Vulnerabilities of Modern Web Applications

F. Holik, S. Neradova
University of Pardubice, Faculty of electrical engineering and informatics, Pardubice, Czech Republic
filip.holik@student.upce.cz

Abstract – The security of modern web applications is becoming increasingly important with their growing usage. As millions of people use these services, the availability, integrity, and confidentiality are critical. This paper describes the process of penetration testing of these applications. The goal of such testing is to detect application flaws and vulnerabilities and to propose a solution to mitigate them. The paper analyses current penetration testing tools and subsequently tests them on a use case web application, build specifically with present security flaws. The process of penetration testing is described in detail and the performance of each tool is evaluated. In the last section, recommended practices to mitigate found flaws are summarized.

I. INTRODUCTION

The trend in modern application design is to move these applications into a remote server instead of running them locally. This will ensure consistent and quick updates, application monitoring and lower requirements on local hardware performance. A lot of companies are using these advantages for office applications like Google Docs, Sheets, and Slides; or Microsoft Office 365. On the other hand, this infrastructure also has its specific requirements - especially for good quality of Internet connection. Another broadly discussed topic is security.

The first security consideration is data location. Some constraints and legislative requirements specify, whether data can be stored outside a state boundary. The second issue is public availability of servers, which host these well-known services. The risk of attacks on those servers is much higher than on private computers [1]. It is therefore much more important to test these applications for vulnerabilities, for example like in [2] or [3].

The paper is further organized as follows: the second section introduces related work in web application security. The third section briefly summarizes the most used penetration testing tools. These tools are categorized by penetration testing phases. The fourth section presents penetration testing of a use case application with the most common examples of security flaws. The fifth section recommends the correct setting of web application technologies to protect them against the found security flaws.

II. RELATED WORK

Current work in web application security focuses especially on general security flaw analysis, or on implementation of specific security solutions. The approach to improve the evaluation of security characteristics was described in [4]. A precise evaluation is important during all phases of the application life-cycle and can be especially useful in finding the flaws during early stages of the application development. Early detection of the flaws can bring significant financial and time savings.

An analysis of key critical requirements for enhancing web application security was researched in [5]. The authors analysed the following areas: application and infrastructure security, communication and traffic inspection, zero-days attacks, dynamic application policies, sensitive data leakage, and user protection.

The more specific area of web application security – input validation – was researched in [3]. This area includes one of the most dangerous attacks: SQL injection and cross-site scripting. The authors proposed a systematic approach to secure this area using the security patterns approach.

Finally, one of the most important areas of web application security is user authentication. This was thoroughly analysed in [6], where authors compare existing solutions and propose a new scheme for mutual authentication using encryption primitives.

Although the mentioned research aims at specific areas of web application security, there is no research on a process of penetration testing, which would describe the most common security flaws in modern web applications.

III. PENETRATION TESTING TOOLS

The higher security risk of remotely run applications has to be verified, expressed and minimized. This process is known as penetration testing, or ethical hacking. The goal of the testing is to conduct a series of experimental attacks on the application. Based on found vulnerabilities, a level of risks is evaluated and a procedure to improve security issues is created. The main goal of the testing is therefore to improve the application security via pointing out its security flaws.

There are a large number of tools for penetration testing of web based applications. Moreover, most of these tools work on different security layers. Typically, there are two basic phases of the testing - reconnaissance and application exploitation.

A. Reconnaisance phase – passive

The first reconnaissance phase is conducted before the actual testing begins. The goal of this phase is to gather as much information about the target network as possible.

Unlike in other phases, the network itself is not accessed. Instead, only the publicly available databases and search engines are used to gather the useful information like web sub-domains, IP ranges, user emails etc.

Maltego [7] is a tool of OSINT (Open-source intelligence) type, which represents a group of tools using publicly available information sources. The tool uses lists of indexes and databases to search for relevant information.

Discover Scripts [8] is another tool of OSINT type. It integrates search and scanning utilities present in *Kali Linux* OS and therefore creates an automatized framework. This framework targets the following four areas: recon (used for passive scanning), scanning, web, and misc.

B. