Team-Too Final Report

EnDe - Final Report on all Testing and Deliverable Chapters

Ethan Price, Joshua Lusk, Lauren Donner, Zachary Kidd

Table of Contents

Introduction .... **pg 3**

Chapter 1 **OWASP's EnDE** .... **pg 4**
- The Main Project .... pg 4
- Installation .... pg 4
- Initial Testing .... pg 5

Chapter 2 **EnDe and Test Cases** .... **pg 7**
- The Testing Process: Major Phases .... pg 7
- Hardware and software requirements .... pg 7
- Test Recording Procedures .... pg 8
- Tested Items .... pg 9
- Constraints .... pg 12

Chapter 3 **Implementing Testing Framework** .... **pg 13**
- Architecture .... pg 13
- How To: Framework .... pg 14
- Some More Testing - Summarized .... pg 14

Chapter 4 **Automated Testing** .... **pg 18**
- Automated Testing Framework .... pg 18
- The Test Case Text Files .... pg 21
- Errors and Manual Checks .... pg 21

Chapter 5 **Fault Injection** .... **pg 24**
- Making a plan for code to fail, and how we did it .... pg 24
- Faulty Tests .... pg 24
- The Sample Run .... pg 25
- Update to the Original Testing Framework .... pg 26

Chapter 6 **Overall Experiences** .... **pg 27**

Introduction

We originally had to decide between three candidates for our project. At first it was a toss up between quite a few, but we soon narrowed it down to three.

The three candidates we chose for our project were (as stated in our GitHub Wiki):

1. OWASP - The project we would prefer to work on. One of our group members is interested in software security, so there's a good chance that interest can spread to the other members. Also, there is good documentation and multiple sub-projects to work on under it. Most projects are on github as well.
2. Open Data Kit - This one seemed to have more documentation and resources for developers than many of the other projects listed. It seems like a good product for data management and may be useful in the future. The tools they already have may be helpful as well.
3. FreeNet - The anti-censorship subject matter looks like it would be an interesting project to work on. This project also has good instructions on how to get the source code.

In the end, we chose OWASP, and more specifically, OWASP's EnDe section.

Chapter 1: OWASP's EnDE

The Main Project

Since we've decided to start with OWASP, and due to the large number of projects within OWASP, we've selected a sub-project, EnDe.

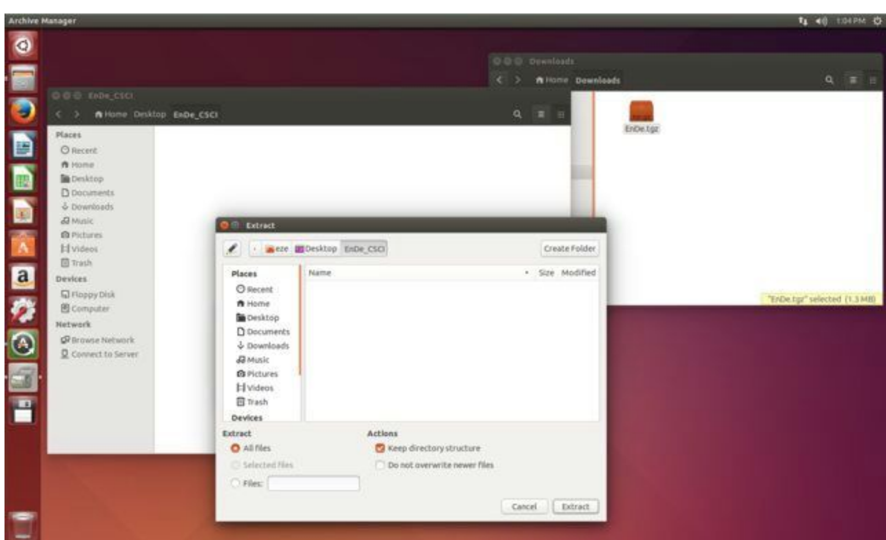
OWASP EnDe Project is a project that provides an encoder, a decoder, a converter, a transformer, calculator all at the click of a mouse. It contains a collection of functions for various codings, encodings, decodings and conversions used world wide. One of the aims of the project was to meet the requirements for HTTP/HTML-based functionality. Copy & paste must be possible, and some functions should be able to be called with a single click.

Since there is no single way to encode, decode, convert, etc.. any one thing, the project itself is coded in several languages, and is run by HTML. The file "index.html" pulls from the resources in the containing folder to run all the functions this project intends to use, and runs simply in a web browser, as to make the project widely available to everyone intending to use it regardless of which language development kits they have installed.

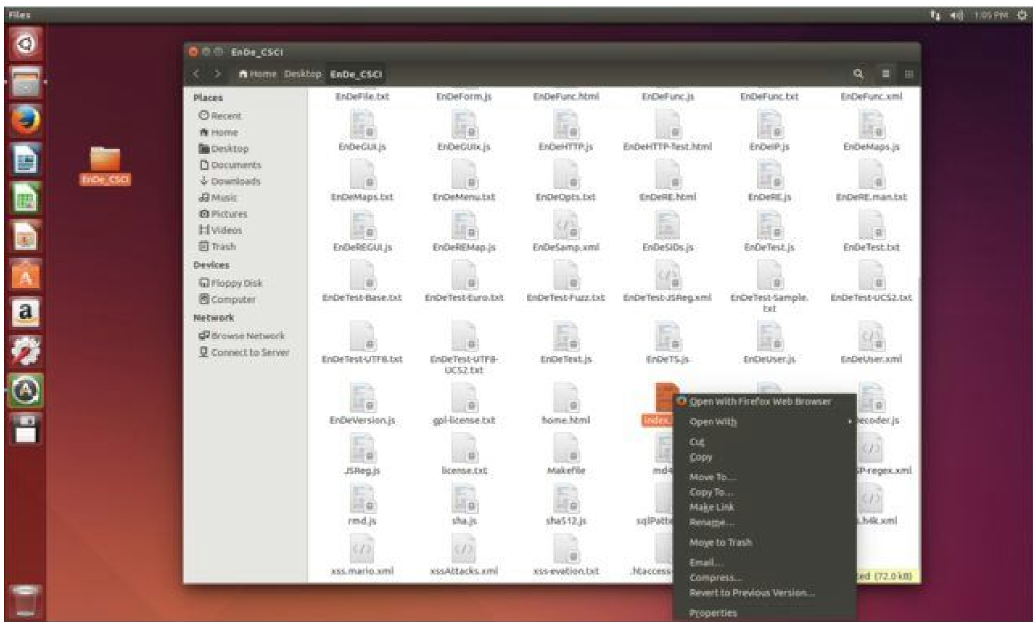
Installation:

