**Index and description of AG Locator tool files and directories:**

**help.html** – Static HTML page displaying usage/help information

**index.html** – Main landing page of site. Contains search box linked to search.php

**modifycontent.php** – Page used to modify existing entries. Accessed from viewentry.php when clicking the “Modify this entry” button. This page renders its own HTML inline (which looks very similar to the static html pages used for other functions), but is a PHP-format file as it invokes actions on-load to fill fields from the entry that the user selected to modify.

**requestkey.html** – Page used to input email address and generate an author key be emailed to the address entered. Linked from modifycontent.php and help.html pages.

**success-add.html –** Static HTML page displayed after a new entry is successfully added to the database.

**success-del.html –** Static HTML page displayed after an entry is successfully deleted from the database.

**success-key.html** – Static HTML page displayed after an author key request is successfully processed.

**success-mod.html** – Static HTML page displayed after an existing entry is successfully updated (modified) in the database.

**updatecontent.html** – Page used to add NEW entries.

**viewentry.php** – Page used to display full details of an entry. Expects an entry’s ID to be submitted as an argument, then pulls data for that entry on-load to build form. Renders its own HTML inline similar to modifycontent.php description.

**/assets/**

**favicon.ico** – Site icon (favicon) that appears in browser tab.

**upvote.png** – Not used in AG Locator. See also: *votesystem.js*

**downvote.png** – Not used in AG Locator. See also: *votesystem.js*

**css/**

**bootstrap.css** – CSS definitions to support elements from Bootstrap framework.

**style.css** – CSS definitions to support general style elements.

**style-responsive.css** – CSS definitions to control responsive style elements.

**table-responsive.css** – CSS definitions to control table rendering.

**font-awesome/** -- Font support files.

**fonts/** -- Font support files.

js/ -- Contains Javascript files for supporting functions that use bootstrap and jquery.

**/php/**

authorkey-salts.php-example – Contains an example of author key salts, the static values used in conjunction with author email address for the Author Key algorithm. Other PHP scripts that need to calculate the author key will reference authorkey-salts.php and load it (“require\_once authorkey-salts.php”) so the static salt values can be used. The actual authorkey-salts.php file should be configured and stored in a location not accessible by the web server (/var/www/secretfiles/cheatsheet/authorkey-salts.php is used from the PROD cheatsheet loation).

**calc-updatekey.php** – Used by request-key.html. This script contains the source code that generates unique author keys via a triple-salted double hash function. Queries database for existence of a provided email address. If that email address is in the database, it generates the author key for the provided email address, then renders the contents of an HTML email which contains the generated author key output. The email is sent via a call to inbuilt PHP mail() function, also known as sendmail, which relies on the server’s internal mailserver configuration. Presently the server is configured to send mail via SMTP on anonymous account using SAS’ internal mailserver, *mailhost.fyi.sas.com* . Port 25.

**db.php-example** – Contains an example of connection information and credentials to access database used by AG Locator. Most other PHP scripts in the tool reference db.php and load it (“require once /path/to/db.php”) so database operations can be performed. The actual db.php file should be configured and stored in a location not accessible by the web server (/var/www/secretfiles/cheatsheet/db.php is used for the PROD cheatsheet location).

**downvote.php –** Not used in AG Locator. See also *votesystem.js*

**insert.php** – Takes entries from updatecontent.html page’s form POST, used when adding NEW entries to the database. Form contents are run through PHP functions to sanitize and validate. All fields run through mysqli\_real\_escape\_string which escapes database control characters to HTML-safe equivalents and prevents most SQL-related exploits. Required fields are checked for length/consistency to verify no forced-submit of the form has occurred or otherwise passed form validation functions. After all validations, runs SQL INSERT to add new entry, then redirects to success-add.html if all went ok.

**modify.php** – Takes entries from modifycontent.php pages’s form POST and argument of entry ID. Used when modifying existing entries of the database OR when deleting existing entries of the database. Performs initial sanitization and validation as was described on *insert.php*, then queries database for the author email of the specified id (injection exploit mitigation). If id is returned match with expected contents, then generates the expected author key and performs comparison with the user-provided author key value. Because the author key must be compared to the user-provided value, this script contains the same source code from *calc-updatekey.php* that generates the author key via a triple-salted double hash function. If the author key can be validated, references the form submit button selected by the user (‘save modified entry’ or ‘delete entry from database’), then runs SQL on database based on that action. If ‘save modified entry’, performs SQL UPDATE for specified id, overwriting fields that user changed on the form submit, and finally redirects user to *success-mod.html*. If ‘delete entry from database’, performs SQL DELETE for specified id, and finally redirects user to *success-del.html*.

