10/08

OWASP
The Open Web Application Security Project

Establishing a Security API for Your Enterprise

Java EE 2008 Edition

(based on OWASP ESAPI 3.1 release version)

alpha

Forward

The Open Web Application Security Project (OWASP) is an open community dedicated to enabling organizations to develop, purchase, and maintain applications that can be trusted. All of the OWASP tools, documents, forums, and chapters are free and open to anyone interested in improving application security. We advocate approaching application security as a people, process, and technology problem because the most effective approaches to application security include improvements in all of these areas. We can be found at www.owasp.org.

OWASP is a new kind of organization. Our freedom from commercial pressures allows us to provide unbiased, practical, cost-effective information about application security. OWASP is not affiliated with any technology company, although we support the informed use of commercial security technology. Similar to many open-source software projects, OWASP produces many types of materials in a collaborative, open way. The OWASP Foundation is a not-for-profit entity that ensures the project's long-term success.

We have found that organizations that standardize their application security controls significantly reduce their vulnerabilities in those areas. We've also noticed that many critical application security controls, such as input validation, output encoding, and error handling are often ignored and left to be reinvented by developers. So if you're tired of securing one application at a time, and wrestling with the same vulnerabilities again and again, establishing your organization's ESAPI is one of the best things you can do.

About the author

The OWASP ESAPI Project is led by Jeff Williams, founder and CEO of Aspect Security, specializing exclusively in application security services. Jeff also serves as the volunteer Chair of the Open Web Application Security Project (OWASP). In addition to the Enterprise Security API Project, Jeff has made extensive contributions to the application security community through OWASP, including the Top Ten, WebGoat, Secure Software Contract Annex, OWASP Risk Rating Methodology, the worldwide local chapters program., and coining the term "Reflected XSS." Jeff has been developing software for 25 years and is lucky to have 4 inspiring children, 3 loyal labradors, and a wonderful wife.

Table of Contents

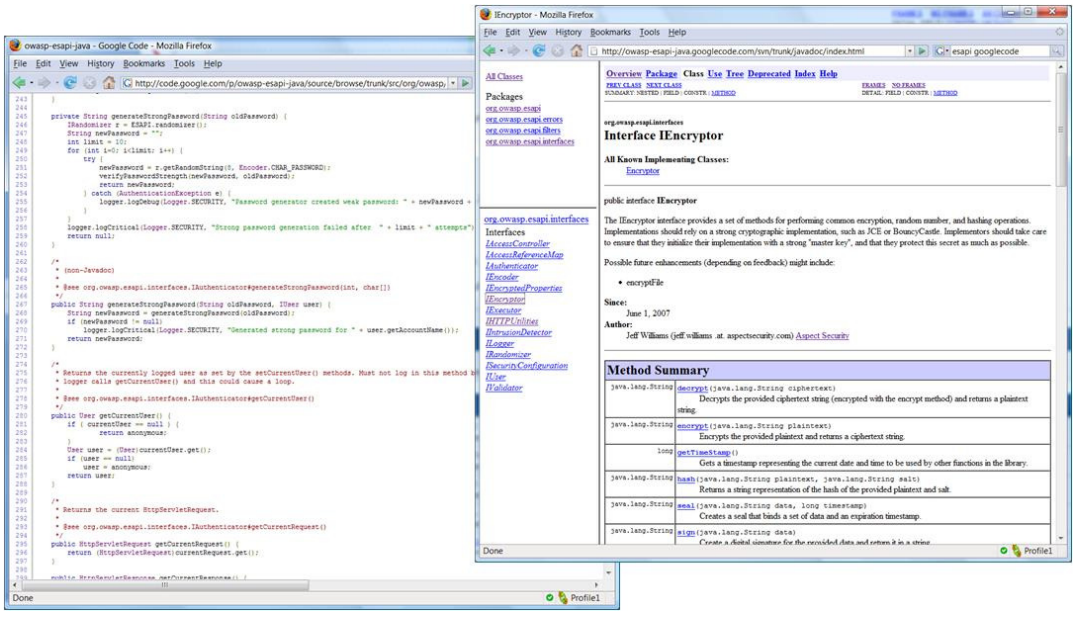
Introduction

Software is at a tipping point. The rapid increase in connectivity, combined with a dramatic rise in the value of assets in our systems, and the increasing use of new protocols and technologies has resulted in applications that represent significant risk to the organizations that build and use them.

One key part of the problem is that application security is a large and complex discipline that most developers will not be able to master, particularly if they are busy learning the latest framework or language. We believe that a key part of the road forward must be to simplify application security for developers.

To help organizations accomplish this, OWASP has defined a security API that covers all the security controls a typical enterprise web application or web service project might need. There are about 120 methods across all the different security controls, organized into a simple intuitive set of interfaces. We've worked very hard to make this API as clear and obvious as possible, to make it easy for developers to make the right decision.

We're building this library in a completely free and open way. All ESAPI projects are offered under the open source under the BSD license. This is absolutely critical for such important code. Only with complete openness can we achieve the level of assurance required. And only with freedom can we help organizations adopt our approach.



The library builds on the excellent security libraries available, such as Java Logging, JCE, and Adobe Commons FileUpload. It uses the concepts from many of the security packages out there, such as Spring Security, Apache Commons Validator, Microsoft's AntiXSS library, and many many more. This library provides a single consistent interface to security functions that is intuitive for enterprise developers. Used properly, the ESAPI provides enough functions to protect against most of the OWASP Top Ten and quite a few more common vulnerabilities.

We want organizations to create their own security API for their enterprise. We recognize that every organization has complex platforms, systems, directories, databases, and infrastructure. We are not trying to replace any of that. We're trying to simplify the application security problem for your developers by providing a simple consistent API to your security infrastructure.

You can find the ESAPI Project on the OWASP website. Currently, the Java version is complete and several organizations are already using it. Versions for .NET, PHP, Haskell, and Classic ASP are in development.

Background

Application security is managing the risks from your software. To achieve secure applications, you'll need to enable your developers, verify your applications, and manage cost and risk. The ESAPI project supports the first of these goals.

Why is application security hard?

Most developers are not taught security in school, so they only know a very small number of the 695 weakness types that MITRE has catalogued. Most never really receive any security feedback on the job, because by the time a vulnerability in their code is found, they're already off on the next project.

In order to write secure code today, developers an impossible amount to learn about application specific attacks, security controls, and vulnerabilities. You can't expect developers to wake up one day and think to himself, "hey – SQL injection! I bet an attacker could insert SQL code in this field that would modify the meaning of my queries!" All of those 695 weakness types are like that. They're not obvious to people, and many are actually quite tricky for the experts.

Which applications are at risk?

What's most important to secure, Internet-facing applications or intranet apps? Most people think that the biggest risks come from the Internet-facing applications, but that may not be true. Intranet applications are less likely to be attacked because the pool of threat agents (insiders) is smaller. But the functions and assets are often quite a bit more critical – private data, trade secrets, financials, and business plans. So the risk is frequently higher for intranet applications.

This risk rating calculus used to be fairly simple. You used to be able to identify which applications contained the most sensitive assets and were exposed to the most attackers, and that would tell you which ones were the most risky.

Unfortunately, in most enterprises, many applications are connected. There are many types of connections. The simplest is when two applications actually talk to each other. But if a user can access both through their browser, they're also connected. If you have single sign on (SSO) in your enterprise, then all your applications are connected through your employee's browsers. Another way applications are connected is when they are cohosted, sharing an operating system. A weakness in one application could compromise another more sensitive application. Or perhaps applications share a backend system, and an attack could propagate from one system to another.

Only in the past few years have many organizations started to realize the risk that vulnerabilities in their "internal" applications pose to their business.

Enabling developers to write secure code

Let's discuss enabling your developers to write secure code. An organization has to provide developers with the things they need in order to be able to produce secure code. Far too many organizations simply expect secure code without doing anything to make it possible for developers to build it.

The first thing to do is to work out the set of security controls available to developers in their environment. Every organization has certain security controls in their infrastructure, such as encryption libraries, logging servers, authentication servers, and more. Developers need easy access to these controls and it helps to have a standard way to do this. Organizations that establish standard security controls can cut security costs across the lifecycle, including training, design, implementation, and testing.

Once you understand the security controls available, you can work out a secure coding guideline for your organization. This is the set of rules for developers to follow when developing applications. This guideline is specific and contains many code snippets and examples of how to program securely. In addition, the guideline is tailored for your environment, your policies, and the technologies that you use. We've found that higher level policies do not communicate well to developers. If you want a particular programming practice followed then you better show it in code.

The last thing to do to support developers is to get them some hands on secure coding training. You want instructors that are experienced developers with application security expertise – developers will immediately detect a trainer who is not really a programmer. The course should cover when and how to use all the major security controls, giving examples of the common security vulnerabilities associated with each control and how to follow the secure coding guidelines in order to use the controls to avoid these vulnerabilities.

ESAPI cuts application security costs

Security can be very expensive. In fact, it can be prohibitively expensive if not provided in the proper manner. This requires us to focus on security as early as possible in the development lifecycle since the costs of dealing with a security vulnerability in production are roughly 100 times greater than detecting and eliminating them during design.

Let's look at how vulnerabilities get introduced. Every vulnerability in production actually represents a series of mistakes that occurred during the software development process. The worst way to discover an application security problem is when it gets exploited by an attacker.

Using an ESAPI has cost benefits across the software development lifecycle. From training and requirements all the way through implementation, testing, and deployment, the costs associated with security are dramatically smaller when your enterprise has an ESAPI in place. In the table below, we

made some conservative estimates of the cost savings associated with an ESAPI in a company with roughly 1000 applications. For example, the cost associated with doing security training is significantly smaller, because you only have to teach the security APIs and where to use them, instead of everything about security. Also, the costs of security testing are significantly decreased, because you no longer have to verify that the security controls are correct since you've already tested them, just that they're used properly.

Cost Area	Typical	With ESAPI
AppSec Training (semiannual)	$270K	$135K
AppSec Requirements	250 days ($150K)	50 days ($30K)
AppSec Design (Threat Model, Arch Review)	500 days ($300K)	250 days ($150K)
AppSec Implementation (Build and Use Controls)	1500 days ($900K)	500 days ($300K)
AppSec Verification (Scan, Code Review, Pen Test)	500 days ($300K)	250 days ($150K)
AppSec Remediation	500 days ($300K)	150 days ($90K)
AppSec Standards and Guidelines	100 days ($60K)	20 days ($12K)
AppSec Inventory, Metrics, and Management	250 days ($150K)	200 days ($120K)
Totals	$2.43M	$1.00M

A "Positive" Approach to Application Security

Designing an API is no trivial task. It requires an extremely deep understanding of how security Is typically done in the enterprise and how mistakes get made. We've spent a great deal of time studying how developers make mistakes, and we've designed the API to make these mistakes more difficult.

Chasing vulnerabilities

Many organizations approach application security with penetration testing and vulnerability scanning – essentially hacking. While it is generally easy to find serious holes this way, it's impossible to hack yourself secure.

We've analyzed the security vulnerabilities discovered in a decade of application security verification with code review and security testing. We found that almost all vulnerabilities fall into one of four categories:

A huge percentage (35%) of security problems are due to missing controls – like applications that simply don't do output encoding, or that don't encrypt sensitive data, or fail to use parameterized database queries. So simply by making the right security controls available in an application, you can eliminate quite a lot of vulnerabilities.

We also see far too many (30%) security controls that are simply broken. One application we looked at recently had three different HTML entity encoding routines, all of which were blacklist and seriously broken. As we'll see, security controls are hard – you don't want to be reimplementing them all the time. Developers need a strong set of standard security controls to use.

We also see vulnerabilities happen (20%) where security controls are available, but the developer simply didn't use them. Encryption is a great control but it doesn't help if you don't encrypt the sensitive information. We need to provide controls that are easy to use. Ideally, they would automatically be used, which is possible in some situations, like automatically logging when security checks fail, if those checks are provided in standard security controls.

The last case is when the developer has controls available but uses or configures them incorrectly (15%). Frequently, this is a direct result of the API being difficult to understand.

Based on these findings, we believe that we can eliminate huge numbers of vulnerabilities simply by providing developers with well-implemented and well-designed security controls and doing everything possible to make sure that developers use them in all the right places.

A "positive" approach to application security

This "positive" approach to security works because the number of attacks and categories of vulnerability are so extensive. MITRE has catalogued almost 700 different categories of programming mistakes that can lead to vulnerability. Each of these weaknesses is targeted by a variety of attacks. Addressing these problems requires only a small number – a dozen or so – well-designed security controls. The positive approach tends to be easier to manage and more cost-effective for organizations.

Don't slip into thinking that as long as your web application does what it's supposed to, anything else it does is okay. Instead, think of your application as an API that you're exposing to attackers. What shows up in your user's browsers is irrelevant, since attackers can invoke any method with any parameters.

ESAPI takes a positive approach to the design decisions inherent in creating a set of security controls. All the controls use a fail-closed, deny by default, whitelist approach to implementing their functions.

Focus on fundamental security controls

Both ESAPI and this book are organized around fundamental security controls. We have not created chapters around common vulnerabilities or attacks, such as XSS, CSRF, SQL injection, etc… Books organized this way are sexy, but don't encourage the best architecture possible.

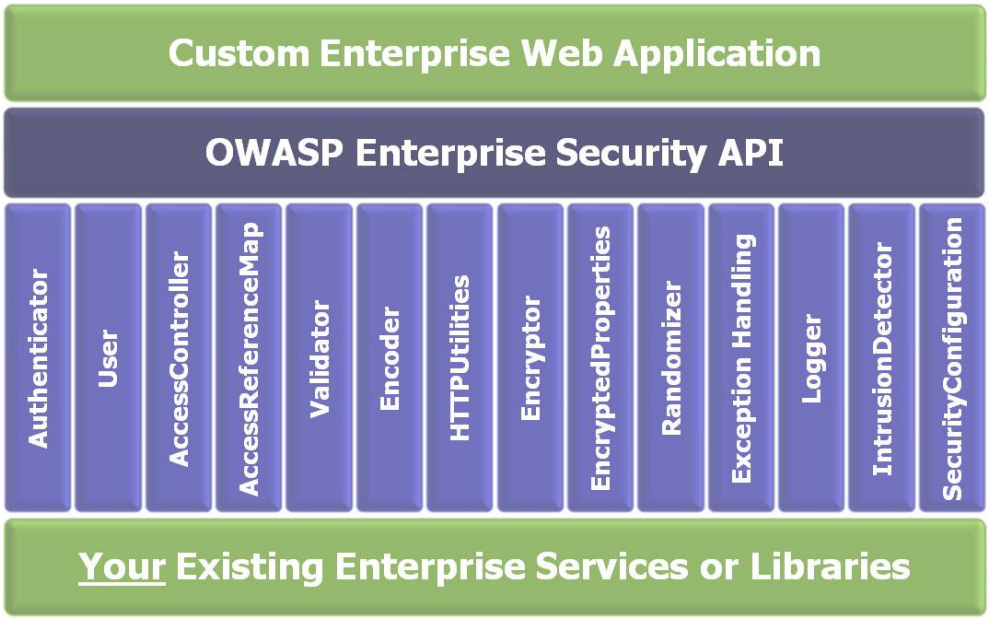
For example, if you are trying to stamp out XSS from your enterprise, the right way to do it is to establish great input validation and output encoding mechanisms that developers can use easily. We discuss these two fundamental controls in our book. But those same controls help to stop a wide range of other attacks as well. We believe that focusing on getting the right controls in place is the path to improved software security.

Designing an Enterprise Security API

This section discusses the rationale for some of the key design decisions that went into creating the ESAPI. We hope to share our approach so that you will understand why these decisions were made and can consider the issues when creating your own ESAPI.

Separating interfaces from implementation

ESAPI is primarily a set of interfaces designed to make security easy to use. These interfaces are usable by anyone who cares to implement them for their enterprise. By keeping the interfaces separate, we are trying to encourage organizations to create their own implementations.



However, we didn't stop there. We built a complete set of reference implementations. The reference implementations are complete and well tested, although there are a few limitations. In general, the reference implementation is designed to run without requiring any other infrastructure, such as database or LDAP servers.

The Authenticator and AccessController implementations in particular work of simple text file policies, which may not be appropriate for enterprise applications. However, the rest of the reference implementation is scalable and appropriate for enterprise applications. You will likely want to hook up the Logger to your logging infrastructure.

Implementating the ESAPI interfaces is not difficult, and can be done without a great deal of security knowledge. In our experience, it takes only a few hours to hook up a new implementation of an ESAPI interface to an existing backend system, such as a user repository for example.

Supporting remediation

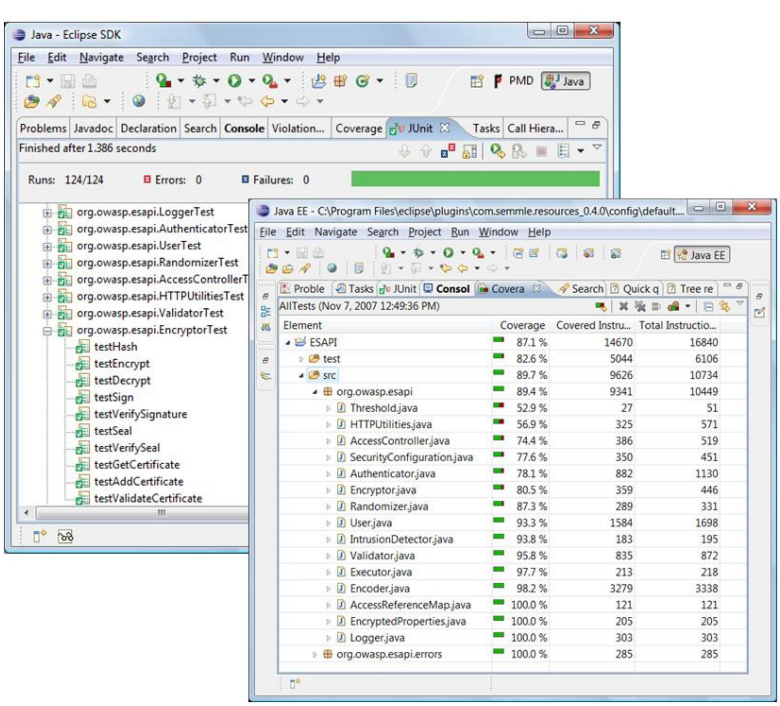
TODO

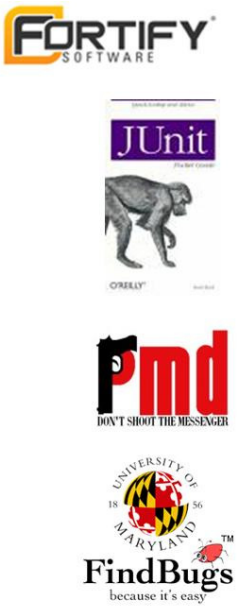
Fundamental security controls, not a framework

TODO

Assurance

It's absolutely critical to get your applications security controls right. An error could expose many applications. Confidence in your security controls comes from evidence like design documentation, code review, security testing, and other analysis.





The ESAPI project involves a world-class team of software security experts from vendors and industry. The reference implementation is small and well structured – about 5,000 lines of well-documented and extensively reviewed code. The code is clean in all the major static analysis tools, including FindBugs, PMD, Ounce, and Fortify. The project also includes about 600 test cases that test all aspects of the security mechanisms.

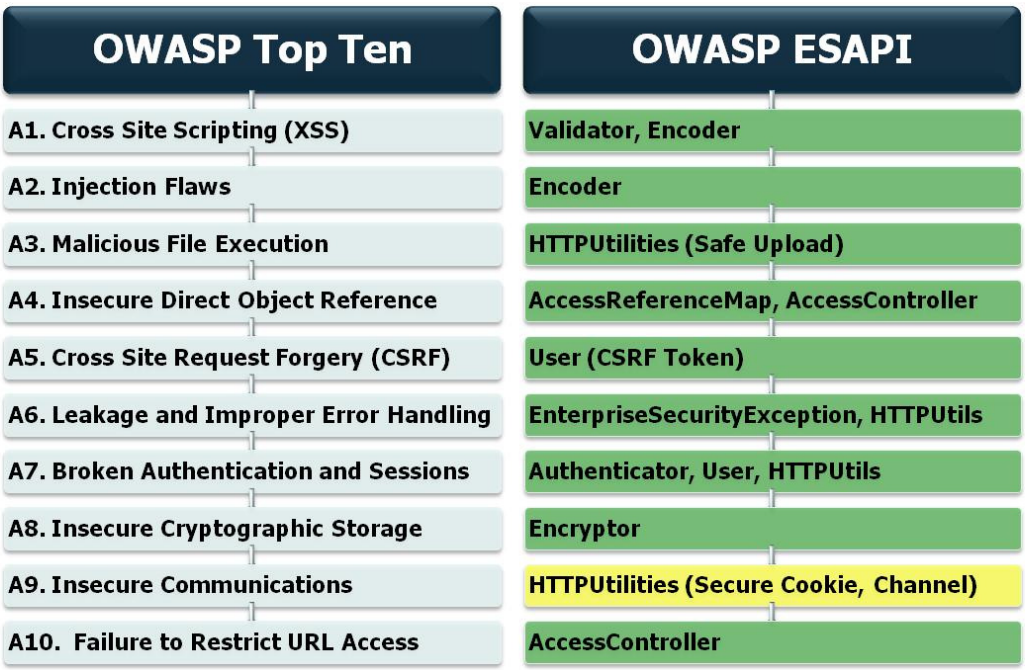
Supporting YOUR software development process

According to IDC, we create several billion lines of new code every year. We have to involve development teams more in securing this code, as there simply are not enough application security specialists to handle that much code, not to mention the trillion lines of code that IDC says have already been developed.

Using an ESAPI is a way to leverage the application security specialists that you have. There are dramatic cost savings associated with the use of an ESAPI throughout the software lifecycle. Training, requirements, design, implementation, testing, deployment, operation, and remediation all take less time, effort, and expertise because your security knowledge has been institutionalized in your ESAPI library.

Establishing YOUR Enterprise Security API

Ask yourself what security controls your developers need to build your applications? Here's a good way to figure it out. Take a look at the common application security vulnerabilities you're finding in penetration testing and code reviews. Then list the security controls that developers need to prevent those holes. You'll end up with a list that includes authentication, session management, access control, input validation, canonicalization, output encoding, parameterized interfaces, encryption, hashing, random numbers, logging, and error handling.



Ask yourself what coverage of the OWASP Top Ten you really need. Then make sure you have an API that will enable developers to avoid those pitfalls.

Why you shouldn't build your own security controls

Writing security controls is time-consuming and extremely prone to mistakes. MITRE's CWE project lists over 600 different types of security mistakes that developers can make, and most of them are not at all obvious. Most people recognize that developers should not build their own encryption mechanisms, but the same argument applies to all the security controls.

Why you shouldn't use security libraries directly

There are plenty of libraries and frameworks out there that provide various security functions – Log4j, Java Cryptographic Extension (JCE), JAAS, Acegi, and dozens more. Some of them are even pretty good at what they do. But there are several reasons why enterprise developers should not use them directly.

Most importantly, these libraries are overpowerful. Most developers only need a very limited set of security functions and don't need a complex interface. Further, many of these libraries contain security holes themselves – such as encoding libraries that don't canonicalize or authentication libraries that don't use strong cryptographic functions. Because many security controls use features from other controls, using security libraries that aren't integrated together is a mistake.

Why you shouldn't rely on security in the platform or framework

Unfortunately, the web application platforms and frameworks do not protect against many well-known attacks. For example, consider header injection – allowing carriage return (CR) and line-feed (LF) characters to be used in HTTP headers. This injection changes the way these HTTP messages are parsed, allowing the attacker to affect how they are interpreted. This type of injection can allow attackers to download malicious files to the user's desktop among other serious impacts.

The platforms could protect against this attack which violates the HTTP specification, but many do not. Instead, your application developers must know to carefully validate and filter user data before using it in HTTP headers.

The platforms and frameworks are a shifting environment. Your code may be protected by something in your infrastructure today, but that environment may change tomorrow. The best approach is to build your applications so that they provide their own protection. If this results in some overlapping protection, also known as defense-in-depth, that's a good outcome.

Create a security API that matches YOUR enterprise

Even if you don't trust open source code, please consider the concept of establishing an ESAPI. With the OWASP project as a model, it would not take much time at all to create a custom ESAPI for your organization. You could adopt just the ESAPI interfaces and use parts of the reference implementation that make sense for you.

Using ESAPI

The full instructions for setting up ESAPI and using it are on the OWASP ESAPI website at http://www.owasp.org/index.php/ESAPI. But the fundamentals are very simple.

First, you should download the latest ESAPI distribution and unzip it into your application's lib directory. In addition to the latest ESAPI.jar file, there are several libraries that are used by the ESAPI reference implementation that you'll need. Be careful if these libraries collide with libraries you are already using in your application.

Next, you should ensure that your ESAPI resources are properly set up. The ESAPI reference implementation uses a "resources" directory containing several files. The resources directory can be located anywhere on your classpath or can be specified with an environment variable on the Java command line as follows: -D org.owasp.esapi.resources="C:\resources"

The most important file in the resources directory is the ESAPI.properties file. In this file, you MUST change the master password for your ESAPI installation. You may also want to add validation patterns, logging configurations, encryption algorithms, and exception thresholds.

Once this is set up, you can call ESAPI from your code.

```
import org.owasp.esapi.ESAPI
...
String input = request.getParameter( "ssn" );
String validInput = ESAPI.validator().getValidInput( "ssn", input, "ssn", 25, false)
String safeOutput = ESAPI.encoder().encodeForHTML( validInput );
out.println( <div>SSN: <b>" + safeOutput + "</b></div>" );
...
```

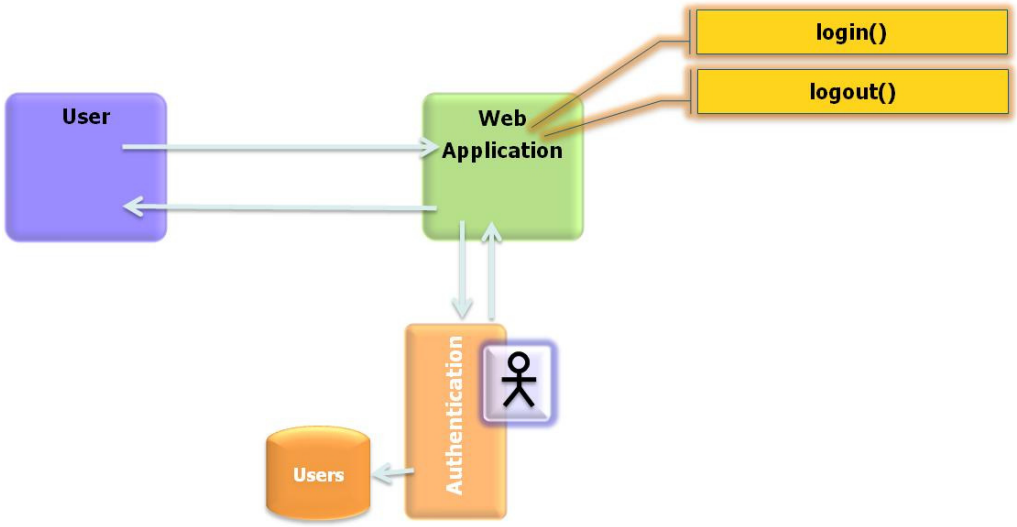
Positive Authentication

Authentication is one of the cornerstones of an enterprise security program. Many other security controls depend on proper authentication, including access control and accountability. Many enterprises have a powerful authentication infrastructure in place, possibly including ActiveDirectory, LDAP, custom databases, SiteMinder, or Tivoli, but have not done everything possible to make it easy for developers to properly use this infrastructure.

Currently, ESAPI supports only username and password authentication. However, we recognize the limitations of this approach to authentication and are investigating how to extend the API to support more sophisticated of authentication, including certificates, federated approaches, and more.

Design

The ESAPI Authenticator is designed to work closely with the ESAPI User interface. Most of the work is done in the Authenticator, while the User interface is a simple data transfer object. For simple applications, the developer will only need to call the "login()" and "logout()" methods and authentication will be handled securely.



Identity everywhere

One design decision in the Authenticator greatly simplifies the implementation of identity management through ESAPI. The "getCurrentUser()" method is quite important as it makes it possible to easily get the identity of the currently logged in user. Without this capability, developers would have to pass around the request, session, or User object throughout their applications. This would virtually guarantee that

the currently logged in user is not available when it is needed for logging or access control decisions. The ESAPI approach makes the identity available everywhere, so that these security activities can be performed without any additional information.

User

ESAPI provides a User interface that models the information required for security. This interface is intended to be essentially a data transfer object, so that all the details of establishing a database or registry of Users is part of the Authenticator interface.

Many organizations already have a User class that they have created internally. You may want to compare against the methods available in the ESAPI User interface to see if there are any capabilities missing in your User implementation.

Probably the easiest thing to do is to add the ESAPI User interface to the list of interfaces implemented by your User class. Then just implement the methods so that they work with your user repository.

Creating a user from your code is easy. You can simply create a user with an account name and a password. This is designed to take information directly from a registration page. The username and password strength checks are called by this method.

```
User user = ESAPI.authenticator().createUser( accountName, password1, password2 );
```

Establishing strong credentials

ESAPI has considerable support for strong credentials. There are methods to check the strength of both usernames and passwords. There is also support for generating strong passwords using a cryptographically strong source of randomness and a character set that minimizes the likelihood of confusion for the user.

```
String newPassword = ESAPI.authenticator().generateStrongPassword();
```

If you already have a User, then you can generate a strong password that takes into account their username and previous passwords.

```
String newPassword = ESAPI.authenticator().generateStrongPassword(user, oldPassword );
```

ESAPI makes it simple to verify username and password strength. It uses an algorithm that checks the number of different character sets used and the length to calculate a strength.

```
ESAPI.authenticator().verifyAccountNameStrength( accountName );
ESAPI.authenticator().verifyPasswordStrength( oldPassword, newPassword );
```

Protecting credentials

ESAPI ensures that credentials are stored in a hashed form to prevent disclosure if the user repository is disclosed. To stop rainbow table attacks, the hashes are salted and iterated properly. ESAPI also ensures that all pages from login form to logout confirmation must be transmitted over SSL to prevent credential and session disclosure. See the Session Management chapter for more information

Interface Authenticator

org.owasp.esapi

All Known Implementing Classes:

FileBasedAuthenticator

```
public interface Authenticator
```

The Authenticator interface defines a set of methods for generating and handling account credentials and session identifiers. The goal of this interface is to encourage developers to protect credentials from disclosure to the maximum extent possible.