OWASP EnDE is made to be easy to use, so installation should be just as easy. The only requirements of installing the project is that you have the web-browser Firefox installed, and a single folder to install the files into. Firefox is "required" since Chrome and other web browsers have security features that inhibit some functions of EnDe, as well as some not being supported on Linux. Once you have the files from the EnDe site downloaded, extract them and place them in the install folder. Open up EnDe by clicking on "index.html", and you are done.

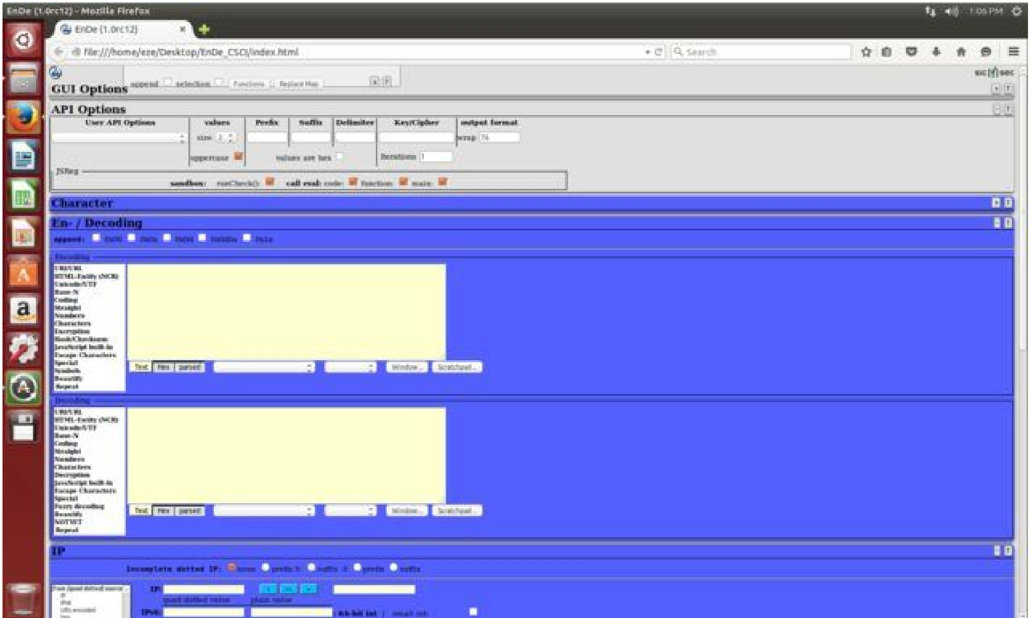
- **Step 1:** download the zipped file needed, and extract it to an installation folder you've created.



- **Step 2:** Open index.html, located in the installation folder, with your web browser.



- **Step 3:** Enjoy using the project EnDe!



Initial Testing:

Simple encoding and decoding worked very well. Placing in a few of our own samples for Base64 encoding and decoding showed up exactly as they should have. The text "VGVzdCBFbmNvZGluZw==" decoded to "Test Encoding" and vice versa when encoding.

Tests: Other examples with given results were used to ensure that the project was working correctly.

Simple Example 1: Using the string "Euro"

CRC-32 ------------- 1fb0ba15
base64 ------------- RXVybw==
urlUTF-8 ----------- Euro
MD5(hex) ----------- 3E823FAC7473E42888932C7761C224FC

- Trying these encodings and hashes in various tools should always return the same result. Not very surprising, as it's expected to do so.

Simple Example 2 Replacing the 'E' in "Euro" with an '€' which results in "€uro"

CRC-32 ----------- b4c7b4a7
base64 ----------- IKx1cm8=
urlUTF-8 --------- %e2%82%acuro
MD5(hex) --------- 08c1d37a0b3394da278b77116cc4e615

- Trying these in various tools will likely return different results depending on the tool used.

Chapter 2: EnDe and Test Cases

The Testing Process: Major Phases

Since testing actual encryption and decryption methods of EnDe would be a larger, and more daunting task, we decided that one of the first things we want to test is the compatibility of EnDe with various web browsers and HTML interfaces. One method of testing, or automating these tests is through a "browser-based regression automation" system called Selenium. It comes with a Web driver, for making the browser work the way it is supposed to, and an IDE to create quick bug reproduction scripts as needed. This way, we could launch a browser and attempt to recreate bugs much quicker than manually re-creating each one as we find them. Selenium can be found here: http://www.seleniumhq.org/

One of the secondary options for major phases of testing is what was previously mentioned, encryption and decryption. We should be able to tell how accurate it is (can it be replicated), and if the information decrypted will be the same as the information decrypted regardless of the browser interface. Again, Selenium will help with this process, but again, this is secondary to testing browser interface, as encryption and decryption methods are well known, and not easily re-written (ie: most methods already in place are the best at what they do)

Testing schedule:

September 30 - October 22: Design and build an automated testing framework

October 23 - November 12: Complete implementation of testing framework

November 13 - November 24: Inject faults into code to fail 5 tests, analyze results

November 25 - December 1: Complete all tests by December 1st deadline

Hardware and software requirements:

We will need various platforms to test EnDe on. For a time, "Chromium" was the web browser of choice for Linux users who wanted to use Google's "Chrome" browser. It was not official, but the closest thing to Chrome Linux users could obtain. That in mind, we will need Linux, Windows, and OSX for various tests. Each web browser that is on various operating systems acts differently based on the system it runs on. Currently, the official Linux build for Chrome is "stable" but does not contain all of the features that Chrome on a Windows or Mac operating system would have. This applies for each browser on these operating

systems, including Mozilla Firefox, Apple Safari, Microsoft Internet Explorer, and SeaMonkey, as well as the previously mentioned Chrome browser.

