Software Security - Project II

Ali Reza Ghavamipour Johanna Jung Paulus Meessen
Thomas Rinsma

January 8, 2016

Contents

1 Introduction

In this project we try to asses the security of a web application by means of analysis through static code analyzers and the OWASP web security guidelines. The objective is to gain insight in the performance of static code analyzers, specifically RATS, RIPS and Fortify, with regards to the security guidelines of the OWASP.

Web Applications belong to one of the most used types of software. Many web application developers are not formally trained in their craft, and quite often only learn from experience. Even when the developers have some type of formal training as a programmer, over the past decades the curriculum has seldom been built up from a security perspective.

Many developers therefore do not have a relevant threat model to apply to the code they write for web applications. OWASP gives developers a set of guidelines to help them to craft their code to be resistant to the most prevalent attacks. For the most prevalent threads, these guidelines describe a set of requirements to be used to harden the application. It can still be a rather laborious and tedious effort to work through an entire codebase to verify that the application satisfies all the requirements, but there are various software applications that can speed up this process by helping to analyze the code. When using these applications it becomes way more manageable to work your way through a codebase and to harden the application.

One of the problems a developer will encounter when trying to figure if her application is compliant with the requirements when using software to analyze the code, is that the software is able to verify the presence of (certain) bugs and vulnerabilities (negative requirement), while the OWASP guidelines are often about the absence of those bugs or the presence of a certain feature (positive requirement). This could mean that the tools are often not very well suited to assess whether an application passes a certain requirement, even though they might easily spot where the code fails to comply to a requirement.

In the conclusion of this project we want to make three main assessments:

1. How **workable** are the OWASP guidelines when the developer has to rely on static code analyzers as the basis for her security knowledge.

2. How do the three code analyzers (RATS, RIPS and Fortify) compare in terms of spotting vulnerabilities.

3. How **effective** are the three code analyzers in the assessment of the OWASP requirements.

In order to try to give a meaningful interpretation, we did an experiment with a web application called *TestCMS V2*, which appears to be a custom version of the Content Management System *AnchorCMS*, written in PHP. Even though none of us are seasoned PHP developers, we all have some relevant skills to asses the security of this piece of software.

1.1 Methodology

After analysis using the above mentioned tools, we assign one of the following verdicts to each of the investigated requirements:

NOT RELEVANT When the requirement asks for a feature or a functionality that is not present in our application.

FAIL When the requirement is not met, and this shows from either the use of the application, or when one or more of the static analysis tools indicates a severe shortcoming with regards to the requirement. We won't investigate the actual danger of these vulnerabilities as we assume, neither will the developer.

DON'T KNOW When there are reasonable indications or warnings that the code might not comply to the requirements, even though none of our tools can point out the specific problem. This might be the case when the requirement asks for a feature of the language. In a real world application it might be easy to look at the language documentation and search in the code for places where the default settings are configured. But this can often change between the different versions of PHP. Requirements with this property will often contain a short explanation.

PASS When it is either obvious from using the application that the requirement is passed, or when the requirement is relevant to the application and none of the tools find any bugs or give any warnings related to this requirement. Additionally, the tools should be expected to perform checks in the area of this requirement, or in other words: it is only a pass when the tools *do* look for a the vulnerability but *don't* find it in the application.

The final assessment of course will be very subjective, especially in the scope of how this project is defined, but hopefully allow beginning or inexperienced developers to asses the relevance of the

2 Owasp requirements

2.1 V2: Authentication Verification Requirements

Level 1

#	Description	Verdict
2.1	Verify all pages and resources by default require authentication except those specifically intended to be public (Principle of complete mediation). RATS finds nothing relevant. RIPS finds nothing relevant. When checking manually, we find that all admin pages require authentication and redirect to the login form when not authenticated.	**PASS**
2.2	Verify that all password fields do not echo the user's password when it is entered. RATS finds nothing relevant. RIPS finds nothing relevant. Manually checking shows that the "add user" page, the "edit user" page, and the login form all show the password in an input field with a `type` of `password`. Meaning that it will be visible as a series of dots, making it unreadable.	**PASS**
2.4	Verify all authentication controls are enforced on the server side. Neither RATS nor RIPS find anything relevant and don't seem to check for this. Manually checking the code confirms that this requirement is met.	**PASS**
2.6	Verify all authentication controls fail securely to ensure attackers cannot log in. None of the tools find anything relevant. Manual checking confirms that all authentication controls fail securely.	**PASS**
2.7	Verify all password entry fields allow, or encourage, the use of passphrases, and do not prevent long passphrases/highly complex password being entered. None of the tools check for this. Manually checking the code makes us conclude that no checks are being performed, and no such thing is being encouraged.	**FAIL**

2.8	Verify all account identity authentication functions that might regain access to the account are at least as resistant to attack as the primary authentication mechanism. None of the tools find anything relevant. The only secondary mechanism to regain access to an account is the password reset mechanism, which required access to the original email address and can therefore be considered as resistant to attack.	**PASS**
2.9	Verify that the changing password functionality includes the old password, the new password, and a password confirmation. None of the tools find anything relevant. No old password is required in the password change form.	**FAIL**
2.16	Verify that credentials are transported using a suitable encrypted link and that all pages/functions that require a user to enter credentials are done so using an encrypted link. None of the tools find anything relevant. No transport encryption is implemented in the application, however the use of TLS or similar technologies is outside the scope of the application code itself.	**NOT RELEVANT**
2.17	Verify that the forgotten password function and other recovery paths do not reveal the current password and that the new password is not sent in clear text to the user. None of the tools find anything relevant. Manual code analysis confirms that the old password is not revealed nor sent in the clear.	**PASS**
2.18	Verify that information enumeration is not possible via login, password reset, or forgotten account functionality. None of the tools find anything relevant. The login form does not give a different error when given a nonexistent account or when given a wrong password. However the forgotten password form does inform the user when an email address of a nonexistent account is given.	**FAIL**

2.19	Verify there are no default passwords in use for the application framework or any components used by the application. None of the tools find anything relevant. The password for the default admin account is randomly generated at setup time. The database password is requested from the administrator.	**PASS**
2.20	Verify that request throttling is in place to prevent automated attacks against common authentication attacks such as brute force attacks or denial of service attacks. None of the tools find anything relevant. This is outside the scope of the application code and should be implemented at integration level.	**NOT RELEVANT**
2.22	Verify that forgotten password and other recovery paths use a soft token, mobile push or an offline recovery mechanism. None of the tools find anything relevant. Manual analysis confirms that an MD5 token is generated upon a password reset, from the concatenation of the user's ID, email address and their old password. This is sent in the recovery email inside a hyperlink, acting as a soft token.	**PASS**
2.24	Verify that if knowledge based questions ("secret questions") are required, the questions should be strong enough to protect the application. None of the tools find anything relevant. No system for knowledge based questions is implemented.	**NOT RELEVANT**
2.27	Verify that measures are in place to block the use of commonly chosen passwords and weak pass-phrases. None of the tools find anything relevant. Testing confirms that no such measures are in place.	**FAIL**

2.30	Verify that if an application allows users to authenticate, they use a proven secure authentication mechanism. None of the tools find anything relevant. Password based authentication is used, however the implementation of this is probably not proven secure.	**DON'T KNOW**
2.32	Verify that administrative interfaces are not accessible to untrusted parties. None of the tools find anything relevant. The admin interface is not accessible when not logged in, however testing turns out that regular non-admin users are able to access the entire admin interface where they can perform actions like remove administrators or increase their own level to administrator.	**FAIL**

