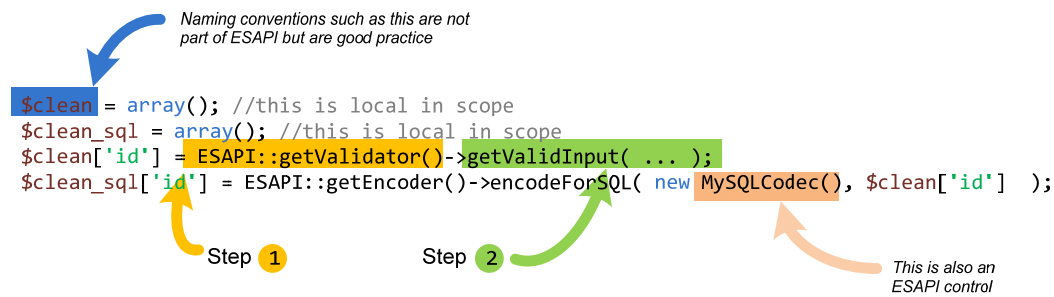


Figure 1: How ESAPI works out of the box from a programmer's perspective

Allowing for language-specific differences, all OWASP ESAPI versions have the same basic 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.

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 document explores these three design patterns, including situations where taking more than one approach may be appropriate.

2 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.

For example:

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

```

Developers would call ESAPI in this example as follows:

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

```

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

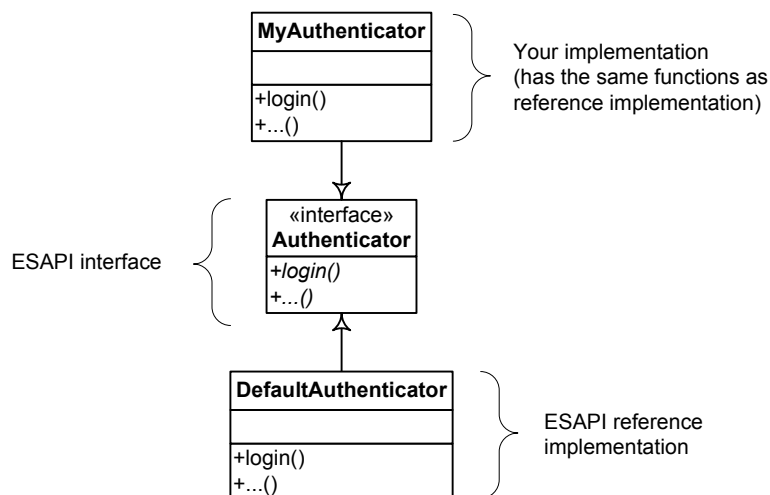


Figure 2: Built-In Singleton Pattern Example

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.

3 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.

For example:

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

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.

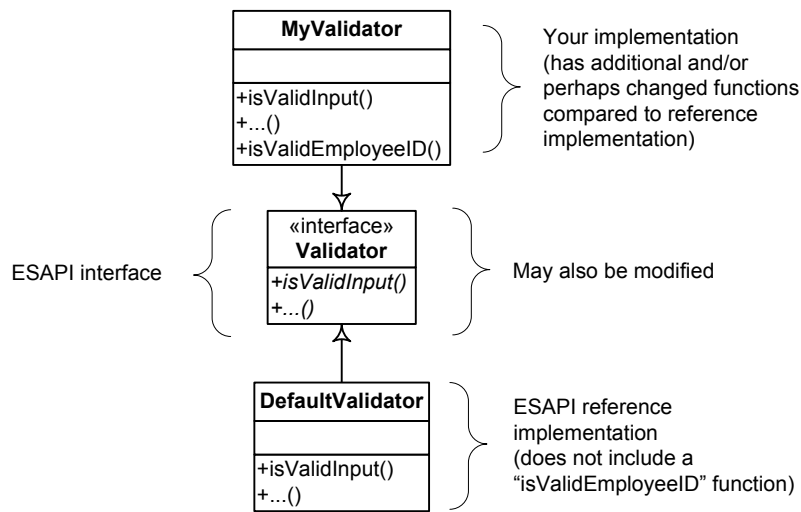


Figure 3: Extended Singleton Pattern Example

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).

4 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.

For example:

In the ESAPI locator class:

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

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

In the new security control class' interface:

```
...
interface Adapter { //new interface
    function getValidEmployeeID($eid);
    function isValidEmployeeID($eid);
}
}
```

In the new security control class:

```

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

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

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

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

```

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.

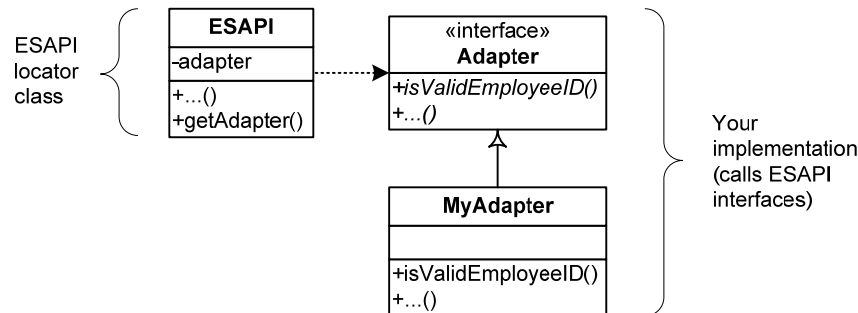


Figure 4: Extended Factory Pattern Example

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).

5 Where to Go From Here

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

This page is intentionally blank

THE BELOW ICONS REPRESENT WHAT OTHER VERSIONS ARE AVAILABLE IN PRINT FOR THIS TITLE BOOK.

ALPHA: "Alpha Quality" book content is a working draft. Content is very rough and in development until the next level of publication.

BETA: "Beta Quality" book content is the next highest level. Content is still in development until the next publishing.

RELEASE: "Release Quality" book content is the highest level of quality in a books title's lifecycle, and is a final product.



YOU ARE FREE:



to **share** - to copy, distribute and transmit the work



to **Remix** - to adapt the work

UNDER THE FOLLOWING CONDITIONS:



Attribution. You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).



Share Alike. - If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.



The Open Web Application Security Project (OWASP) is a worldwide free and open community focused on improving the security of application software. Our mission is to make application security "visible," so that people and organizations can make informed decisions about application security risks. Everyone is free to participate in OWASP and all of our materials are available under a free and open software license. The OWASP Foundation is a 501c3 not-for-profit charitable organization that ensures the ongoing availability and support for our work.