MediaWiki (v-1.30.0) ASVS 7-9

May 2018

A first analysis of the application has been done using two static source code analyzers, RIPS and RATS, which are both able to detect common security vulnerabilities in PHP scripts without executing any statement. Initially we've run the tools on the whole application source code to detect common problems automatically and, later, we inspected the source code deeper in order to verify the application satisfies every requirement of the OWASP standard assigned to our group.

Figure 1: RIPS stats

The screenshot above summarizes those warnings generated by RIPS tool. As can be seen, several sinks are detected but, however, once we checked the code they were related to, we realized the tool were generating lot of false positives. For example, command injection warnings are about maintenance scripts, which are not normally executed by the server, the SQL injection warning is due to an external library function call which runs a query without using prepared statements, but however, untrusted parameters used are properly escaped using the available mysql_real_escape_string function.
Additionally, for MediaWiki software static analysis RATS tool was used and set with warning level 3 (maximum). Source code is too vast to perform a full inspection with maximum warning level (too many false positives to handle), so the strategy applied were to use searching keywords to detect and identify the behaviour of the modules to look for to analyze, testing those ASVS submodules

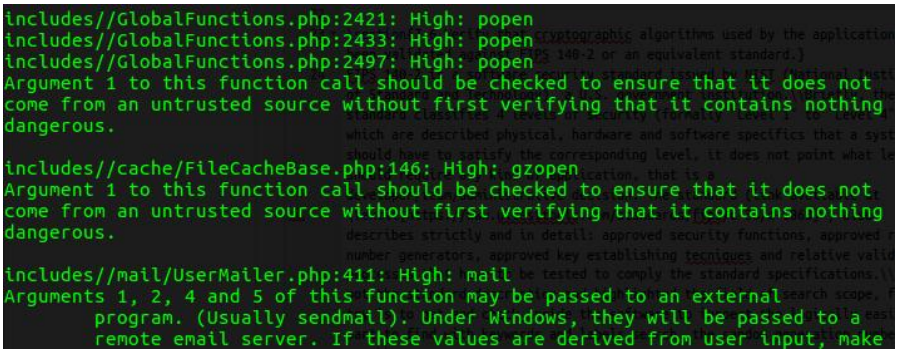

Figure 2: RATS only noticable warnings (5 out of more than 3500).

that from our point of view made sense to statically analyze without need of running the application.

1 7.2 Verify that cryptographic error are handled correctly and padding oracle attack is not guessable

1 Padding oracle attack is a vulnerability strictly related to cryptographic primitives that use Block Ciphers to encrypt data, in short: after block division of data to encrypt getting performed, last block (if number of total bytes to encrypt is not a multiple of single block size) reserve a part of its size to padding data to get the fixed single block size.
Attacks based on padding oracle rely on weak cryptographic algos that, in a recursive way, starting from the padding bytes, can perform decryption of the whole blocks data.

From a static analysis Point of View, search for padding oracle vulnerabilities may consist in detect methods of encryption for: SSL/TLS modules do not contain weak block ciphers inside the default suite list (like the in-famous RC4...) to prevent MITM attacks, password generation modules do not contain padding oracle vulnerable ciphers to prevent password spoofing and subsequent decryption and that password generation errors are handled properly.
Inside **Session** class it is possible to find the whole mechanism used to prevent padding oracle attacks, the session gets created after successful authentication, and all data gets encrypted using *aes-256* algorythm, the two possible variants are *AES-256-ctr* and *AES-256-cbc*, the second one may be vulnerable to padding oracle attacks (ctr variant practically transforms AES block encryption algorithm into a stream one), but inside session there is a mechanism to mitigate padding oracle attacks.
An Encrypt-then-HMAC system (HMAC is a mechanism to enforce data integrity and

