Evaluation of Security Solutions

for CSCI-4970 Capstone Project “Kintsugi”

and IASC-4460 Network Vulnerability Discovery Semester Project

by Jon von Kampen

October 31st, 2013

# Purpose

Project Kintsugi must meet security requirements from the customer, and additional requirements specified by the developers. This document justifies the methods and tools used to meet these requirements. Solutions are considered in light of the application’s functionality and architecture.

# Application Architecture

**Web server:** Apache 2.2.22

**Application language:** Python 3

**Application framework:** Django 1.5

**Database server:** BaseX (native XML database)

# Security Requirements

Kintsugi is a web application that provides a searchable interface to the Common Weaknesses Enumeration (CWE) database. The following “must-have” security requirements were produced during the requirements engineering process:

1. Project must encode any CWE sample code, such that it will not execute if displayed in a web browser.
2. Project must encode any user input displayed on-page (*e.g.* search terms) to prevent cross-site scripting attacks.
3. Project must encode any user input in searches to prevent database injection attacks.
4. Project must verify that search filters originating from static values (*e.g.* dropdown list items) have not been modified to contain unexpected data.

## Related Security Concerns

The following additional concerns relate to the above requirements:

1. The application should include full-text search functionality, which may include searches for executable code strings.
2. The application may handle user-controlled GET and POST parameters as part of the search functionality.
3. Multiple character encodings must be properly handled for input sanitization and/or safe encoding to be effective.

# Solutions

## Django HTML entity encoding

When using Django templates, certain HTML entities that allow cross-site scripting (<, >, ‘, “, &)[[1]](#footnote-1)are automatically encoded to safe equivalents as pages are generated[[2]](#footnote-2). However, this only prevents injection into certain contexts, namely HTML blocks. For example, if an HTML tag attribute is built from user input, and is not surrounded in quotes, the user can inject additional attributes.

At the present time, CWE sample code or user input should be placed only within the HTML block context. As such, Django’s automatic protection should be sufficient.

**Addresses requirements: 1, 2, 5, 6**

### Alternative solutions

1. Stripping potentially dangerous characters is not an option, since input and output may include code.
2. Python functions can be used to escape characters (e.g., the html.escape function[[3]](#footnote-3)). However, developers must remember to use the selected function on every occasion. Django encodes output automatically as long as template syntax is used (which offers other benefits).

## Django form option validation

Django will be used to build and validate the contents of form fields, including search filters. A Django *ChoiceField* form object[[4]](#footnote-4) is built from a set of tuples of the format (‘value’, ‘human-readable name’). ChoiceField validation will throw an *invalid\_choice* error if a submitted value is not found within the tuple set.

**Addresses requirements:** **2, 4, 6**

### Alternative solutions

Developers could maintain data structures with acceptable static values, and verify that submitted values exist within the structures. This is essentially a less elegant duplication of Django’s functionality.

## Django character encoding setting

Django’s *DEFAULT\_CHARSET* setting can be used to instruct browsers to interpret output as UTF-8[[5]](#footnote-5). Django specifies the encoding via an HTTP header and an HTML <meta> tag.

**Addresses requirement: 7**

### Alternative solutions

Custom headers can be added to Django-generated responses[[6]](#footnote-6), and the <meta> tag could likewise be added. The Django-native solution is much simpler.

## Query binding and null byte escaping

BaseX has a Python API that supports query binding. This would seem to be all we need to prevent query injection, because binding typically prevents data from being interpreted as code. Unfortunately, the API’s bind method allows injection of null bytes, which are significant as field and command separators in the BaseX server protocol[[7]](#footnote-7). I determined this by reviewing the API code[[8]](#footnote-8) and writing a quick proof of concept that achieved arbitrary command execution (returning the results of the “help” command):

import BaseXClient

session = BaseXClient.Session('localhost', 1984, 'myLogin', 'myPassword')

input = "declare variable $name external; for $i in 1 to 10 return element { $name } { $i }"

query = session.query(input)

query.bind("$name", "number" + chr(0) + chr(0) + 'help')

print(query.execute())

query.close()

session.close()

(The crucial null byte and command injection is highlighted in red.)

The API will be modified to properly escape null bytes by prefixing them with 0xFF bytes, per protocol documentation. Additionally, changes will be pushed to the API GitHub repository (at the discretion of other BaseX developers) so that others can benefit from our work. If not accepted upstream, we will have to maintain a separate version of the API, adding the encoding feature to new releases.

**Addresses requirements: 3, 5, 6**

### Alternative solutions

1. We could search for another XML database solution with a more robust API. However, BaseX satisfies all feature requirements; choosing another database would take time from the development process; and the identified vulnerability solution is trivial to implement.
2. No additional Python 3 BaseX API exists, so switching APIs is not an option. Additionally, developing a new API from scratch exceeds the project scope and resources.

1. <https://docs.djangoproject.com/en/1.5/topics/templates/#automatic-html-escaping> [↑](#footnote-ref-1)
2. <https://docs.djangoproject.com/en/1.5/topics/security/#cross-site-scripting-xss-protection> [↑](#footnote-ref-2)
3. <http://docs.python.org/3/library/html.html#html.escape> [↑](#footnote-ref-3)
4. <https://docs.djangoproject.com/en/1.5/ref/forms/fields/#choicefield> [↑](#footnote-ref-4)
5. <http://security.stackexchange.com/a/34116> [↑](#footnote-ref-5)
6. <http://stackoverflow.com/questions/14377050/custom-http-header-in-django> [↑](#footnote-ref-6)
7. <http://docs.basex.org/wiki/Server_Protocol> [↑](#footnote-ref-7)
8. <https://github.com/BaseXdb/basex/blob/master/basex-api/src/main/python3/BaseXClient.py> – especially lines 164-165: “api won't escape 0x00, 0xff automatically, so you must do it yourself explicitly.” [↑](#footnote-ref-8)