**search.php** – Primary code used in index.html / main page search function. On-load, builds table header for output fields and initializes an HTML table. On user-entry into search box, performs sanitization on input (to mitigate potential SQL-related exploits), then runs SQL SELECT query based on the data entered in search. There is an IF/ELSE which currently always will resolve to ELSE (the IF allows pull to metrics calculation to track user search activity – there was a deprecated PHP function from PHP5 that was used in this code block and did not work when forced to update to PHP7 – rewrite of this IF would be required should that be implemented again. Inside the always-run ELSE, there is still some basic variable assignment to support search-activity tracking that presently doesn’t actually go/get used anywhere. But after that, under the comment ‘//check for results’, is the actual output of the script. The output of SQL SELECT is run through PHP’s htmlspecialchars function to enable browser-accurate formatting of output, then assigned to intermediary PHP data variables (named *d\_<field-descriptor>* ). The database output for ‘notes’ field (additional notes on an entry) is also run through a regex function and PHP preg\_replace in order to render URLs inside the notes as actual/clickable hyperlinks. After setting intermediary data variables for all database output we intend to use, perform PHP str\_replace and inject the intermediary data variables into output variables for the HTML table rows (variable *$o* ). Finally, output *$o* as an echo in PHP, rendering it back to browser output / visible to the user. A catch IF/ELSE is set to where if the results of the query were empty (there were no hits on the search), the output table is left empty and the first field shows an HTML ‘danger’ label (a red box, basically) stating that no matching results were located.

**upvote.php** – Not used in AG locator. See also *votesystem.js*

**/scripts/**

**triggers.js** – Support functions for search triggers. Controls wait times and other aspects to enable live-search functionality (where the results dynamically update as a user enters their search input)

**votesystem.js** – Support functions for upvote/downvote system, which was a feature implemented in the SSLAZY project which AG Locator was cloned from; enabled result sorting based on user-provided feedback. The vote system is not implemented in AG Locator.

**About the author key system design:**

The author key system is a method which allows users to self-moderate their own AG Locator database entries, but prevents the ability to edit another users’ entries. To prevent password-storage requirement of the tool and avoid users forgetting their passwords, the author key system implements a validation similar to the secondary authentication mechanism popular in contemporary 2FA implementations.

In the author key system, a user can enter an email address and have the author key for that email address sent to them. The only place for this email to be sent, however, is the email address they entered. In practice this means that a user can only obtain an author key for an email address for which they are able to access the mailbox of. In other words, a user who can only access their own email address can only access their own author keys.

The calculation of author keys is performed server-side in PHP, after the user submits the POST form on request-key.html page. Because calculations occur on the server, there is no method by which the user could obtain the output of those calculations within their browser as the form does not return any data.

As a practical example of the author key system design/function, consider the following:

* Alice has submitted an entry to the database using her email, [alice@sas.com](mailto:alice@sas.com) .
* Bob has never submitted an entry to the database. Bob’s email is [bob@sas.com](mailto:bob@sas.com) .
* Eve has submitted an entry to the database using her email, [eve@sas.com](mailto:eve@sas.com) .

If Alice would like to edit her entry, she can use the request-key.html page to request her author key. The system will ask which email address Alice wants the author key for. Alice will enter her email, [alice@sas.com](mailto:alice@sas.com) , and submit the page. The system will calculate the author key for [alice@sas.com](mailto:alice@sas.com) (for the sake of the example, let’s say the key was *H3110A1IC3*), then generate an email that is automatically sent to [alice@sas.com](mailto:alice@sas.com) . Alice will be instructed after submitting the page to check her email, [alice@sas.com](mailto:alice@sas.com) , and she should see an email that displays her author key of *H3110A1IC3* . Alice can use *H3110A1IC3* to edit any entries that have an author of [alice@sas.com](mailto:alice@sas.com) .

----

If Bob would like to edit Alice’s entry, he will be prompted for Alice’s author key. Unless Alice shared her author key with Bob, he should not know it, so he would need to use the request-key.html page to attempt and obtain it. If Bob enters his email address [bob@sas.com](mailto:bob@sas.com) into the key request, the system will flag Bob as never having been an entry author. The system will refuse to generate an author key for Bob, and no email will be sent to him.

If Bob was feeling clever and decided he would instead enter Alice’s email address, [alice@sas.com](mailto:alice@sas.com) , the form would submit. The generated author key would be *H3110A1IC3* still, but the email that contained it would be sent to [alice@sas.com](mailto:alice@sas.com), as this was the email address entered in the form. Unless Bob had access to Alice’s email inbox, he would not be able to obtain the Author Key which was sent there.

----

If Eve would like to edit Alice’s entry, she will be prompted for Alice’s author key. Much like Bob, there should not be a reason for Eve to know Alice’s author key.