source authentication of a packet exchanging a secret key between the two communication end-points included inside the cipher suite to establish for example a TLS tunnel, cipher itself composes of 3 elements: key exchange algorythm, type of block/stream to know how to handle data send/recieved, type of MAC used for guarantee integrity and source of packet) strongly opposes to an hypothetical attacker to gain information about the padding part of a message, this because as implemented in *Session* class, HMAC gets calculated after the encryption of message, this prevents leaks of information about padding or plain text if the HMAC of a specific data packet is guessed right by an hypothetical attacker.
Session class handles also exceptions related to some cryptographic errors, performing a complete scan with RATS on whole */include/session*folder no warnings were given, the class itself even handles as example a case where if OpenSSL is not installed on the machine, *BadMethodCallException* it outputs verbose exception log regard no encrypted exchange messages mechanism and even implementing possibility to suppress those warnings and make session work unsafely unencrypted

2 7.6 Verify that cryptographic algorithms used by the application have been validated against FIPS 140-2 or an equivalent standard.

FIPS 140-2 is a software security standard issued by NIST (National Institute of Standard and Technology), a U.S. government institution.
Briefly, the standard classifies 4 levels of security (formally "Level 1" to "Level 4") in which are described physical, hardware and software specifics that a system should have to satisfy the corresponding level, it does not point what level should require any kind of application, that is a developer/team/administrative decision. The standard (link available at *https://www.ultesttools.com/standards/fips/c-38/c-1863*) also describes strictly and in detail: approved security functions, approved random number generators, approved key establishing tecniques and relative validation processes that have to be tested to comply the standard specifications.
Once got the standard description and highlighted the field of search scope, first try is to get and check inside software to inspect the logically easiest part to find with keywords and labels search, the random generation numbers one.
A first hint grepping the word "random" on the whole MediaWiki code folders is gotten by class **CryptRand**, which in its description is described as a random numbers generator class which partially uses an external Drupal library extended with self-made code, inspecting class code and skipping procedures like time-lock of specific milliseconds to avoid time-based attacks against random generation functions and going straight to the part relative to functions used, inside code description we found the Figure 3 comment section with the relative random-generator handling code right after.
We can conclude it may be FIPS 140-2 compliant (but not validated!) or not based on the O.S. specifics, for example if no option is available, OpenSSL im-

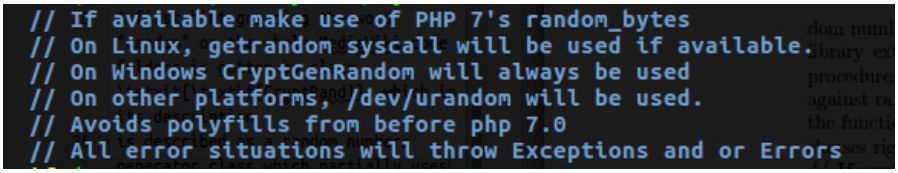

Figure 3: *Comment section in class CryptRand.php before random generation code handling, OS related system calls may be performed.*

plementation choice available (found out inspecting code instead of what the description comment points out...) gives possibility to run the library in "FIPS mode" calling a FIPS 140-2 compliant version of its random generator function, for linux distributions unfortunately I did not find any resource/documentation about its compliance except for Oracle's Solaris 11 that claims its compliance and validation to the standards, as for some versions of Windows Server (e.g. 2003).
After pointing out that there is no security compliance without a compliant O.S. for random generation bytes even if encryptioon functions used are conform to the standard, I proceeded with the validation of all cryptographic modules used by the MediaWiki software, keeping in mind that being FIPS 140-2 standard compliant does not mean to be FIPS 140-2 validated, the compliancy part permit to adequate code to FIPS standard, validation is issued by NIST after checking and validating the standard requirements that the issuer source code must have.

2.1 Symmetric Key Encryption and Decryption

Symmetric encryption/decryption is used when sending/receiving messages inside an already established session with client.
Algorythm proposed to encrypt/decrypt messages are: ***AES256-cbc, AES256-ctr, rijndael128-cbc.rijndael128-ctr*** The two version of AES are taken from OpenSSL library, which if used in "FIPS mod" is FIPS 140-2 compliant, rijndael variants instead are part of a deprecated library from PHP 7.1 and above, so are neither FIPS 140-2 compliant, nor validated.