RAW Hardware/Software Requirements:

- Compatible Mac, Windows, and Linux operating systems
- Mozilla Firefox, Apple Safari, Microsoft Internet Explorer, SeaMonkey, and Chrome web browsers
- Machines capable of running each of these softwares either on 64 bit or 32 bit architecture (whichever is made more widely available by the browser developers)
- Python (versions below 3.x)
- Selenium to run the GUI tests (in terminal: “sudo easy_install selenium”)

Test recording procedures:

Running the tests should automatically generate a report of which tests pass and which fail.

Test Results

Run on 11/11/15 at 13:42:02.

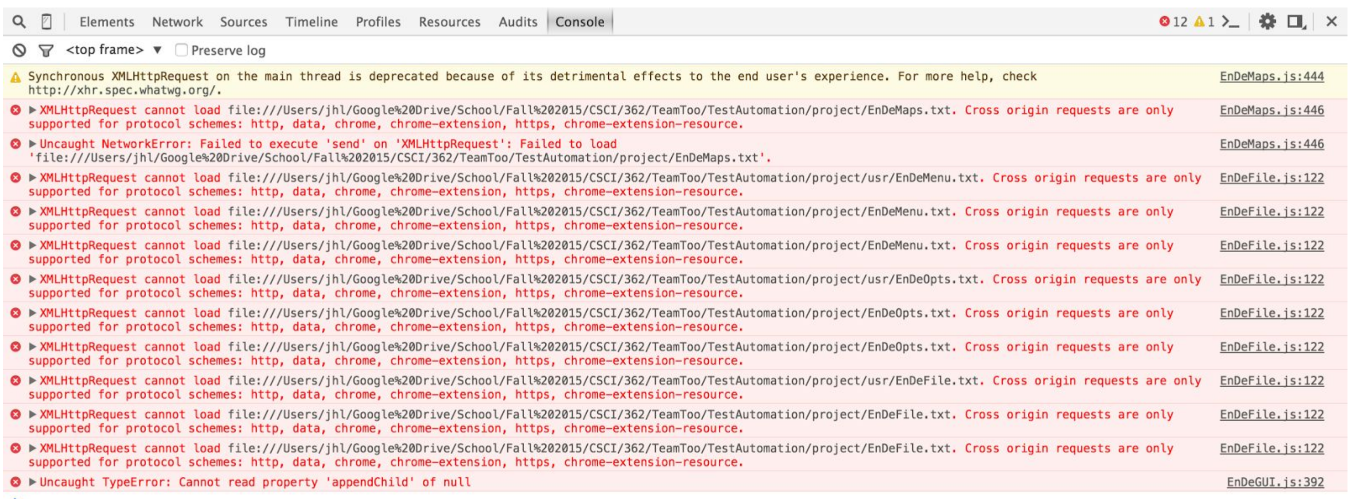
Test ID Number	Requirement Being Tested	Component Being Tested	Method Being Tested	Test Inputs, including command line arguments	Expected Outcomes	Actual Outcome
1	Encode an input string into base64, display the resulting string.	the EnDe graphical interface - index.html	NA	Euro	RXVybw==	RXVybw==
2	Convert an input string into hexidecimal, display the resulting hexidecimal value as a string.	the EnDe graphical interface - index.html	NA	Hex Test	4865782054657374	48,65,78,20,54,65,73,74
3	Convert an integer to its binary value and display the resulting binary representation.	the EnDe graphical interface - index.html	NA	42	101010	101010
4	Encode an input string into Rot13, display the resulting string.	the EnDe graphical interface - index.html	NA	Testing	Grfgvat	Grfgvat
5	Convert an input string into morse code, display the resulting string.	the EnDe graphical interface - index.html	NA	SOS	... ___ ...	... ___ ..

An example of the test results generated in our ‘reports’ folder.

It is an ‘html’ file that opens with most browsers.

Tested items:

First off, in our testing, Chrome has already had issues with EnDe. We have tested with various other browsers such as Firefox and IE (yes, it’s terrible, we know), and issues occur in both Chrome and IE, both dealing with security detecting scripts as either malicious, or simply not running them. It may be a protective feature; however, it inhibits EnDe from fully functioning. (see graphic below)



The current errors found in Google Chrome, all seeming to stem from cross origin errors

Secondly, since Chrome is more unstable on Linux, it will not install on some distributions of Linux, and must be tested to the best of our ability. Ubuntu is currently having issues installing Chrome, and we have already had the web browser crash twice on launch. For now, we are sticking to testing it on Windows 7 and Mac OSX. We have found that in both operating systems, EnDe has a similar bug generated by Chrome. Disabling security features all together fixes the issue, but is obviously not a complete fix. We are still in the process of going through each security feature, and noting which one causes the bug or crash of the system.

Since a large majority of the HTML runs JavaScript, we can test the JavaScript code itself or we can test how well it is implemented in the different web browsers. As far as testing went, our initial test cases, and the tests are below:

5 / 25 Test Cases - Initial Tests:

Section	Description
Test Case #	1
Summary	Verify that the base64 encoding works through the EnDe interface
Prerequisites	Web browser can properly load EnDe suite (currently, not Chrome)
Procedure	1. User enters encoding text into the Encoding Text area
	2. User hovers over 'Base-N' option in left-hand pane
	3. From the pop-up menu the User selects the 'base64' option
Test Data	Encoding Text: Euro
Oracle	Decoded Text: RXVybw== (found using python's base64 algorithm, not EnDe's)

Section	Description
Test Case #	2
Summary	Verify EnDe's morse code encoding
Prerequisites	Web browser can properly load EnDe suite (currently, not Chrome)
Procedure	1. User enters encoding text into the Encoding Text area
	2. User hovers over 'Symbols' option in left-hand pane
	3. From the pop-up menu the User selects the 'Morse' option
Test Data	Encoding Text: SOS
Oracle	Decoded Text: ... ___ ...