One possible implementation relies on the use of a thread local variable to store the current user's identity. The application is responsible for calling setCurrentUser() as soon as possible after each HTTP request is received. The value of getCurrentUser() is used in several other places in this API. This eliminates the need to pass a user object to methods throughout the library. For example, all of the logging, access control, and exception calls need access to the currently logged in user.

The goal is to minimize the responsibility of the developer for authentication. In this example, the user simply calls authenticate with the current request and the name of the parameters containing the username and password. The implementation should verify the password if necessary, create a session if necessary, and set the user as the current user.

```
public void doPost(ServletRequest request, ServletResponse response) {
try {
User user = ESAPI.authenticator().login(request, response);
// continue with authenticated user
} catch (AuthenticationException e) {
// handle failed authentication (it's already been logged)
}
```

Method Summary	
void	**changePassword**(User user, String currentPassword, String newPassword, String newPassword2) Changes the password for the specified user.
void	**clearCurrent**() Clear the current user.

User	**createUser**(String accountName, String password1, String password2) Creates a new User with the information provided.
boolean	**exists**(String accountName) Determine if the account exists.
String	**generateStrongPassword**() Generate a strong password.
String	**generateStrongPassword**(User user, String oldPassword) Generate strong password that takes into account the user's information and old password.
User	**getCurrentUser**() Returns the currently logged in User.
User	**getUser**(String accountName) Returns the User matching the provided accountName.
User	**getUser**(long accountId) Returns the User matching the provided accountId.
Set	**getUserNames**() Gets a collection containing all the existing user names.
String	**hashPassword**(String password, String accountName) Returns a string representation of the hashed password, using the accountName as the salt.
User	**login**(HttpServletRequest request, HttpServletResponse response) Authenticates the user's credentials from the HttpServletRequest if necessary, creates a session if necessary, and sets the user as the current user.
void	**logout**() Logs out the current user.
void	**removeUser**(String accountName) Removes the account.
void	**setCurrentUser**(User user) Sets the currently logged in User.

void	**verifyAccountNameStrength**(String accountName) Ensures that the account name passes site-specific complexity requirements, like minimum length.
boolean	**verifyPassword**(User user, String password) Verify that the supplied password matches the password for this user.
void	**verifyPasswordStrength**(String oldPassword, String newPassword) Ensures that the password meets site-specific complexity requirements.

Method Detail

clearCurrent

```
void clearCurrent()
```

Clear the current user. This allows the thread to be reused safely.

---

login

```
User login(HttpServletRequest request,
           HttpServletResponse response)
    throws AuthenticationException
```

Authenticates the user's credentials from the HttpServletRequest if necessary, creates a session if necessary, and sets the user as the current user.

Parameters:

`request` - the current HTTP request

`response` - the response

Returns:

the user

Throws:

`AuthenticationException` - if the credentials are not verified, or if the account is disabled, locked, expired, or timed out

verifyPassword

```
boolean verifyPassword(User user,
                       String password)
```

Verify that the supplied password matches the password for this user. This method is typically used for "reauthentication" for the most sensitive functions, such as transactions, changing email address, and changing other account information.

Parameters:

`user` - the user

`password` - the password

Returns:

true, if the password is correct for the specified user

---

logout

```
void logout()
```

Logs out the current user.

---

createUser

```
User createUser(String accountName,
                String password1,
                String password2)
         throws AuthenticationException
```

Creates a new User with the information provided. Implementations should check the accountName and password for proper format and strength against brute force attacks. Two copies of the new password are required to encourage user interface designers to include a "re-type password" field in their forms. Implementations should verify that both are the same.

Parameters:

`accountName` - the account name of the new user

`password1` - the password of the new user

`password2` - the password of the new user. This field is to encourage user interface designers to include two password fields in their forms.

Returns:

the User that has been created

Throws:

`AuthenticationException` - if user creation fails

---

generateStrongPassword

```
String generateStrongPassword()
```

Generate a strong password. Implementations should use a large character set that does not include confusing characters, such as i I 1 l 0 o and O. There are many algorithms to generate strong memorable passwords that have been studied in the past.

Returns:

a password with strong password strength

---

generateStrongPassword

```
String generateStrongPassword(User user,
                              String oldPassword)
```

Generate strong password that takes into account the user's information and old password. Implementations should verify that the new password does not include information such as the username, fragments of the old password, and other information that could be used to weaken the strength of the password.

Parameters:

`user` - the user whose information to use when generating password

`oldPassword` - the old password to use when verifying strength of new password. The new password may be checked for fragments of oldPassword.

Returns:

a password with strong password strength

---

changePassword

```
void changePassword(User user,
                    String currentPassword,
                    String newPassword,
                    String newPassword2)
             throws AuthenticationException
```

Changes the password for the specified user. This requires the current password, as well as the password to replace it with. This new password must be repeated to ensure that the user has typed it in correctly.

Parameters:

`user` - the user to change the password for

`currentPassword` - the current password for the specified user

`newPassword` - the new password to use

`newPassword2` - a verification copy of the new password

Throws:

`AuthenticationException` - if any errors occur

---

getUser

```
User getUser(long accountId)
```

Returns the User matching the provided accountId.

Parameters:

`accountId` - the account id

Returns:

the matching User object, or null if no match exists

---

getUser

`User` **`getUser`**`(String accountName)`

Returns the User matching the provided accountName.

Parameters:

`accountName` - the account name

Returns:

the matching User object, or null if no match exists

---

getUserNames

`Set` **`getUserNames`**`()`

Gets a collection containing all the existing user names.

Returns:

a set of all user names

---

getCurrentUser

`User` **`getCurrentUser`**`()`

Returns the currently logged in User.

Returns:

the matching User object, or the Anonymous user if no match exists

---

setCurrentUser

```
void setCurrentUser(User user)
```

Sets the currently logged in User.

Parameters:

`user` - the user to set as the current user

---

hashPassword

```
String hashPassword(String password,
                    String accountName)
             throws EncryptionException
```

Returns a string representation of the hashed password, using the accountName as the salt. The salt helps to prevent against "rainbow" table attacks where the attacker pre-calculates hashes for known strings. This method specifies the use of the user's account name as the "salt" value. The Encryptor.hash method can be used if a different salt is required.

Parameters:

`password` - the password to hash

`accountName` - the account name to use as the salt

Returns:

the hashed password

Throws:

`EncryptionException`

---

removeUser

```
void removeUser(String accountName)
         throws AuthenticationException
```

Removes the account.

Parameters:

`accountName` - the account name to remove

Throws:

`AuthenticationException` - the authentication exception if user does not exist

---

verifyAccountNameStrength

```
void verifyAccountNameStrength(String accountName)
                        throws AuthenticationException
```

Ensures that the account name passes site-specific complexity requirements, like minimum length.

Parameters:

`accountName` - the account name

Throws:

`AuthenticationException` - if account name does not meet complexity requirements

---

verifyPasswordStrength

```
void verifyPasswordStrength(String oldPassword,
                            String newPassword)
                     throws AuthenticationException
```

Ensures that the password meets site-specific complexity requirements. This method takes the old password so that the algorithm can analyze the new password to see if it is too similar to the old password. Note that this has to be invoked when the user has entered the old password, as the list of old credentials stored by ESAPI is all hashed.

Parameters:

`oldPassword` - the old password

`newPassword` - the new password

Returns:

true, if the new password meets complexity requirements and is not too similar to the old password

Throws:

`AuthenticationException` - if newPassword is too similar to oldPassword or if newPassword does not meet complexity requirements

---

exists

```
boolean exists(String accountName)
```

Determine if the account exists.

Parameters:

`accountName` - the account name

Returns:

true, if the account exists

Interface User

org.owasp.esapi

All Superinterfaces:

Principal

All Known Implementing Classes:

DefaultUser

```
public interface User

extends Principal
```

The User interface represents an application user or user account. There is quite a lot of information that an application must store for each user in order to enforce security properly. There are also many rules that govern authentication and identity management.

A user account can be in one of several states. When first created, a User should be disabled, not expired, and unlocked. To start using the account, an administrator should enable the account. The account can be locked for a number of reasons, most commonly because they have failed login for too many times. Finally, the account can expire after the expiration date has been reached. The User must be enabled, not expired, and unlocked in order to pass authentication.

Field Summary	
User	**ANONYMOUS** The ANONYMOUS user is used to represent an unidentified user.

Method Summary	
void	**addRole**(String role) Adds a role to an account.
void	**addRoles**(Set newRoles) Adds a set of roles to an account.

void	**changePassword**(String oldPassword, String newPassword1, String newPassword2) Sets the user's password, performing a verification of the user's old password, the equality of the two new passwords, and the strength of the new password.
void	**disable**() Disable account.
void	**enable**() Enable account.
long	**getAccountId**() Gets the account id.
String	**getAccountName**() Gets the account name.
String	**getCSRFToken**() Gets the CSRF token.
Date	**getExpirationTime**() Returns the date that the current user's account will expire, usually when the account will be disabled.
int	**getFailedLoginCount**() Returns the number of failed login attempts since the last successful login for an account.
Date	**getLastFailedLoginTime**() Returns the date of the last failed login time for a user.
String	**getLastHostAddress**() Returns the last host address used by the user.
Date	**getLastLoginTime**() Returns the date of the last successful login time for a user.
Date	**getLastPasswordChangeTime**() Gets the date of user's last password change.

Set	**getRoles**() Gets the roles assigned to a particular account.
String	**getScreenName**() Gets the screen name.
void	**incrementFailedLoginCount**() Increment failed login count.
boolean	**isAnonymous**() Checks if user is anonymous.
boolean	**isEnabled**() Checks if an account is currently enabled.
boolean	**isExpired**() Checks if an account is expired.
boolean	**isInRole**(String role) Checks if an account has been assigned a particular role.
boolean	**isLocked**() Checks if an account is locked.
boolean	**isLoggedIn**() Tests to see if the user is currently logged in.
boolean	**isSessionAbsoluteTimeout**() Tests to see if the user's session has exceeded the absolute time out.
boolean	**isSessionTimeout**() Tests to see if the user's session has timed out from inactivity.
void	**lock**() Lock the user's account.
void	**loginWithPassword**(String password) Login with password.
void	**logout**() Logout this user.

void	**removeRole**(String role) Removes a role from an account.
String	**resetCSRFToken**() Returns a token to be used as a prevention against CSRF attacks.
void	**setAccountName**(String accountName) Sets the account name.
void	**setExpirationTime**(Date expirationTime) Sets the time when this user's account will expire.
void	**setLastFailedLoginTime**(Date lastFailedLoginTime) Set the time of the last failed login for this user.
void	**setLastHostAddress**(String remoteHost) Set the last remote host address used by this user.
void	**setLastLoginTime**(Date lastLoginTime) Set the time of the last successful login for this user.
void	**setLastPasswordChangeTime**(Date lastPasswordChangeTime) Set the time of the last password change for this user.
void	**setRoles**(Set roles) Sets the roles of this account.
void	**setScreenName**(String screenName) Sets the screen name.
void	**unlock**() Unlock account.
boolean	**verifyPassword**(String password) Verify that the supplied password matches the password for this user.

Field Detail

ANONYMOUS

```
public static final User ANONYMOUS
```

The ANONYMOUS user is used to represent an unidentified user. Since there is always a real user, the ANONYMOUS user is better than using null to represent this.

Method Detail

addRole

```
void addRole(String role)
      throws AuthenticationException
```

Adds a role to an account.

Parameters:

`role` - the role to add

Throws:

`AuthenticationException` - the authentication exception

---

addRoles

```
void addRoles(Set newRoles)
       throws AuthenticationException
```

Adds a set of roles to an account.

Parameters:

`newRoles` - the new roles to add

Throws:

`AuthenticationException` - the authentication exception

---

changePassword

```
void changePassword(String oldPassword,
                    String newPassword1,
                    String newPassword2)
             throws AuthenticationException,
                    EncryptionException
```

Sets the user's password, performing a verification of the user's old password, the equality of the two new passwords, and the strength of the new password.

Parameters:

`oldPassword` - the old password

`newPassword1` - the new password

`newPassword2` - the new password - used to verify that the new password was typed correctly

Throws:

`AuthenticationException` - if newPassword1 does not match newPassword2, if oldPassword does not match the stored old password, or if the new password does not meet complexity requirements

`EncryptionException`

---

disable

```
void disable()
```

Disable account.

Throws:

`AuthenticationException` - the authentication exception

---

enable

```
void enable()
```

Enable account.

Throws:

AuthenticationException - the authentication exception

---

getAccountId

```
long getAccountId()
```

Gets the account id.

Returns:

the account id

---

getAccountName

```
String getAccountName()
```

Gets the account name.

Returns:

the account name

---

getCSRFToken

```
String getCSRFToken()
```

Gets the CSRF token.

Returns:

the CSRF token

---

getExpirationTime

```
Date getExpirationTime()
```

Returns the date that the current user's account will expire, usually when the account will be disabled.

Returns:

Date representing the account expiration time.

---

getFailedLoginCount

```
int getFailedLoginCount()
```

Returns the number of failed login attempts since the last successful login for an account. This method is intended to be used as a part of the account lockout feature, to help protect against brute force attacks. However, the implementor should be aware that lockouts can be used to prevent access to an application by a legitimate user, and should consider the risk of denial of service.

Returns:

the number of failed login attempts since the last successful login

---

getLastHostAddress

```
String getLastHostAddress()
```

Returns the last host address used by the user. This will be used in any log messages generated by the processing of this request.

Returns:

the last host address used by the user

---

getLastFailedLoginTime

```
Date getLastFailedLoginTime()
                    throws AuthenticationException
```

Returns the date of the last failed login time for a user. This date should be used in a message to users after a successful login, to notify them of potential attack activity on their account.

Returns:

date of the last failed login

Throws:

AuthenticationException - the authentication exception

---

getLastLoginTime

```
Date getLastLoginTime()
```

Returns the date of the last successful login time for a user. This date should be used in a message to users after a successful login, to notify them of potential attack activity on their account.

Returns:

date of the last successful login

---

getLastPasswordChangeTime

```
Date getLastPasswordChangeTime()
```

Gets the date of user's last password change.

Returns:

the date of last password change

---

getRoles

```
Set getRoles()
```

Gets the roles assigned to a particular account.

Returns:

an immutable set of roles

---

getScreenName

```
String getScreenName()
```

Gets the screen name.

Returns:

the screen name

---

incrementFailedLoginCount

```
void incrementFailedLoginCount()
```

Increment failed login count.

---

isAnonymous

```
boolean isAnonymous()
```

Checks if user is anonymous.

Returns:

true, if user is anonymous

---

isEnabled

```
boolean isEnabled()
```

Checks if an account is currently enabled.

Returns:

true, if account is enabled

---

isExpired

```
boolean isExpired()
```

Checks if an account is expired.

Returns:

true, if account is expired

---

isInRole

```
boolean isInRole(String role)
```

Checks if an account has been assigned a particular role.

Parameters:

`role` - the role for which to check

Returns:

true, if role has been assigned to user

---

isLocked

```
boolean isLocked()
```

Checks if an account is locked.

Returns:

true, if account is locked

isLoggedIn

```
boolean isLoggedIn()
```

Tests to see if the user is currently logged in.

Returns:

true, if the user is logged in

---

isSessionAbsoluteTimeout

```
boolean isSessionAbsoluteTimeout()
```

Tests to see if the user's session has exceeded the absolute time out.

Returns:

true, if user's session has exceeded the absolute time out

---

isSessionTimeout

```
boolean isSessionTimeout()
```

Tests to see if the user's session has timed out from inactivity.

Returns:

true, if user's session has timed out from inactivity

---

lock

```
void lock()
```

Lock the user's account.

---

loginWithPassword

```
void loginWithPassword(String password)
                throws AuthenticationException
```

Login with password.

Parameters:

`password` - the password

Throws:

`AuthenticationException` - if login fails

---

logout

```
void logout()
```

Logout this user.

---

removeRole

```
void removeRole(String role)
         throws AuthenticationException
```

Removes a role from an account.

Parameters:

`role` - the role to remove

Throws:

`AuthenticationException` - the authentication exception

---

resetCSRFToken

```
String resetCSRFToken()
               throws AuthenticationException
```

Returns a token to be used as a prevention against CSRF attacks. This token should be added to all links and forms. The application should verify that all requests contain the token, or they may have been generated by a CSRF attack. It is generally best to perform the check in a centralized location, either a filter or controller. See the verifyCSRFToken method.

Returns:

the new CSRF token

Throws:

`AuthenticationException` - the authentication exception

---

setAccountName

```
void setAccountName(String accountName)
```

Sets the account name.

Parameters:

`accountName` - the new account name

---

setExpirationTime

```
void setExpirationTime(Date expirationTime)
```

Sets the time when this user's account will expire.

Parameters:

`expirationTime` - the new expiration time

---

setRoles

```
void setRoles(Set roles)
       throws AuthenticationException
```

Sets the roles of this account.

Parameters:

`roles` - the new roles

Throws:

`AuthenticationException` - the authentication exception

---

setScreenName

```
void setScreenName(String screenName)
```

Sets the screen name.

Parameters:

`screenName` - the new screen name

---

unlock

```
void unlock()
```

Unlock account.

---

verifyPassword

```
boolean verifyPassword(String password)
                throws EncryptionException
```

Verify that the supplied password matches the password for this user. This method is typically used for "reauthentication" for the most sensitive functions, such as transactions, changing email address, and changing other account information.

Parameters:

`password` - the password that the user entered

Returns:

true, if the password passed in matches the account's password

Throws:

EncryptionException

---

setLastFailedLoginTime

```
void setLastFailedLoginTime(Date lastFailedLoginTime)
```

Set the time of the last failed login for this user.

---

setLastHostAddress

```
void setLastHostAddress(String remoteHost)
```

Set the last remote host address used by this user.

---

setLastLoginTime

```
void setLastLoginTime(Date lastLoginTime)
```

Set the time of the last successful login for this user.

---

setLastPasswordChangeTime

```
void setLastPasswordChangeTime(Date lastPasswordChangeTime)
```

Set the time of the last password change for this user.

Positive Session Management

Everyone knows that protecting passwords is important, but do you take the same care with your session identifiers? Session identifier cookies are often exposed and just as important. Here's how they work - when you log into a web application, you exchange your credentials for a session identifier cookie. This cookie gets sent with every subsequent request from your browser until you log out or the session times out. During that window, if an attacker steals your session identifier, they have full access to your account.

What is your session identifier? Log into a website and type "javascript:alert(document.cookie)" into your browser. That number is very important and must be kept secret. Anyone who has it can hijack your account. So what do you need to do to protect your application's session identifiers?

Using SSL

Some web applications use SSL for the username and password, but then fall back to a non-SSL connection after authentication. Unfortunately, this means that the session identifier is transmitted in the clear in every HTTP request, where it can be easily read by anyone with access to the network. This attack is called "sidejacking" and there are simple tools available to exploit this weakness.

Don't forget your Ajax requests, as they may also contain a session identifier. Gmail has this problem, as their application sometimes falls back to non-SSL for Ajax requests, exposing the user's Gmail session identifier on the wire. Google recently added a setting to "always use SSL" that you should enable right now.

Despite performance issues, the only solution to protecting your session identifiers on the wire is to use SSL for every single page from your login form to your logout confirmation.

ESAPI handles much of this automatically in the Authenticator by verifying that any authenticated request uses SSL.

```
ESAPI.authenticator().login()
```

If you want to verify this yourself, you can use the support method in the HTTPUtilities.

```
ESAPI.httpUtilities().assertSecureRequest( request );
```

Use "Secure" and "HttpOnly" cookie flags

Even if your application always uses SSL, attackers may try to trick the browser into exposing the session identifier over a non-SSL connection by getting victims to view a page including the following type of tag:

```
<img src="http://www.example.com">
```

When the browser sees this tag (even in the attacker's page) it will generate a non-SSL request and send it to www.example.com. The request will include the session identifier, and the attacker can sidejack the user's session. The solution is to use the "Secure" flag on your session identifier. This flag tells the browser to send the cookie only over an SSL connection.

Another way an attacker might steal the session identifier is to use an XSS attack. The injected script simply accesses the cookie and sends it to the attacker. Another cookie flag called HttpOnly can protect against this attack, as it tells the browser not to allow scripts to access the session identifier. While most browsers respect the HttpOnly flag, many development environments do not yet make it easy to set.

ESAPI handles this automatically for all cookies in the SafeResponse class, which is automatically used if you enable the ESAPIFilter.

```
HttpSession session = new SafeRequest( request ).getSession();
```

Avoid URL rewriting with session identifier

Back in the early days of the web, the U.S. Government (wrongly) concluded that cookies were a privacy violation and they banned them. Developers quickly came up with a workaround that involved including the session identifier directly in the URL. Unfortunately, URLs are frequently disclosed via bookmarks, referrers, web logs, cut-and-paste, and more. So the cure was much worse than the disease.

Given the small number of web users that do not allow session cookies and the high risk of session identifier exposure, this URL rewriting technique should not be used. Disabling this may not be trivial, as many frameworks fall back to this technique when cookies are not accepted. You can test your environment by disabling cookies and browsing your site.

ESAPI overrides these methods in the SafeResponse class, which is automatically used if you use the ESAPIFilter. If you want to call these methods manually:

```
String badInput = request.getParameter( "badinput" );
new SafeResponse( response ).setHeader( "inject", badInput );
```

Change session identifier on login

One creative way for attackers to steal session identifiers is to grab them before the user logs in. For example, imagine an evil coworker goes to your desk, browses to a sensitive internal application, and writes down your session identifier. Then when you come in to work and log in, the session identifier is now authenticated, and the attacker can use it to access the application as you.

In principle, the attack is easy to defeat. All you have to do is change the session identifier when a user logs in. Unfortunately, most platforms do not provide an easy way to actually accomplish this. Fortunately, there is a simple method in ESAPI to do just this. This method is called automatically if you are using ESAPI for authentication, but it can also be used as a standalone method.

```
ESAPI.httpUtilities().changeSessionId();
```

Get Logout Right

Many web applications don't properly invalidate the user's session when they hit the logout function. This is important because logout is the only way that a user can end their session and minimize the danger from an attacker that sniffs the session id, steals the session id with an XSS attack, or misuses the session with a CSRF attack.

The most important thing is to actually invalidate the session on the server site. Clearing the session id cookie from the browser is a nice feature, but it's not critical from a security perspective.

```
ESAPI.authenticator().logout();
```

Cross Site Request Forgery (CSRF)

Did you know hackers can force your browser to send requests to any site they want? It's not even hard. All they have to do is get you to view an email or a webpage. Unless the site has specifically protected against this (and almost nobody has), attackers can make your browser do anything you can do – and they can use your credentials and your access privileges. They can set preferences, create payees, write checks, change passwords, etc…

Generally, browsers stop cross site communication by following the "same-origin policy." This rule is pretty simple: if your site has a different origin (protocol, domain, and port don't all match), you aren't allowed to access information from or send requests to the other site. Without this simple rule, there would be no security on the Internet. Every website could access data from every other one – you'd need a separate web browser for every website.

Unfortunately, the same-origin policy is nowhere near airtight. Attackers don't even need an exploit to bypass it. They can simply embed an IMG, SCRIPT, IFRAME, or FORM tag that references the targeted website in an HTML page. When the victim's browser renders this tag, it generates a request and sends it to the targeted website – right around the same-origin policy. This is a feature of all browsers – it's used by many applications to grab images from other sites and to post form data to services.

Attackers can use this loophole to forge requests that appear to be coming from a legitimate user. These are called "cross-site request forgeries" or CSRF for short. Take a look at this old CSRF attack on Neflix. The attacker can post this image tag anywhere: a blog, a wiki, a website, even an email message. Anyone who accidentally browses a page containing the attack while logged into Netflix will have a movie silently added to their queue.

```
<img src=http://www.netflix.com/AddToQueue?movieid=70011204
    width="1" height="1" border="0">
```

Attackers can submit forms on your behalf as well.

```
<body onload="document.forms[0].submit()">
<form method="POST" action="https://bank.com/transfer.do">
       <input type="hidden" name="account" value="123456789"/>
       <input type="hidden" name="amount" value="2500"/>
</form>
```

More complex attacks involve a sending series of requests and delays, allowing the attacker to execute a series of actions in a row. In a strange way, a CSRF attack is a sort of program that executes web instructions on behalf of the attacker. You can test your applications for CSRF vulnerabilities with the OWASP CSRFTester.

What If your browser went renegade? Imagine your browser suddenly turned evil and started trying to mess up your life. What could it do? It might raid the email account you're currently logged into. Maybe it would go after any bank accounts and healthcare sites you're using. Did you click "remember my login" on any websites? Guess what, your renegade browser is logged in and can take those accounts over.

And what if you're at work or connected to the VPN? Your browser can go after the corporate portal. Do you have single sign-on? That means you're logged into every web application on your intranet, and your renegade browser can go after any of them.

Using CSRF, an attacker can attack all of these targets and can do just about anything you can do through your browser. All these attacks can be done remotely and basically anonymously.

What can you do to protect yourself? First, don't stay logged into web sites. You have to actually hit the logout button, not just close the browser. Second, stop CSRF from getting to your critical web sites by using a separate browser to access them. Companies should start thinking about using a separate browser for accessing intranet applications.

If your web application is attacked by a CSRF, all you'll see is normal transactions being performed by authenticated and authorized users. There won't be any way to tell that the user didn't actually execute the transaction. Probably the only way you'll find out that you have a CSRF problem is when users start complaining about phantom transactions on their account. The attacker can cover their tracks easily by removing the attack once it has worked.

Taken alone, CSRF attacks are simple and powerful. However, most attackers use CSRF and XSS in conjunction. Together, these two techniques allow attackers to invade a victim's browser and execute malicious programs using the credentials of site the user is logged into. This combination is devastating and I'm frankly surprised that a cross-application CSRF-XSS worm hasn't already been developed.

The best solution to CSRF is to require a random token in each business function in your application. You can generate the random token when the user logs in and store it in their session. When you generate links and forms, simply add it to the URL or put it in a hidden form field. For example:

```
http://www.example.com?token=8FD41A&data=1
```

Requests that show up without the right token are forged and you can reject them. If you want to add protection without modifying code, the OWASP CSRFGuard is a filter that sits in front of your application and adds token support. In ESAPI, there's built in support for generating these tokens and automatically adding them to a url.

```
String url = ESAPI.httpUtilities().addCSRFToken( "/example/action?t=1" );
```

If you like, you can get the token and add it to a hidden field in a form.

```
String token = ESAPI.httpUtilities().getCSRFToken();
```

That's about the best solution we know of today. OWASP is already working to get CSRF defenses into the web and container standards, but it's not likely to actually happen anytime soon. If you have an application framework, you may want to investigate integrating CSRF protection like CSRFGuard. But don't wait – it will be extremely difficult to roll out protection against forged requests after you're already being attacked.

Positive HTTP Protection

TODO

Class SafeRequest

org.owasp.esapi.filters

```
java.lang.Object
  └ org.owasp.esapi.filters.SafeRequest
```

All Implemented Interfaces:

HttpServletRequest, ServletRequest

```
public class SafeRequest
extends Object
implements HttpServletRequest
```

This request wrapper simply overrides unsafe methods in the HttpServletRequest API with safe versions that return canonicalized data where possible. The wrapper returns a safe value when a validation error is detected, including stripped or empty strings.

Constructor Summary
SafeRequest(HttpServletRequest request) Construct a safe request that overrides the default request methods with safer versions.

Method Summary	
Object	**getAttribute**(String name) Same as HttpServletRequest, no security changes required.
Enumeration	**getAttributeNames**() Same as HttpServletRequest, no security changes required.
String	**getAuthType**() Same as HttpServletRequest, no security changes required.
String	**getCharacterEncoding**() Same as HttpServletRequest, no security changes required.

int	**getContentLength**() Same as HttpServletRequest, no security changes required.
String	**getContentType**() Same as HttpServletRequest, no security changes required.
String	**getContextPath**() Returns the context path from the HttpServletRequest after canonicalizing and filtering out any dangerous characters.
Cookie[]	**getCookies**() Returns the array of Cookies from the HttpServletRequest after canonicalizing and filtering out any dangerous characters.
long	**getDateHeader**(String name) Same as HttpServletRequest, no security changes required.
String	**getHeader**(String name) Returns the named header from the HttpServletRequest after canonicalizing and filtering out any dangerous characters.
Enumeration	**getHeaderNames**() Returns the enumeration of header names from the HttpServletRequest after canonicalizing and filtering out any dangerous characters.
Enumeration	**getHeaders**(String name) Returns the enumeration of headers from the HttpServletRequest after canonicalizing and filtering out any dangerous characters.
ServletInputStream	**getInputStream**() Same as HttpServletRequest, no security changes required.
int	**getIntHeader**(String name) Same as HttpServletRequest, no security changes required.
String	**getLocalAddr**() Same as HttpServletRequest, no security changes required.
Locale	**getLocale**() Same as HttpServletRequest, no security changes required.

Enumeration	**getLocales**() Same as HttpServletRequest, no security changes required.
String	**getLocalName**() Same as HttpServletRequest, no security changes required.
int	**getLocalPort**() Same as HttpServletRequest, no security changes required.
String	**getMethod**() Same as HttpServletRequest, no security changes required.
String	**getParameter**(String name) Returns the named parameter from the HttpServletRequest after canonicalizing and filtering out any dangerous characters.
Map	**getParameterMap**() Returns the parameter map from the HttpServletRequest after canonicalizing and filtering out any dangerous characters.
Enumeration	**getParameterNames**() Returns the enumeration of parameter names from the HttpServletRequest after canonicalizing and filtering out any dangerous characters.
String[]	**getParameterValues**(String name) Returns the array of matching parameter values from the HttpServletRequest after canonicalizing and filtering out any dangerous characters.
String	**getPathInfo**() Returns the path info from the HttpServletRequest after canonicalizing and filtering out any dangerous characters.
String	**getPathTranslated**() Same as HttpServletRequest, no security changes required.
String	**getProtocol**() Same as HttpServletRequest, no security changes required.

String	**getQueryString**() Returns the query string from the HttpServletRequest after canonicalizing and filtering out any dangerous characters.
BufferedReader	**getReader**() Same as HttpServletRequest, no security changes required.
String	**getRealPath**(String path) Same as HttpServletRequest, no security changes required.
String	**getRemoteAddr**() Same as HttpServletRequest, no security changes required.
String	**getRemoteHost**() Same as HttpServletRequest, no security changes required.
int	**getRemotePort**() Same as HttpServletRequest, no security changes required.
String	**getRemoteUser**() Returns the name of the ESAPI user associated with this request.
RequestDispatcher	**getRequestDispatcher**(String path) Checks to make sure the path to forward to is within the WEB-INF directory and then returns the dispatcher.
String	**getRequestedSessionId**() Returns the URI from the HttpServletRequest after canonicalizing and filtering out any dangerous characters.
String	**getRequestURI**() Returns the URI from the HttpServletRequest after canonicalizing and filtering out any dangerous characters.
StringBuffer	**getRequestURL**() Returns the URL from the HttpServletRequest after canonicalizing and filtering out any dangerous characters.

String	**getScheme**() Returns the scheme from the HttpServletRequest after canonicalizing and filtering out any dangerous characters.
String	**getServerName**() Returns the server name (host header) from the HttpServletRequest after canonicalizing and filtering out any dangerous characters.
int	**getServerPort**() Returns the server port (after the : in the host header) from the HttpServletRequest after parsing and checking the range 0-65536.
String	**getServletPath**() Returns the server path from the HttpServletRequest after canonicalizing and filtering out any dangerous characters.
HttpSession	**getSession**() Returns a session, creating it if necessary, and sets the HttpOnly flag on the JSESSIONID cookie.
HttpSession	**getSession**(boolean create) Returns a session, creating it if necessary, and sets the HttpOnly flag on the JSESSIONID cookie.
Principal	**getUserPrincipal**() Returns the ESAPI User associated with this request.
boolean	**isRequestedSessionIdFromCookie**() Same as HttpServletRequest, no security changes required.
boolean	**isRequestedSessionIdFromUrl**() Same as HttpServletRequest, no security changes required.
boolean	**isRequestedSessionIdFromURL**() Same as HttpServletRequest, no security changes required.
boolean	**isRequestedSessionIdValid**() Same as HttpServletRequest, no security changes required.

boolean	**isSecure**() Same as HttpServletRequest, no security changes required.
boolean	**isUserInRole**(String role) Returns true if the ESAPI User associated with this request has the specified role.
void	**removeAttribute**(String name) Same as HttpServletRequest, no security changes required.
void	**setAttribute**(String name, Object o) Same as HttpServletRequest, no security changes required.
void	**setCharacterEncoding**(String enc) Sets the character encoding to the ESAPI configured encoding.

Constructor Detail

SafeRequest

```
public SafeRequest(HttpServletRequest request)
```

Construct a safe request that overrides the default request methods with safer versions.

Method Detail

getAttribute

```
public Object getAttribute(String name)
```

Same as HttpServletRequest, no security changes required.

Specified by:

`getAttribute` in interface `ServletRequest`

---

getAttributeNames

```
public Enumeration getAttributeNames()
```

Same as HttpServletRequest, no security changes required.

Specified by:

`getAttributeNames` in interface `ServletRequest`

---

getAuthType

```
public String getAuthType()
```

Same as HttpServletRequest, no security changes required.

Specified by:

`getAuthType` in interface `HttpServletRequest`

---

getCharacterEncoding

```
public String getCharacterEncoding()
```

Same as HttpServletRequest, no security changes required.

Specified by:

`getCharacterEncoding` in interface `ServletRequest`

---

getContentLength

```
public int getContentLength()
```

Same as HttpServletRequest, no security changes required.

Specified by:

`getContentLength` in interface `ServletRequest`

---

getContentType

```
public String getContentType()
```

Same as HttpServletRequest, no security changes required.

Specified by:

`getContentType` in interface `ServletRequest`

---

getContextPath

```
public String getContextPath()
```

Returns the context path from the HttpServletRequest after canonicalizing and filtering out any dangerous characters.

Specified by:

`getContextPath` in interface `HttpServletRequest`

---

getCookies

```
public Cookie[] getCookies()
```

Returns the array of Cookies from the HttpServletRequest after canonicalizing and filtering out any dangerous characters.

Specified by:

`getCookies` in interface `HttpServletRequest`

---

getDateHeader

```
public long getDateHeader(String name)
```

Same as HttpServletRequest, no security changes required.

Specified by:

`getDateHeader` in interface `HttpServletRequest`

---

getHeader

```
public String getHeader(String name)
```

Returns the named header from the HttpServletRequest after canonicalizing and filtering out any dangerous characters.

Specified by:

`getHeader` in interface `HttpServletRequest`

---

getHeaderNames

```
public Enumeration getHeaderNames()
```

Returns the enumeration of header names from the HttpServletRequest after canonicalizing and filtering out any dangerous characters.

Specified by:

`getHeaderNames` in interface `HttpServletRequest`

---

getHeaders

```
public Enumeration getHeaders(String name)
```

Returns the enumeration of headers from the HttpServletRequest after canonicalizing and filtering out any dangerous characters.

Specified by:

`getHeaders` in interface `HttpServletRequest`

---

getInputStream

```
public ServletInputStream getInputStream()
                                  throws IOException
```

Same as HttpServletRequest, no security changes required. Note that this input stream may contain attacks and the developer is responsible for canonicalizing, validating, and encoding any data from this stream.

Specified by:

`getInputStream` in interface `ServletRequest`

Throws:

`IOException`

---

getIntHeader

```
public int getIntHeader(String name)
```

Same as HttpServletRequest, no security changes required.

Specified by:

`getIntHeader` in interface `HttpServletRequest`

---

getLocalAddr

```
public String getLocalAddr()
```

Same as HttpServletRequest, no security changes required.

Specified by:

`getLocalAddr` in interface `ServletRequest`

---

getLocale

```
public Locale getLocale()
```

Same as HttpServletRequest, no security changes required.

Specified by:

`getLocale` in interface `ServletRequest`

---

getLocales

```
public Enumeration getLocales()
```

Same as HttpServletRequest, no security changes required.

Specified by:

`getLocales` in interface `ServletRequest`

---

getLocalName

```
public String getLocalName()
```

Same as HttpServletRequest, no security changes required.

Specified by:

`getLocalName` in interface `ServletRequest`

---

getLocalPort

```
public int getLocalPort()
```

Same as HttpServletRequest, no security changes required.

Specified by:

`getLocalPort` in interface `ServletRequest`

---

getMethod

```
public String getMethod()
```

Same as HttpServletRequest, no security changes required.

Specified by:

`getMethod` in interface `HttpServletRequest`

---

getParameter

```
public String getParameter(String name)
```

Returns the named parameter from the HttpServletRequest after canonicalizing and filtering out any dangerous characters.

Specified by:

`getParameter` in interface `ServletRequest`

---

getParameterMap

```
public Map getParameterMap()
```

Returns the parameter map from the HttpServletRequest after canonicalizing and filtering out any dangerous characters.

Specified by:

`getParameterMap` in interface `ServletRequest`

---

getParameterNames

```
public Enumeration getParameterNames()
```

Returns the enumeration of parameter names from the HttpServletRequest after canonicalizing and filtering out any dangerous characters.

Specified by:

`getParameterNames` in interface `ServletRequest`

---

getParameterValues

```
public String[] getParameterValues(String name)
```

Returns the array of matching parameter values from the HttpServletRequest after canonicalizing and filtering out any dangerous characters.

Specified by:

`getParameterValues` in interface `ServletRequest`

---

getPathInfo

```
public String getPathInfo()
```

Returns the path info from the HttpServletRequest after canonicalizing and filtering out any dangerous characters.

Specified by:

`getPathInfo` in interface `HttpServletRequest`

---

getPathTranslated

```
public String getPathTranslated()
```

Same as HttpServletRequest, no security changes required.

Specified by:

`getPathTranslated` in interface `HttpServletRequest`

---

getProtocol

```
public String getProtocol()
```

Same as HttpServletRequest, no security changes required.

Specified by:

`getProtocol` in interface `ServletRequest`

---

getQueryString

```
public String getQueryString()
```

Returns the query string from the HttpServletRequest after canonicalizing and filtering out any dangerous characters.

Specified by:

`getQueryString` in interface `HttpServletRequest`

---

getReader

```
public BufferedReader getReader()
                        throws IOException
```

Same as HttpServletRequest, no security changes required. Note that this reader may contain attacks and the developer is responsible for canonicalizing, validating, and encoding any data from this stream.

Specified by:

`getReader` in interface `ServletRequest`

Throws:

`IOException`

---

getRealPath

```
public String getRealPath(String path)
```

Same as HttpServletRequest, no security changes required.

Specified by:

`getRealPath` in interface `ServletRequest`

---

getRemoteAddr

```
public String getRemoteAddr()
```

Same as HttpServletRequest, no security changes required.

Specified by:

`getRemoteAddr` in interface `ServletRequest`

---

getRemoteHost

```
public String getRemoteHost()
```

Same as HttpServletRequest, no security changes required.

Specified by:

`getRemoteHost` in interface `ServletRequest`

---

getRemotePort

```
public int getRemotePort()
```

Same as HttpServletRequest, no security changes required.

Specified by:

`getRemotePort` in interface `ServletRequest`

getRemoteUser

```
public String getRemoteUser()
```

Returns the name of the ESAPI user associated with this request.

Specified by:

`getRemoteUser` in interface `HttpServletRequest`

getRequestDispatcher

```
public RequestDispatcher getRequestDispatcher(String path)
```

Checks to make sure the path to forward to is within the WEB-INF directory and then returns the dispatcher. Otherwise returns null.

Specified by:

`getRequestDispatcher` in interface `ServletRequest`

getRequestedSessionId

```
public String getRequestedSessionId()
```

Returns the URI from the HttpServletRequest after canonicalizing and filtering out any dangerous characters. Code must be very careful not to depend on the value of a requested session id reported by the user.

Specified by:

`getRequestedSessionId` in interface `HttpServletRequest`

getRequestURI

```
public String getRequestURI()
```

Returns the URI from the HttpServletRequest after canonicalizing and filtering out any dangerous characters.

Specified by:

`getRequestURI` in interface `HttpServletRequest`

---

getRequestURL

```
public StringBuffer getRequestURL()
```

Returns the URL from the HttpServletRequest after canonicalizing and filtering out any dangerous characters.

Specified by:

`getRequestURL` in interface `HttpServletRequest`

---

getScheme

```
public String getScheme()
```

Returns the scheme from the HttpServletRequest after canonicalizing and filtering out any dangerous characters.

Specified by:

`getScheme` in interface `ServletRequest`

---

getServerName

```
public String getServerName()
```

Returns the server name (host header) from the HttpServletRequest after canonicalizing and filtering out any dangerous characters.

Specified by:

`getServerName` in interface `ServletRequest`

getServerPort

```
public int getServerPort()
```

Returns the server port (after the : in the host header) from the HttpServletRequest after parsing and checking the range 0-65536.

Specified by:

`getServerPort` in interface `ServletRequest`

---

getServletPath

```
public String getServletPath()
```

Returns the server path from the HttpServletRequest after canonicalizing and filtering out any dangerous characters.

Specified by:

`getServletPath` in interface `HttpServletRequest`

---

getSession

```
public HttpSession getSession()
```

Returns a session, creating it if necessary, and sets the HttpOnly flag on the JSESSIONID cookie.

Specified by:

`getSession` in interface `HttpServletRequest`

---

getSession

```
public HttpSession getSession(boolean create)
```

Returns a session, creating it if necessary, and sets the HttpOnly flag on the JSESSIONID cookie.

Specified by:

`getSession` in interface `HttpServletRequest`

---

getUserPrincipal

`public Principal` **`getUserPrincipal`**`()`

Returns the ESAPI User associated with this request.

Specified by:

`getUserPrincipal` in interface `HttpServletRequest`

---

isRequestedSessionIdFromCookie

`public boolean` **`isRequestedSessionIdFromCookie`**`()`

Same as HttpServletRequest, no security changes required.

Specified by:

`isRequestedSessionIdFromCookie` in interface `HttpServletRequest`

---

isRequestedSessionIdFromUrl

`public boolean` **`isRequestedSessionIdFromUrl`**`()`

Same as HttpServletRequest, no security changes required.

Specified by:

`isRequestedSessionIdFromUrl` in interface `HttpServletRequest`

---

isRequestedSessionIdFromURL

`public boolean` **`isRequestedSessionIdFromURL`**`()`

Same as HttpServletRequest, no security changes required.

Specified by:

`isRequestedSessionIdFromURL` in interface `HttpServletRequest`

---

isRequestedSessionIdValid

```
public boolean isRequestedSessionIdValid()
```

Same as HttpServletRequest, no security changes required.

Specified by:

`isRequestedSessionIdValid` in interface `HttpServletRequest`

---

isSecure

```
public boolean isSecure()
```

Same as HttpServletRequest, no security changes required.

Specified by:

`isSecure` in interface `ServletRequest`

---

isUserInRole

```
public boolean isUserInRole(String role)
```

Returns true if the ESAPI User associated with this request has the specified role.

Specified by:

`isUserInRole` in interface `HttpServletRequest`

---

removeAttribute

```
public void removeAttribute(String name)
```

Same as HttpServletRequest, no security changes required.

Specified by:

`removeAttribute` in interface `ServletRequest`

---

setAttribute

```
public void setAttribute(String name,
                         Object o)
```

Same as HttpServletRequest, no security changes required.

Specified by:

`setAttribute` in interface `ServletRequest`

---

setCharacterEncoding

```
public void setCharacterEncoding(String enc)
                          throws UnsupportedEncodingException
```

Sets the character encoding to the ESAPI configured encoding.

Specified by:

`setCharacterEncoding` in interface `ServletRequest`

Throws:

`UnsupportedEncodingException`

Class SafeResponse

org.owasp.esapi.filters

java.lang.Object

└ **org.owasp.esapi.filters.SafeResponse**

All Implemented Interfaces:

HttpServletResponse, ServletResponse

---

public class **SafeResponse**

extends Object

implements HttpServletResponse

This response wrapper simply overrides unsafe methods in the HttpServletResponse API with safe versions.

Constructor Summary
SafeResponse(HttpServletResponse response) Construct a safe response that overrides the default response methods with safer versions.
SafeResponse(HttpServletResponse response, String mode)

Method Summary	
void	**addCookie**(Cookie cookie) Add a cookie to the response after ensuring that there are no encoded or illegal characters in the name and name and value.
void	**addDateHeader**(String name, long date) Add a cookie to the response after ensuring that there are no encoded or illegal characters in the name.

void	**addHeader**(String name, String value) Add a header to the response after ensuring that there are no encoded or illegal characters in the name and name and value.
void	**addIntHeader**(String name, int value) Add an int header to the response after ensuring that there are no encoded or illegal characters in the name and name.
boolean	**containsHeader**(String name) Same as HttpServletResponse, no security changes required.
String	**encodeRedirectUrl**(String url) Return the URL without any changes, to prevent disclosure of the JSESSIONID.
String	**encodeRedirectURL**(String url) Return the URL without any changes, to prevent disclosure of the JSESSIONID The default implementation of this method can add the JSESSIONID to the URL if support for cookies is not detected.
String	**encodeUrl**(String url) Return the URL without any changes, to prevent disclosure of the JSESSIONID The default implementation of this method can add the JSESSIONID to the URL if support for cookies is not detected.
String	**encodeURL**(String url) Return the URL without any changes, to prevent disclosure of the JSESSIONID The default implementation of this method can add the JSESSIONID to the URL if support for cookies is not detected.
void	**flushBuffer**() Same as HttpServletResponse, no security changes required.
int	**getBufferSize**() Same as HttpServletResponse, no security changes required.
String	**getCharacterEncoding**() Same as HttpServletResponse, no security changes required.

String	**getContentType**() Same as HttpServletResponse, no security changes required.
Locale	**getLocale**() Same as HttpServletResponse, no security changes required.
ServletOutputStream	**getOutputStream**() Same as HttpServletResponse, no security changes required.
PrintWriter	**getWriter**() Same as HttpServletResponse, no security changes required.
boolean	**isCommitted**() Same as HttpServletResponse, no security changes required.
void	**reset**() Same as HttpServletResponse, no security changes required.
void	**resetBuffer**() Same as HttpServletResponse, no security changes required.
void	**sendError**(int sc) Override the error code with a 200 in order to confound attackers using automated scanners.
void	**sendError**(int sc, String msg) Override the error code with a 200 in order to confound attackers using automated scanners.
void	**sendRedirect**(String location) This method generates a redirect response that can only be used to redirect the browser to safe locations, as configured in the ESAPI security configuration.
void	**setBufferSize**(int size) Same as HttpServletResponse, no security changes required.
void	**setCharacterEncoding**(String charset) Sets the character encoding to the ESAPI configured encoding.

void	**setContentLength**(int len) Same as HttpServletResponse, no security changes required.
void	**setContentType**(String type) Same as HttpServletResponse, no security changes required.
void	**setDateHeader**(String name, long date) Add a date header to the response after ensuring that there are no encoded or illegal characters in the name.
void	**setHeader**(String name, String value) Add a header to the response after ensuring that there are no encoded or illegal characters in the name and value.
void	**setIntHeader**(String name, int value) Add an int header to the response after ensuring that there are no encoded or illegal characters in the name.
void	**setLocale**(Locale loc) Same as HttpServletResponse, no security changes required.
void	**setStatus**(int sc) Override the status code with a 200 in order to confound attackers using automated scanners.
void	**setStatus**(int sc, String sm) Override the status code with a 200 in order to confound attackers using automated scanners.

Constructor Detail

SafeResponse

```
public SafeResponse(HttpServletResponse response)
```

Construct a safe response that overrides the default response methods with safer versions.

SafeResponse

```
public SafeResponse(HttpServletResponse response,
                    String mode)
```

Method Detail

addCookie

```
public void addCookie(Cookie cookie)
```

Add a cookie to the response after ensuring that there are no encoded or illegal characters in the name and name and value. This method also sets the secure and HttpOnly flags on the cookie. This implementation uses a custom "set-cookie" header instead of using Java's cookie interface which doesn't allow the use of HttpOnly.

Specified by:

`addCookie` in interface `HttpServletResponse`

---

addDateHeader

```
public void addDateHeader(String name,
                          long date)
```

Add a cookie to the response after ensuring that there are no encoded or illegal characters in the name.

Specified by:

`addDateHeader` in interface `HttpServletResponse`

---

addHeader

```
public void addHeader(String name,
                      String value)
```

Add a header to the response after ensuring that there are no encoded or illegal characters in the name and name and value. This implementation follows the following recommendation: "A recipient MAY replace any linear white space with a single SP before interpreting the field value or forwarding the message downstream." http://www.w3.org/Protocols/rfc2616/rfc2616-sec2.html#sec2.2

Specified by:

`addHeader` in interface `HttpServletResponse`

---

addIntHeader

```
public void addIntHeader(String name,
                         int value)
```

Add an int header to the response after ensuring that there are no encoded or illegal characters in the name and name.

Specified by:

`addIntHeader` in interface `HttpServletResponse`

---

containsHeader

```
public boolean containsHeader(String name)
```

Same as HttpServletResponse, no security changes required.

Specified by:

`containsHeader` in interface `HttpServletResponse`

---

encodeRedirectUrl

```
public String encodeRedirectUrl(String url)
```

Return the URL without any changes, to prevent disclosure of the JSESSIONID. The default implementation of this method can add the JSESSIONID to the URL if support for cookies is not detected. This exposes the JSESSIONID credential in bookmarks, referer headers, server logs, and more.

Specified by:

`encodeRedirectUrl` in interface `HttpServletResponse`

Returns:

original url

---

encodeRedirectURL

```
public String encodeRedirectURL(String url)
```

Return the URL without any changes, to prevent disclosure of the JSESSIONID The default implementation of this method can add the JSESSIONID to the URL if support for cookies is not detected. This exposes the JSESSIONID credential in bookmarks, referer headers, server logs, and more.

Specified by:

`encodeRedirectURL` in interface `HttpServletResponse`

Returns:

original url

---

encodeUrl

```
public String encodeUrl(String url)
```

Return the URL without any changes, to prevent disclosure of the JSESSIONID The default implementation of this method can add the JSESSIONID to the URL if support for cookies is not detected. This exposes the JSESSIONID credential in bookmarks, referer headers, server logs, and more.

Specified by:

`encodeUrl` in interface `HttpServletResponse`

Returns:

original url

---

encodeURL

```
public String encodeURL(String url)
```

Return the URL without any changes, to prevent disclosure of the JSESSIONID The default implementation of this method can add the JSESSIONID to the URL if support for cookies is not detected. This exposes the JSESSIONID credential in bookmarks, referer headers, server logs, and more.

Specified by:

`encodeURL` in interface `HttpServletResponse`

Returns:

original url

---

flushBuffer

```
public void flushBuffer()
                throws IOException
```

Same as HttpServletResponse, no security changes required.

Specified by:

`flushBuffer` in interface `ServletResponse`

Throws:

`IOException`

---

getBufferSize

```
public int getBufferSize()
```

Same as HttpServletResponse, no security changes required.

Specified by:

`getBufferSize` in interface `ServletResponse`

getCharacterEncoding

```
public String getCharacterEncoding()
```

Same as HttpServletResponse, no security changes required.

Specified by:

`getCharacterEncoding` in interface `ServletResponse`

getContentType

```
public String getContentType()
```

Same as HttpServletResponse, no security changes required.

Specified by:

`getContentType` in interface `ServletResponse`

getLocale

```
public Locale getLocale()
```

Same as HttpServletResponse, no security changes required.

Specified by:

`getLocale` in interface `ServletResponse`

getOutputStream

```
public ServletOutputStream getOutputStream()
                                    throws IOException
```

Same as HttpServletResponse, no security changes required.

Specified by:

`getOutputStream` in interface `ServletResponse`

Throws:

`IOException`

---

getWriter

```
public PrintWriter getWriter()
                     throws IOException
```

Same as HttpServletResponse, no security changes required.

Specified by:

`getWriter` in interface `ServletResponse`

Throws:

`IOException`

---

isCommitted

```
public boolean isCommitted()
```

Same as HttpServletResponse, no security changes required.

Specified by:

`isCommitted` in interface `ServletResponse`

---

reset

```
public void reset()
```

Same as HttpServletResponse, no security changes required.

Specified by:

`reset` in interface `ServletResponse`

---

resetBuffer

```
public void resetBuffer()
```

Same as HttpServletResponse, no security changes required.

Specified by:

`resetBuffer` in interface `ServletResponse`

---

sendError

```
public void sendError(int sc)
               throws IOException
```

Override the error code with a 200 in order to confound attackers using automated scanners.

Specified by:

`sendError` in interface `HttpServletResponse`

Throws:

`IOException`

---

sendError

```
public void sendError(int sc,
                      String msg)
               throws IOException
```

Override the error code with a 200 in order to confound attackers using automated scanners. The message is canonicalized and filtered for dangerous characters.

Specified by:

`sendError` in interface `HttpServletResponse`

Throws:

`IOException`

---

sendRedirect

```
public void sendRedirect(String location)
                  throws IOException
```

This method generates a redirect response that can only be used to redirect the browser to safe locations, as configured in the ESAPI security configuration. This method does not that redirect requests can be modified by attackers, so do not rely information contained within redirect requests, and do not include sensitive information in a redirect.

Specified by:

`sendRedirect` in interface `HttpServletResponse`

Throws:

`IOException`

---

setBufferSize

```
public void setBufferSize(int size)
```

Same as HttpServletResponse, no security changes required.

Specified by:

`setBufferSize` in interface `ServletResponse`

---

setCharacterEncoding

```
public void setCharacterEncoding(String charset)
```

Sets the character encoding to the ESAPI configured encoding.

Specified by:

`setCharacterEncoding` in interface `ServletResponse`

---

setContentLength

```
public void setContentLength(int len)
```

Same as HttpServletResponse, no security changes required.

Specified by:

`setContentLength` in interface `ServletResponse`

---

setContentType

```
public void setContentType(String type)
```

Same as HttpServletResponse, no security changes required.

Specified by:

`setContentType` in interface `ServletResponse`

---

setDateHeader

```
public void setDateHeader(String name,
                          long date)
```

Add a date header to the response after ensuring that there are no encoded or illegal characters in the name.

Specified by:

`setDateHeader` in interface `HttpServletResponse`

---

setHeader

```
public void setHeader(String name,
                      String value)
```

Add a header to the response after ensuring that there are no encoded or illegal characters in the name and value. "A recipient MAY replace any linear white space with a single SP before interpreting the field value or forwarding the message downstream." http://www.w3.org/Protocols/rfc2616/rfc2616-sec2.html#sec2.2

Specified by:

`setHeader` in interface `HttpServletResponse`

---

setIntHeader

```
public void setIntHeader(String name,
                         int value)
```

Add an int header to the response after ensuring that there are no encoded or illegal characters in the name.

Specified by:

`setIntHeader` in interface `HttpServletResponse`

---

setLocale

```
public void setLocale(Locale loc)
```

Same as HttpServletResponse, no security changes required.

Specified by:

`setLocale` in interface `ServletResponse`

---

setStatus

```
public void setStatus(int sc)
```

Override the status code with a 200 in order to confound attackers using automated scanners.

Specified by:

`setStatus` in interface `HttpServletResponse`

---

setStatus

```
public void setStatus(int sc,
                      String sm)
```

Override the status code with a 200 in order to confound attackers using automated scanners. The message is canonicalized and filtered for dangerous characters.

Specified by:

`setStatus` in interface `HttpServletResponse`

Interface HTTPUtilities

org.owasp.esapi

All Known Implementing Classes:

DefaultHTTPUtilities

`public interface` **`HTTPUtilities`**

The HTTPUtilities interface is a collection of methods that provide additional security related to HTTP requests, responses, sessions, cookies, headers, and logging.

Field Summary	
String	**REMEMBER_TOKEN_COOKIE_NAME** Key for remember token cookie

Method Summary	
String	**addCSRFToken**(String href) Adds the current user's CSRF token (see User.getCSRFToken()) to the URL for purposes of preventing CSRF attacks.
void	**assertSecureRequest**(HttpServletRequest request) Ensures that the current request uses SSL and POST to protect any sensitive parameters in the querystring from being sniffed or logged.
HttpSession	**changeSessionIdentifier**(HttpServletRequest request) Invalidate the old session after copying all of its contents to a newly created session with a new session id.
String	**decryptHiddenField**(String encrypted) Decrypts an encrypted hidden field value and returns the cleartext.
Map	**decryptQueryString**(String encrypted) Takes an encrypted querystring and returns a Map containing the original parameters.

Map	**decryptStateFromCookie**(HttpServletRequest request) Retrieves a map of data from a cookie encrypted with encryptStateInCookie().
String	**encryptHiddenField**(String value) Encrypts a hidden field value for use in HTML.
String	**encryptQueryString**(String query) Takes a querystring (everything after the question mark in the URL) and returns an encrypted string containing the parameters.
void	**encryptStateInCookie**(HttpServletResponse response, Map cleartext) Stores a Map of data in an encrypted cookie.
Cookie	**getCookie**(HttpServletRequest request, String name) Get the first cookie with the matching name.
String	**getCSRFToken**() Returns the current user's CSRF token.
SafeRequest	**getCurrentRequest**() Retrieves the current HttpServletRequest
SafeResponse	**getCurrentResponse**() Retrieves the current HttpServletResponse
List	**getSafeFileUploads**(HttpServletRequest request, File tempDir, File finalDir) Extract uploaded files from a multipart HTTP requests.
void	**killAllCookies**(HttpServletRequest request, HttpServletResponse response) Kill all cookies received in the last request from the browser.
void	**killCookie**(HttpServletRequest request, HttpServletResponse response, String name) Kills the specified cookie by setting a new cookie that expires immediately.
void	**logHTTPRequest**(HttpServletRequest request, Logger logger) Format the Source IP address, URL, URL parameters, and all form parameters into a string suitable for the log file.

void	**logHTTPRequest**(HttpServletRequest request, Logger logger, List parameterNamesToObfuscate) Format the Source IP address, URL, URL parameters, and all form parameters into a string suitable for the log file.
void	**safeSendForward**(HttpServletRequest request, HttpServletResponse response, String context, String location) This method perform a forward to any resource located inside the WEB-INF directory.
void	**safeSetContentType**(HttpServletResponse response) Sets the content type on each HTTP response, to help protect against cross-site scripting attacks and other types of injection into HTML documents.
void	**setCurrentHTTP**(HttpServletRequest request, HttpServletResponse response) Stores the current HttpRequest and HttpResponse so that they may be readily accessed throughout ESAPI (and elsewhere)
void	**setNoCacheHeaders**(HttpServletResponse response) Set headers to protect sensitive information against being cached in the browser.
String	**setRememberToken**(HttpServletRequest request, HttpServletResponse response, String password, int maxAge, String domain, String path) Set a cookie containing the current User's remember me token for automatic authentication.
void	**verifyCSRFToken**(HttpServletRequest request) Checks the CSRF token in the URL (see User.getCSRFToken()) against the user's CSRF token and throws an IntrusionException if it is missing.

Field Detail

REMEMBER_TOKEN_COOKIE_NAME

```
public static final String REMEMBER_TOKEN_COOKIE_NAME
```

Key for remember token cookie

Method Detail

assertSecureRequest

```
void assertSecureRequest(HttpServletRequest request)
                  throws AccessControlException
```

Ensures that the current request uses SSL and POST to protect any sensitive parameters in the querystring from being sniffed or logged. For example, this method should be called from any method that uses sensitive data from a web form. This method uses `getCurrentRequest()` to obtain the current `HttpServletRequest` object

Throws:

`AccessControlException` - if security constraints are not met

---

addCSRFToken

```
String addCSRFToken(String href)
```

Adds the current user's CSRF token (see User.getCSRFToken()) to the URL for purposes of preventing CSRF attacks. This method should be used on all URLs to be put into all links and forms the application generates.

Parameters:

`href` - the URL to which the CSRF token will be appended

Returns:

the updated URL with the CSRF token parameter added

---

getCookie

```
Cookie getCookie(HttpServletRequest request,
                 String name)
```

Get the first cookie with the matching name.

Returns:

the requested cookie

---

getCSRFToken

```
String getCSRFToken()
```

Returns the current user's CSRF token. If there is no current user then return null.

Returns:

the current users CSRF token

---

changeSessionIdentifier

```
HttpSession changeSessionIdentifier(HttpServletRequest request)
                            throws AuthenticationException
```

Invalidate the old session after copying all of its contents to a newly created session with a new session id. Note that this is different from logging out and creating a new session identifier that does not contain the existing session contents. Care should be taken to use this only when the existing session does not contain hazardous contents. This method uses `getCurrentRequest()` to obtain the current `HttpSession` object

Returns:

the new HttpSession with a changed id

Throws:

`AuthenticationException`

`EnterpriseSecurityException` - the enterprise security exception

---

verifyCSRFToken

```
void verifyCSRFToken(HttpServletRequest request)
             throws IntrusionException
```

Checks the CSRF token in the URL (see User.getCSRFToken()) against the user's CSRF token and throws an IntrusionException if it is missing.

Throws:

`IntrusionException` - if CSRF token is missing or incorrect

---

decryptHiddenField

```
String decryptHiddenField(String encrypted)
```

Decrypts an encrypted hidden field value and returns the cleartext. If the field does not decrypt properly, an IntrusionException is thrown to indicate tampering.

Parameters:

`encrypted` - hidden field value to decrypt

Returns:

decrypted hidden field value stored as a String

---

setRememberToken

```
String setRememberToken(HttpServletRequest request,
                        HttpServletResponse response,
                        String password,
                        int maxAge,
                        String domain,
                        String path)
```

Set a cookie containing the current User's remember me token for automatic authentication. The use of remember me tokens is generally not recommended, but this method will help do it as safely as possible. The user interface should strongly warn the user that this should only be enabled on computers where no other users will have access. The username can be retrieved with: User username = ESAPI.authenticator().getCurrentUser();

Parameters:

`password` - the user's password

`maxAge` - the length of time that the token should be valid for in relative seconds

`domain` - the domain to restrict the token to or null

`path` - the path to restrict the token to or null

Returns:

encrypted "Remember Me" token stored as a String

---

encryptHiddenField

```
String encryptHiddenField(String value)
                  throws EncryptionException
```

Encrypts a hidden field value for use in HTML.

Parameters:

`value` - the cleartext value of the hidden field

Returns:

the encrypted value of the hidden field

Throws:

`EncryptionException`

---

encryptQueryString

```
String encryptQueryString(String query)
                  throws EncryptionException
```

Takes a querystring (everything after the question mark in the URL) and returns an encrypted string containing the parameters.

Parameters:

`query` - the querystring to encrypt

Returns:

encrypted querystring stored as a String

Throws:

EncryptionException

---

decryptQueryString

```
Map decryptQueryString(String encrypted)
                throws EncryptionException
```

Takes an encrypted querystring and returns a Map containing the original parameters.

Parameters:

`encrypted` - the encrypted querystring to decrypt

Returns:

a Map object containing the decrypted querystring

Throws:

EncryptionException

---

getSafeFileUploads

```
List getSafeFileUploads(HttpServletRequest request,
                        File tempDir,
                        File finalDir)
                 throws ValidationException
```

Extract uploaded files from a multipart HTTP requests. Implementations must check the content to ensure that it is safe before making a permanent copy on the local filesystem. Checks should include length and content checks, possibly virus checking, and path and name checks. Refer to the file checking methods in Validator for more information. This method uses `getCurrentRequest()` to obtain the `HttpServletRequest` object

Parameters:

`tempDir` - the temporary directory

`finalDir` - the final directory

Returns:

List of new File objects from upload

Throws:

`ValidationException` - if the file fails validation

---

decryptStateFromCookie

```
Map decryptStateFromCookie(HttpServletRequest request)
                    throws EncryptionException
```

Retrieves a map of data from a cookie encrypted with encryptStateInCookie().

Returns:

a map containing the decrypted cookie state value

Throws:

`EncryptionException`

---

killAllCookies

```
void killAllCookies(HttpServletRequest request,
                    HttpServletResponse response)
```

Kill all cookies received in the last request from the browser. Note that new cookies set by the application in this response may not be killed by this method.

---

killCookie

```
void killCookie(HttpServletRequest request,
                HttpServletResponse response,
                String name)
```

Kills the specified cookie by setting a new cookie that expires immediately. Note that this method does not delete new cookies that are being set by the application for this response.

---

encryptStateInCookie

```
void encryptStateInCookie(HttpServletResponse response,
                          Map cleartext)
                   throws EncryptionException
```

Stores a Map of data in an encrypted cookie. Generally the session is a better place to store state information, as it does not expose it to the user at all. If there is a requirement not to use sessions, or the data should be stored across sessions (for a long time), the use of encrypted cookies is an effective way to prevent the exposure.

Throws:

EncryptionException

---

safeSendForward

```
void safeSendForward(HttpServletRequest request,
                     HttpServletResponse response,
                     String context,
                     String location)
              throws AccessControlException,
                     ServletException,
                     IOException
```

This method perform a forward to any resource located inside the WEB-INF directory. Forwarding to publicly accessible resources can be dangerous, as the request will have already passed the URL based access control check. This method ensures that you can only forward to non-publicly accessible resources.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`location` - the URL to forward to

Throws:

AccessControlException

ServletException

IOException

---

safeSetContentType

```
void safeSetContentType(HttpServletResponse response)
```

Sets the content type on each HTTP response, to help protect against cross-site scripting attacks and other types of injection into HTML documents.

---

setNoCacheHeaders

```
void setNoCacheHeaders(HttpServletResponse response)
```

Set headers to protect sensitive information against being cached in the browser. Developers should make this call for any HTTP responses that contain any sensitive data that should not be cached within the browser or any intermediate proxies or caches. Implementations should set headers for the expected browsers. The safest approach is to set all relevant headers to their most restrictive setting. These include:

```
Cache-Control: no-store

Cache-Control: no-cache

Cache-Control: must-revalidate

Expires: -1
```

Note that the header "pragma: no-cache" is only useful in HTTP requests, not HTTP responses. So even though there are many articles recommending the use of this header, it is not helpful for preventing browser caching. For more information, please refer to the relevant standards:

- HTTP/1.1 Cache-Control "no-cache"
- HTTP/1.1 Cache-Control "no-store"
- HTTP/1.0 Pragma "no-cache"

- HTTP/1.0 Expires
- IE6 Caching Issues
- Firefox browser.cache.disk_cache_ssl
- Mozilla

This method uses `getCurrentResponse()` to obtain the `HttpServletResponse` object

---

setCurrentHTTP

```
void setCurrentHTTP(HttpServletRequest request,
                    HttpServletResponse response)
```

Stores the current HttpRequest and HttpResponse so that they may be readily accessed throughout ESAPI (and elsewhere)

Parameters:

`request` - the current request

`response` - the current response

---

getCurrentRequest

```
SafeRequest getCurrentRequest()
```

Retrieves the current HttpServletRequest

Returns:

the current request

---

getCurrentResponse

```
SafeResponse getCurrentResponse()
```

Retrieves the current HttpServletResponse

Returns:

the current response

---

logHTTPRequest

```
void logHTTPRequest(HttpServletRequest request,
                    Logger logger)
```

Format the Source IP address, URL, URL parameters, and all form parameters into a string suitable for the log file. Be careful not to log sensitive information, and consider masking with the logHTTPRequest(List parameterNamesToObfuscate) method.

Parameters:

`logger` - the logger to write the request to

---

logHTTPRequest

```
void logHTTPRequest(HttpServletRequest request,
                    Logger logger,
                    List parameterNamesToObfuscate)
```

Format the Source IP address, URL, URL parameters, and all form parameters into a string suitable for the log file. The list of parameters to obfuscate should be specified in order to prevent sensitive information from being logged. If a null list is provided, then all parameters will be logged. If HTTP request logging is done in a central place, the parameterNamesToObfuscate could be made a configuration parameter. We include it here in case different parts of the application need to obfuscate different parameters. This method uses `getCurrentResponse()` to obtain the `HttpServletResponse` object

Parameters:

`logger` - the logger to write the request to

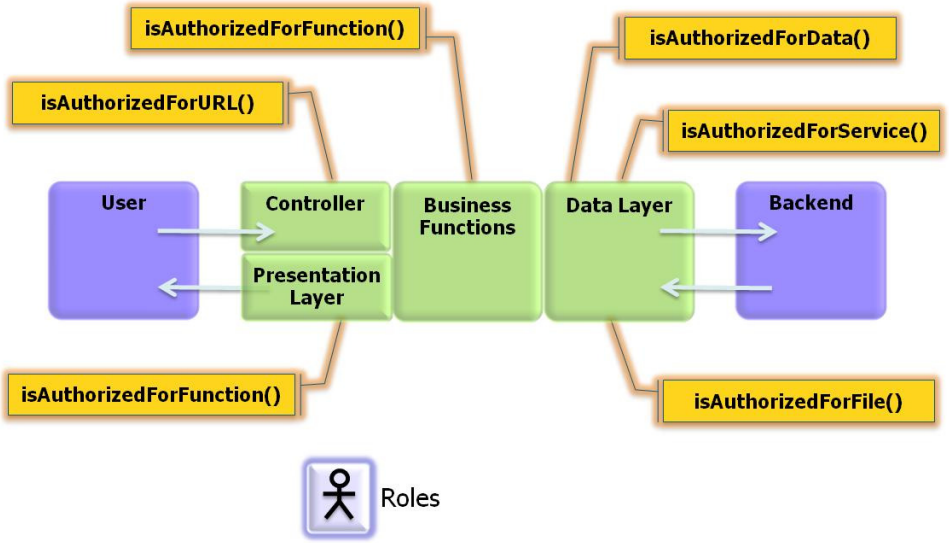
`parameterNamesToObfuscate` - the sensitive parameters

Positive Access Control

Access control is one of the most complex areas for developers to implement in their applications. The primary difficulty is that access control decisions must be made at many levels in an application. This is similar to the real world, where you might have a gate at the physical entrance to a facility, a guard at the building, locked doors within a facility, and locked desks within offices.

Design

Access control mechanisms can be difficult to implement. In ESAPI all the access control methods follow a simple pattern. Each level has two methods, assertIsAuthorizedForX() and isAuthorizedForX(). The boolean method simply delegates to the assert method.



Each of these methods uses a deny by default approach, that prevents access unless there is a specific rule authorizing it.

In most cases, failing an access control check represents an intrusion in progress and should be flagged as an attack. An appropriate response might be to log a special message, log out the current user, or disable the current user's account. However, in some cases, failing an access control check does not indicate an attack. For example, if a web page uses an access control check to determine whether or not to display links or buttons for administrators, the failure is expected for non-admin users.

Controlling access to URLs

Unless every URL in an application is public, then there is a need for performing URL based access control. A carefully constructed web application might be able to perform all of its access control by checking the URL. However, most applications need "multi-role" URLs, and therefore need other access control methods.

Many of the access control products on the market perform URL-based access control. If you use these products, you should be sure that the policy they enforce is easy to access and change. Knowing these rules is critical for developers to make the right choices implementing access control.

```
try {
    ESAPI.accessController().assertAuthorizedForURL( "/admin" ) {
     // allow access to the admin functions
 } catch (AccessControlException ace) {
     ... attack in progress
 }
```

Controlling access to functions

In many applications, business functions will be restricted by role or other access control decision. If possible, the best approach is to adopt a pattern of checking access inside every single business function.

```
try {
     ESAPI.accessController().assertAuthorizedForFunction( BUSINESS_FUNCTION );
     // execute BUSINESS_FUNCTION
 } catch (AccessControlException ace) {
     ... attack in progress
 }
```

Controlling access to services

In some cases, the access to backend services should only be allowed on behalf of certain users. In these cases, the assertAuthorizedForService() call can be used to verify access is authorized.

```
try {
     ESAPI.accessController().assertAuthorizedForService( “PaymentGateway” );
     // access service
 } catch (AccessControlException ace) {
     ... attack in progress
 }
```

Controlling access to files

Many applications access files from the local filesystem. In many cases, these files are only authorized for a particular user. ESAPI allows you to control access easily.

```
try {
     ESAPI.accessController().assertAuthorizedForService( “PaymentGateway” );
     // access service
 } catch (AccessControlException ace) {
     ... attack in progress
 }
```

Controlling access to data

Data is particularly critical for applications. TODO.

```
try {
     Report report = persistenceLayer.getReport( “test” );
     ESAPI.accessController().assertAuthorizedForData( “read”, report );
     // use the data
 } catch (AccessControlException ace) {
     ... attack in progress
 }
```

Controlling direct object references

Most applications use parameters or form fields that reference data on the server by its name or id. Attackers love to try to access unauthorized data by tampering with these “direct” references. For example, imagine a URL that references a file on the server:

```
http://www.example.com/banking/fetchReport?fn=03102008.xls
```

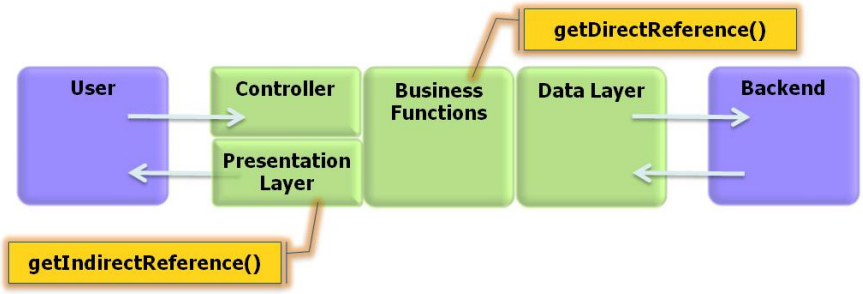
Attackers will immediately attempt to manipulate the filename to access other files on the host. If the code takes the “fn” parameter and appends it to a filepath, the attacker might try sending

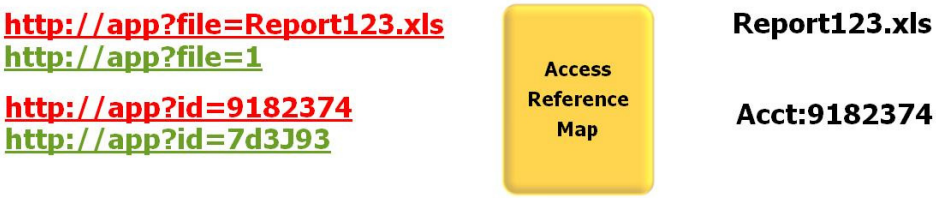
"?fn=../../../../etc/password%00" to gain access to the host's password file. The %00 at the end of the URL will be decoded by the container into a null byte that terminates the string, leaving only the attackers path in the parameter. There are hundreds of attack variants, so don't try to filter out attacks.

Database keys are also often exposed. Attackers will attempt to manipulate the "uid" parameter in the URL below to access other user accounts.

```
http://www.example.com/banking/genReport?uid=10347343
```

A simple approach to solving direct reference issues is replacing them with indirect references. You can use an "access reference map" to assign integers to the authorized resources, and then use the integers in the web page instead of the direct reference. When the indirect integer reference is returned, your code can look up the corresponding direct reference in the map. This prevents tampering with the parameter and significantly restricts the application's attack surface.





```
// Create a new list of files
Set fileSet = new HashSet();

// add direct references (e.g. File objects)
File file1 = new File( ... );
fileSet.add( file1 );
...

// create an access reference map
AccessReferenceMap map = new AccessReferenceMap( fileSet );

// store the map somewhere safe - like the session!
session.setAttribute( “filemap”, map );

// get the indirect reference for a file and add it to the map
String indRef = map.getIndirectReference( file1 );
String href = “http://www.example.com/esapi?file=” + indRef );
```

When the request is returned from the browser, look up the indirect reference and get the File object.

```
// if the indirect reference doesn't exist, it's very likely an attack
// getDirectReference throws an AccessControlException
// you should handle as appropriate

String indref = request.getParameter( “file” );
File file = (File)map.getDirectReference( indref );

// use the file object without fear of tampering
```

Presentation Layer

Many of the absolute worst application security holes are related to access control. The complexity is often through the roof. Frequently the schemes evolve over a long period of time. One extremely common problem is that an access control checks are made in the presentation layer, but the corresponding check is not made in the business logic.

For example, imagine that an online bank has special features that are only displayed to customers with over $250,000 in savings. The presentation layer has careful access control checks so that these functions are not exposed to less affluent users. When a rich customer selects one of these features, a request containing the parameter “function=special1” is generated that invokes a special business function. This parameter is not particularly secret and may be disclosed by authorized users. asdasdf

Interface AccessController

org.owasp.esapi

All Known Implementing Classes:

FileBasedAccessController

```
public interface AccessController
```

The AccessController interface defines a set of methods that can be used in a wide variety of applications to enforce access control. In most applications, access control must be performed in multiple different locations across the various application layers. This class provides access control for URLs, business functions, data, services, and files.

The implementation of this interface will need to access the current User object (from Authenticator.getCurrentUser()) to determine roles or permissions. In addition, the implementation will also need information about the resources that are being accessed. Using the user information and the resource information, the implementation should return an access control decision.

Implementers are encouraged to build on existing access control mechanisms, such as methods like isUserInRole() or hasPrivilege(). While powerful, these methods can be confusing, as users may be in multiple roles or possess multiple overlapping privileges. These methods encourage the use of complex boolean tests throughout the code. The point of this interface is to centralize access control logic so that it is easy to use and easy to verify.

```
try {
    ESAPI.accessController().assertAuthorizedForFunction( BUSINESS_FUNCTION );
    // execute BUSINESS_FUNCTION
} catch (AccessControlException ace) {
... attack in progress
}
```

Note that in the user interface layer, access control checks can be used to control whether particular controls are rendered or not. These checks are supposed to fail when an unauthorized user is logged in, and do not represent attacks. Remember that regardless of how the user interface appears, an attacker can attempt to invoke any business function or access any data in your application. Therefore, access control checks in the user interface should be repeated in both the business logic and data layers.

```
<% if ( ESAPI.accessController().isAuthorizedForFunction( ADMIN_FUNCTION ) ) { %>
<a href="/doAdminFunction">ADMIN</a>
<% } else { %>
<a href="/doNormalFunction">NORMAL</a>
<% } %>
```

Method Summary	
void	**assertAuthorizedForData**(String key) Checks if the current user is authorized to access the referenced data.
void	**assertAuthorizedForData**(String action, Object data)
void	**assertAuthorizedForFile**(String filepath) Checks if an account is authorized to access the referenced file.
void	**assertAuthorizedForFunction**(String functionName) Checks if an account is authorized to access the referenced function.
void	**assertAuthorizedForService**(String serviceName) Checks if an account is authorized to access the referenced service.
void	**assertAuthorizedForURL**(String url) Checks if an account is authorized to access the referenced URL.
boolean	**isAuthorizedForData**(String key) Checks if an account is authorized to access the referenced data.
boolean	**isAuthorizedForData**(String action, Object data)
boolean	**isAuthorizedForFile**(String filepath) Checks if an account is authorized to access the referenced file.
boolean	**isAuthorizedForFunction**(String functionName) Checks if an account is authorized to access the referenced function.
boolean	**isAuthorizedForService**(String serviceName) Checks if an account is authorized to access the referenced service.
boolean	**isAuthorizedForURL**(String url) Checks if an account is authorized to access the referenced URL.

Method Detail

isAuthorizedForURL

boolean **isAuthorizedForURL**(String url)

Checks if an account is authorized to access the referenced URL. Generally, this method should be invoked in the application's controller or a filter as follows:

```
ESAPI.accessController().isAuthorizedForURL(request.getRequestURI().toString())
;
```

The implementation of this method should call assertAuthorizedForURL(String url), and if an AccessControlException is not thrown, this method should return true.

Parameters:

`url` - the URL as returned by request.getRequestURI().toString()

Returns:

true, if is authorized for URL

---

isAuthorizedForFunction

```
boolean isAuthorizedForFunction(String functionName)
```

Checks if an account is authorized to access the referenced function. The implementation of this method should call assertAuthorizedForFunction(String functionName), and if an AccessControlException is not thrown, this method should return true.

Parameters:

`functionName` - the function name

Returns:

true, if is authorized for function

---

isAuthorizedForData

```
boolean isAuthorizedForData(String key)
```

Checks if an account is authorized to access the referenced data. The implementation of this method should call assertAuthorizedForData(String key), and if an AccessControlException is not thrown, this method should return true.

Parameters:

`key` - the key

Returns:

true, if is authorized for data

---

isAuthorizedForData

```
boolean isAuthorizedForData(String action,
                            Object data)
```

---

isAuthorizedForFile

```
boolean isAuthorizedForFile(String filepath)
```

Checks if an account is authorized to access the referenced file. The implementation of this method should call assertAuthorizedForFile(String filepath), and if an AccessControlException is not thrown, this method should return true.

Parameters:

`filepath` - the path of the file to be checked, including filename

Returns:

true, if is authorized for file

---

isAuthorizedForService

```
boolean isAuthorizedForService(String serviceName)
```

Checks if an account is authorized to access the referenced service. This can be used in applications that provide access to a variety of backend services. The implementation of this method should call assertAuthorizedForService(String serviceName), and if an AccessControlException is not thrown, this method should return true.

Parameters:

`serviceName` - the service name

Returns:

true, if is authorized for service

---

assertAuthorizedForURL

```
void assertAuthorizedForURL(String url)
                    throws AccessControlException
```

Checks if an account is authorized to access the referenced URL. The implementation should allow access to be granted to any part of the URL. Generally, this method should be invoked in the application's controller or a filter as follows:

```
ESAPI.accessController().assertAuthorizedForURL(request.getRequestURI().toStrin
g());
```

This method throws an AccessControlException if access is not authorized, or if the referenced URL does not exist. If the User is authorized, this method simply returns. Specification: The implementation should do the following: 1) Check to see if the resource exists and if not, throw an AccessControlException 2) Use available information to make an access control decision a. Ideally, this policy would be data driven b. You can use the current User, roles, data type, data name, time of day, etc. c. Access control decisions must deny by default 3) If access is not permitted, throw AccessControlException with details

Parameters:

`url` - the URL as returned by request.getRequestURI().toString()

Throws:

`AccessControlException` - if access is not permitted

---

assertAuthorizedForFunction

```
void assertAuthorizedForFunction(String functionName)
                        throws AccessControlException
```

Checks if an account is authorized to access the referenced function. The implementation should define the function "namespace" to be enforced. Choosing something simple like the class name of action classes or menu item names will make this implementation easier to use. This method throws an AccessControlException if access is not authorized, or if the referenced function does not exist. If the User is authorized, this method simply returns. Specification: The implementation should do the following: 1) Check to see if the function exists and if not, throw an AccessControlException 2) Use available information to make an access control decision a. Ideally, this policy would be data driven b. You can use the current User, roles, data type, data name, time of day, etc. c. Access control decisions must deny by default 3) If access is not permitted, throw AccessControlException with details

Parameters:

`functionName` - the function name

Throws:

`AccessControlException` - if access is not permitted

---

assertAuthorizedForData

```
void assertAuthorizedForData(String key)
                      throws AccessControlException
```

Checks if the current user is authorized to access the referenced data. This method simply returns if access is authorized. It throws an AccessControlException if access is not authorized, or if the referenced data does not exist. Specification: The implementation should do the following: 1) Check to see if the resource exists and if not, throw an AccessControlException 2) Use available information to make an access control decision a. Ideally, this policy would be data driven b. You can use the current User, roles, data type, data name, time of day, etc. c. Access control decisions must deny by default 3) If access is not permitted, throw AccessControlException with details

Parameters:

`key` - the name of the target data object

Throws:

`AccessControlException` - if access is not permitted

assertAuthorizedForData

```
void assertAuthorizedForData(String action,
                             Object data)
                      throws AccessControlException
```

Throws:

AccessControlException

assertAuthorizedForFile

```
void assertAuthorizedForFile(String filepath)
                      throws AccessControlException
```

Checks if an account is authorized to access the referenced file. The implementation should validate and canonicalize the input to be sure the filepath is not malicious. This method throws an AccessControlException if access is not authorized, or if the referenced File does not exist. If the User is authorized, this method simply returns. Specification: The implementation should do the following: 1) Check to see if the File exists and if not, throw an AccessControlException 2) Use available information to make an access control decision a. Ideally, this policy would be data driven b. You can use the current User, roles, data type, data name, time of day, etc. c. Access control decisions must deny by default 3) If access is not permitted, throw AccessControlException with details

Parameters:

`filepath` - the path of the file to be checked, including filename

Throws:

AccessControlException - if access is not permitted

See Also:

Encoder.canonicalize(String)

assertAuthorizedForService

```
void assertAuthorizedForService(String serviceName)
                         throws AccessControlException
```

Checks if an account is authorized to access the referenced service. This can be used in applications that provide access to a variety of backend services. This method throws an AccessControlException if access is not authorized, or if the referenced service does not exist. If the User is authorized, this method simply returns. Specification: The implementation should do the following: 1) Check to see if the service exists and if not, throw an AccessControlException 2) Use available information to make an access control decision a. Ideally, this policy would be data driven b. You can use the current User, roles, data type, data name, time of day, etc. c. Access control decisions must deny by default 3) If access is not permitted, throw AccessControlException with details

Parameters:

`serviceName` - the service name

Throws:

`AccessControlException` - if access is not permitted

Interface AccessReferenceMap

org.owasp.esapi

All Known Implementing Classes:

IntegerAccessReferenceMap, RandomAccessReferenceMap

```
public interface AccessReferenceMap
```

The AccessReferenceMap interface is used to map from a set of internal direct object references to a set of indirect references that are safe to disclose publicly. This can be used to help protect database keys, filenames, and other types of direct object references. As a rule, developers should not expose their direct object references as it enables attackers to attempt to manipulate them.

Indirect references are handled as strings, to facilitate their use in HTML. Implementations can generate simple integers or more complicated random character strings as indirect references. Implementations should probably add a constructor that takes a list of direct references.

Note that in addition to defeating all forms of parameter tampering attacks, there is a side benefit of the AccessReferenceMap. Using random strings as indirect object references, as opposed to simple integers makes it impossible for an attacker to guess valid identifiers. So if per-user AccessReferenceMaps are used, then request forgery (CSRF) attacks will also be prevented.

```
Set fileSet = new HashSet();
fileSet.addAll(...); // add direct references (e.g. File objects)
AccessReferenceMap map = new AccessReferenceMap( fileSet );
// store the map somewhere safe - like the session!
String indRef = map.getIndirectReference( file1 );
String href = "http://www.aspectsecurity.com/esapi?file=" + indRef );
...
// if the indirect reference doesn't exist, it's likely an attack
// getDirectReference throws an AccessControlException
// you should handle as appropriate
String indref = request.getParameter( "file" );
File file = (File)map.getDirectReference( indref );
```

Method Summary	
String	**addDirectReference**(Object direct) Adds a direct reference to the AccessReferenceMap and generates an associated indirect reference.

Object	**getDirectReference**(String indirectReference) Get the original direct object reference from an indirect reference.
String	**getIndirectReference**(Object directReference) Get a safe indirect reference to use in place of a potentially sensitive direct object reference.
Iterator	**iterator**() Get an iterator through the direct object references.
String	**removeDirectReference**(Object direct) Removes a direct reference and its associated indirect reference from the AccessReferenceMap.
void	**update**(Set directReferences) Updates the access reference map with a new set of directReferences, maintaining any existing indirectReferences associated with items that are in the new list.

Method Detail

iterator

```
Iterator iterator()
```

Get an iterator through the direct object references. No guarantee is made as to the order of items returned.

Returns:

the iterator

---

getIndirectReference

```
String getIndirectReference(Object directReference)
```

Get a safe indirect reference to use in place of a potentially sensitive direct object reference. Developers should use this call when building URL's, form fields, hidden fields, etc... to help protect their private implementation information.

Parameters:

`directReference` - the direct reference

Returns:

the indirect reference

---

getDirectReference

```
Object getDirectReference(String indirectReference)
                    throws AccessControlException
```

Get the original direct object reference from an indirect reference. Developers should use this when they get an indirect reference from a request to translate it back into the real direct reference. If an invalid indirectReference is requested, then an AccessControlException is thrown.

Parameters:

`indirectReference` - the indirect reference

Returns:

the direct reference

Throws:

`AccessControlException` - if no direct reference exists for the specified indirect reference

---

addDirectReference

```
String addDirectReference(Object direct)
```

Adds a direct reference to the AccessReferenceMap and generates an associated indirect reference.

Parameters:

`direct` - the direct reference

Returns:

the corresponding indirect reference

---

removeDirectReference

```
String removeDirectReference(Object direct)
                      throws AccessControlException
```

Removes a direct reference and its associated indirect reference from the AccessReferenceMap.

Parameters:

`direct` - the direct reference to remove

Returns:

the corresponding indirect reference

Throws:

AccessControlException

---

update

```
void update(Set directReferences)
```

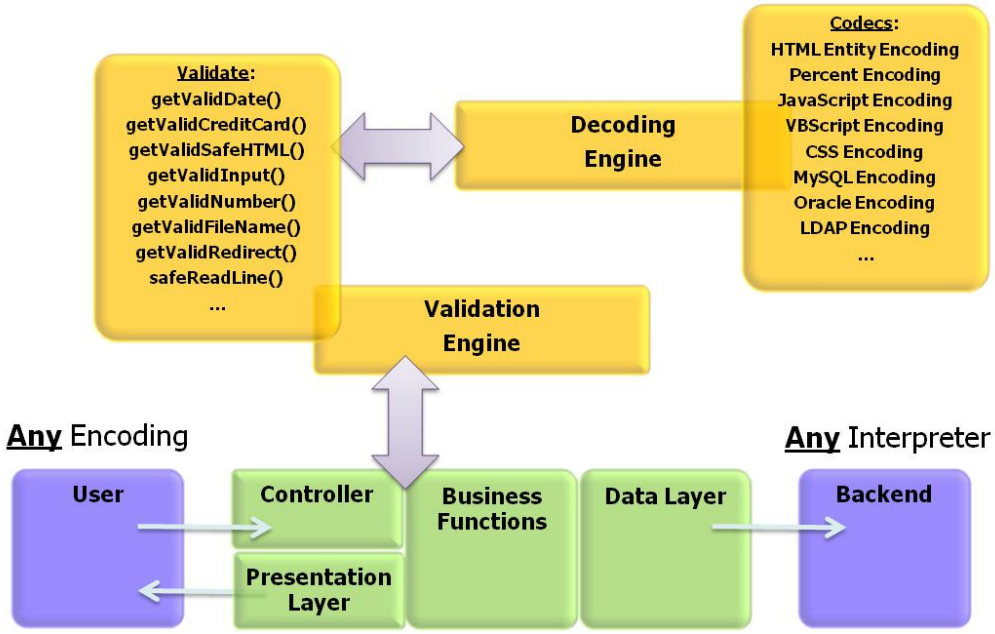
Updates the access reference map with a new set of directReferences, maintaining any existing indirectReferences associated with items that are in the new list.

Positive Input Validation

Validation is all about minimizing the attack surface of your application. An unvalidated form field can contain virtually any attack against a downstream interpreter or application. By enforcing a specific positive pattern, you dramatically reduce the attacker's freedom. Of course you shouldn't forget about using parameterized interfaces or output encoding/escaping, but checking the input is a critical first step.

Design

TODO



TODO

Minimize

The easiest way to make sure you don't accept bad input is to eliminate the need for accepting the input in the first place.

You should consider eliminating hidden fields entirely. Hidden fields are form values that aren't displayed to the user. When the user submits a form, the hidden fields are submitted just like any other form field. Attackers can easily change hidden field values to anything they want with browser tools like TamperData or WebDeveloper. Hidden fields are frequently quite vulnerable to attack because they're often overlooked when implementing validation.

You could use normal input validation techniques to check the hidden field, but the best way to check is to do a direct comparison with the value that you just set in the web page. Of course, if you can do this, there's really no point in having the hidden fields at all, so consider just getting rid of them and reducing your attack surface.

Global validation

There are hundreds of thousands of Unicode code points and dozens of different encodings. This creates a huge attack surface for your application.

As a first stage, your application should enforce a global set of allowed characters. You can make the list of allowed characters as broad as it has to be, but you'll bound your attack surface. If you want to get fancy, you can create individual whitelists for the various parts of the HTTP request.

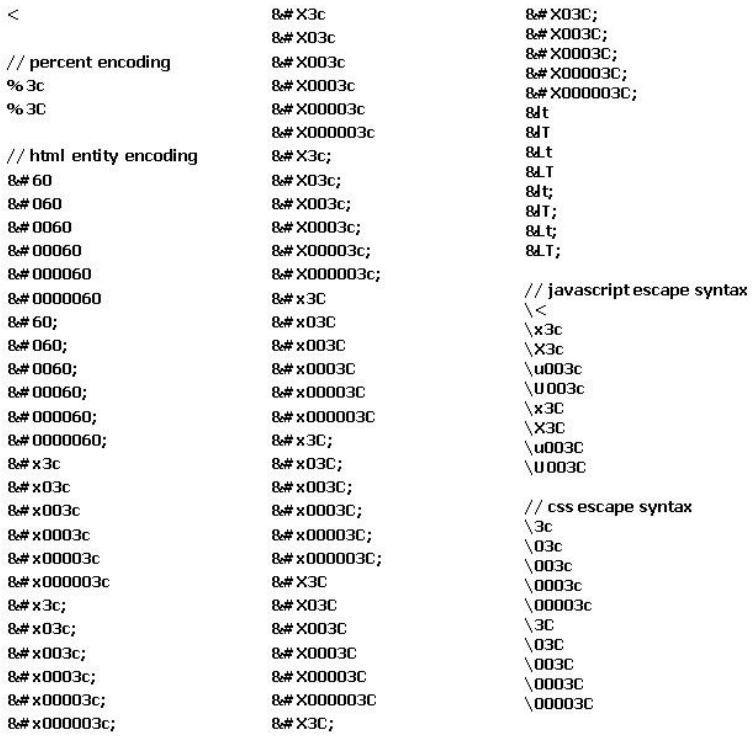
The global character whitelist isn't enough though. Your application will probably have to allow in some dangerous characters like apostrophe for example (to support poor Mr. O'Malley). Therefore, you also need to do specific validation on a field-by-field basis.

In ESAPI this is easy. ESAPI provides a single method that checks all parts of the HTTP request, including the querystring, headers, cookies, and form fields to make sure they match a predefined set of patterns. The specific patterns for these are defined in ESAPI.properties. If you'd like to make this happen automatically on every request, you can add this call to your controller or use a Filter in front of your application to apply this validation.

```
ESAPI.validator().assertIsValidHTTPRequest();
```

Canonicalize

It's a time honored principle that you cannot validate data until it is in its canonical form. However, actually canonicalizing user data is incredibly tricky these days. The table below shows many of the possible ways to encode a less-than sign



There are also many ways to double encode data.

```
< -> &lt; -> &#26;lt&#59 (double entity)
\ -> %5c -> %255c(double percent)
etc...
```

There are also double encoding with multiple schemes, for example:

```
< -> &lt; -> %26lt%3b (first entity, then percent)
< -> %26 -> &#25;26 (first percent, then entity)
etc...
```

And nested encoding, for example:

```
%3c -> %253c (nested encode % with percent)
%3c -> %%33%63 (nested encode percent both nibbles)
%3c -> %%33c (nested encode first nibble with percent)
%3c -> %3%63 (nested encode second nibble with percent)
< -> &&108;t; (nested encode l with entity)
etc...
```

And even nested encoding with multiple schemes:

```
< -> &%6ct; (nested encode l with percent)
%3c -> %&x33;c (nested encode 3 with entity)
etc...
```

And the order of applying the decoding is critical. If you decode in one order, the output is safe. If you decode in another order, the output is dangerous.

Original	Decode 1	Partial	Decode 2	Final
%26lt;	HTML Entity	%26lt;	URL Decode	<
%26lt;	URL Decode	<	HTML Entity	<
&#25;3c	HTML Entity	%3c	URL Decode	<
&#25;3c	URL Decode	&#25;3c	HTML Entity	%3c
%2526lt%253B	HTML Entity	%2526lt%253B	URL Decode	%26lt%3B
%2526lt%253B	URL Decode	%26lt%3B	HTML Entity	%26lt%3B
%2526lt%253B	2x URL Decode	<	HTML Entity	<

TODO

Handling validation errors

TODO

Rich data

All web applications allow some form of rich data, but it has become a key part of Web 2.0. Data is "rich" if it allows markup, special characters, images, formatting, and other complex syntax. This richness allows users create new and innovative content and services.

Unfortunately, richness also affords attackers an unprecedented opportunity to bury attacks targeting downstream users and systems. At least half the vulnerabilities that plague web applications and web services involve some form of injection. If your application forwards one of these buried attacks to innocent users downstream, you're potentially liable for the damage.

Unscrambling the Egg

One of the oldest security principles in the book is that you should always keep code and data separate. Once you mix them together, it's almost impossible separate them again. Unfortunately, most of the data formats and protocols we're using today mixing code and data like a bad DJ. That's why injection is going to be with us for a long time.

HTML is one of the worst offenders. JavaScript code can be placed in a huge number of places with dozens of different forms and encodings. See the XSS cheatsheet for some examples. HTML allows JavaScript in the header, body, dozens of event handlers, links, CSS definitions, and style attributes. There's no simple validation that can detect all the variants of code in all these places. You have to have a full security parser to validate HTML data before you can use it.

Untrusted Data Is Code

Almost everything connected to the Internet will execute "data" if an attacker buries the right kind of code in it. The code might be JavaScript, VB, SQL, LDAP, XPath, shell script, machine code or a hundred others depending on where that data goes. SQL injection is just an attacker sneaking malicious SQL inside user input that gets concatenated into a query. Injected code isn't just a snippet anymore – it might be a huge program.

What's important to remember is that every piece of untrusted data – every form field, every URL parameter, every cookie, and every XML parameter might contain injected code for some downstream system. If you're not absolutely sure there is no code in the data – and that's pretty much impossible – then for all you know, that data is really a little program. There is no such thing as plain old "data" anymore.

Think about HTML for a minute. We don't really "view" web pages anymore. They're programs. We "run" them in our browser. Even a tiny fragment of HTML can contain a script. Even without Javascript, web pages can be used for CSRF attacks that perform a series of functions for the attacker. That's a kind of program too. Would you open an HTML document sent to you?

Getting Worse

Attackers have even started to chain these attacks and use them in multiple stages. Consider the recent massive bot attacks that use SQL injection to jam JavaScript code into all the strings in a database. The infected data gets used in a webpage and the attack redirects the victim's browser to a site that installs malware. You can imagine attacks that are passed from system to system before they are ever executed and their payload is realized.

One factor that makes detecting these attacks difficult is that the web enables so many different types of encoding. There are over 100 different character encodings, and we've added higher level encodings such as percent-encoding, HTML-entity encoding, and bbcode on top of those. The real nightmare here is that anywhere downstream, systems may decode this data and reawaken a dormant attack.

So, even if your application isn't vulnerable to injection, someone might use the data from your application or service. As "Web 2.0" continues to mashup data from different sources, the likelihood of these attacks increases.

Stamping Out Injection

Developers should treat untrusted data as though it's malicious code with three easy steps - validate, separate, and encode.

1. Validate means that you should have a "whitelist" input validation rule for every input – no exceptions. Not just for form fields, but hidden fields, URL parameters, headers, cookies, and all backend systems.
2. Separate means you shouldn't mix up the data into command strings. Wherever possibly you should use parameterized interfaces, such as PreparedStatement in Java, that prevent injection by keeping code and data separate.
3. Encode means that you should encode untrusted data for the destination. One thing you absolutely have to do, but almost nobody does, is specify the character set that you'll be using. Then you'll need a set of methods that apply the proper encoding for the destination, such as an HTML page, HTML attribute, JavaScript, XPath query, LDAP query, etc…

Why do we care about downstream injection? Injection is not a new problem – we've known about it for decades. The body of knowledge on XSS and SQL injection is extensive. If your system has a flaw that forwards an attack to an innocent victim who is harmed because of your negligence, courts are not going to have any trouble holding you liable.

Rich Content

More and more, the data passed around the Internet is "rich", meaning that it contains markup and the data is intended to be parsed, rendered, and sometimes executed. Ensuring that this rich data does not contain any malicious instructions is extremely difficult. Nowhere is this problem more significant than in HTML, the worst scramble of code and data of all time.

Validating rich content is incredibly difficult. You cannot use simple string searches or even regular expressions to validate this type of data. You need a real parser. Actually, you need a parser that is tolerant of malformed markup and other syntax problems.

ESAPI protects against scripts embedded in rich data in a new way. Rather than trying to search through the input for dangerous characters and patterns, ESAPI takes advantage of another OWASP project called AntiSamy, which fully parses the rich content and has an extensive set of rules for which tags and attributes are allowed.

The difficulty of verifying whether rich content contains attacks is increasing rapidly as we use more and more complex formats. Using a robust parser and a whitelist set of rules is the right approach for detecting and preventing attacks. In addition to HTML, AntiSamy supports CSS, which is particularly challenging to parse and validate.

To use ESAPI to validate rich HTML data, use the following approach:

```
String dirty = request.getParameter( “input” );
String safeMarkup = ESAPI.validator().getValidSafeHTML( "input", dirty, 2500, true );
// store, use, or render the safeMarkup
```

Interface Validator

org.owasp.esapi

All Known Implementing Classes:

DefaultValidator

```
public interface Validator
```

The Validator interface defines a set of methods for canonicalizing and validating untrusted input. Implementors should feel free to extend this interface to accommodate their own data formats. Rather than throw exceptions, this interface returns boolean results because not all validation problems are security issues. Boolean returns allow developers to handle both valid and invalid results more cleanly than exceptions.

Implementations must adopt a "whitelist" approach to validation where a specific pattern or character set is matched. "Blacklist" approaches that attempt to identify the invalid or disallowed characters are much more likely to allow a bypass with encoding or other tricks.

Method Summary	
void	**assertIsValidHTTPRequest**() Validates the current HTTP request by comparing parameters, headers, and cookies to a predefined whitelist of allowed characters.
void	**assertIsValidHTTPRequestParameterSet**(String context, Set required, Set optional) Validates that the parameters in the current request contain all required parameters and only optional ones in addition.
void	**assertIsValidHTTPRequestParameterSet**(String context, Set required, Set optional, ValidationErrorList errorList) Validates that the parameters in the current request contain all required parameters and only optional ones in addition.
void	**assertValidFileUpload**(String context, String filepath, String filename, byte[] content, int maxBytes, boolean allowNull) Validates the filepath, filename, and content of a file.

void	**assertValidFileUpload**(String context, String filepath, String filename, byte[] content, int maxBytes, boolean allowNull, ValidationErrorList errorList) Validates the filepath, filename, and content of a file.
String	**getValidCreditCard**(String context, String input, boolean allowNull) Returns a canonicalized and validated credit card number as a String.
String	**getValidCreditCard**(String context, String input, boolean allowNull, ValidationErrorList errorList) Returns a canonicalized and validated credit card number as a String.
Date	**getValidDate**(String context, String input, DateFormat format, boolean allowNull) Returns a valid date as a Date.
Date	**getValidDate**(String context, String input, DateFormat format, boolean allowNull, ValidationErrorList errorList) Returns a valid date as a Date.
String	**getValidDirectoryPath**(String context, String input, boolean allowNull) Returns a canonicalized and validated directory path as a String.
String	**getValidDirectoryPath**(String context, String input, boolean allowNull, ValidationErrorList errorList) Returns a canonicalized and validated directory path as a String.
Double	**getValidDouble**(String context, String input, double minValue, double maxValue, boolean allowNull) Returns a validated real number as a double.
Double	**getValidDouble**(String context, String input, double minValue, double maxValue, boolean allowNull, ValidationErrorList errorList) Returns a validated real number as a double.
byte[]	**getValidFileContent**(String context, byte[] input, int maxBytes, boolean allowNull) Returns validated file content as a byte array.
byte[]	**getValidFileContent**(String context, byte[] input, int maxBytes, boolean allowNull, ValidationErrorList errorList) Returns validated file content as a byte array.

String	**getValidFileName**(String context, String input, boolean allowNull) Returns a canonicalized and validated file name as a String.
String	**getValidFileName**(String context, String input, boolean allowNull, ValidationErrorList errorList) Returns a canonicalized and validated file name as a String.
String	**getValidInput**(String context, String input, String type, int maxLength, boolean allowNull) Returns canonicalized and validated input as a String.
String	**getValidInput**(String context, String input, String type, int maxLength, boolean allowNull, ValidationErrorList errorList) Returns canonicalized and validated input as a String.
Integer	**getValidInteger**(String context, String input, int minValue, int maxValue, boolean allowNull) Returns a validated integer.
Integer	**getValidInteger**(String context, String input, int minValue, int maxValue, boolean allowNull, ValidationErrorList errorList) Returns a validated integer.
String	**getValidListItem**(String context, String input, List list) Returns the list item that exactly matches the canonicalized input.
String	**getValidListItem**(String context, String input, List list, ValidationErrorList errorList) Returns the list item that exactly matches the canonicalized input.
Double	**getValidNumber**(String context, String input, long minValue, long maxValue, boolean allowNull) Returns a validated number as a double within the range of minValue to maxValue.
Double	**getValidNumber**(String context, String input, long minValue, long maxValue, boolean allowNull, ValidationErrorList errorList) Returns a validated number as a double within the range of minValue to maxValue.
byte[]	**getValidPrintable**(String context, byte[] input, int maxLength, boolean allowNull) Returns canonicalized and validated printable characters as a byte array.

byte[]	**getValidPrintable**(String context, byte[] input, int maxLength, boolean allowNull, ValidationErrorList errorList) Returns canonicalized and validated printable characters as a byte array.
String	**getValidPrintable**(String context, String input, int maxLength, boolean allowNull) Returns canonicalized and validated printable characters as a String.
String	**getValidPrintable**(String context, String input, int maxLength, boolean allowNull, ValidationErrorList errorList) Returns canonicalized and validated printable characters as a String.
String	**getValidRedirectLocation**(String context, String input, boolean allowNull) Returns a canonicalized and validated redirect location as a String.
String	**getValidRedirectLocation**(String context, String input, boolean allowNull, ValidationErrorList errorList) Returns a canonicalized and validated redirect location as a String.
String	**getValidSafeHTML**(String context, String input, int maxLength, boolean allowNull) Returns canonicalized and validated "safe" HTML.
String	**getValidSafeHTML**(String context, String input, int maxLength, boolean allowNull, ValidationErrorList errorList) Returns canonicalized and validated "safe" HTML.
boolean	**isValidCreditCard**(String context, String input, boolean allowNull) Returns true if input is a valid credit card.
boolean	**isValidDate**(String context, String input, DateFormat format, boolean allowNull) Returns true if input is a valid date according to the specified date format.
boolean	**isValidDirectoryPath**(String context, String input, boolean allowNull) Returns true if input is a valid directory path.
boolean	**isValidDouble**(String context, String input, double minValue, double maxValue, boolean allowNull) Returns true if input is a valid double within the range of minValue to maxValue.

boolean	**isValidFileContent**(String context, byte[] input, int maxBytes, boolean allowNull) Returns true if input is valid file content.
boolean	**isValidFileName**(String context, String input, boolean allowNull) Returns true if input is a valid file name.
boolean	**isValidFileUpload**(String context, String filepath, String filename, byte[] content, int maxBytes, boolean allowNull) Returns true if a file upload has a valid name, path, and content.
boolean	**isValidHTTPRequest**() Validate the current HTTP request by comparing parameters, headers, and cookies to a predefined whitelist of allowed characters.
boolean	**isValidHTTPRequestParameterSet**(String context, Set required, Set optional) Returns true if the parameters in the current request contain all required parameters and only optional ones in addition.
boolean	**isValidInput**(String context, String input, String type, int maxLength, boolean allowNull) Returns true if input is valid according to the specified type.
boolean	**isValidInteger**(String context, String input, int minValue, int maxValue, boolean allowNull) Returns true if input is a valid integer within the range of minValue to maxValue.
boolean	**isValidListItem**(String context, String input, List list) Returns true if input is a valid list item.
boolean	**isValidNumber**(String context, String input, long minValue, long maxValue, boolean allowNull) Returns true if input is a valid number within the range of minValue to maxValue.
boolean	**isValidPrintable**(String context, byte[] input, int maxLength, boolean allowNull) Returns true if input contains only valid printable ASCII characters.
boolean	**isValidPrintable**(String context, String input, int maxLength, boolean allowNull) Returns true if input contains only valid printable ASCII characters (32-126).

boolean	**isValidRedirectLocation**(String context, String input, boolean allowNull) Returns true if input is a valid redirect location, as defined by "ESAPI.properties".
boolean	**isValidSafeHTML**(String context, String input, int maxLength, boolean allowNull) Returns true if input is "safe" HTML.
String	**safeReadLine**(InputStream inputStream, int maxLength) Reads from an input stream until end-of-line or a maximum number of characters.

Method Detail

isValidInput

```
boolean isValidInput(String context,
                     String input,
                     String type,
                     int maxLength,
                     boolean allowNull)
              throws IntrusionException
```

Returns true if input is valid according to the specified type. The type parameter must be the name of a defined type in the ESAPI configuration or a valid regular expression. Implementers should take care to make the type storage simple to understand and configure.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`input` - The actual user input data to validate.

`type` - The regular expression name that maps to the actual regular expression from "ESAPI.properties".

`maxLength` - The maximum post-canonicalized String length allowed.

`allowNull` - If allowNull is true then an input that is NULL or an empty string will be legal. If allowNull is false then NULL or an empty String will throw a ValidationException.

Returns:

true, if the input is valid based on the rules set by 'type'

Throws:

IntrusionException

---

getValidInput

```
String getValidInput(String context,
                     String input,
                     String type,
                     int maxLength,
                     boolean allowNull)
              throws ValidationException,
                     IntrusionException
```

Returns canonicalized and validated input as a String. Invalid input will generate a descriptive ValidationException, and input that is clearly an attack will generate a descriptive IntrusionException.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`input` - The actual user input data to validate.

`type` - The regular expression name that maps to the actual regular expression from "ESAPI.properties".

`maxLength` - The maximum post-canonicalized String length allowed.

`allowNull` - If allowNull is true then an input that is NULL or an empty string will be legal. If allowNull is false then NULL or an empty String will throw a ValidationException.

Returns:

The canonicalized user input.

Throws:

ValidationException

IntrusionException

---

getValidInput

```
String getValidInput(String context,
                     String input,
                     String type,
                     int maxLength,
                     boolean allowNull,
                     ValidationErrorList errorList)
              throws IntrusionException
```

Returns canonicalized and validated input as a String. Invalid input will generate a descriptive ValidationException, and input that is clearly an attack will generate a descriptive IntrusionException. Instead of throwing a ValidationException on error, this variant will store the exception inside of the ValidationErrorList.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`input` - The actual user input data to validate.

`type` - The regular expression name that maps to the actual regular expression from "ESAPI.properties".

`maxLength` - The maximum post-canonicalized String length allowed.

`allowNull` - If allowNull is true then an input that is NULL or an empty string will be legal. If allowNull is false then NULL or an empty String will throw a ValidationException.

`errorList` - If validation is in error, resulting error will be stored in the errorList by context

Returns:

The canonicalized user input.

Throws:

IntrusionException

---

isValidDate

```
boolean isValidDate(String context,
                    String input,
                    DateFormat format,
                    boolean allowNull)
             throws IntrusionException
```

Returns true if input is a valid date according to the specified date format.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`input` - The actual user input data to validate.

`format` - Required formatting of date inputted.

`allowNull` - If allowNull is true then an input that is NULL or an empty string will be legal. If allowNull is false then NULL or an empty String will throw a ValidationException.

Returns:

true, if input is a valid date according to the format specified by 'format'

Throws:

IntrusionException

---

getValidDate

```
Date getValidDate(String context,
                  String input,
                  DateFormat format,
                  boolean allowNull)
           throws ValidationException,
                  IntrusionException
```

Returns a valid date as a Date. Invalid input will generate a descriptive ValidationException, and input that is clearly an attack will generate a descriptive IntrusionException.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`input` - The actual user input data to validate.

`format` - Required formatting of date inputted.

`allowNull` - If allowNull is true then an input that is NULL or an empty string will be legal. If allowNull is false then NULL or an empty String will throw a ValidationException.

Returns:

A valid date as a Date

Throws:

ValidationException

IntrusionException

---

getValidDate

```
Date getValidDate(String context,
                  String input,
                  DateFormat format,
                  boolean allowNull,
                  ValidationErrorList errorList)
           throws IntrusionException
```

Returns a valid date as a Date. Invalid input will generate a descriptive ValidationException and store it inside of the errorList argument, and input that is clearly an attack will generate a descriptive IntrusionException. Instead of throwing a ValidationException on error, this variant will store the exception inside of the ValidationErrorList.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`input` - The actual user input data to validate.

`format` - Required formatting of date inputted.

`allowNull` - If allowNull is true then an input that is NULL or an empty string will be legal. If allowNull is false then NULL or an empty String will throw a ValidationException.

`errorList` - If validation is in error, resulting error will be stored in the errorList by context

Returns:

A valid date as a Date

Throws:

`IntrusionException`

---

isValidSafeHTML

```
boolean isValidSafeHTML(String context,
                        String input,
                        int maxLength,
                        boolean allowNull)
                 throws IntrusionException
```

Returns true if input is "safe" HTML. Implementors should reference the OWASP AntiSamy project for ideas on how to do HTML validation in a whitelist way, as this is an extremely difficult problem.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`input` - The actual user input data to validate.

`maxLength` - The maximum post-canonicalized String length allowed.

`allowNull` - If allowNull is true then an input that is NULL or an empty string will be legal. If allowNull is false then NULL or an empty String will throw a ValidationException.

Returns:

true, if input is valid safe HTML

Throws:

`IntrusionException`

---

getValidSafeHTML

```
String getValidSafeHTML(String context,
                        String input,
                        int maxLength,
                        boolean allowNull)
                 throws ValidationException,
                        IntrusionException
```

Returns canonicalized and validated "safe" HTML. Implementors should reference the OWASP AntiSamy project for ideas on how to do HTML validation in a whitelist way, as this is an extremely difficult problem.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`input` - The actual user input data to validate.

`maxLength` - The maximum post-canonicalized String length allowed.

`allowNull` - If allowNull is true then an input that is NULL or an empty string will be legal. If allowNull is false then NULL or an empty String will throw a ValidationException.

Returns:

Valid safe HTML

Throws:

ValidationException

IntrusionException

---

getValidSafeHTML

```
String getValidSafeHTML(String context,
                        String input,
                        int maxLength,
                        boolean allowNull,
                        ValidationErrorList errorList)
                 throws IntrusionException
```

Returns canonicalized and validated "safe" HTML. Implementors should reference the OWASP AntiSamy project for ideas on how to do HTML validation in a whitelist way, as this is an extremely difficult problem. Instead of throwing a ValidationException on error, this variant will store the exception inside of the ValidationErrorList.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`input` - The actual user input data to validate.

`maxLength` - The maximum post-canonicalized String length allowed.

`allowNull` - If allowNull is true then an input that is NULL or an empty string will be legal. If allowNull is false then NULL or an empty String will throw a ValidationException.

`errorList` - If validation is in error, resulting error will be stored in the errorList by context

Returns:

Valid safe HTML

Throws:

IntrusionException

isValidCreditCard

```
boolean isValidCreditCard(String context,
                          String input,
                          boolean allowNull)
                   throws IntrusionException
```

Returns true if input is a valid credit card. Maxlength is mandated by valid credit card type.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`input` - The actual user input data to validate.

`allowNull` - If allowNull is true then an input that is NULL or an empty string will be legal. If allowNull is false then NULL or an empty String will throw a ValidationException.

Returns:

true, if input is a valid credit card number

Throws:

IntrusionException

getValidCreditCard

```
String getValidCreditCard(String context,
                          String input,
                          boolean allowNull)
                   throws ValidationException,
                          IntrusionException
```

Returns a canonicalized and validated credit card number as a String. Invalid input will generate a descriptive ValidationException, and input that is clearly an attack will generate a descriptive IntrusionException.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`input` - The actual user input data to validate.

`allowNull` - If allowNull is true then an input that is NULL or an empty string will be legal. If allowNull is false then NULL or an empty String will throw a ValidationException.

Returns:

A valid credit card number

Throws:

ValidationException

IntrusionException

---

getValidCreditCard

```
String getValidCreditCard(String context,
                          String input,
                          boolean allowNull,
                          ValidationErrorList errorList)
                   throws IntrusionException
```

Returns a canonicalized and validated credit card number as a String. Invalid input will generate a descriptive ValidationException, and input that is clearly an attack will generate a descriptive IntrusionException. Instead of throwing a ValidationException on error, this variant will store the exception inside of the ValidationErrorList.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`input` - The actual input data to validate.

`allowNull` - If allowNull is true then an input that is NULL or an empty string will be legal. If allowNull is false then NULL or an empty String will throw a ValidationException.

`errorList` - If validation is in error, resulting error will be stored in the errorList by context

Returns:

A valid credit card number

Throws:

`IntrusionException`

---

isValidDirectoryPath

```
boolean isValidDirectoryPath(String context,
                             String input,
                             boolean allowNull)
                      throws IntrusionException
```

Returns true if input is a valid directory path.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`input` - The actual input data to validate.

`allowNull` - If allowNull is true then an input that is NULL or an empty string will be legal. If allowNull is false then NULL or an empty String will throw a ValidationException.

Returns:

true, if input is a valid directory path

Throws:

`IntrusionException`

---

getValidDirectoryPath

```
String getValidDirectoryPath(String context,
                             String input,
                             boolean allowNull)
                      throws ValidationException,
                             IntrusionException
```

Returns a canonicalized and validated directory path as a String. Invalid input will generate a descriptive ValidationException, and input that is clearly an attack will generate a descriptive IntrusionException.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`input` - The actual input data to validate.

`allowNull` - If allowNull is true then an input that is NULL or an empty string will be legal. If allowNull is false then NULL or an empty String will throw a ValidationException.

Returns:

A valid directory path

Throws:

ValidationException

IntrusionException

---

getValidDirectoryPath

```
String getValidDirectoryPath(String context,
                             String input,
                             boolean allowNull,
                             ValidationErrorList errorList)
                      throws IntrusionException
```

Returns a canonicalized and validated directory path as a String. Invalid input will generate a descriptive ValidationException, and input that is clearly an attack will generate a descriptive

IntrusionException. Instead of throwing a ValidationException on error, this variant will store the exception inside of the ValidationErrorList.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`input` - The actual input data to validate.

`allowNull` - If allowNull is true then an input that is NULL or an empty string will be legal. If allowNull is false then NULL or an empty String will throw a ValidationException.

`errorList` - If validation is in error, resulting error will be stored in the errorList by context

Returns:

A valid directory path

Throws:

`IntrusionException`

---

isValidFileName

```
boolean isValidFileName(String context,
                        String input,
                        boolean allowNull)
                 throws IntrusionException
```

Returns true if input is a valid file name.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`input` - The actual input data to validate.

`allowNull` - If allowNull is true then an input that is NULL or an empty string will be legal. If allowNull is false then NULL or an empty String will throw a ValidationException.

Returns:

true, if input is a valid file name

Throws:

`IntrusionException`

---

getValidFileName

```
String getValidFileName(String context,
                        String input,
                        boolean allowNull)
                 throws ValidationException,
                        IntrusionException
```

Returns a canonicalized and validated file name as a String. Implementors should check for allowed file extensions here, as well as allowed file name characters, as declared in "ESAPI.properties". Invalid input will generate a descriptive ValidationException, and input that is clearly an attack will generate a descriptive IntrusionException.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`input` - The actual input data to validate.

`allowNull` - If allowNull is true then an input that is NULL or an empty string will be legal. If allowNull is false then NULL or an empty String will throw a ValidationException.

Returns:

A valid file name

Throws:

`ValidationException`

IntrusionException

---

getValidFileName

```
String getValidFileName(String context,
                        String input,
                        boolean allowNull,
                        ValidationErrorList errorList)
                    throws IntrusionException
```

Returns a canonicalized and validated file name as a String. Implementors should check for allowed file extensions here, as well as allowed file name characters, as declared in "ESAPI.properties". Invalid input will generate a descriptive ValidationException, and input that is clearly an attack will generate a descriptive IntrusionException. Instead of throwing a ValidationException on error, this variant will store the exception inside of the ValidationErrorList.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`input` - The actual input data to validate.

`allowNull` - If allowNull is true then an input that is NULL or an empty string will be legal. If allowNull is false then NULL or an empty String will throw a ValidationException.

`errorList` - If validation is in error, resulting error will be stored in the errorList by context

Returns:

A valid file name

Throws:

IntrusionException

---

isValidNumber

```
boolean isValidNumber(String context,
                      String input,
                      long minValue,
                      long maxValue,
                      boolean allowNull)
               throws IntrusionException
```

Returns true if input is a valid number within the range of minValue to maxValue.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`input` - The actual input data to validate.

`minValue` - Lowest legal value for input.

`maxValue` - Highest legal value for input.

`allowNull` - If allowNull is true then an input that is NULL or an empty string will be legal. If allowNull is false then NULL or an empty String will throw a ValidationException.

Returns:

true, if input is a valid number

Throws:

IntrusionException

---

getValidNumber

```
Double getValidNumber(String context,
                      String input,
                      long minValue,
                      long maxValue,
                      boolean allowNull)
               throws ValidationException,
                      IntrusionException
```

Returns a validated number as a double within the range of minValue to maxValue. Invalid input will generate a descriptive ValidationException, and input that is clearly an attack will generate a descriptive IntrusionException.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`input` - The actual input data to validate.

`minValue` - Lowest legal value for input.

`maxValue` - Highest legal value for input.

`allowNull` - If allowNull is true then an input that is NULL or an empty string will be legal. If allowNull is false then NULL or an empty String will throw a ValidationException.

Returns:

A validated number as a double.

Throws:

`ValidationException`

`IntrusionException`

---

getValidNumber

```
Double getValidNumber(String context,
                      String input,
                      long minValue,
                      long maxValue,
                      boolean allowNull,
                      ValidationErrorList errorList)
               throws IntrusionException
```

Returns a validated number as a double within the range of minValue to maxValue. Invalid input will generate a descriptive ValidationException, and input that is clearly an attack will generate a descriptive IntrusionException. Instead of throwing a ValidationException on error, this variant will store the exception inside of the ValidationErrorList.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`input` - The actual input data to validate.

`minValue` - Lowest legal value for input.

`maxValue` - Highest legal value for input.

`allowNull` - If allowNull is true then an input that is NULL or an empty string will be legal. If allowNull is false then NULL or an empty String will throw a ValidationException.

`errorList` - If validation is in error, resulting error will be stored in the errorList by context

Returns:

A validated number as a double.

Throws:

`IntrusionException`

---

isValidInteger

```
boolean isValidInteger(String context,
                       String input,
                       int minValue,
                       int maxValue,
                       boolean allowNull)
                throws IntrusionException
```

Returns true if input is a valid integer within the range of minValue to maxValue.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`input` - The actual input data to validate.

`minValue` - Lowest legal value for input.

`maxValue` - Highest legal value for input.

`allowNull` - If allowNull is true then an input that is NULL or an empty string will be legal. If allowNull is false then NULL or an empty String will throw a ValidationException.

Returns:

true, if input is a valid integer

Throws:

`IntrusionException`

---

getValidInteger

```
Integer getValidInteger(String context,
                        String input,
                        int minValue,
                        int maxValue,
                        boolean allowNull)
                 throws ValidationException,
                        IntrusionException
```

Returns a validated integer. Invalid input will generate a descriptive ValidationException, and input that is clearly an attack will generate a descriptive IntrusionException.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`input` - The actual input data to validate.

`minValue` - Lowest legal value for input.

`maxValue` - Highest legal value for input.

`allowNull` - If allowNull is true then an input that is NULL or an empty string will be legal. If allowNull is false then NULL or an empty String will throw a ValidationException.

Returns:

A validated number as an integer.

Throws:

ValidationException

IntrusionException

---

getValidInteger

```
Integer getValidInteger(String context,
                        String input,
                        int minValue,
                        int maxValue,
                        boolean allowNull,
                        ValidationErrorList errorList)
                    throws IntrusionException
```

Returns a validated integer. Invalid input will generate a descriptive ValidationException, and input that is clearly an attack will generate a descriptive IntrusionException. Instead of throwing a ValidationException on error, this variant will store the exception inside of the ValidationErrorList.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`input` - The actual input data to validate.

`minValue` - Lowest legal value for input.

`maxValue` - Highest legal value for input.

`allowNull` - If allowNull is true then an input that is NULL or an empty string will be legal. If allowNull is false then NULL or an empty String will throw a ValidationException.

`errorList` - If validation is in error, resulting error will be stored in the errorList by context

Returns:

A validated number as an integer.

Throws:

IntrusionException

---

isValidDouble

```
boolean isValidDouble(String context,
                      String input,
                      double minValue,
                      double maxValue,
                      boolean allowNull)
               throws IntrusionException
```

Returns true if input is a valid double within the range of minValue to maxValue.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`input` - The actual input data to validate.

`minValue` - Lowest legal value for input.

`maxValue` - Highest legal value for input.

`allowNull` - If allowNull is true then an input that is NULL or an empty string will be legal. If allowNull is false then NULL or an empty String will throw a ValidationException.

Returns:

true, if input is a valid double.

Throws:

IntrusionException

---

getValidDouble

```
Double getValidDouble(String context,
                      String input,
                      double minValue,
                      double maxValue,
                      boolean allowNull)
               throws ValidationException,
                      IntrusionException
```

Returns a validated real number as a double. Invalid input will generate a descriptive ValidationException, and input that is clearly an attack will generate a descriptive IntrusionException.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`input` - The actual input data to validate.

`minValue` - Lowest legal value for input.

`maxValue` - Highest legal value for input.

`allowNull` - If allowNull is true then an input that is NULL or an empty string will be legal. If allowNull is false then NULL or an empty String will throw a ValidationException.

Returns:

A validated real number as a double.

Throws:

`ValidationException`

`IntrusionException`

getValidDouble

```
Double getValidDouble(String context,
                      String input,
                      double minValue,
                      double maxValue,
                      boolean allowNull,
                      ValidationErrorList errorList)
               throws IntrusionException
```

Returns a validated real number as a double. Invalid input will generate a descriptive ValidationException, and input that is clearly an attack will generate a descriptive IntrusionException. Instead of throwing a ValidationException on error, this variant will store the exception inside of the ValidationErrorList.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`input` - The actual input data to validate.

`minValue` - Lowest legal value for input.

`maxValue` - Highest legal value for input.

`allowNull` - If allowNull is true then an input that is NULL or an empty string will be legal. If allowNull is false then NULL or an empty String will throw a ValidationException.

`errorList` - If validation is in error, resulting error will be stored in the errorList by context

Returns:

A validated real number as a double.

Throws:

`IntrusionException`

isValidFileContent

```
boolean isValidFileContent(String context,
                           byte[] input,
                           int maxBytes,
                           boolean allowNull)
                    throws IntrusionException
```

Returns true if input is valid file content. This is a good place to check for max file size, allowed character sets, and do virus scans.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`input` - The actual input data to validate.

`maxBytes` - The maximum number of bytes allowed in a legal file.

`allowNull` - If allowNull is true then an input that is NULL or an empty string will be legal. If allowNull is false then NULL or an empty String will throw a ValidationException.

Returns:

true, if input contains valid file content.

Throws:

IntrusionException

---

getValidFileContent

```
byte[] getValidFileContent(String context,
                           byte[] input,
                           int maxBytes,
                           boolean allowNull)
                    throws ValidationException,
                           IntrusionException
```

Returns validated file content as a byte array. This is a good place to check for max file size, allowed character sets, and do virus scans. Invalid input will generate a descriptive

ValidationException, and input that is clearly an attack will generate a descriptive IntrusionException.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`input` - The actual input data to validate.

`maxBytes` - The maximum number of bytes allowed in a legal file.

`allowNull` - If allowNull is true then an input that is NULL or an empty string will be legal. If allowNull is false then NULL or an empty String will throw a ValidationException.

Returns:

A byte array containing valid file content.

Throws:

`ValidationException`

`IntrusionException`

---

getValidFileContent

```
byte[] getValidFileContent(String context,
                           byte[] input,
                           int maxBytes,
                           boolean allowNull,
                           ValidationErrorList errorList)
                    throws IntrusionException
```

Returns validated file content as a byte array. This is a good place to check for max file size, allowed character sets, and do virus scans. Invalid input will generate a descriptive ValidationException, and input that is clearly an attack will generate a descriptive IntrusionException. Instead of throwing a ValidationException on error, this variant will store the exception inside of the ValidationErrorList.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`input` - The actual input data to validate.

`maxBytes` - The maximum number of bytes allowed in a legal file.

`allowNull` - If allowNull is true then an input that is NULL or an empty string will be legal. If allowNull is false then NULL or an empty String will throw a ValidationException.

`errorList` - If validation is in error, resulting error will be stored in the errorList by context.

Returns:

A byte array containing valid file content.

Throws:

`IntrusionException`

---

isValidFileUpload

```
boolean isValidFileUpload(String context,
                          String filepath,
                          String filename,
                          byte[] content,
                          int maxBytes,
                          boolean allowNull)
                   throws IntrusionException
```

Returns true if a file upload has a valid name, path, and content.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`filepath` - The file path of the uploaded file.

`filename` - The filename of the uploaded file

`content` - A byte array containing the content of the uploaded file.

`maxBytes` - The max number of bytes allowed for a legal file upload.

`allowNull` - If allowNull is true then an input that is NULL or an empty string will be legal. If allowNull is false then NULL or an empty String will throw a ValidationException.

Returns:

true, if a file upload has a valid name, path, and content.

Throws:

`IntrusionException`

---

assertValidFileUpload

```
void assertValidFileUpload(String context,
                           String filepath,
                           String filename,
                           byte[] content,
                           int maxBytes,
                           boolean allowNull)
                    throws ValidationException,
                           IntrusionException
```

Validates the filepath, filename, and content of a file. Invalid input will generate a descriptive ValidationException, and input that is clearly an attack will generate a descriptive IntrusionException.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`filepath` - The file path of the uploaded file.

`filename` - The filename of the uploaded file

`content` - A byte array containing the content of the uploaded file.

`maxBytes` - The max number of bytes allowed for a legal file upload.

`allowNull` - If allowNull is true then an input that is NULL or an empty string will be legal. If allowNull is false then NULL or an empty String will throw a ValidationException.

Throws:

`ValidationException`

`IntrusionException`

---

assertValidFileUpload

```
void assertValidFileUpload(String context,
                           String filepath,
                           String filename,
                           byte[] content,
                           int maxBytes,
                           boolean allowNull,
                           ValidationErrorList errorList)
                    throws IntrusionException
```

Validates the filepath, filename, and content of a file. Invalid input will generate a descriptive ValidationException, and input that is clearly an attack will generate a descriptive IntrusionException. Instead of throwing a ValidationException on error, this variant will store the exception inside of the ValidationErrorList.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`filepath` - The file path of the uploaded file.

`filename` - The filename of the uploaded file

`content` - A byte array containing the content of the uploaded file.

`maxBytes` - The max number of bytes allowed for a legal file upload.

`allowNull` - If allowNull is true then an input that is NULL or an empty string will be legal. If allowNull is false then NULL or an empty String will throw a ValidationException.

errorList - If validation is in error, resulting error will be stored in the errorList by context

Throws:

IntrusionException

---

isValidHTTPRequest

```
boolean isValidHTTPRequest()
                    throws IntrusionException
```

Validate the current HTTP request by comparing parameters, headers, and cookies to a predefined whitelist of allowed characters. See the SecurityConfiguration class for the methods to retrieve the whitelists.

Returns:

true, if is a valid HTTP request

Throws:

IntrusionException

---

assertIsValidHTTPRequest

```
void assertIsValidHTTPRequest()
                        throws ValidationException,
                               IntrusionException
```

Validates the current HTTP request by comparing parameters, headers, and cookies to a predefined whitelist of allowed characters. Invalid input will generate a descriptive ValidationException, and input that is clearly an attack will generate a descriptive IntrusionException.

Throws:

ValidationException

IntrusionException

---

isValidListItem

```
boolean isValidListItem(String context,
                        String input,
                        List list)
                 throws IntrusionException
```

Returns true if input is a valid list item.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`input` - The value to search 'list' for.

`list` - The list to search for 'input'.

Returns:

true, if 'input' was found in 'list'.

Throws:

IntrusionException

---

getValidListItem

```
String getValidListItem(String context,
                        String input,
                        List list)
                 throws ValidationException,
                        IntrusionException
```

Returns the list item that exactly matches the canonicalized input. Invalid or non-matching input will generate a descriptive ValidationException, and input that is clearly an attack will generate a descriptive IntrusionException.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`input` - The value to search 'list' for.

`list` - The list to search for 'input'.

Returns:

The list item that exactly matches the canonicalized input.

Throws:

`ValidationException`

`IntrusionException`

---

getValidListItem

```
String getValidListItem(String context,
                        String input,
                        List list,
                        ValidationErrorList errorList)
                 throws IntrusionException
```

Returns the list item that exactly matches the canonicalized input. Invalid or non-matching input will generate a descriptive ValidationException, and input that is clearly an attack will generate a descriptive IntrusionException. Instead of throwing a ValidationException on error, this variant will store the exception inside of the ValidationErrorList.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`input` - The value to search 'list' for.

`list` - The list to search for 'input'.

`errorList` - If validation is in error, resulting error will be stored in the errorList by context

Returns:

The list item that exactly matches the canonicalized input.

Throws:

`IntrusionException`

---

isValidHTTPRequestParameterSet

```
boolean isValidHTTPRequestParameterSet(String context,
                                       Set required,
                                       Set optional)
                                throws IntrusionException
```

Returns true if the parameters in the current request contain all required parameters and only optional ones in addition.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`required` - parameters that are required to be in HTTP request

`optional` - additional parameters that may be in HTTP request

Returns:

true, if all required parameters are in HTTP request and only optional parameters in addition. Returns false if parameters are found in HTTP request that are not in either set (required or optional), or if any required parameters are missing from request.

Throws:

`IntrusionException`

---

assertIsValidHTTPRequestParameterSet

```
void assertIsValidHTTPRequestParameterSet(String context,
                                          Set required,
                                          Set optional)
                                   throws ValidationException,
                                          IntrusionException
```

Validates that the parameters in the current request contain all required parameters and only optional ones in addition. Invalid input will generate a descriptive ValidationException, and input that is clearly an attack will generate a descriptive IntrusionException.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`required` - parameters that are required to be in HTTP request

`optional` - additional parameters that may be in HTTP request

Throws:

ValidationException

IntrusionException

---

assertIsValidHTTPRequestParameterSet

```
void assertIsValidHTTPRequestParameterSet(String context,
                                          Set required,
                                          Set optional,
                                          ValidationErrorList errorList)
                                   throws IntrusionException
```

Validates that the parameters in the current request contain all required parameters and only optional ones in addition. Invalid input will generate a descriptive ValidationException, and input that is clearly an attack will generate a descriptive IntrusionException. Instead of throwing a ValidationException on error, this variant will store the exception inside of the ValidationErrorList.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`required` - parameters that are required to be in HTTP request

`optional` - additional parameters that may be in HTTP request

`errorList` - If validation is in error, resulting error will be stored in the errorList by context

Throws:

`IntrusionException`

---

isValidPrintable

```
boolean isValidPrintable(String context,
                         byte[] input,
                         int maxLength,
                         boolean allowNull)
                  throws IntrusionException
```

Returns true if input contains only valid printable ASCII characters.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`input` - data to be checked for validity

`maxLength` - Maximum number of bytes stored in 'input'

`allowNull` - If allowNull is true then an input that is NULL or an empty string will be legal. If allowNull is false then NULL or an empty String will throw a ValidationException.

Returns:

true, if 'input' is less than maxLength and contains only valid, printable characters

Throws:

IntrusionException

---

getValidPrintable

```
byte[] getValidPrintable(String context,
                         byte[] input,
                         int maxLength,
                         boolean allowNull)
                  throws ValidationException
```

Returns canonicalized and validated printable characters as a byte array. Invalid input will generate a descriptive ValidationException, and input that is clearly an attack will generate a descriptive IntrusionException.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`input` - data to be returned as valid and printable

`maxLength` - Maximum number of bytes stored in 'input'

`allowNull` - If allowNull is true then an input that is NULL or an empty string will be legal. If allowNull is false then NULL or an empty String will throw a ValidationException.

Returns:

a byte array containing only printable characters, made up of data from 'input'

Throws:

ValidationException

---

getValidPrintable

```
byte[] getValidPrintable(String context,
                         byte[] input,
                         int maxLength,
                         boolean allowNull,
                         ValidationErrorList errorList)
                  throws IntrusionException
```

Returns canonicalized and validated printable characters as a byte array. Invalid input will generate a descriptive ValidationException, and input that is clearly an attack will generate a descriptive IntrusionException. Instead of throwing a ValidationException on error, this variant will store the exception inside of the ValidationErrorList.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`input` - data to be returned as valid and printable

`maxLength` - Maximum number of bytes stored in 'input'

`allowNull` - If allowNull is true then an input that is NULL or an empty string will be legal. If allowNull is false then NULL or an empty String will throw a ValidationException.

`errorList` - If validation is in error, resulting error will be stored in the errorList by context

Returns:

a byte array containing only printable characters, made up of data from 'input'

Throws:

`IntrusionException`

isValidPrintable

```
boolean isValidPrintable(String context,
                         String input,
                         int maxLength,
                         boolean allowNull)
                  throws IntrusionException
```

Returns true if input contains only valid printable ASCII characters (32-126).

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`input` - data to be checked for validity

`maxLength` - Maximum number of bytes stored in 'input' after canonicalization

`allowNull` - If allowNull is true then an input that is NULL or an empty string will be legal. If allowNull is false then NULL or an empty String will throw a ValidationException.

Returns:

true, if 'input' is less than maxLength after canonicalization and contains only valid, printable characters

Throws:

IntrusionException

---

getValidPrintable

```
String getValidPrintable(String context,
                         String input,
                         int maxLength,
                         boolean allowNull)
                  throws ValidationException
```

Returns canonicalized and validated printable characters as a String. Invalid input will generate a descriptive ValidationException, and input that is clearly an attack will generate a descriptive IntrusionException.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`input` - data to be returned as valid and printable

`maxLength` - Maximum number of bytes stored in 'input' after canonicalization

`allowNull` - If allowNull is true then an input that is NULL or an empty string will be legal. If allowNull is false then NULL or an empty String will throw a ValidationException.

Returns:

a String containing only printable characters, made up of data from 'input'

Throws:

`ValidationException`

---

getValidPrintable

```
String getValidPrintable(String context,
                         String input,
                         int maxLength,
                         boolean allowNull,
                         ValidationErrorList errorList)
                  throws IntrusionException
```

Returns canonicalized and validated printable characters as a String. Invalid input will generate a descriptive ValidationException, and input that is clearly an attack will generate a descriptive IntrusionException. Instead of throwing a ValidationException on error, this variant will store the exception inside of the ValidationErrorList.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`input` - data to be returned as valid and printable

`maxLength` - Maximum number of bytes stored in 'input' after canonicalization

`allowNull` - If allowNull is true then an input that is NULL or an empty string will be legal. If allowNull is false then NULL or an empty String will throw a ValidationException.

`errorList` - If validation is in error, resulting error will be stored in the errorList by context

Returns:

a String containing only printable characters, made up of data from 'input'

Throws:

`IntrusionException`

---

isValidRedirectLocation

```
boolean isValidRedirectLocation(String context,
                                String input,
                                boolean allowNull)
                         throws IntrusionException
```

Returns true if input is a valid redirect location, as defined by "ESAPI.properties".

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`input` - redirect location to be checked for validity, according to rules set in "ESAPI.properties"

`allowNull` - If allowNull is true then an input that is NULL or an empty string will be legal. If allowNull is false then NULL or an empty String will throw a ValidationException.

Returns:

true, if 'input' is a valid redirect location, as defined by "ESAPI.properties", false otherwise.

Throws:

IntrusionException

---

getValidRedirectLocation

```
String getValidRedirectLocation(String context,
                                String input,
                                boolean allowNull)
                         throws ValidationException,
                                IntrusionException
```

Returns a canonicalized and validated redirect location as a String. Invalid input will generate a descriptive ValidationException, and input that is clearly an attack will generate a descriptive IntrusionException.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`input` - redirect location to be returned as valid, according to encoding rules set in "ESAPI.properties"

`allowNull` - If allowNull is true then an input that is NULL or an empty string will be legal. If allowNull is false then NULL or an empty String will throw a ValidationException.

Returns:

A canonicalized and validated redirect location, as defined in "ESAPI.properties"

Throws:

ValidationException

IntrusionException

---

getValidRedirectLocation

```
String getValidRedirectLocation(String context,
                                String input,
                                boolean allowNull,
                                ValidationErrorList errorList)
                         throws IntrusionException
```

Returns a canonicalized and validated redirect location as a String. Invalid input will generate a descriptive ValidationException, and input that is clearly an attack will generate a descriptive IntrusionException. Instead of throwing a ValidationException on error, this variant will store the exception inside of the ValidationErrorList.

Parameters:

`context` - A descriptive name of the parameter that you are validating (e.g., LoginPage_UsernameField). This value is used by any logging or error handling that is done with respect to the value passed in.

`input` - redirect location to be returned as valid, according to encoding rules set in "ESAPI.properties"

`allowNull` - If allowNull is true then an input that is NULL or an empty string will be legal. If allowNull is false then NULL or an empty String will throw a ValidationException.

`errorList` - If validation is in error, resulting error will be stored in the errorList by context

Returns:

A canonicalized and validated redirect location, as defined in "ESAPI.properties"

Throws:

`IntrusionException`

---

safeReadLine

```
String safeReadLine(InputStream inputStream,
                    int maxLength)
             throws ValidationException
```

Reads from an input stream until end-of-line or a maximum number of characters. This method protects against the inherent denial of service attack in reading until the end of a line. If an attacker doesn't ever send a newline character, then a normal input stream reader will read until all memory is exhausted and the platform throws an OutOfMemoryError and probably terminates.

Parameters:

`inputStream` - The InputStream from which to read data

`maxLength` - Maximum characters allowed to be read in per line

Returns:

a String containing the current line of inputStream

Throws:

`ValidationException`

Class ValidationErrorList

org.owasp.esapi

```
java.lang.Object
  └ org.owasp.esapi.ValidationErrorList
```

```
public class ValidationErrorList

extends Object
```

The ValidationErrorList class defines a well-formed collection of ValidationExceptions so that groups of validation functions can be called in a non-blocking fashion.

To use the ValidationErrorList to execute groups of validation attempts, your controller code would look something like:

```
 ValidationErrorList() errorList = new ValidationErrorList();.
 String name  = getValidInput("Name", form.getName(), "SomeESAPIRegExName1", 255, fals
e, errorList);
 String address = getValidInput("Address", form.getAddress(), "SomeESAPIRegExName2", 2
55, false, errorList);
 Integer weight = getValidInteger("Weight", form.getWeight(), 1, 1000000000, false, er
rorList);
 Integer sortOrder = getValidInteger("Sort Order", form.getSortOrder(), -
100000, +100000, false, errorList);
 request.setAttribute( "ERROR_LIST", errorList );
```

The at your view layer you would be able to retrieve all of your error messages via a helper function like:

```
 public static ValidationErrorList getErrors() {
     HttpServletRequest request = ESAPI.httpUtilities().getCurrentRequest();
     ValidationErrorList errors = new ValidationErrorList();
     if (request.getAttribute(Constants.ERROR_LIST) != null) {
        errors = (ValidationErrorList)request.getAttribute("ERROR_LIST");
     }
     return errors;
 }
```

You can list all errors like:

```
 %
      for (Object vo : errorList.errors()) {
         ValidationException ve = (ValidationException)vo;
 %>
 %= ESAPI.encoder().encodeForHTML(ve.getMessage()) %>
```

```
>
 %
      }
 %>
```

And even check if a specific UI component is in error via calls like:

```
ValidationException e = errorList.getError("Name");
```

Constructor Summary
ValidationErrorList()

Method Summary	
void	**addError**(String context, ValidationException ve) Adds a new error to list with a unique named context.
List	**errors**() Returns list of ValidationException, or empty list of no errors exist.
ValidationException	**getError**(String context) Retrieves ValidationException for given context if one exists.
boolean	**isEmpty**() Returns true if no error are present.
int	**size**() Returns the numbers of errors present.

Constructor Detail

ValidationErrorList

```
public ValidationErrorList()
```

Method Detail

addError

```
public void addError(String context,
                     ValidationException ve)
```

Adds a new error to list with a unique named context. No action taken if either element is null. Existing contexts will be overwritten.

Parameters:

`context` - unique named context for this ValidationErrorList

---

errors

```
public List errors()
```

Returns list of ValidationException, or empty list of no errors exist.

Returns:

List

---

getError

```
public ValidationException getError(String context)
```

Retrieves ValidationException for given context if one exists.

Parameters:

`context` - unique name for each error

Returns:

ValidationException or null for given context

---

isEmpty

```
public boolean isEmpty()
```

Returns true if no error are present.

Returns:

boolean

---

size

```
public int size()
```

Returns the numbers of errors present.

Returns:

boolean

Positive Output Encoding/Escaping

Output encoding and escaping is one of the most important and most widely ignored security controls. The goal is to prevent injection into renderers, interpreters, and decoders anywhere in or downstream from the application. Injection is at the root of many of the most serious application security flaws, including XSS, SQL injection, command injection, and more.

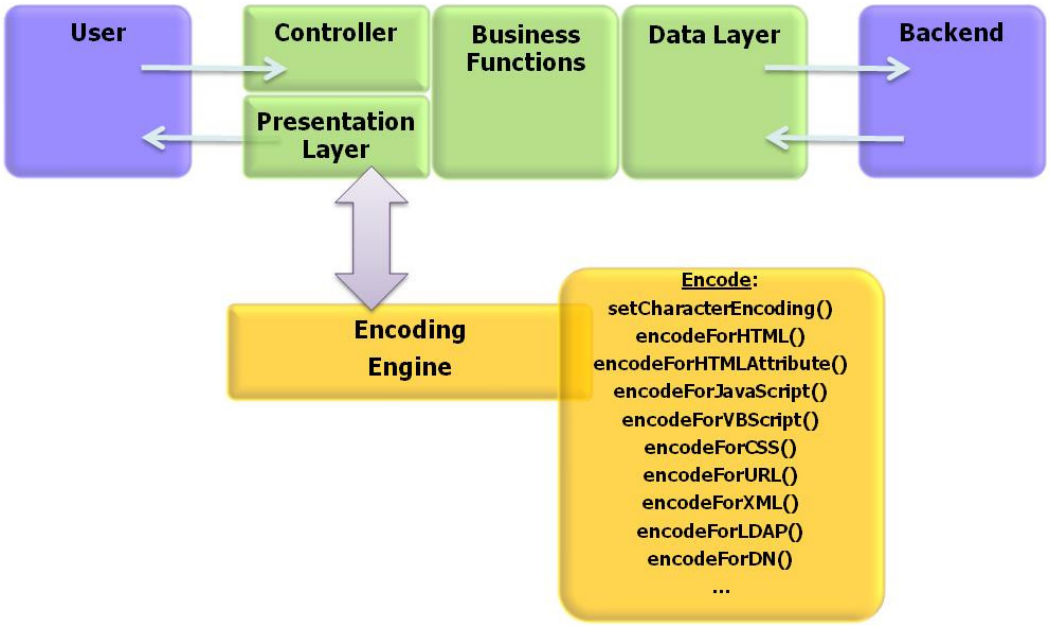
Design

Users should understand that the best defense against injection is to use a parameterized interface. This is an interface that keeps the parameters, typically user data, separate from the actual commands to be executed by the interpreter. The alternative is to use a "dynamic" query or command that concatenates all the user data into the command, so that the interpreter can't tell them apart.

Unfortunately, many interpreters don't have a parameterized interface, so we have to use the next best thing, encoding or escaping. Encoding and escaping work by changing all the characters that are relevant to the interpreter into harmless characters that just appear as data.

So for example, if the single quote character (') is used to end a quoted string in a SQL interpreter, that character might be escaped with a backslash (\'), so that it is no longer significant to the interpreter. This prevents the attacker from modifying the meaning of the query.

In ESAPI there are quite a variety of encoders that apply to data that is to be used in different contexts.



XSS and Injection

Think you've protected your web applications from cross-site scripting (XSS) vulnerabilities? The odds are against you. Roughly 90% of web applications have this problem, and it's getting worse.

XSS happens any time your application uses input from an HTTP request directly in HTML output. This includes everything in the HTTP request, including the querystring, form fields, hidden fields, pulldown menus, checkboxes, cookies, and headers. It doesn't matter if you immediately send the input back to the user who sent it ("Reflected XSS") or store it for a while and send it later to someone else ("Stored XSS").

XSS is a fairly serious vulnerability. By sending just the right input, typically a few special characters like " and > and then some javascript, an attacker can get a script running in the context of your web page. That script can disclose your application's cookies, rewrite your HTML, perform a phishing attack, or steal data from your forms. Attackers can even install an "XSS proxy" inside the victim's browser, allowing them to control your users' browsers remotely.

Security problems like XSS are inevitable when you don't keep code and data apart – and HTML is the worst mashup of code and data of all time. There's no way good way to keep JavaScript code and HTML data separate from each other. To prevent SQL injection, you can use a parameterized query to keep the data and code separate. But there's nothing equivalent for HTML. Just about every HTML element allows JavaScript code in attributes and event handlers, even CSS:

```
<div style="xss:expression(alert('xss'))" />
```

Even though proper output encoding protects against injection, you still must validate. One way to keep code out of user input is "whitelist input validation." All you have to do is verify that each input matches a strict definition of what you expect, like a tight regular expression. Attempting to take a shortcut and apply a global filter for attacks is known as a "blacklist" approach and never works. Unfortunately, even the tightest input validation can't completely defeat XSS, as some input fields require the same characters that are significant in HTML.

```
?name=O'Connell&comment=I "like" your website; it's great!
```

You'll also have to canonicalize the data before you validate. Attackers will use tricks to attempt to bypass your validation. By using various kinds of encoding, either alone or in combination, they hide code inside data. You have to decode this data to its simplest form ("canonicalize") before validating, or

you may allow code to reach some backend system that decodes it and enables the attack. But canonicalizing is easier said than done. Take a string that's been doubly encoded using multiple different (possibly unknown) encoding schemes, add any number of backend parsers, consider the browser's permissive parsing, and you have a real mess:

```
?input=test%2522%2Bonblur%253D%26quotalert&amp%23x28document.cookie%2529
```

You can use encoding to tell the browser that data shouldn't be run as code. "Whitelist output encoding" is simply replacing everything except a small list of safe characters (such as alphanumerics) with HTML entities before sending to the browser. The browser will render these entities instead of interpreting them, which defuses XSS attacks. Except that it doesn't always work. First, many frameworks and libraries use "blacklist" encoding where only a very small set of characters is encoded. Also, there are some places in HTML, such as href and src attributes, which allow HTML entities to be executed. So this insane URL is actually executed by browsers:

```
<a href="&#106ava&#115cript&#58&#x61ler&#116&#40&#39test&#39&#41&#59">test</a>
```

Don't forget to assign the proper character set: Even if you're doing great input validation and output encoding, attackers may try to trick the browser into using the wrong character set to interpret the data. An innocuous string in one character set may be interpreted as a dangerous string in a different character set. Many browsers will attempt to guess the character set if one is not specified. So, the harmless-looking string +ADw- will be interpreted as a < character if the browser decides to use UTF-7. This is why you should always be careful to set a sane character set like UTF-8.

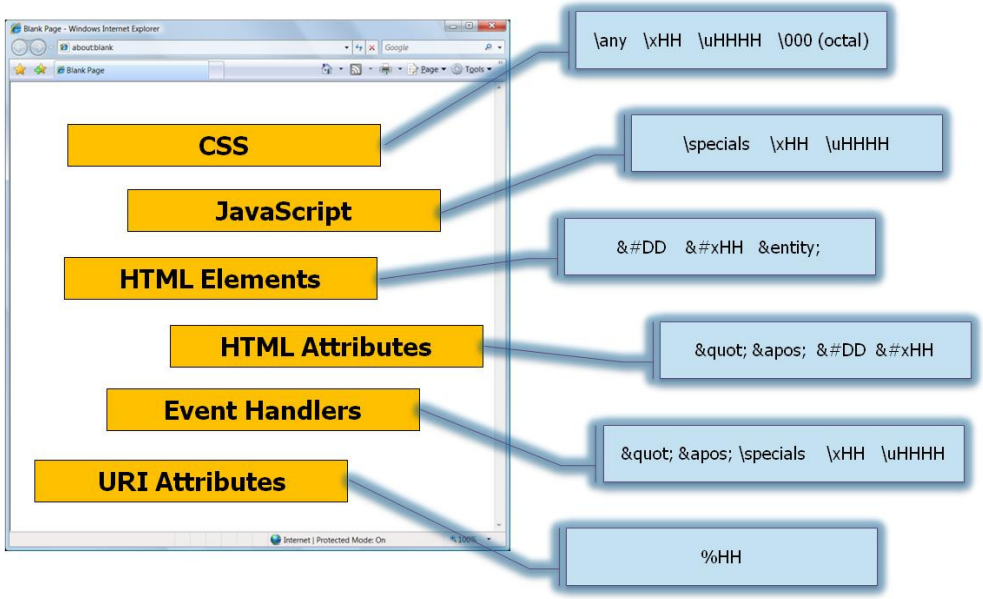
```
Content-Type: text/html; charset=UTF-8
```

As web applications and web services share more and more data, XSS is getting worse. Many frameworks and libraries are encoding, decoding, and re-encoding with all kinds of schemes and sending data through new protocols. Ajax and other "rich" applications are complicating this situation. One thing is for sure, XSS isn't going away any time soon.

Contexts

Understanding the context for where user data will land is critically important. If you're sending to a simple interpreter, then you just need to understand where in the command string the user data will end up. Is it inside a quoted string? A filename? A parameter name? Frequently, the rules are different

depending on the context. Nowhere is this problem more obvious than in HTML, which contains multiple parsers and dozens of contexts. Here are just a few.



Output encoding

In many applications, any text that you put in a form field can become a part of the webpage.

```
String name = request.getParameter("name");
...
<p>Hello, <%=name></p>
```

Notice that the input will show up in the text of the page right after the word "Hello." If you enter a script in the field, it will become a part of the page and will run. This is a textbook example of reflected XSS.

To fix the problem, we can wrap the user input contained in the "name" parameter with a call to the ESAPI Encoder.encodeForHTML() method. The encodeForHTML() method uses a "whitelist" HTML entity encoding algorithm to ensure that they cannot be interpreted as script. The Encoder ensures that there are no double-encoded characters in the input. This call should be used to wrap any user input being rendered in HTML element content.

```
<p>Hello, <%=ESAPI.encoder().encodeForHTML(name)%></p>
```

For the HTML attribute, the updated page uses ESAPI.encoder().encodeForHTMLAttribute(name). This call does a slightly different encoding that is more effective with attributes.

```
<form action="main?function=OutputUserInput&secure" method="POST">
    <p>Enter your name:</p>
    <input name='name' value='<%=ESAPI.encoder().encodeForHTMLAttribute(name) %>'>
    <input type='submit' value='submit'>
</form>
```

Interface Encoder

org.owasp.esapi

All Known Implementing Classes:

DefaultEncoder

`public interface` **`Encoder`**

The Encoder interface contains a number of methods related to encoding input so that it will be safe for a variety of interpreters. To prevent double-encoding, all encoding methods should first check to see that the input does not already contain encoded characters. There are a few methods related to decoding that are used for canonicalization purposes. See the Validator class for more information.

All of the methods here must use a "whitelist" or "positive" security model, meaning that all characters should be encoded, except for a specific list of "immune" characters that are known to be safe.

Field Summary	
char[]	**CHAR_ALPHANUMERICS**
char[]	**CHAR_DIGITS**
char[]	**CHAR_LETTERS**
char[]	**CHAR_LOWERS** Standard character sets
char[]	**CHAR_PASSWORD_DIGITS**
char[]	**CHAR_PASSWORD_LETTERS**
char[]	**CHAR_PASSWORD_LOWERS** Password character set, is alphanumerics (without l, i, I, o, O, and 0) selected specials like + (bad for URL encoding, | is like i and 1, etc...)
char[]	**CHAR_PASSWORD_SPECIALS**
char[]	**CHAR_PASSWORD_UPPERS**
char[]	**CHAR_SPECIALS**
char[]	**CHAR_UPPERS**

Method Summary	
String	**canonicalize**(String input) This method performs canonicalization on data received to ensure that it has been reduced to its most basic form before validation.
String	**canonicalize**(String input, boolean strict)
byte[]	**decodeFromBase64**(String input) Decode data encoded with BASE-64 encoding.
String	**decodeFromURL**(String input) Decode from URL.
String	**encodeForBase64**(byte[] input, boolean wrap) Encode for Base64.
String	**encodeForCSS**(String input) Encode data for use in Cascading Style Sheets (CSS) content.
String	**encodeForDN**(String input) Encode data for use in an LDAP distinguished name.
String	**encodeForHTML**(String input) Encode data for use in HTML content.
String	**encodeForHTMLAttribute**(String input) Encode data for use in HTML attributes.
String	**encodeForJavaScript**(String input) Encode data for insertion inside a data value in JavaScript.
String	**encodeForLDAP**(String input) Encode data for use in LDAP queries.
String	**encodeForOS**(Codec codec, String input) Encode for an operating system command shell according to the selected codec (appropriate codecs include the WindowsCodec and UnixCodec).
String	**encodeForSQL**(Codec codec, String input) Encode input for use in a SQL query (this method is not recommended), according to the selected codec (appropriate codecs include the MySQLCodec and OracleCodec).

String	**encodeForURL**(String input) Encode for use in a URL.
String	**encodeForVBScript**(String input) Encode data for insertion inside a data value in a visual basic script.
String	**encodeForXML**(String input) Encode data for use in an XML element.
String	**encodeForXMLAttribute**(String input) Encode data for use in an XML attribute.
String	**encodeForXPath**(String input) Encode data for use in an XPath query.
String	**normalize**(String input) Reduce all non-ascii characters to their ASCII form so that simpler validation rules can be applied.

Field Detail

CHAR_LOWERS

```
public static final char[] CHAR_LOWERS
```

Standard character sets

CHAR_UPPERS

```
public static final char[] CHAR_UPPERS
```

CHAR_DIGITS

```
public static final char[] CHAR_DIGITS
```

CHAR_SPECIALS

```
public static final char[] CHAR_SPECIALS
```

CHAR_LETTERS

```
public static final char[] CHAR_LETTERS
```

CHAR_ALPHANUMERICS

```
public static final char[] CHAR_ALPHANUMERICS
```

CHAR_PASSWORD_LOWERS

```
public static final char[] CHAR_PASSWORD_LOWERS
```

Password character set, is alphanumerics (without l, i, I, o, O, and 0) selected specials like + (bad for URL encoding, | is like i and 1, etc...)

CHAR_PASSWORD_UPPERS

```
public static final char[] CHAR_PASSWORD_UPPERS
```

CHAR_PASSWORD_DIGITS

```
public static final char[] CHAR_PASSWORD_DIGITS
```

CHAR_PASSWORD_SPECIALS

```
public static final char[] CHAR_PASSWORD_SPECIALS
```

CHAR_PASSWORD_LETTERS

```
public static final char[] CHAR_PASSWORD_LETTERS
```

Method Detail

canonicalize

```
String canonicalize(String input)
            throws EncodingException
```

This method performs canonicalization on data received to ensure that it has been reduced to its most basic form before validation. For example, URL-encoded data received from ordinary "application/x-www-url-encoded" forms so that it may be validated properly.

Canonicalization is simply the operation of reducing a possibly encoded string down to its simplest form. This is important, because attackers frequently use encoding to change their input in a way that will bypass validation filters, but still be interpreted properly by the target of the attack. Note that data encoded more than once is not something that a normal user would generate and should be regarded as an attack.

For input that comes from an HTTP servlet request, there are generally two types of encoding to be concerned with. The first is "applicaton/x-www-url-encoded" which is what is typically used in most forms and URI's where characters are encoded in a %xy format. The other type of common character encoding is HTML entity encoding, which uses several formats:

`&lt`

`,`

`&#117`

, and

`&#x3a`

.

Note that all of these formats may possibly render properly in a browser without the trailing semicolon.

Double-encoding is a particularly thorny problem, as applying ordinary decoders may introduce encoded characters, even characters encoded with a different encoding scheme. For example %26lt; is a character which has been entity encoded and then the first character has been url-encoded. Implementations should throw an IntrusionException when double-encoded characters are detected.

Note that there is also "multipart/form" encoding, which allows files and other binary data to be transmitted. Each part of a multipart form can itself be encoded according to a "Content-Transfer-Encoding" header. See the HTTPUtilties.getSafeFileUploads() method.

For more information on form encoding, please refer to the W3C specifications.

Parameters:

`input` - the text to canonicalize

Returns:

a String containing the canonicalized text

Throws:

`EncodingException` - if canonicalization fails

---

canonicalize

```
String canonicalize(String input,
                    boolean strict)
             throws EncodingException
```

Parameters:

`input` - the text to canonicalize

`strict` - true if checking for double encoding is desired, false otherwise

Returns:

a String containing the canonicalized text

Throws:

`EncodingException` - if canonicalization fails

---

normalize

```
String normalize(String input)
```

Reduce all non-ascii characters to their ASCII form so that simpler validation rules can be applied. For example, an accented-e character will be changed into a regular ASCII e character.

Parameters:

`input` - the text to normalize

Returns:

a normalized String

---

encodeForCSS

`String` **`encodeForCSS`**`(String input)`

Encode data for use in Cascading Style Sheets (CSS) content.

Parameters:

`input` - the text to encode for CSS

Returns:

input encoded for CSS

---

encodeForHTML

`String` **`encodeForHTML`**`(String input)`

Encode data for use in HTML content.

Parameters:

`input` - the text to encode for HTML

Returns:

input encoded for HTML

---

encodeForHTMLAttribute

`String` **`encodeForHTMLAttribute`**`(String input)`

Encode data for use in HTML attributes.

Parameters:

`input` - the text to encode for an HTML attribute

Returns:

input encoded for use as an HTML attribute

---

encodeForJavaScript

`String` **`encodeForJavaScript`**`(String input)`

Encode data for insertion inside a data value in JavaScript. Putting user data directly inside a script is quite dangerous. Great care must be taken to prevent putting user data directly into script code itself, as no amount of encoding will prevent attacks there.

Parameters:

`input` - the text to encode for JavaScript

Returns:

input encoded for use in JavaScript

---

encodeForVBScript

`String` **`encodeForVBScript`**`(String input)`

Encode data for insertion inside a data value in a visual basic script. Putting user data directly inside a script is quite dangerous. Great care must be taken to prevent putting user data directly into script code itself, as no amount of encoding will prevent attacks there.

Parameters:

`input` - the text to encode for VBScript

Returns:

input encoded for use in VBScript

---

encodeForSQL

```
String encodeForSQL(Codec codec,
                    String input)
```

Encode input for use in a SQL query (this method is not recommended), according to the selected codec (appropriate codecs include the MySQLCodec and OracleCodec). The use of the PreparedStatement interface is and preferred approach. However, if for some reason this is impossible, then this method is provided as a weaker alternative. The best approach is to make sure any single-quotes are double-quoted. Another possible approach is to use the {escape} syntax described in the JDBC specification in section 1.5.6 (see http://java.sun.com/j2se/1.4.2/docs/guide/jdbc/getstart/statement.html). However, this syntax does not work with all drivers, and requires modification of all queries.

Parameters:

`codec` - a Codec that declares which database 'input' is being encoded for (ie. MySQL, Oracle, etc.)

`input` - the text to encode for SQL

Returns:

input encoded for use in SQL

---

encodeForOS

```
String encodeForOS(Codec codec,
                   String input)
```

Encode for an operating system command shell according to the selected codec (appropriate codecs include the WindowsCodec and UnixCodec).

Parameters:

`codec` - a Codec that declares which database 'input' is being encoded for (ie. Windows, Unix, etc.)

`input` - the text to encode for the command shell

Returns:

input encoded for use in command shell

---

encodeForLDAP

```
String encodeForLDAP(String input)
```

Encode data for use in LDAP queries.

Parameters:

`input` - the text to encode for LDAP

Returns:

input encoded for use in LDAP

---

encodeForDN

```
String encodeForDN(String input)
```

Encode data for use in an LDAP distinguished name.

Parameters:

`input` - the text to encode for an LDAP distinguished name

Returns:

input encoded for use in an LDAP distinguished name

---

encodeForXPath

```
String encodeForXPath(String input)
```

Encode data for use in an XPath query.

Parameters:

`input` - the text to encode for XPath

Returns:

input encoded for use in XPath

---

encodeForXML

```
String encodeForXML(String input)
```

Encode data for use in an XML element. The implementation should follow the XML Encoding Standard from the W3C.

The use of a real XML parser is strongly encouraged. However, in the hopefully rare case that you need to make sure that data is safe for inclusion in an XML document and cannot use a parse, this method provides a safe mechanism to do so.

Parameters:

`input` - the text to encode for XML

Returns:

input encoded for use in XML

---

encodeForXMLAttribute

```
String encodeForXMLAttribute(String input)
```

Encode data for use in an XML attribute. The implementation should follow the XML Encoding Standard from the W3C.

The use of a real XML parser is highly encouraged. However, in the hopefully rare case that you need to make sure that data is safe for inclusion in an XML document and cannot use a parse, this method provides a safe mechanism to do so.

Parameters:

`input` - the text to encode for use as an XML attribute

Returns:

input encoded for use in an XML attribute

---

encodeForURL

```
String encodeForURL(String input)
            throws EncodingException
```

Encode for use in a URL. This method performs URL encoding" on the entire string.

Parameters:

`input` - the text to encode for use in a URL

Returns:

input encoded for use in a URL

Throws:

`EncodingException` - if encoding fails

---

decodeFromURL

```
String decodeFromURL(String input)
             throws EncodingException
```

Decode from URL. Implementations should first canonicalize and detect any double-encoding. If this check passes, then the data is decoded using URL decoding.

Parameters:

`input` - the text to decode from an encoded URL

Returns:

the decoded URL value

Throws:

EncodingException - if decoding fails

---

encodeForBase64

```
String encodeForBase64(byte[] input,
                       boolean wrap)
```

Encode for Base64.

Parameters:

input - the text to encode for Base64

Returns:

input encoded for Base64

---

decodeFromBase64

```
byte[] decodeFromBase64(String input)
                 throws IOException
```

Decode data encoded with BASE-64 encoding.

Parameters:

input - the Base64 text to decode

Returns:

input decoded from Base64

Throws:

IOException

Package org.owasp.esapi.codecs

Interface Summary	
Codec	The Codec interface defines a set of methods for encoding and decoding application level encoding schemes, such as HTML entity encoding and percent encoding (aka URL encoding).

Class Summary	
Base64	Encodes and decodes to and from Base64 notation.
CSSCodec	Implementation of the Codec interface for backslash encoding used in CSS.
HTMLEntityCodec	Implementation of the Codec interface for HTML entity encoding.
JavaScriptCodec	Implementation of the Codec interface for backslash encoding in JavaScript.
MySQLCodec	Implementation of the Codec interface for MySQL strings.
OracleCodec	Implementation of the Codec interface for Oracle strings.
PercentCodec	Implementation of the Codec interface for percent encoding (aka URL encoding).
PushbackString	The pushback string is used by Codecs to allow them to push decoded characters back onto a string for further decoding.
UnixCodec	Implementation of the Codec interface for '\' encoding from Unix command shell.
VBScriptCodec	Implementation of the Codec interface for 'quote' encoding from VBScript.
WindowsCodec	Implementation of the Codec interface for '^' encoding from Windows command shell.

Interface Executor

org.owasp.esapi

All Known Implementing Classes:

DefaultExecutor

```
public interface Executor
```

The Executor interface is used to run an OS command with reduced security risk. Implementations should do as much as possible to minimize the risk of injection into either the command or parameters. In addition, implementations should timeout after a specified time period in order to help prevent denial of service attacks. The class should perform logging and error handling as well. Finally, implementation should handle errors and generate an ExecutorException with all the necessary information.

Method Summary	
String	**executeSystemCommand**(File executable, List params, File workdir, Codec codec) Executes a system command after checking that the executable exists and escaping all the parameters to ensure that injection is impossible.

Method Detail

executeSystemCommand

```
String executeSystemCommand(File executable,
                            List params,
                            File workdir,
                            Codec codec)
                     throws ExecutorException
```

Executes a system command after checking that the executable exists and escaping all the parameters to ensure that injection is impossible. Implementations must change to the specified working directory before invoking the command.

Parameters:

`executable` - the command to execute

`params` - the parameters of the command being executed

`workdir` - the working directory

`codec` - the codec to use to encode for the particular OS in use

Returns:

the output of the command being run

Throws:

`ExecutorException` - the service exception

Class SafeFile

org.owasp.esapi

```
java.lang.Object
  └ java.io.File
      └ org.owasp.esapi.SafeFile
```

All Implemented Interfaces:

Comparable, Serializable

```
public class SafeFile
extends File
```

Extension to java.io.File to prevent against null byte injections and other unforeseen problems resulting from unprintable characters causing problems in path lookups. This does _not_ prevent against directory traversal attacks.

Constructor Summary
SafeFile(File parent, String child)
SafeFile(String path)
SafeFile(String parent, String child)
SafeFile(URI uri)

Constructor Detail

SafeFile

```
public SafeFile(String path)
         throws ValidationException
```

Throws:

ValidationException

SafeFile

```
public SafeFile(String parent,
                String child)
         throws ValidationException
```

Throws:

ValidationException

SafeFile

```
public SafeFile(File parent,
                String child)
         throws ValidationException
```

Throws:

ValidationException

SafeFile

```
public SafeFile(URI uri)
         throws ValidationException
```

Throws:

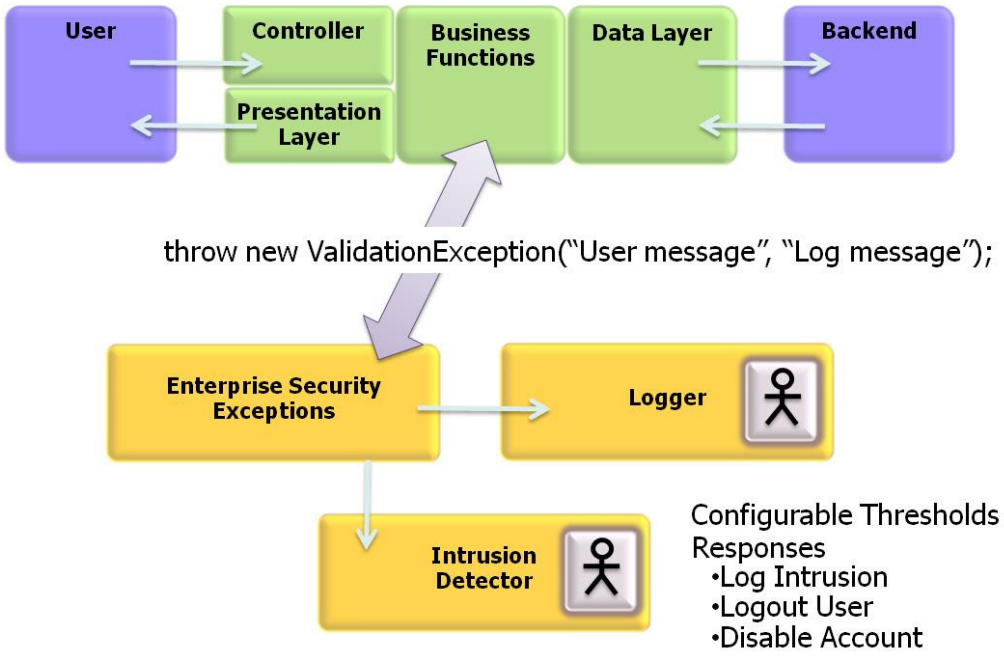
ValidationException

Positive Error Handling, Logging, and Intrusion Detection

Today, most web applications will let you attack them forever without taking any action to stop you. Most will have no idea that you are attacking them and will not even log the details necessary to let someone know of the attack. It is a rare application today that logs the content of form fields in the HTTP request, where many attacks can be found.

Design

ESAPI establishes a powerful infrastructure for detecting, handling, logging, and responding to attacks.



TODO

Package org.owasp.esapi.errors

A set of exception classes designed to model the error conditions that frequently arise in enterprise web applications and web services.

See:

Description

Exception Summary	
AccessControlException	An AccessControlException should be thrown when a user attempts to access a resource that they are not authorized for.
AuthenticationAccountsException	An AuthenticationException should be thrown when anything goes wrong during login or logout.
AuthenticationCredentialsException	An AuthenticationException should be thrown when anything goes wrong during login or logout.
AuthenticationException	An AuthenticationException should be thrown when anything goes wrong during login or logout.
AuthenticationHostException	An AuthenticationHostException should be thrown when there is a problem with the host involved with authentication, particularly if the host changes unexpectedly.
AuthenticationLoginException	An AuthenticationException should be thrown when anything goes wrong during login or logout.
AvailabilityException	An AvailabilityException should be thrown when the availability of a limited resource is in jeopardy.
CertificateException	A CertificateException should be thrown for any problems that arise during processing of digital certificates.
EncodingException	An ExecutorException should be thrown for any problems that occur when encoding or decoding data.
EncryptionException	An EncryptionException should be thrown for any problems related to encryption, hashing, or digital signatures.

EnterpriseSecurityException	EnterpriseSecurityException is the base class for all security related exceptions.
ExecutorException	An ExecutorException should be thrown for any problems that arise during the execution of a system executable.
IntegrityException	An AvailabilityException should be thrown when the availability of a limited resource is in jeopardy.
IntrusionException	An IntrusionException should be thrown anytime an error condition arises that is likely to be the result of an attack in progress.
ValidationAvailabilityException	
ValidationException	A ValidationException should be thrown to indicate that the data provided by the user or from some other external source does not match the validation rules that have been specified for that data.
ValidationUploadException	

Class EnterpriseSecurityException

org.owasp.esapi.errors

```
java.lang.Object
  └ java.lang.Throwable
      └ java.lang.Exception
          └ org.owasp.esapi.errors.EnterpriseSecurityException
```

All Implemented Interfaces:

Serializable

Direct Known Subclasses:

AccessControlException, AuthenticationException, AvailabilityException, CertificateException, EncodingException, EncryptionException, ExecutorException, IntegrityException, ValidationException

---

```
public class EnterpriseSecurityException
extends Exception
```

EnterpriseSecurityException is the base class for all security related exceptions. You should pass in the root cause exception where possible. Constructors for classes extending EnterpriseSecurityException should be sure to call the appropriate super() method in order to ensure that logging and intrusion detection occur properly.

All EnterpriseSecurityExceptions have two messages, one for the user and one for the log file. This way, a message can be shown to the user that doesn't contain sensitive information or unnecessary implementation details. Meanwhile, all the critical information can be included in the exception so that it gets logged.

Note that the "logMessage" for ALL EnterpriseSecurityExceptions is logged in the log file. This feature should be used extensively throughout ESAPI implementations and the result is a fairly complete set of security log records. ALL EnterpriseSecurityExceptions are also sent to the IntrusionDetector for use in detecting anomolous patterns of application usage.

Constructor Summary
EnterpriseSecurityException(String userMessage, String logMessage) Creates a new instance of EnterpriseSecurityException.
EnterpriseSecurityException(String userMessage, String logMessage, Throwable cause) Creates a new instance of EnterpriseSecurityException that includes a root cause Throwable.

Method Summary	
String	**getLogMessage**() Returns a message that is safe to display in logs, but probably not to users
String	**getUserMessage**() Returns message that is safe to display to users

Constructor Detail

EnterpriseSecurityException

```
public EnterpriseSecurityException(String userMessage,
                                   String logMessage)
```

Creates a new instance of EnterpriseSecurityException. This exception is automatically logged, so that simply by using this API, applications will generate an extensive security log. In addition, this exception is automatically registered with the IntrusionDetector, so that quotas can be checked.

Parameters:

`userMessage` - the message displayed to the user

`logMessage` - the message logged

EnterpriseSecurityException

```
public EnterpriseSecurityException(String userMessage,
                                   String logMessage,
                                   Throwable cause)
```

Creates a new instance of EnterpriseSecurityException that includes a root cause Throwable.

Parameters:

`userMessage` - the message displayed to the user

`logMessage` - the message logged

`cause` - the cause

Method Detail

getUserMessage

```
public String getUserMessage()
```

Returns message that is safe to display to users

Returns:

a String containing a message that is safe to display to users

---

getLogMessage

```
public String getLogMessage()
```

Returns a message that is safe to display in logs, but probably not to users

Returns:

a String containing a message that is safe to display in logs, but probably not to users

Class IntrusionException

org.owasp.esapi.errors

```
java.lang.Object
  └ java.lang.Throwable
      └ java.lang.Exception
          └ java.lang.RuntimeException
              └ org.owasp.esapi.errors.IntrusionException
```

All Implemented Interfaces:

Serializable

```
public class IntrusionException
extends RuntimeException
```

An IntrusionException should be thrown anytime an error condition arises that is likely to be the result of an attack in progress. IntrusionExceptions are handled specially by the IntrusionDetector, which is equipped to respond by either specially logging the event, logging out the current user, or invalidating the current user's account.

Unlike other exceptions in the ESAPI, the IntrusionException is a RuntimeException so that it can be thrown from anywhere and will not require a lot of special exception handling.

Constructor Summary
IntrusionException(String userMessage, String logMessage) Creates a new instance of IntrusionException.
IntrusionException(String userMessage, String logMessage, Throwable cause) Instantiates a new intrusion exception.

Method Summary	
String	**getLogMessage**() Returns a String that is safe to display in logs, but probably not to users
String	**getUserMessage**() Returns a String containing a message that is safe to display to users

Constructor Detail

IntrusionException

```
public IntrusionException(String userMessage,
                          String logMessage)
```

Creates a new instance of IntrusionException.

Parameters:

`userMessage` - the message to display to users

`logMessage` - the message logged

IntrusionException

```
public IntrusionException(String userMessage,
                          String logMessage,
                          Throwable cause)
```

Instantiates a new intrusion exception.

Parameters:

`userMessage` - the message to display to users

`logMessage` - the message logged

`cause` - the cause

Method Detail

getUserMessage

```
public String getUserMessage()
```

Returns a String containing a message that is safe to display to users

Returns:

a String containing a message that is safe to display to users

---

getLogMessage

```
public String getLogMessage()
```

Returns a String that is safe to display in logs, but probably not to users

Returns:

a String containing a message that is safe to display in logs, but probably not to users

Interface Logger

org.owasp.esapi

public interface **Logger**

The Logger interface defines a set of methods that can be used to log security events. Implementors should use a well established logging library as it is quite difficult to create a high-performance logger.

The order of logging levels is:

- trace
- debug
- info
- warn
- error
- fatal (the most serious)

In the default implementation, this interface is implemented by JavaLogger.class, which is an inner class in JavaLogFactory.java

Field Summary	
String	**FUNCTIONALITY**
String	**PERFORMANCE**
String	**SECURITY**
String	**USABILITY**

Method Summary	
void	**debug**(String type, String message) Log debug.
void	**debug**(String type, String message, Throwable throwable) Log debug.

void	**error**(String type, String message) Log error.
void	**error**(String type, String message, Throwable throwable) Log error.
void	**fatal**(String type, String message) Log critical.
void	**fatal**(String type, String message, Throwable throwable) Log critical.
void	**info**(String type, String message) Log success.
void	**info**(String type, String message, Throwable throwable) Log success.
boolean	**isDebugEnabled**() Allows the caller to determine if messages logged at this level will be discarded, to avoid performing expensive processing
boolean	**isErrorEnabled**() Allows the caller to determine if messages logged at this level will be discarded, to avoid performing expensive processing
boolean	**isFatalEnabled**() Allows the caller to determine if messages logged at this level will be discarded, to avoid performing expensive processing
boolean	**isInfoEnabled**() Allows the caller to determine if messages logged at this level will be discarded, to avoid performing expensive processing
boolean	**isTraceEnabled**() Allows the caller to determine if messages logged at this level will be discarded, to avoid performing expensive processing

boolean	**isWarningEnabled**() Allows the caller to determine if messages logged at this level will be discarded, to avoid performing expensive processing
void	**trace**(String type, String message) Log trace.
void	**trace**(String type, String message, Throwable throwable) Log trace.
void	**warning**(String type, String message) Log warning.
void	**warning**(String type, String message, Throwable throwable) Log warning.

Field Detail

SECURITY

```
public static final String SECURITY
```

USABILITY

```
public static final String USABILITY
```

PERFORMANCE

```
public static final String PERFORMANCE
```

FUNCTIONALITY

```
public static final String FUNCTIONALITY
```

Method Detail

fatal

```
void fatal(String type,
           String message)
```

Log critical.

Parameters:

`type` - the type of event

`message` - the message to log to log

---

fatal

```
void fatal(String type,
           String message,
           Throwable throwable)
```

Log critical.

Parameters:

`type` - the type of event of event

`message` - the message to log to log

`throwable` - the exception thrown

---

isFatalEnabled

```
boolean isFatalEnabled()
```

Allows the caller to determine if messages logged at this level will be discarded, to avoid performing expensive processing

Returns:

true if fatal messages will be output to the log

---

debug

```
void debug(String type,
           String message)
```

Log debug.

Parameters:

`type` - the type of event

`message` - the message to log

---

debug

```
void debug(String type,
           String message,
           Throwable throwable)
```

Log debug.

Parameters:

`type` - the type of event

`message` - the message to log

`throwable` - the exception thrown

---

isDebugEnabled

```
boolean isDebugEnabled()
```

Allows the caller to determine if messages logged at this level will be discarded, to avoid performing expensive processing

Returns:

true if debug messages will be output to the log

---

error

```
void error(String type,
           String message)
```

Log error.

Parameters:

`type` - the type of event

`message` - the message to log

---

error

```
void error(String type,
           String message,
           Throwable throwable)
```

Log error.

Parameters:

`type` - the type of event

`message` - the message to log

`throwable` - the exception thrown

---

isErrorEnabled

```
boolean isErrorEnabled()
```

Allows the caller to determine if messages logged at this level will be discarded, to avoid performing expensive processing

Returns:

true if error messages will be output to the log

---

info

```
void info(String type,
          String message)
```

Log success.

Parameters:

`type` - the type of event

`message` - the message to log

---

info

```
void info(String type,
          String message,
          Throwable throwable)
```

Log success.

Parameters:

`type` - the type of event

`message` - the message to log

`throwable` - the exception thrown

---

isInfoEnabled

```
boolean isInfoEnabled()
```

Allows the caller to determine if messages logged at this level will be discarded, to avoid performing expensive processing

Returns:

true info if messages will be output to the log

---

trace

```
void trace(String type,
           String message)
```

Log trace.

Parameters:

`type` - the type of event

`message` - the message to log

---

trace

```
void trace(String type,
           String message,
           Throwable throwable)
```

Log trace.

Parameters:

`type` - the type of event

`message` - the message to log

`throwable` - the exception thrown

---

isTraceEnabled

```
boolean isTraceEnabled()
```

Allows the caller to determine if messages logged at this level will be discarded, to avoid performing expensive processing

Returns:

true if trace messages will be output to the log

---

warning

```
void warning(String type,
             String message)
```

Log warning.

Parameters:

`type` - the type of event

`message` - the message to log

---

warning

```
void warning(String type,
             String message,
             Throwable throwable)
```

Log warning.

Parameters:

`type` - the type of event

`message` - the message to log

`throwable` - the exception thrown

---

isWarningEnabled

```
boolean isWarningEnabled()
```

Allows the caller to determine if messages logged at this level will be discarded, to avoid performing expensive processing

Returns:

true if warning messages will be output to the log

Interface IntrusionDetector

org.owasp.esapi

All Known Implementing Classes:

DefaultIntrusionDetector

```
public interface IntrusionDetector
```

The IntrusionDetector interface is intended to track security relevant events and identify attack behavior. The implementation can use as much state as necessary to detect attacks, but note that storing too much state will burden your system.

The interface is currently designed to accept exceptions as well as custom events. Implementations can use this stream of information to detect both normal and abnormal behavior.

Method Summary	
void	**addEvent**(String eventName, String logMessage) Adds the event to the IntrusionDetector.
void	**addException**(Exception exception) Adds the exception to the IntrusionDetector.

Method Detail

addException

```
void addException(Exception exception)
        throws IntrusionException
```

Adds the exception to the IntrusionDetector.

Parameters:

`exception` - the exception

Throws:

`IntrusionException` - the intrusion exception

addEvent

```
void addEvent(String eventName,
              String logMessage)
       throws IntrusionException
```

Adds the event to the IntrusionDetector.

Parameters:

`eventName` - the event

`logMessage` - the message to log with the event

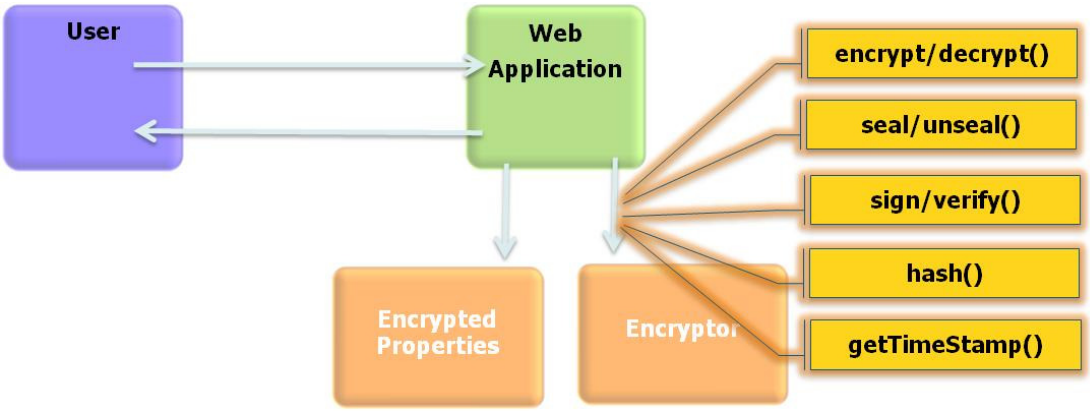
Throws:

`IntrusionException` - the intrusion exception

Positive Encryption, Hashing, and Random Numbers

Design

TODO



TODO

Using strong cryptography

TODO

Interface Encryptor

org.owasp.esapi

All Known Implementing Classes:

JavaEncryptor

```
public interface Encryptor
```

The Encryptor interface provides a set of methods for performing common encryption, random number, and hashing operations. Implementations should rely on a strong cryptographic implementation, such as JCE or BouncyCastle. Implementors should take care to ensure that they initialize their implementation with a strong "master key", and that they protect this secret as much as possible.

Possible future enhancements (depending on feedback) might include:

- encryptFile

Method Summary	
String	**decrypt**(String ciphertext) Decrypts the provided ciphertext string (encrypted with the encrypt method) and returns a plaintext string.
String	**encrypt**(String plaintext) Encrypts the provided plaintext and returns a ciphertext string.
long	**getRelativeTimeStamp**(long offset) Gets an absolute timestamp representing an offset from the current time to be used by other functions in the library.
long	**getTimeStamp**() Gets a timestamp representing the current date and time to be used by other functions in the library.
String	**hash**(String plaintext, String salt) Returns a string representation of the hash of the provided plaintext and salt.

String	**seal**(String data, long timestamp) Creates a seal that binds a set of data and includes an expiration timestamp.
String	**sign**(String data) Create a digital signature for the provided data and return it in a string.
String	**unseal**(String seal) Unseals data (created with the seal method) and throws an exception describing any of the various problems that could exist with a seal, such as an invalid seal format, expired timestamp, or decryption error.
boolean	**verifySeal**(String seal) Verifies a seal (created with the seal method) and throws an exception describing any of the various problems that could exist with a seal, such as an invalid seal format, expired timestamp, or data mismatch.
boolean	**verifySignature**(String signature, String data) Verifies a digital signature (created with the sign method) and returns the boolean result.

Method Detail

hash

```
String hash(String plaintext,
            String salt)
     throws EncryptionException
```

Returns a string representation of the hash of the provided plaintext and salt. The salt helps to protect against a rainbow table attack by mixing in some extra data with the plaintext. Some good choices for a salt might be an account name or some other string that is known to the application but not to an attacker. See this article for more information about hashing as it pertains to password schemes.

Parameters:

`plaintext` - the plaintext String to encrypt

`salt` - the salt

Returns:

the encrypted hash of 'plaintext' stored as a String

Throws:

EncryptionException - the encryption exception

---

encrypt

```
String encrypt(String plaintext)
        throws EncryptionException
```

Encrypts the provided plaintext and returns a ciphertext string.

Parameters:

plaintext - the plaintext String to encrypt

Returns:

the encrypted String

Throws:

EncryptionException - the encryption exception

---

decrypt

```
String decrypt(String ciphertext)
        throws EncryptionException
```

Decrypts the provided ciphertext string (encrypted with the encrypt method) and returns a plaintext string.

Parameters:

ciphertext - the ciphertext

Returns:

the decrypted ciphertext

Throws:

EncryptionException - the encryption exception

---

sign

```
String sign(String data)
     throws EncryptionException
```

Create a digital signature for the provided data and return it in a string.

Parameters:

data - the data to sign

Returns:

the digital signature stored as a String

Throws:

EncryptionException - the encryption exception

---

verifySignature

```
boolean verifySignature(String signature,
                        String data)
```

Verifies a digital signature (created with the sign method) and returns the boolean result.

Parameters:

signature - the signature to verify

data - the data to verify

Returns:

true, if the signature is verified

Throws:

EncryptionException - the encryption exception

---

seal

```
String seal(String data,
            long timestamp)
     throws IntegrityException
```

Creates a seal that binds a set of data and includes an expiration timestamp.

Parameters:

`data` - the data to seal

`timestamp` - the absolute expiration date of the data, expressed as seconds since the epoch

Returns:

the seal

Throws:

IntegrityException

EncryptionException - the encryption exception

---

unseal

```
String unseal(String seal)
       throws EncryptionException
```

Unseals data (created with the seal method) and throws an exception describing any of the various problems that could exist with a seal, such as an invalid seal format, expired timestamp, or decryption error.

Parameters:

`seal` - the sealed data

Returns:

the original data

Throws:

EncryptionException

ExcryptionException - if the unsealed data cannot be retrieved for any reason

---

verifySeal

`boolean` **`verifySeal`**`(String seal)`

Verifies a seal (created with the seal method) and throws an exception describing any of the various problems that could exist with a seal, such as an invalid seal format, expired timestamp, or data mismatch.

Parameters:

`seal` - the seal

Returns:

true, if the seal is valid

---

getRelativeTimeStamp

`long` **`getRelativeTimeStamp`**`(long offset)`

Gets an absolute timestamp representing an offset from the current time to be used by other functions in the library.

Parameters:

`offset` - the offset to add to the current time

Returns:

the absolute timestamp

---

getTimeStamp

```
long getTimeStamp()
```

Gets a timestamp representing the current date and time to be used by other functions in the library.

Returns:

the timestamp

Interface EncryptedProperties

org.owasp.esapi

All Known Implementing Classes:

DefaultEncryptedProperties

```
public interface EncryptedProperties
```

The EncryptedProperties interface represents a properties file where all the data is encrypted before it is added, and decrypted when it retrieved. This interface can be implemented in a number of ways, the simplest being extending Properties and overloading the getProperty and setProperty methods.

Method Summary	
String	**getProperty**(String key) Gets the property value from the encrypted store, decrypts it, and returns the plaintext value to the caller.
Set	**keySet**() Key set.
void	**load**(InputStream in) Load.
String	**setProperty**(String key, String value) Encrypts the plaintext property value and stores the ciphertext value in the encrypted store.
void	**store**(OutputStream out, String comments) Store.

Method Detail

getProperty

```
String getProperty(String key)
            throws EncryptionException
```

Gets the property value from the encrypted store, decrypts it, and returns the plaintext value to the caller.

Parameters:

`key` - the key

Returns:

the decrypted property value

Throws:

`EncryptionException` - the encryption exception

---

setProperty

```
String setProperty(String key,
                   String value)
            throws EncryptionException
```

Encrypts the plaintext property value and stores the ciphertext value in the encrypted store.

Parameters:

`key` - the key

`value` - the value

Returns:

the encrypted property value

Throws:

`EncryptionException` - the encryption exception

---

keySet

```
Set keySet()
```

Key set.

Returns:

the set

---

load

```
void load(InputStream in)
          throws IOException
```

Load.

Parameters:

`in` - the in

Throws:

`IOException` - Signals that an I/O exception has occurred.

---

store

```
void store(OutputStream out,
           String comments)
    throws IOException
```

Store.

Parameters:

`out` - the out

`comments` - the comments

Throws:

`IOException` - Signals that an I/O exception has occurred.

Interface Randomizer

org.owasp.esapi

All Known Implementing Classes:

DefaultRandomizer

```
public interface Randomizer
```

The IRandomizer interface defines a set of methods for creating cryptographically random numbers and strings. Implementers should be sure to use a strong cryptographic implementation, such as the JCE or BouncyCastle. Weak sources of randomness can undermine a wide variety of security mechanisms.

Method Summary	
boolean	**getRandomBoolean**() Returns a random boolean.
String	**getRandomFilename**(String extension) Returns an unguessable random filename with the specified extension.
String	**getRandomGUID**() Generates a random GUID.
int	**getRandomInteger**(int min, int max) Gets the random integer.
long	**getRandomLong**() Gets the random long.
float	**getRandomReal**(float min, float max) Gets the random real.
String	**getRandomString**(int length, char[] characterSet) Gets a random string of a desired length and character set.

Method Detail

getRandomString

```
String getRandomString(int length,
                       char[] characterSet)
```

Gets a random string of a desired length and character set.

Parameters:

`length` - the length of the string

`characterSet` - the character set

Returns:

the random string

---

getRandomBoolean

```
boolean getRandomBoolean()
```

Returns a random boolean.

Returns:

true or false, randomly

---

getRandomInteger

```
int getRandomInteger(int min,
                     int max)
```

Gets the random integer.

Parameters:

`min` - the minimum integer that will be returned

`max` - the maximum integer that will be returned

Returns:

the random integer

---

getRandomLong

```
long getRandomLong()
```

Gets the random long.

Returns:

the random long

---

getRandomFilename

```
String getRandomFilename(String extension)
```

Returns an unguessable random filename with the specified extension.

Parameters:

`extension` - extension to add to the random filename

Returns:

a random unguessable filename ending with the specified extension

---

getRandomReal

```
float getRandomReal(float min,
                    float max)
```

Gets the random real.

Parameters:

`min` - the minimum real number that will be returned

`max` - the maximum real number that will be returned

Returns:

the random real

---

getRandomGUID

```
String getRandomGUID()
            throws EncryptionException
```

Generates a random GUID.

Returns:

the GUID

Throws:

EncryptionException

Positive Communication Security

ESAPI does not provide much in the way of communication security. Unfortunately, the use of secure connections is typically out of the reach of software developers.

ESAPI does provide a simple function to check whether the channel used is secure or not.

Positive HTTP Security

Service Challenges (pattern?)

Positive Security Configuration

TODO

Interface SecurityConfiguration

org.owasp.esapi

All Known Implementing Classes:

DefaultSecurityConfiguration

`public interface` **`SecurityConfiguration`**

The SecurityConfiguration interface stores all configuration information that directs the behavior of the ESAPI implementation.

Protection of this configuration information is critical to the secure operation of the application using the ESAPI. You should use operating system access controls to limit access to wherever the configuration information is stored. Please note that adding another layer of encryption does not make the attackers job much more difficult. Somewhere there must be a master "secret" that is stored unencrypted on the application platform. Creating another layer of indirection doesn't provide any real additional security.

Nested Class Summary	
static class	**SecurityConfiguration.Threshold** Models a simple threshold as a count and an interval, along with a set of actions to take if the threshold is exceeded.

Method Summary	
List	**getAllowedFileExtensions**() Gets the allowed file extensions.
int	**getAllowedFileUploadSize**() Gets the allowed file upload size.
int	**getAllowedLoginAttempts**() Gets the allowed login attempts.

String	**getApplicationName**() Gets the application name, used for logging
String	**getCharacterEncoding**() Gets the character encoding.
String	**getDigitalSignatureAlgorithm**() Gets the digital signature algorithm.
String	**getEncryptionAlgorithm**() Gets the encryption algorithm.
String	**getHashAlgorithm**() Gets the hashing algorithm.
File	**getKeystore**() Gets the keystore.
boolean	**getLogEncodingRequired**() Returns whether HTML entity encoding should be applied to log entries.
char[]	**getMasterPassword**() Gets the master password.
byte[]	**getMasterSalt**() Gets the master salt.
int	**getMaxOldPasswordHashes**() Gets the max old password hashes.
String	**getPasswordParameterName**() Gets the password parameter name.
SecurityConfiguration.Threshold	**getQuota**(String eventName) Gets an intrusion detection Quota.
String	**getRandomAlgorithm**() Gets the random number generation algorithm.

long	**getRememberTokenDuration**() Gets the time window allowed for the remember token in milliseconds.
String	**getResourceDirectory**() Gets the ESAPI resource directory as a String.
String	**getResponseContentType**() Gets the content-type set for responses.
String	**getUsernameParameterName**() Gets the username parameter name.
void	**setResourceDirectory**(String dir) Sets the ESAPI resource directory.

Method Detail

getApplicationName

`String` **`getApplicationName`**`()`

Gets the application name, used for logging

Returns:

the application name

---

getMasterPassword

`char[]` **`getMasterPassword`**`()`

Gets the master password.

Returns:

the master password

---

getKeystore

`File` **`getKeystore`**`()`

Gets the keystore.

Returns:

the keystore

---

getMasterSalt

`byte[]` **`getMasterSalt`**`()`

Gets the master salt.

Returns:

the master salt

---

getAllowedFileExtensions

`List` **`getAllowedFileExtensions`**`()`

Gets the allowed file extensions.

Returns:

the allowed file extensions

---

getAllowedFileUploadSize

`int` **`getAllowedFileUploadSize`**`()`

Gets the allowed file upload size.

Returns:

the allowed file upload size

getPasswordParameterName

`String` **`getPasswordParameterName`**`()`

Gets the password parameter name.

Returns:

the password parameter name

---

getUsernameParameterName

`String` **`getUsernameParameterName`**`()`

Gets the username parameter name.

Returns:

the username parameter name

---

getEncryptionAlgorithm

`String` **`getEncryptionAlgorithm`**`()`

Gets the encryption algorithm.

Returns:

the encryption algorithm

---

getHashAlgorithm

`String` **`getHashAlgorithm`**`()`

Gets the hashing algorithm.

Returns:

the hashing algorithm

---

getCharacterEncoding

```
String getCharacterEncoding()
```

Gets the character encoding.

Returns:

encoding character name

---

getDigitalSignatureAlgorithm

```
String getDigitalSignatureAlgorithm()
```

Gets the digital signature algorithm.

Returns:

the digital signature algorithm

---

getRandomAlgorithm

```
String getRandomAlgorithm()
```

Gets the random number generation algorithm.

Returns:

random number generation algorithm

---

getAllowedLoginAttempts

```
int getAllowedLoginAttempts()
```

Gets the allowed login attempts.

Returns:

the allowed login attempts

---

getMaxOldPasswordHashes

`int` **`getMaxOldPasswordHashes`**`()`

Gets the max old password hashes.

Returns:

the max old password hashes

---

getQuota

`SecurityConfiguration.Threshold` **`getQuota`**`(String eventName)`

Gets an intrusion detection Quota.

Parameters:

`eventName` - the event whose quota is desired

Returns:

the matching Quota for eventName

---

getResourceDirectory

`String` **`getResourceDirectory`**`()`

Gets the ESAPI resource directory as a String.

Returns:

the ESAPI resource directory

setResourceDirectory

```
void setResourceDirectory(String dir)
```

Sets the ESAPI resource directory.

Parameters:

`dir` - location of the resource directory

---

getResponseContentType

```
String getResponseContentType()
```

Gets the content-type set for responses.

---

getRememberTokenDuration

```
long getRememberTokenDuration()
```

Gets the time window allowed for the remember token in milliseconds.

---

getLogEncodingRequired

```
boolean getLogEncodingRequired()
```

Returns whether HTML entity encoding should be applied to log entries.

A Note on "Rich Internet Applications"

Most new applications are distributed over the Internet and have client and server portions. This makes the term "Rich Internet Applications" a bit strange, but whatever we call them, this form of application is here to stay. There are a number of interesting security challenges with RIA.

Introduction

"Rich Internet Application" or "RIA" is what we call applications that run inside your browser or on your desktop that interact with web applications or web services. RIA platforms include Javascript (Ajax), Adobe AIR, Microsoft Silverlight, Java applets, and Java JFX. They look pretty, but should we trust these applications? These applications are downloaded from websites – some good, some bad, and some already compromised. So what protects users and server applications from a renegade RIA?

To get a good picture of RIA security, think of your browser as a multi-user operating system that is running your RIA along with many other hostile programs. There are two key security protections that limit what an RIA can do. They are known as the "same origin policy" and the "sandbox." Without these protections, a rogue RIA could access anything you see on the web and could also ravage your computer. Unfortunately, both policies have a long history of problems and they were never designed to handle RIAs.

The "Same Origin Policy"

If two sites share the same protocol ("http"), domain ("www.foo.com"), and port (80), we say they are from the "same origin." Browsers are supposed to restrict RIA from accessing anything that is not from the same origin. As the number of data structures and services in the browser increases, there's an explosion of places where the same origin policy must be applied. You can test it with "DOM Checker" which runs almost 1,400 test cases. The latest Internet Explorer fails 13 of these checks and the latest Firefox fails 25. But even if enforcement were perfect, we would still have to deal with "cross-site" problems like cross-site request forgery (CSRF) and cross-site scripting (XSS) which bypass the policy.

The "Sandbox"

If the same origin policy is like walls within the browser, then the sandbox is the floor, protecting the operating system from malicious RIAs. The sandbox is also absolutely critical for RIA. Without it, malicious RIAs could compromise the entire operating system and all the data and applications on it. The sandbox is also quite complex, as the browser does have legitimate need to access local files and launch local executables.

Plugin Policy

Getting these policies implemented would be hard enough once. Unfortunately, each RIA framework has its own environment that installs into the browser, and they can all communicate through the DOM. Each plugin is responsible for enforcing its own version of the same origin and sandbox policies.

There is also quite a lot of pressure on the folks building RIA platforms to allow even more cross-domain and cross-sandbox functionality. New use cases for sharing data and creating "mashups" are soon going to finally stretch these policies that were never intended for anything more than Web 1.0 past the breaking point.

Recommendations

First, be sure that you understand exactly what policies your RIA framework is trying to enforce. You may find that it simply doesn't provide the security controls you thought it did. There is absolutely no way to create a secure application without understanding what security the environment provides.

For now, recognize that your RIA is going to be running in a hostile environment with sketchy walls and a rotten floor. When you design your RIA, make sure that critical security decisions (particularly authentication and access control) are not made in the client-side code. These decisions should be made on the server side, where such decisions and code are safe from tampering. You should also minimize the data that is stored on the client, and provide a way to clear out the data as soon as it is no longer needed. Input validation and encoding are, as always, critical security controls that you will need to implement carefully.

Remember, these RIA platforms and frameworks are still very new. They all have lots of parsers and interpreters in them, and they have to enforce very complex security policies. The more you can minimize the number of technologies you are using, the better for security.

Wrapup

Don't think of security as restricting what RIA can do. The same origin policy and the sandbox are the security mechanisms that have allowed us to have such a diverse Internet available. We need to continue to enhance security controls for RIA so that we can take advantage of their awesome features. With so much demand for RIAs that can access multiple data sources, we need to bring the same origin policy up to the application layer and make decisions based on the true "originator" of services and data, not just the last "origin" they happened to come from.

AJAX

Fundamentally, there's nothing terribly new about Ajax security, but we need to do some work to apply all those good old security principles to this new technology. Unfortunately, there are an awful lot of devils in the details.

One major security challenge for Ajax applications is that moving your code to the client involves a ton of data formats, protocols, parsers, and interpreters. These include Javascript, VBScript, Flash, JSON, XML, REST, XmlHttpRequest, XSLT, CSS, and HTML in addition to your existing server side technologies. As if that wasn't enough, each of the Ajax frameworks has its own data formats and custom framework formats.

An application's "attack surface" approximates the ways in which an attacker can cause damage to your application or its users. The more technologies you use, the bigger your attack surface. To help hold the line on your Ajax application's attack surface, we offer the following rules.

Know what runs where

Ajax is making it increasingly difficult to be sure where your code is going to run. Take the Google Web Toolkit (GWT) for example. You program in Java and the environment takes some of that code and compiles it to Javascript that runs on the client. If you make a mistake and implement authentication, access control, validation, or other security checks in the code that runs on the client, an attacker can simply bypass them with Firebug.

Imagine you've carefully coded rules to be sure that administrative functions are never shown to ordinary users. This sounds good, but you forgot that the user interface code is running on the client. So the attacker uses Firebug to invoke the administrative functions. If the proper checks aren't in place on the server side, the attacker just gained administrative rights.

Many Rich Internet Application (RIA) frameworks also have this issue. The solution is to be very careful about making the client-server boundary clear.

Keep data separate from code

Hackers frequently use a technique called "injection" to bury commands inside innocent-looking data and getting interpreters to execute their commands for them. This simple trick is at the heart of many security attacks including SQL injection, command injection, LDAP injection, XSS, and buffer overflows. Preventing injection in a target rich environment like the modern browser takes discipline.

The key to stopping injection attacks is never executing data that might contain code. But with Ajax, lots of data and code get passed around and mashed together in the DOM. There has never been a data structure that mixes together code and data more than modern HTML. So be very careful with data that might include user input. Assume it's an attack unless you've carefully canonicalized, validated, and properly encoded.

Imagine you're invoking a REST interface and the request contains user data. The response you receive is a JSON string that includes that user data. Don't eval that string until you're sure that there can't be anything but safe data in there. Even just adding that data to the DOM might be enough to get it to execute if there's Javascript code buried in there.

Beware encoding

Encoding makes everything more complicated. Attackers can hide their attacks inside innocent looking data by encoding it. Backend systems may recognize the encoding used and execute the attack. Or they may decode the attack and pass it on to a system that's vulnerable to it.

Attackers may use multiple different encoding schemes, or even double encode to tunnel their attacks through innocent systems. There are dozens and dozens of encoding schemes and no way to tell which schemes will be recognized by the interpreters you're using. This makes recognizing attacks very difficult, if not impossible.

Every time you send or receive data both sides have to know the intended encoding. Never try to make a "best effort" attempt to guess the right encoding. You can't prevent an attacker from sending data with some other encoding through the channel, but you don't have to execute it. A few examples:

Set HTTP encoding in header:

```
Content-Type: text/xml, charset=utf-8
```

Or use a "meta" tag in the HTML:

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
```

Set XML encoding in the first line of XML documents:

```
<?xml version="1.0" encoding="utf-8"?>
```

Remember

Your Ajax application's attack surface is under your control. The choices you make can drastically affect the size of your application's attack surface. Be sure you ask questions about where your code runs, what data formats and protocols are involved, and which parsers and interpreters get invoked. And most importantly, be sure to nail down how you're going to keep code and data separate.

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

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

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 book title's lifecycle, and is a final product.

ALPHA **BETA** **RELEASE**

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.

OWASP
The Open Web Application Security Project

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.