2.2 Secure Hash Standards

Since the already discovered class CryptRand uses part of a Durpal FIPS 140-2 validated class library but mostly own source code to generate hashes, algo used is SHA-1 wich may be compliant but as aforementioned not FIPS140-2 validated.

3 7.6 Verify that all random numbers, random file names, random GUIDs, and random strings are generated using the cryptographic modules approved random number generator when these random values are intended to be not guessable by an attacker.

3.1 Random Session number ID

Rand num: most crucial random number generator is session user ID, generated in *resources/src/mediawiki/mediawiki.user.js*.
It uses Math.random() function in a proper and extended way with 64-bit generated ID to make collisions over 500 millions number (potentially 500 millions active users at same time) with less of 1% of probability to happen.

3.2 Random GUIDs

For GUIDs the only noticable one we found interesting is inside Input Boxes generator, to avoid pages displayed by a session that have multiple input boxeses (login, checkbox etc...) to have the same box ID, it is present in: *extensions/InputBox/InputBox.classes.php*; and uses the included global function wfrandom() that generates a pseudorand number between 0 and 2$\hat{}$31 -1, which in turn uses php function mt_rand(min value, max value) which uses a Mersenne Twister Random Number Generator that is a pretty good algorythm to generate pseudorandom numbers with a semi-uniform probability distribution.

3.3 Random Password Strings

For default random password generation we can find reference in:
includes/password/PasswordFactory.php, where function generateRandomPasswordString(minDigitsNumber) calls back the already seen class MWCryptRand which uses OpenSSL/server OS libraries to generate a pseudorandom secret (user password in this case).

3.4 HMAC generator

Hash function used for generating HMAC to validate source and integrity of messages cand find its reference in *includes/DefaultSettings.php* where **SHA-256** is set as default value for such purpose.

V 9.1 Sensitive data does not get cached

Pre-filled forms may be cached by browser even when using HTTPS.
Forms that handles sensitive data should have disabled both client-side caching and fields autocomplete features. Even if this features could (apparently) seem harmless, combined with a XSS vulnerability they could lead to disclosure of sensitive informations without any user interaction. In order to verify that every form which handles sensitive data have both the aforementioned features disabled, we started by detecting where those forms were defined and then we inspected the source code involved. Once those forms have been identified, we analysed server-side scripts to check HTTP Control-Cache and Pragma headers were sent properly with sensitive pages and the HTML code rendered by browser for the auto-complete form (and fields) feature.

Hard-coded forms identified with the use of *grep* command line utility using different keywords, (i.e. *<form*, *form*, *autocomplete*, and fields ID obtained by HTML source code review) do not handle sensitive data, thus we identified where those forms were generated and what headers server sends with them. Sensitive forms identified are those involved with the following procedures: login, signup and password change.
After some more source code analysis, we identified a PHP class (named OutputPage.php) which is responsible to prepare final page rendering and to add cache headers to the packets sent by the server. Inspecting this class source code through different *grep* and *ctrl+f* commands, we found a function that is supposed to enable/disable client-side caching (enableClientCache) by setting this variable *mEnableClientCache* to true/false, respectively. The mentioned variable is true by default albeit it is modified whenever Web pages used for sending sensitive data have to be sent to clients. We verified the function is properly called every time it is necessary. For example, it is properly set at every signin, change password, and signup procedure, as it can be seen by inspecting browser cache (i.e. on Mozilla browser by writing about:cache into the URL bar) after those procedure are terminated. On the other hand, the autocomplete feature (which default value is true) is not explicitly turned off as it can be seen inspecting the HTML source code of web pages containing sensitive forms.

V 9.3 Sensitive data does not get sent in the URL

Sending sensitive data in the URL will lead to them being available in the browser history, in logs by the application server and any potential intermediaries (proxies). Tests regarding this requirement has been performed manually. After surfing for pages containing sensitive forms, those related to login, account creation, and password change procedures, by inspecting HTML source code it can be clearly seen that every form containing sensitive data has the *method* attribute set as POST and, consequently, no sensitive data is sent into an URL.