If Eve requests an author key for her email address of [eve@sas.com](mailto:eve@sas.com), the form will submit, as unlike Bob there are entries in the database which were authored by Eve. The author key will be calculated using the input of [eve@sas.com](mailto:eve@sas.com), which (again for the sake of the example) could be *B0NJ0UR3V3* . Eve will get an email stating that her author key is *B0NJ0UR3V3* .

If Eve tries to edit Alice’s entry and provides the system with her key of *B0NJ0UR3V3* , the system will reject the modification. When Eve submits the form to update Alice’s entry, the system will reference the entry and see that its author email is [alice@sas.com](mailto:alice@sas.com). The author key will be calculated for [alice@sas.com](mailto:alice@sas.com), and then compared to the author key value that Eve submitted in the update form. Essentially, it will check and see that *B0NJ0UR3V3* ***!=***  *H3110A1IC3* , and refuse to update the entry due to a key mismatch.

Much like Bob, if Eve attempted to request an author key and entered Alice’s email of [alice@sas.com](mailto:alice@sas.com), she would not be able to obtain the key as that generated value would only be emailed to Alice’s inbox at [alice@sas.com](mailto:alice@sas.com) (which we hope Eve does not have access to).

----

For clarity it should be noted that the author key system, with no user-defined passwords, should only be considered a single-factor authentication. The underlying authentication from the email system provider that will receive the generated author key, however, could be considered a secondary/2FA or tertiary if that system itself has 2FA implemented.

**How the author key is actually calculated:**

The source code for generating an author key can be found in both *php/calc-updatekey.php* (used to calculate Author Key to send to user email address) and *php/modify.php* (used to calculate Author Key from database field author\_email for comparison). The code is PHP and runs on the server; it is the same code in both files.

The actual author key salt values will control the output (what is considered a valid author key). Changing any/all the static salt values will change the author key for every user. To protect these secret values, the author key salts are stored in a separate file (currently /var/www/secretfiles/cheatsheet/authorkey-salts.php) and loaded only in the PHP scripts that need them. The salt file is stored outside of the web server root, so only code executing on the web server has access to it. There are three salt variables: *$Sugar*, *$Spice*, and *$EverythingNice*.

The author key generation is a basic salting and hashing algorithm, performed in multiple passes to increase obfuscation. It can be considered a triple-salted double-hashed value.

First, the user’s email address is taken as a variable, *$authoremail* .

The value of *$authoremail* is padded with two static values, saved as variable *$parsley*:

* Before the email address, the text represented by variable *$Sugar* is added.
* After the email address, the text represented by variable *$Spice* is added.

The value of *$parsley* is hashed using SHA256, and saved as variable *$sage* .

The value of *$sage* is padded with a third static value, then saved as variable *$rosemary* :

* After the value of *$sage* , the text represented by variable *$EverythingNice* is added.

The value of *$rosemary* is hashed using CRC32, and saved as variable *$thyme* .

The variable *$thyme* is equal to the actual Author Key for the provided email address.

* When generated in *calc-updatekey.php*, the *$thyme* variable is put inside an email and sent via PHP mail() function.
* When generated in *modify.php*, the *$thyme* variable is compared against the user-provided value for Author Key, and the rest of the modify script only runs if they are a match.

--------

While probably not critical for the purpose and environment in which AG Locator is being utilized, it should be noted that the use of static salt values is not considered cryptographically secure. A bad actor could theoretically reverse-engineer the static salt(s) by observing the output value and how it changes for different input text. The design of the Author Key system does however mitigate that in some ways:

* The use of two separate hashes (with some static salts being added before initial hash, and some static salts being added after initial hash) does provide a further level of obfuscation, making a reverse-engineering effort significantly more difficult.
* The generation of output values server-side and internal distribution only to an email address taken as the sole input value makes pattern-observation methods of reverse-engineering significantly more difficult. A bad actor would need to have an actual email address which was deliverable to for every value they needed to test in order to attempt understanding the algorithm.

Likely the attack vector which would be most at-risk of exploit in this system is access to source code stored on server. The PHP scripts performing the calculations of Author Key load the static salt files from a remote source file to obfuscate their values, but does have access to the static salt value variables as plaintext. If a malicious actor was able to access and modify this source code, they could write a program which echoes the salts, then implement the same salts and hashes and would produce the same output values as actual Author Keys. Efforts have been taken using .htaccess files to prevent web server leakage of source code, and the static salt file is stored outside of webserver root location to prevent any potential browser exploits from leaking these values. Future updates to AG Locator tool may wish to improve this design by further obfuscating the static salts in source code (or implementing non-static salts).