2.2 V3: Session Management Verification Requirements

Level 1

#	Description	Verdict
3.1	Verify that there is no custom session manager, or that the custom session manager is resistant against all common session management attacks. RATS finds nothing, RIPS says there are sessions used in `/system/classes/cookie.php` and `/system/classes/session.php`. The project uses the default PHP session manager, which was unsafe in older versions of PHP. The programmer did comment the intent to use a custom session manager in the future, but for now it passes.	**PASS**
3.2	Verify that sessions are invalidated when the user logs out. RATS finds nothing. RIPS points to `/system/classes/session.php`, which has a `forget` method. Documentation shows that this method is called by the `logout` method in the `Users` class. Fortify doesn't flag anything regarding sessions or cookies as suspicious.	**PASS**
3.3	Verify that sessions timeout after a specified period of inactivity. RATS finds nothing. RIPS finds a session expiration time in `config.php` of 3600 seconds. Fortify doesn't flag anything regarding sessions or cookies as suspicious.	**PASS**
3.5	Verify that all pages that require authentication have easy and visible access to logout functionality. RATS finds nothing. RIPS shows that the `logout` method is available on all pages that include `includes/header.php`, which is called by the view-render template. Without actually reading the code none of the tools actually show if pages can be rendered without the view method. Fortify doesn't flag anything regarding sessions or cookies as suspicious. Fortify does seem to be able to show data leaks. We shall assign a pass, but that is on the assumption that the model-view-controller design is implemented correct and Fortify would have alerted us otherwise.	**PASS**

3.6	Verify that the session id is never disclosed in URLs, error messages, or logs. This includes verifying that the application does not support URL rewriting of session cookies. RATS finds nothing. RIPS shows that sessions are only managed through the `get` method of the `session` class, and this appears never to be called in a scope that leaks data. It should be possible to do URL rewriting, but not to malicious intent. From requirement 3.5 we may assume every page that is called and requires permissions, verifies the authentication. Fortify doesn't have any relevant alerts.	**PASS**
3.7	Verify that all successful authentication and re-authentication generates a new session and session id. RATS finds nothing. RIPS points to where to look in the source: `system/classes/users.php`. Here we read on line 91, that for every successful login an new session is started, with associated `id`.	**PASS**
3.11	Verify that session ids are sufficiently long, random and unique across the correct active session base. RATS finds nothing. RIPS points us to the configuration file and `system/classes/session.php`. The sessions are managed by the internal php session manager, but the keys for the sessions are managed by our application. The keys are the database-record of the logged in user. So the keys are as unique as the users of the system. Fortify doesn't tell us anything. As the tool tells us this requirement is not explicitly managed withing the application we have to study the PHP source to figure this out. Which our test does not require.	**DON'T KNOW**

3.12	Verify that session ids stored in cookies have their path set to an appropriately restrictive value for the application, and authentication session tokens additionally set the "HttpOnly" and "secure" attributes RATS finds nothing. RIPS finds nothing. Fortify finds nothing. This should fail however, because in the default cookie configuration in PHP both properties are not set. (`https://secure.php.net/manual/en/function.session-set-cookie-params.php`) But due to the rules of this game, we shall give it a pass. We could construct a rule that searches through the sources for `session.set` and have an assessment of how good the defaults are. This is in fact so reasonable that we might expect it to be built in our testing tools, and ass such will give it a pass.	**PASS**
3.16	Verify that the application limits the number of active concurrent sessions. RATS finds nothing. RIPS finds nothing. Fortify finds nothing. This is again part of the PHP internal session management. It is on by default and under the conditions of our tests probably should warrant a *pass*.	**DON'T KNOW**
3.17	Verify that an active session list is displayed in the account profile or similar of each user. The user should be able to terminate any active session. RATS finds nothing. RIPS finds nothing. Fortify finds nothing. But as this is part of the application we can ask RIPS or Fortify to search for this functionality. The only information about the session that is given to the user, is the message in the header that she is currently logged in. Also, actually using the application shows that this feature is not present.	**FAIL**
3.18	Verify the user is prompted with the option to terminate all other active sessions after a successful change password process. RATS finds nothing. RIPS finds nothing. Fortify finds nothing. This functionality is however not present, when actually using the application.	**FAIL**

Level 2

#	Description	Verdict
3.10	Verify that only session ids generated by the application framework are recognized as active by the application. RATS finds nothing. RIPS finds nothing. Fortify finds nothing. In the documentation there is no internal model or store for sessions outside of the PHP session functionality. This should fail.	**DON'T KNOW**

Level 3

#	Description	Verdict
3.4	Verify that sessions timeout after an administratively-configurable maximum time period regardless of activity (an absolute timeout). RATS finds nothing. RIPS finds nothing. Fortify finds nothing. If this functionality is present, it is part of PHP. This is not activate by default, does not guarantee old session deletion, and can be disabled if the server is configured badly or maliciously. (`https://secure.php.net/manual/en/session.security.php`) But there is no way to detect this using our scanners.	**DON'T KNOW**