9.4 Verify that the application sets sufficient anti-caching headers such that any sensitive and personal information displayed by the application or entered by user should not be cached on disk by mainstream modern browsers (e.g. visit about:cache to review disk cache).

For the caching header section I found two interesting papers regarding where the mechanism to not allow caching headers in all most common browser and on how a previous implementation of MediaWiki could allow duplicate sessions when different users used the same proxy to connect to MediaWiki services (or a malicious attacker could have duplicated a legitimate session using same proxy as the target user was doing in the meantime...) .
1st: https://wikitech.wikimedia.org/wiki/MediaWiki_cachingCache_headers
2nd: https://isc.sans.edu/forums/diary/TheSecurityImpactofHTTPCachingHeaders/17033/
Searching for this vulnerability inside CVE MediaWiki list we found out that the flaw was not resolved until version 1.15.5 (CVE-2010-2787).
Checking inside class source code **OutputPage.php** is possible to see those anti proxy-duplication mechanisms inside function sendCacheControl(), purging proxy cache explicitly at every new website visit.
Other built-in header anti-caching mechanisms for all common browser can be visible in the final part of same function as visible in Figure 1. (in a if/else clause where if private configuration of the MediaWiki instance is sat, this because MediaWiki has built-in engine to perform if a page can be accessible by anyone or if it needs special access requirements to be visualized, in the private case no info about session gets cached, more references here: https://www.mediawiki.org/wiki/Manual:Preventing_accessSimple_private_wiki).

```
# In general, the absence of a last modified header should be enough to prevent
# the client from using its cache. We send a few other things just to make sure.
$response->header( 'Expires: ' . gmdate( 'D, d M Y H:i:s', 0 ) . ' GMT' );
$response->header( 'Cache-Control: no-cache, no-store, max-age=0, must-revalidate' );
$response->header( 'Pragma: no-cache' );
```

Figure 4: Anti-caching common php header-building pattern to avoid any session information to get cached on most common browsers.

V 9.5 Temporary server caches of sensitive data are properly cleaned up

In order to state that every sensitive data is properly cleaned up once used and temporary copies are handled safely, we identified where passwords sent via HTML forms are handled by server-side scripts and then we analyzed how those

data are handled. Initially, we found the main class that handles sensitive information, AuthManager.php, and then we manually verified that login, account creation, and password change procedures handle sensitive data properly.
With the use of *grep* and *ctrl+f*, we analyzed the behaviour of several subroutines, such as beginAuthentication, continueAuthentication, and setSessionDataForUser, and we stated that sensitive data is handled and stored correctly. For example, once a password for a specific account is received or validated, the involved subroutine terminates and there's no further processing or referencing to the password's value.

Furthermore, no password is stored as it is received. MediaWiki hashes account passwords so that not even who's able to query the DB directly would be able to read their values.

V 9.9 Client side storage doesn't contain secrets

We verified that after every security critical procedure is performed (such as login, account creation and password change) no secrets are stored into regular cookies or HTML5 local storage. Cookie values have been analyzed through the Cookie Quick Manager add-on, a Mozilla Firefox browser extension which allows to read and modify stored cookies. The screenshot below, taken after having created and authenticated a new account, summarises those values.

Figure 5: Mozilla Cookie-Manager add-on

As it can be seen, no sensitive data is stored in regular cookies. The only stored values are the the MediaWiki session ID, the username and its ID into the application database, cpPosTime (UNIX timestamp) and other two values related to CDN (ContentDeliveryNetwork) caching, which can be all considered not sensitive nor PIIs. However, trusting the username or its ID could lead to several impersonation attacks so it's crucial that the application doesn't use those values for security-related subroutines. We have made sure those data were never trusted by using grep with the keyword $_COOKIE and inspecting the resulting files deeper.

Data maintained into HTML5 local storage has been inspected too, this time using the GoogleChrome inspector. Once the mentioned security-related procedures has been performed, the stored local content has been inspected and, as shown by the following screenshot, no sensitive data had been stored inside.