Section	Description
Test Case #	3
Summary	Verify that the hexidecimal conversion works through the EnDe interface
Prerequisites	Web browser can properly load EnDe suite (currently, not Chrome)
Procedure	1. User enters decimal value into the Encoding Text area
	2. User hovers over 'Straight' option in left-hand pane
	3. From the pop-up menu the User selects the 'Octal' option
Test Data	Encoding Text: 2345
Oracle	Decoded Text: 929

Section	Description
Test Case #	4
Summary	Verify that HTML escaping works through the EnDe interface
Prerequisites	Web browser can properly load EnDe suite (currently, not Chrome)
Procedure	1. User enters SQL code into the Encoding Text area
	2. User hovers over 'Escape Characters' option in left-hand pane
	3. From the pop-up menu the User selects the 'HTML' option
Test Data	Encoding Text: <script src="http://code.jquery.com/jquery-2.1.4.min.js"></script>
Oracle	Decoded Text: <script src="http://code.jquery.com/jquery-2.1.4.min.js"></script>

Section	Description
Test Case #	5
Summary	Verify that the ROT13 encoding works through the EnDe interface
Prerequisites	Web browser can properly load EnDe suite (currently, not Chrome)
Procedure	1. User enters encoding text into the Encoding Text area
	2. User hovers over 'Coding' option in left-hand pane
	3. From the pop-up menu the User selects the 'ROT13' option
Test Data	Encoding Text: notforspammers@example.com
Oracle	Decoded Text: abgsbefcnzzref@rknzcyr.pbz

Constraints:

As mentioned before, we've had issues with Linux and Chrome. Other web browsers are not having as much of an issue, but due to the lower user count of Chrome on Linux distributions, Google does not have it on their top priority to fix. We will either have to wait for a patch, or find "work-arounds" ourselves. This will affect our time constraints, and we will have to worry about fixing Chrome for Linux just to test what we already have. However, Chrome works flawlessly on other operating systems, and there should be no issue there.

Chapter 3: Implementing Testing Framework

Automated Testing Framework: Experiences

For our automated testing framework, we are using Python as our scripting language. Our runAllTests.py script reads through all of the files in the testCases directory and adds all the values in each file to a dictionary that is then pushed onto a list of other test-case dictionary (so a list of dictionaries).

Using the file names, runAllTests.py then uses those names to run the tests themselves from the testCaseExecutables directory. The specific dictionary holding data for that test case is sent along to the text case executable on the command line. This is done by serializing the dictionary in runAllTests and decoding the serialization once it reaches the specific test case executable. This way the data sent to the test case can be dynamic based on the test case's needs. Each test writes to a results file that will eventually be used to show the results of all tests in your browser, once all tests are run.

Architecture:

The folder structure for the project is as follows:

```
/TestAutomation
   /project
      /src
      /bin
   /scripts
      runAllTests.py
   /testCases
      testCase1.txt
      testCase2.txt
      testCase3.txt
      testCase4.txt
      testCase5.txt
   /testCasesExecutables
      testCase1.py
      testCase2.py
      testCase3.py
      testCase4.py
      testCase5.py
   /temp
      results.html
   /oracles
   /docs
      README.txt
   /reports
      testReport.html
```

How To: Framework

To run the automated testing framework, the instructions are simple:

1. From the terminal, change to the TestAutomation directory
2. From TestAutomation, type the following command: "python ./scripts/runAllTests.py"
3. You will see several tests pop up on your browser, then you will see a results page for which tests passed and which tests failed.

Test Results

Run on 11/11/15 at 13:42:02.

Test ID Number	Requirement Being Tested	Component Being Tested	Method Being Tested	Test Inputs, including command line arguments	Expected Outcomes	Actual Outcome
1	Encode an input string into base64, display the resulting string.	the EnDe graphical interface - index.html	NA	Euro	RXVybw==	RXVybw==
2	Convert an input string into hexidecimal, display the resulting hexidecimal value as a string.	the EnDe graphical interface - index.html	NA	Hex Test	4865782054657374	48,65,78,20,54,65,73,74
3	Convert an integer to its binary value and display the resulting binary representation.	the EnDe graphical interface - index.html	NA	42	101010	101010
4	Encode an input string into Rot13, display the resulting string.	the EnDe graphical interface - index.html	NA	Testing	Grfgvat	Grfgvat
5	Convert an input string into morse code, display the resulting string.	the EnDe graphical interface - index.html	NA	SOS	... ___ ...	... ___ ..

Again, here is our sample results after running the tests.

These are also our first 5 test cases.

Some More Testing - Summarized:

Here is our sample text used to test some of the encoding and decoding, as well as encryption and decryption:

jkhviuyv3rcsdf832099874%!$#5*_asldfkjasdhfibv==   lk;'op,huoy8,,

These Base(XX) encoding functions work fine as these are the outputs that return the same when decoded:

Base64:

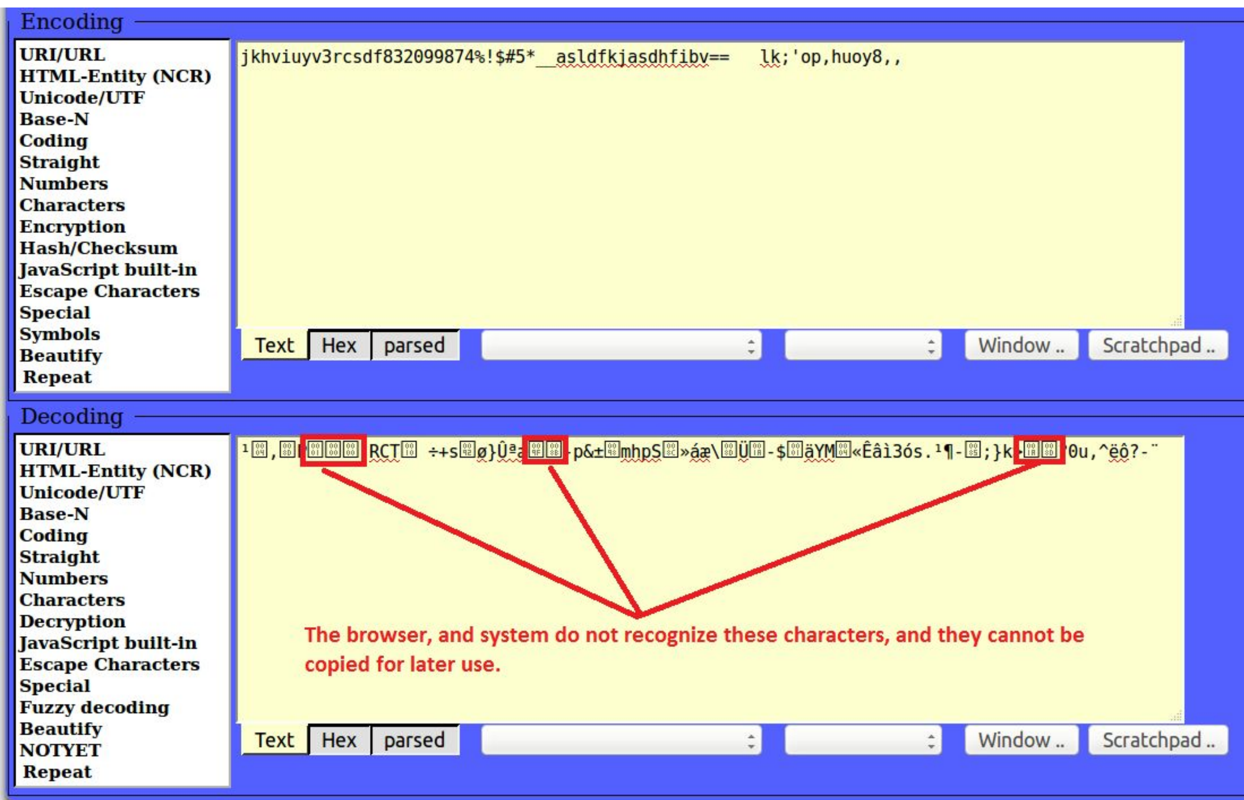
amtodml1eXYzcmNzZGY4MzIwOTk4NzQlISQjNSpfX2FzbGRma2phc2RoZmlidj09ICAgbGs7J29wLGh1b3k4LCw=

Base85:

rQ7pbxCcR@;SCZ1uR@4|9G+rLNB@pQcB@;pQidnWG8D+^uBD7hzuMB0

And so on…

These encoding methods worked fine, however, encryption and decryption tests were somewhat, difficult. The encryption functions sometimes returns characters not identified by either the browser, or the system running the tests, and cannot be placed back into the function correctly. The system records nonsensical characters as the encoded text when copied, or pulled from the page, and outputs content in a similar fashion, as seen below:



These were the generated (output) hex codes when selecting 'Hex' as an output method. Notice how attempting to decode the text returns no results in the above box.

Errors like this occur for all except for ***BLOWFISH*** and ***BLOCK (TEA) ESCAPED*** encryption.

ENDE provides a ENDEtest.js, and a ENDEtest.txt file, however, manually trying the encryption still yields no results. Either this is an issue with the browser and javascript IDE (netbeans), or there may be something wrong with the encryption methods used (ie: javascript/python/C/C# encryption methods) - It would be safe to assume the first would be the issue, that the characters requested for the text are simply not found on the host's system.

To build on this, the following text (tested) provides no results when decrypting AES text using the same method. Keep in mind we are using the same sample text as mentioned above:

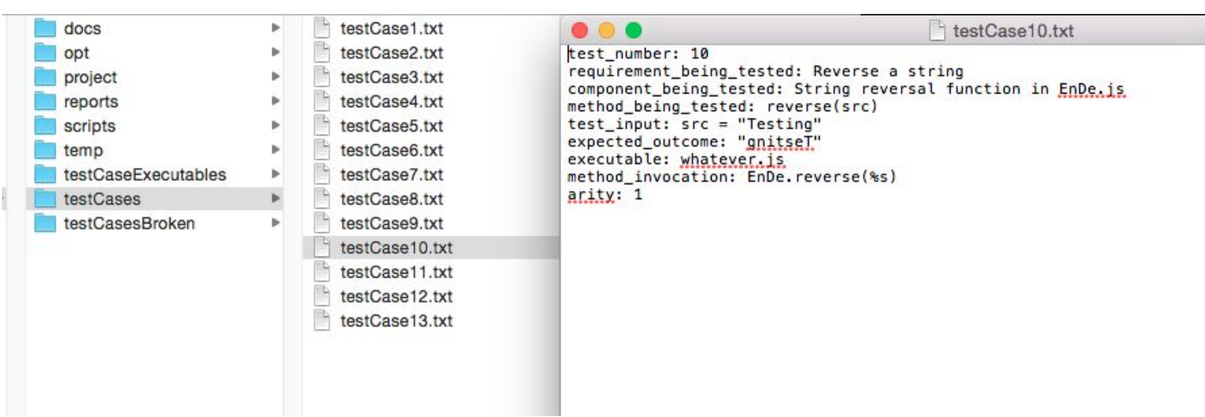
```
Method     Encryption (expected outcome)
#----------+-----------------------------------------------------
aes128     \xa3\x98\x17\xc9\x26\x01\x00\x00\x2d\x7c\x4d\x3b\xfe\x1d\xc2\x01\x07
aes192     \xd5\x53\x18\xc9\x26\x01\x00\x00\x2d\xd0\x9a\x62\x0f\xf2\x75\x90\xc0
aes256     \xda\xf1\x18\xc9\x26\x01\x00\x00\x2d\x7d\x6f\x93\xb7\x01\xde\xed\x7a
aes128r    \x14\x41\x19\xc9\x26\x01\x00\x00\x2d\x90\x64\x70\x6c\xfa\x19\xed\x4f
aes192r    \x23\x8a\x19\xc9\x26\x01\x00\x00\x2d\x65\xbe\x34\x94\xda\x41\x4e\x9c
aes256r    \x0e\xe6\x19\xc9\x26\x01\x00\x00\x2d\x69\xb5\x0c\x8f\x37\x09\x4d\x32
teaesc      !1!!227!!7!!130!!159!!218!!26!!240!
teacor     \x01\xe3\x07\x82\x9f\xda\x1a\xf0
tearaw      \x01\xe3\x07\x82\x9f\xda\x1a\xf0
```

Currently, we have found no way to fix this issue, and it is still a work in progress. However, we can expect to test the methods and code without using the front facing part of the software, but we'll get to that in a later chapter.

Chapter 4: Automated Testing

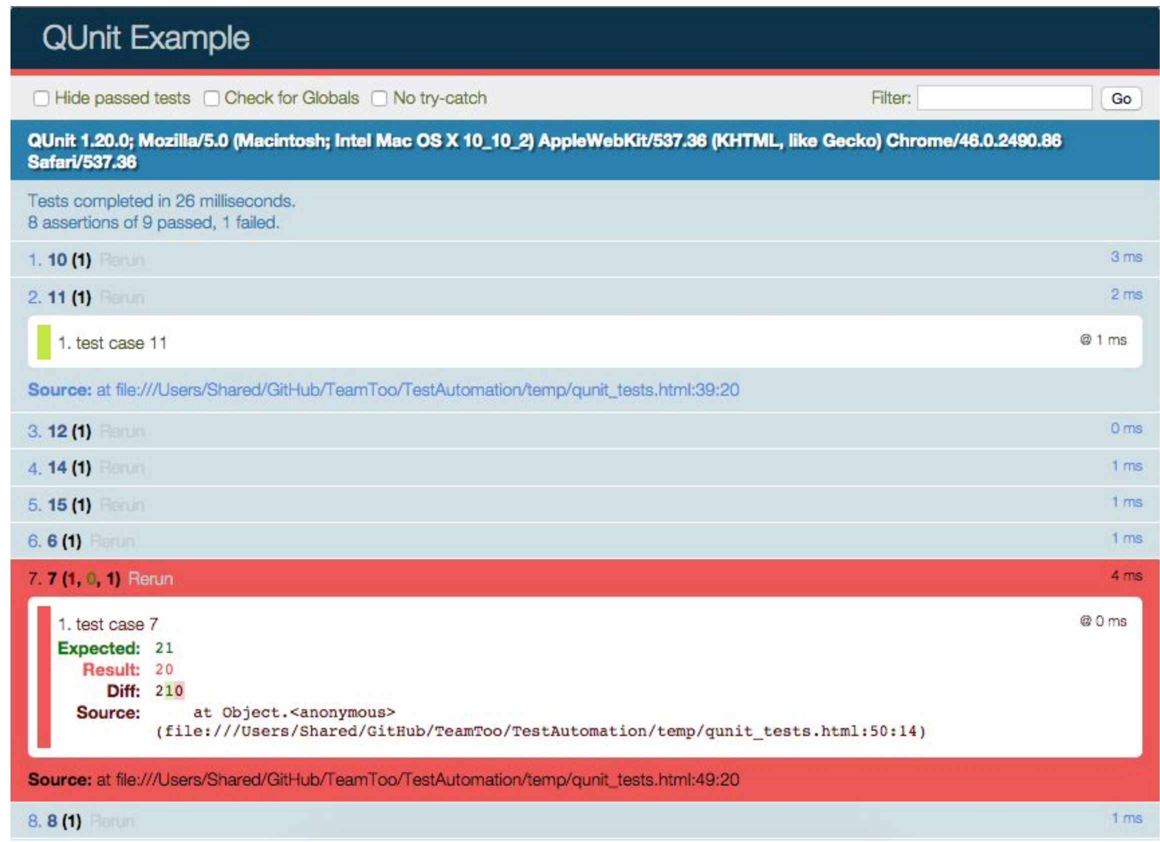
Automated Testing Framework: Adding New Logic for New Tests

Some changes were made in how our testing framework works since Chapter 3. Now, instead of using the names/numbers of the tests to run separate files in the testCaseExecutables directory, we run all tests using information in the .txt files found in the testCases directory.



These files contain all information needed to execute the tests without needing repeated code in the testCaseExecutables files. We use two different types of cases in our testing: tests written in Python that use Selenium to simulate interactions in a browser window to test the user interface, and tests on the JavaScript code that EnDe is largely written in.

In runAllTests.py, we check whether we need to be running Python code to test the user interface or JavaScript code to test methods directly. For the JavaScript tests we used QUnit, which is a JavaScript unit testing framework. We were able to get the results of whether a test passed or failed using QUnit, then used those values in our report.



Here is a sample QUnit output that appears after running all the tests.

Almost all of the functionality for our automated testing framework is in runAllTests.py. This includes searching through the testCases directory, pulling out relevant information from the files and invoking the appropriate methods, and writing and saving the report. This was done mainly to avoid repeated code in separate executable files for each test case. Below is a sample run of our ‘runAllTests.py.

```
python ./scripts/runAllTests.py
Running Test Case 1...
        test input: Euro
        expected outcome: RXVybw==
        EnDe result: RXVybw==
        Equal? True
Running Test Case 13...
        test input: $Monies$
        expected outcome: 1062582A77640053
        EnDe result: 1062582A77640053
        Equal? True
Running Test Case 2...
        test input: Hex Test
        expected outcome: 4865782054657374
        EnDe result: 48,65,78,20,54,65,73,74
        Equal? False
Running Test Case 3...
        test input: 42
        expected outcome: 101010
        EnDe result: 101010
        Equal? True
Running Test Case 4...
        test input: Testing
        expected outcome: Grfgvat
        EnDe result: Grfgvat
        Equal? True
Running Test Case 5...
        test input: SOS
        expected outcome: ... ___ ...
        EnDe result: ... ___ ..
        Equal? False
Compling JavaScript tests...
Test 10...
Test 11...
Test 12...
Test 14...
Test 15...
Test 6...
Test 7...
Test 8...
Test 9...
Compiled...running
```

Python Tests

All infos used are from each testCase#.txt

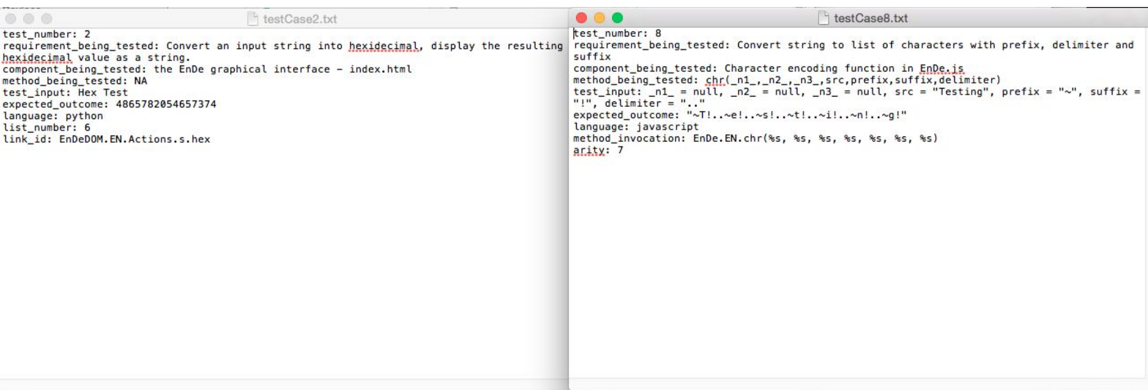
Javascript Tests

This is a sample run on the Mac OSX Terminal.

This is how it would show on Ubuntu's as well, since both are 'linux based' operating systems.

The Test Case Text Files

Our test case text files contain different information depending on whether the information will be used in a Python user interface test or a JavaScript test for functions. All test case files contain a test number, requirement being tested, component being tested, test input, and expected outcome (oracle). For the user interface tests, we also include a list number which allows us to locate which item in the list of types of encryption we need to click in the user interface and a link id that allows us to click the proper specific encryption/decryption method in a sub-menu under that type. For our JavaScript tests, we include the method being tested and how to invoke the method. We also include arity, so if a user enters too many arguments it will only accept the first ones up to the amount required.



Errors and Manual Checks

We did find some issues with using the test case files, however. One issue arose when implementing automated testing for values that were returned not as UTF-8 encoded characters. Of course, this means that the test did fail, but it would also cause the scripts to break, and it would cause issues with the files being written (ie: the results.html). Since this was causing the automated tests to fail, and scripts to be broken, this test had to be checked manually to verify the issue.

The culprit test case:

testCase10_broken.txt — Edited

```
test_number: DON'T USE THIS ONE
requirement_being_tested: AES-128 encoding - verifiying that output matches expected from
web browser
component_being_tested: the EnDe graphical interface - index.html
method_being_tested: AES-128 encoding
test_input: heureca!
expected_outcome: \xa3\x98\x17\xc9\x26\x01\x00\x00\x2d\x7c\x4d\x3b\xfe\x1d\xc2\x01\x07
executable: testCase9.py
list_number: 9
link_id: EnDeDOM.EN.Actions.s.aes128

SERIOUSLY. DON'T USE IT. IT WILL BREAK EVERYTHING.
```

The error(s) being thrown:

OSX:

```
Running Test Case 10...
        test input: heureca!
        expected outcome: \xa3\x98\x17\xc9\x26\x01\x00\x00\x2d\x7c\x4d\x3b\xfe\x1d\xc2\x01\x07
        EnDe result: #µçùP-X,_<Ôôû
        Equal? False
Traceback (most recent call last):
  File "./scripts/runAllTests.py", line 129, in <module>
    testCaseData['expected_outcome'], val))
UnicodeEncodeError: 'ascii' codec can't encode characters in position 302-304: ordinal not in range(128)
```

Linux:

```
Running Test Case 9...
        test input: heureca!
        expected outcome: \xa3\x98\x17\xc9\x26\x01\x00\x00\x2d\x7c\x4d\x3b\xfe\x1d\xc2\x01\x07
        EnDe result: ùP È.ÔòÐD¶
        Equal? False
Traceback (most recent call last):
  File "./scripts/runAllTests.py", line 129, in <module>
    testCaseData['expected_outcome'], val))
UnicodeEncodeError: 'ascii' codec can't encode characters in position 300-303: ordinal not in range(128)
```

Notice how the output recorded in the console only shows 'some' non-standard characters. Other characters aren't shown because the console can't display them. Below, we have the manual input on the EnDe html page.

Sample manual test from the GUI:

Notice in this one the non-standard characters that are shown as boxes. These are what is attempting to be added to the results.html file. The only reason they show up here as boxes is because that is what Firefox has implemented to catch non standard characters like these.

In order to prevent the error from popping up while testing, and causing issues in doing so, entering any value other than an numerical one for the test case number will cause the testing to stop before the values are sent to test. It's a quick and simple way to stop someone from running a test that might pull it from

```
Traceback (most recent call last):
  File "./scripts/runAllTests.py", line 73, in <module>
    testCaseIndex = int(testCaseData['test_number'])
ValueError: invalid literal for int() with base 10: "DON'T FREAKING USE THIS ONE"
```

GitHub.

Sample of the error thrown with an improper test number for the test case text file.

Chapter 5: Fault Injection

Making a plan for code to fail, and how we did it:

For adding the faults in the code, we created "add_faults.sh" and "remove_faults.sh". These two files swap out our code with the faulty code in a quick and simple command './add_faults.sh' or './remove_faults.sh' (see screenshot at the bottom for example execution)

Faulty Tests:

We made the following modifications to the files EnDe.js, EnDeCheck.js, EnDeText.js, and what is swapped with the 'add_faults.sh'. Below are the tests we can now expect to fail based on the faults:

Test 10: Modified the reverse function in EnDe.js to only go to length - 1

```
Changed line from:
var i = src.length;
```

```
Changed line to:
var i = src.length - 1;
```

Test 11: Modified the val2num function in EnDeCheck.js to replace non-digit characters with '-' instead of ''

```
Changed line from:
this.val2num  = function(src) { return((src+='').replace(/[^0-9]/g,'')); };
```

```
Changed line to:
this.val2num  = function(src) { return((src+='').replace(/[^0-9]/g,'-')); };
```

Test 12: Modified the atbash function in EnDe.js to change from -1 to -2

```
Changed line from:
bux += String.fromCharCode((((78-ccc)*2)-1+ccc));
```

```
Changed line to:
bux += String.fromCharCode((((78-ccc)*2)-2+ccc));
```

Test 15: Modified the DELwhite function in EnDeText.js to replace white space with '.' instead of ''

Changed line from:
case 'txtDELwhite': bux = bux.replace(/[\t \r\n]/g, ''); break;

Changed Line to:
case 'txtDELwhite': bux = bux.replace(/[\t \r\n]/g, '.'); break;

Test 18: Modified the stibitz function in EnDe.js to have incorrect value for case 1 from '0100' to '1111'

Changed line from:
case '1' : bux += '0100' + delimiter; break;

Changed line to:
case '1' : bux += '1111' + delimiter; break;

The Sample Run:

These fault changes were saved in EnDe[Check|Test].js.fault, and the preserved, passing versions were saved in EnDe[Check|Test].js.normal and the two sets of versions are copied in and out of the actual files run, EnDe[Check|Test].js. These operations were combined into the two files add_faults.sh and remove_faults.sh which allow for easy switching between the two states.

```
eze@ubuntu:~/Desktop/TeamToo-scratch/TestAutomation$ ./add_faults.sh
eze@ubuntu:~/Desktop/TeamToo-scratch/TestAutomation$ python scripts/runAllTests.
py 10 11 12 15 18
Running Test #10...
        FAILED
done
Running Test #11...
        FAILED
done
Running Test #12...
        FAILED
done
Running Test #15...
        FAILED
done
Running Test #18...
        FAILED
done
eze@ubuntu:~/Desktop/TeamToo-scratch/TestAutomation$
```

Here, you can see a screenshot of the output of individual tests.

These are also our expected faults.

```
eze@ubuntu:~$ cd '/home/eze/Desktop/TeamToo-scratch/TestAutomation'
eze@ubuntu:~/Desktop/TeamToo-scratch/TestAutomation$ python scripts/runAllTests.
py
Running Test #1...
Base64
        PASSED
done
Running Test #2...
Hex
        PASSED
done
Running Test #3...
integer to binary
        PASSED
done
Running Test #4...
Rot13
        PASSED
done
Running Test #5...
Morse
        FAILED
done
Running Test #6...
        PASSED
done
Running Test #7...
        PASSED
done
Running Test #8...
        PASSED
done
Running Test #9...
        PASSED
done
Running Test #10...
        PASSED
done
Running Test #11...
        PASSED
done
Running Test #12...
        PASSED
```

As mentioned, running the command will run all of the tests. The output would look similar to this when running all of the test cases that have been created. This is without using the injected faults.

Update to the original testing framework:

As a side note, we had updated and 'streamlined' our code at this point. Now, each test case runs as 'its own executable'. (see picture above) This allows us to test individual cases over running the entire test each time for one update. For instance 'python ./scripts/runAllTests.py 10 11 12 15 18' will only run those 5 tests. (see previous screenshot for sample execution of individual tests) Also updated the reports so that the chart contains 'pass' and 'fail'. Again, this still runs with the same command, it is just ‘streamlined’ now.

Chapter 6: Overall Experiences

The overall experience for the project wasn't overly terrible. Communication between group members was always a hassle, but somehow or another, we always got our work done as needed (and sometimes more) when it was due. But, that being said, we have had a group member fall short of what was expected. Picking up the slack is always a pain on the group members that have to, and this time was no exception. Regardless of the one exception, everyone who submitted content towards the project did a great job of it, and we are happy of that.

As far as figuring out the testing procedures, you gotta stumble before you can walk, let alone go into full on sprint. Our first obstacle was selecting a browser. There were plenty of options, but we needed something stable. Tackling that was fairly simple.

Next, issues arose when testing, in regard to other group members. Some of the group members got "lost" when we implemented some testing framework. Getting them up to speed was simple fix with a single 'group meeting'. A similar issue was getting everyone on the same page with the operating system. Some virtual machines acted funny, and we had to make sure everyone got the correct settings in place. But again, once it was all said and done, it actually was fairly simple.

We then had a few other issues with testing. Group members would 'push' to GitHub while others are 'pulling' from it, causing one to get improper code or information. Again, miscommunication. Was a simple fix by getting them to redownload the master branch. However, that still didn't fix genuine errors. Fixing those required another unscheduled group meeting, but after that, things were back to normal.

The testing was the fun part, once the framework was implemented. Adding in test cases and seeing each run, then pass or fail was the best. You can finally see the work done coming to fruition. And, since our testing framework was stable enough, adding new tests became a breeze, freeing up a larger amount of time for all group members.

Lastly, the deliverables were the slowest part. Each group member would submit content, and then we would have to format it into one comprehensive report. It may have been tedious, but getting everything into 'that one final report' was gratifying once completed.