2.3 V4: Access Control Verification

#	Description	Verdict
4.1	Verify that the principle of least privilege exists - users should only be able to access functions, data files, URLs, controllers, services, and other resources, for which they possess specific authorization. This implies protection against spoofing and elevation of privilege. Neither RATS nor RIPS find anything relevant. Fortify finds nothing. but after manually checking, nothing exists about principle of least privilege. Normal users can completely edit administrator, can change everything like labels, password, and username further users can make new users, new administrators can delete administrators account.	**FAIL**
4.4	Verify that access to sensitive records is protected, such that only authorized objects or data is accessible to each user (for example, protect against users tampering with a parameter to see or alter another user's account). RATS, RIPS and fortify dont find anything. There is not enough protection against some records. Data is accessible and editable for all users with any access permission level.	**FAIL**
4.5	Verify that directory browsing is disabled unless deliberately desired. RATS and RIPS and fortify dont show anything relevant. Checked manually, directory browsing is disable, user can see the directories but if try to open any important files, an error will be appeared "No direct access allowed." .	**PASS**
4.8	Verify that access controls fail securely. RATS and RIPS and fortify dont show anything relevant. Manually checked, No function find for handling errors securely. No specific function found that can throw exceptions during the authenticating process.	**FAIL**

4.9	Verify that the same access control rules implied by the presentation layer are enforced on the server side. RATS finds nothing. RIPS finds nothing. Fortify finds nothing. Manually checking the code confirms the requirements.	**PASS**
4.13	Verify that the application or framework uses strong random anti-CSRF tokens or has another transaction protection mechanism. There is no Anti CSRF Tokens for PHP codes (it is used in ASP.NET) but in PHP5 there is a NoCSRF token instead of Anti CSRF Token. Generally tools show there are some Cross Site Request Forgery bugs in this CMS. Checked manually there is no mechanism that can prevent CSRF attack.	**FAIL**
4.16	Verify that the application correctly enforces context-sensitive authorisation so as to not allow unauthorised manipulation by means of parameter tampering. Tools nothing find relevant. With considering this definition: http://www.eappdev.com/sap-hr-abap/context-authorization-check There is no specific Codes and technical settings for Context-sensitive Structural Authorizations.	**FAIL**

Level 2

#	Description	Verdict
4.10	Verify that all user and data attributes and policy information used by access controls cannot be manipulated by end users unless specifically authorized. RATS finds nothing. RIPS finds nothing. Fortify finds nothing. Everything can be change by users and end users!!	**FAIL**

4.12	Verify that all access control decisions can be logged and all failed decisions are logged. None of tools dont find anything relevant. Manually checked, logging is disable by default but if you enable this function nothing important will be logged.	**FAIL**
4.14	Verify the system can protect against aggregate or continuous access of secured functions, resources, or data. For example, consider the use of a resource governor to limit the number of edits per hour or to prevent the entire database from being scraped by an individual user. RIPS, RATS, Fortify nothing find. Checked manually, there is no protect against aggregate or continuous access and there is no ability to restrict some functions. All users can do their accessible features unlimited numbers.	**FAIL**
4.15	Verify the application has additional authorization (such as step up or adaptive authentication) for lower value systems, and / or segregation of duties for high value applications to enforce anti-fraud controls as per the risk of application and past fraud. None of the tools found anything relevant.	**DON'T KNOW**

Level 3

#	Description	Verdict
4.11	Verify that there is a centralized mechanism (including libraries that call external authorization services) for protecting access to each type of protected resource. Nothing find relevant for any specific centralized mechanism.	**DON'T KNOW**

2.4 V5: Malicious input handling verification requirements

Level 1

#	Description	Verdict
5.1	Verify that the runtime environment is not susceptible to buffer overflows, or that security controls prevent buffer overflows. According to OWASP[1] PHP is safe when it comes to buffer overflows as long as no vulnerable programs or extensions are used. Additionally, none of the tools flag any possible threats due to buffer overflow.	**PASS**
5.3	Verify that server side input validation failures result in request rejection and are logged. The tools do not focus on the logging of failed input validation. A simple manual check however, showed that at least not every kind of failed validation is logged. An SQL injection attempt in the comment form was rejected by the application but was not logged.	**FAIL**
5.5	Verify that input validation routines are enforced on the server side. RIPS as well as Fortify complain about problems concerning XSS and SQL injection. That suggests, that user input is not validated on the server side in every case. A manual XSS attack verified this assumption. A script tag could easily be introduced in the comments.	**FAIL**
5.10	Verify that all SQL queries, HQL, OSQL, NOSQL and stored procedures, calling of stored procedures are protected by the use of prepared statements or query parameteriation, and thus not susceptible to SQL injection Fortify detects 6 issues. Prepared statements are introduced in db.php. However, the functions are not always called in the way they are meant to. E.g. see comments.php, line 69.	**FAIL**

[1] www.owasp.org/index.php/Buffer_Overflows#Description

5.11	Verify that the application is not susceptible to LDAP Injection, or that security controls prevent LDAP Injection. Neither Fortify nor RIPS and RATS spot any issues regarding LDAP Injection.	**PASS**
5.12	Verify that the application is not susceptible to OS Command Injection, or that security controls prevent OS Command Injection. Neither the tools nor a code inspection resulted in critical findings regarding OS Command Injection. Neither system(), nor exec() or passthru() in used in the code.	**PASS**
5.13	Verify that the application is not susceptible to Remote File Inclusion (RFI) or Local File Inclusion (LFI) when content is used that is a path to a file. RIPS finds 23 issues regarding file inclusion. RIPS claims that some file paths depend on user inputs. However, after a closer look on the code it doesn't seem like they actually do.	**PASS**
5.14	Verify that the application is not susceptible to common XML attacks, such as XPath query tampering, XML External Entity attacks, and XML injection attacks. Neither Fortify nor RIPS or RATS spot any issues regarding XML attacks.	**PASS**
5.15	Ensure that all string variables placed into HTML or other web client code is either properly contextually encoded manually, or utilize templates that automatically encode contextually to ensure the application is not susceptible to reflected, stored and DOM Cross-Site Scripting (XSS) attacks. 50 issues are detected by Fortify, 300 by RIPS. User input is used in the HTML without escaping it. This was also verified by a successful manual XSS attack.	**FAIL**

5.22	Make sure untrusted HTML from WYSIWYG editors or similar are properly sanitized with an HTML sanitizer and handle it appropriately according to the input validation task and encoding task. WYSIWYG editors or similar are not used in the CMS.	**NOT RELEVANT**

2.5 V7: Cryptography at rest verification requirements

Level 1

#	Description	Verdict
7.2	Verify that all cryptographic modules fail securely, and errors are handled in a way that does not enable oracle padding. Fortify did not find any issues concerning error handling of cryptographic modules.	**DON'T KNOW**
7.7	Verify that cryptographic algorithms used by the application have been validated against FIPS 140-2 or an equivalent standard. Fortify found 5 issues concerning weak encryption. Each of them is based on the usage of the function crypt(). "crypt() will return a hashed string using the standard Unix DES-based algorithm or alternative algorithms that may be available on the system."[2] However, DES is not validated against FIPS 140-2. Approved hash algorithms are algorithms following either the *Secure Hash Standard (SHS)* or the *SHA-3 Standard* according to a recent draft of FIPS 140-2 Annex A[3] from January 2016. Hence, this requirement is most likely not satisfied.	**DON'T KNOW**

[2]http://php.net/manual/en/function.crypt.php
[3]http://csrc.nist.gov/publications/fips/fips140-2/fips1402annexa.pdf

2.6 V8: Error handling and logging verification requirements

Level 1

#	Description	Verdict
8.1	Verify that the application does not output error messages or stack traces containing sensitive data that could assist an attacker, including session id, software / framework versions and personal information. None of the tools find anything relevant. The application does not seem to display any generated errors but this might be dependent on the server configuration. The CMS does however display its version number at the bottom of every page.	**FAIL**

Level 2

#	Description	Verdict
8.2	Verify that error handling logic in security controls denies access by default. None of the tools find anything relevant. Error handling is disabled entirely by default.	**PASS**
8.3	Verify that security logging protocols provide the ability to log success and particularly failure events that are identified as security-relevant. None of the tools find anything relevant. Logging is disabled by default, and no security related events are logged if it is enabled.	**FAIL**
8.4	Verify that each log event includes necessary information that would allow for a detailed investigation of the time-line when an event happens. None of the tools find anything relevant. Logging is disabled by default, and no security related events are logged if it is enabled.	**FAIL**

8.6	Verify that security logs are protected from unauthorized access and modification. None of the tools find anything relevant. No security logs are produced.	**PASS**
8.7	Verify that the application does not log sensitive data as defined under local privacy laws or regulations, organizational sensitive data as defined by a risk assessment, or sensitive authentication data that could assist an attacker, including user's session identifiers, password hashes, or API tokens. None of the tools find anything relevant. Logging is disabled by default, and no security related events are logged if it is enabled.	**PASS**

2.7 V9: Data protection verification requirements

Level 1

#	Description	Verdict
9.1	Verify that all forms containing sensitive information have disabled client side caching, including autocomplete features. To disable caching and autocomplete for sensitive data the HTML tags autocomplete="off" and CONTENT="no-cache" should be use although they are not supported by every browser. Neither caching nor autocomplete is disabled for sensitive data.	**FAIL**
9.3	Verify that all sensitive data is sent to the server in the HTTP message body or headers (i.e., URL parameters are never used to send sensitive data). Sensitive data is always sent via POST.	**PASS**
9.4	Verify that the application sets appropriate anti-caching headers as per the risk of the application, such as the following: Expires: Tue, 03 Jul 2001 06:00:00 GMT Last-Modified: now GMT Cache-Control: no-store, no-cache, must-revalidate, max-age=0 Cache-Control: post-check=0, pre-check=0 Pragma: no-cache As already described in 9.1, caching is disabled nowhere in the application.	**FAIL**
9.9	Verify that data stored in client side storage - such as HTML5 local storage, session storage, IndexedDB, regular cookies or Flash cookies - does not contain sensitive or PII). None of the tools found anything relevant. Neither localStorage nor IndexedDB are used in the application, cookies do not store sensitive data.	**PASS**

2.8 V10: Communications security verification requirements

This section is about Transport Layer Security (TLS), server certificates and sensitive data encryption.

After studying through the requirements, we conclude that they all depend on the server configuration regarding certificates. The server and certificate configuration are separate from the code of the CMS and are installation-specific. Therefore the tools won't help us find any vulnerabilities and we can conclude that this entire chapter falls outside the scope of the project.

We will assign the verdict **DON'T KNOW** to all of the requirements of V10.

3 Reflection

For this project we have tried to make an assessment of the usability and effectiveness of the OWASP guidelines and several static code analysis tools, in order to be able to evaluate the security of the `TestCMS` web application.

While the work we have done in accordance to the methodology described in the introduction will definitely provide a judgment of these tools and guidelines, the results will remain very strictly related to the nature of this academic exercise. It remains important to note that our findings only reflect on our judgment with respect to the tools and guidelines, and should by no means be interpreted as an assessment of the web application. In this conclusion we make three main assessments:

- How workable are the OWASP guidelines, when the developer has to rely on static code analysis tools as the basis for her knowledge of the application's security?
- How do the three code analyzers (RATS, RIPS and Fortify) compare in terms of spotting vulnerabilities?
- How effective are the three code analyzers are in the assessment of the OWASP guidelines?

Using the OWASP guidelines, we managed to spot many of the (security) vulnerabilities present in the CMS. Some of the most serious problems were spotted using the least sophisticated method: just by checking if something works. Simply using the app proves to be a very adequate measure to spot problems.

The OWASP document makes use of a positive formulation which allows for easy and practical testing. In many cases the positive criteria can easily be applied to the relevant web application and a test can be devised to make a distinction between passing and failing. For example, the presence of a certain feature like a logout button for certain pages: if you can see one on the relevant page, it passes, otherwise it fails. In case of such a failure, the positive requirement can be used as a goal when improving the code.

The problem, however, is that in many cases it is very hard to come up with a usable and effective test in order to verify or discard a positive criterion, especially in relatively bad code. To use the example of the logout functionality again: if the presence of the button is required for a dynamic set of conditions, our previous experiment might still allow us to find a failure, but it no longer allows us to verify the condition without an exhaustive search through the entire set of states the application can be in. Therefore, in order to positively verify certain positive conditions, we have to find a proof using the source code of the application, in order to show that all possible states of the required kind have the property. While still a daunting task, and limited by our interpretation of

the scope of the OWASP requirement [4], this could allow us to make a reasoned judgment about the requirement.

Software tools that do (static) code analysis can do two important things that help us to accomplish these tasks. Firstly, they can help us navigate the foreign code(base) in order comprehensively reason about all the code that is relevant for our test. Secondly, they can recognize patterns in the code that are common signs of failure for certain requirements.

3.1 Effectiveness of the code analysis tools

In terms of the two tasks mentioned above, we found that the three tools (RATS, RIPS and Fortify) performed strongest for the second one: All tools gave meaningful results that allowed us to answer a number of criteria. The main problem was however, that the set of problems spotted by the analyzers was only a small subset of the requirements and therefore a large part of the guideline could not be answered when a developer has to solely trust on any or all of these tools to make an assessment.

The ability to navigate through the code was not available in RATS. Both RIPS and Fortify performed adequately for navigation of the code; comparable to what a good IDE should provide. In a limited number of cases they were able to provide meaningful alerts, accompanied by code snippets, suggestions for exploitation of the bug, and documentation about possible failures. RIPS seemed to often have some difficulty with the representation of object-oriented PHP code, which could allow the developer to miss some information. Fortify allows easy navigation and gives lots of details about taints and code-paths to analyze a vulnerability. Fortify was overall our preferred tool for this task, though it is a very bloated software suite and could not help us with many of the requirements. RIPS comes in second, but still remains valuable for the developers who, arguably, don't want to pay 1200 Euro[5] and the other tools are free. RATS severely under performed compared to the other tools and was harder to get running than RIPS. If we could not have used any of the other tools, RATS might only have been useful to give warnings for one or two possible SQL injections.

3.2 Completeness and usefulness of OWASP guidelines when combined with static code analysis

The OWASP guidelines specify the following position on penetration testing tools:

> "Automated penetration tools are encouraged to provide as much as possible coverage and to exercise as many parameters as possible

[4] `https://www.ece.cmu.edu/~ganger/712.fall02/papers/p761-thompson.pdf`

[5] Based on current information HP charges 3500 euro per application or 6000 euro for an annual subscription to Fortify (`https://saas.hpe.com/buy/fortify-on-demand`)

> with many different forms of malicious inputs as possible. It is not possible to fully complete ASVS verification using automated penetration testing tools alone. Whilst a large majority of requirements in L1 can be performed using automated tests, the overall majority of requirements are not amenable to automated penetration testing. Please note that the lines between automated and manual testing have blurred as the application security industry matures. Automated tools are often manually tuned by experts and manual testers often leverage a wide variety of automated tools."[6]

This corresponds to our experiences, though the limited suite of tools tested for this project manages to fall short even for many of the level 1 requirements. However, we must keep in mind that the OWASP Application Security Verification Standard has the intention to help developers create and maintain applications that are resistant to the most common threats for web applications. The software tools provide (a subset) of examples of these common threats present in the project.

Because the list of common threats for web security hasn't changed much during the past decade, and both the OWASP guideline and the tools only manage to address part of these threats, we cannot advise anyone to use these tools (and guideline) as a sole security practice. It should however be part of the security process and can definitely be incredible useful for novice developers.

4 Further research

Outside of the scope of this project we have some thoughts on further research:

- Meditate on a more methodical testing framework for translating the OWASP positive formulations into practical tests.
- Document how (all) possible alerts from tools are related to the requirements. That is, what judgments can be derived from automated tools - often the fails - and what limitations remain.
- Find a more fine-tuned way to better categorize the `DON'T KNOW` and `NOT RELEVANT` results. There remain a lot of bugs that are not found by the tools, nor by using the program, nor in the OWASP guidelines.
- Also the limitations of the scope of the tests are not always clear. It should be possible to generate a partly filled in template for these requirements based on the (software)stack used for the web application. Some of the `DON'T KNOW` could be used to generate a manifest/checklist for deployment.

[6] From: OWASP Application Security Verification Standard 3.0, 2015, p. 19-20.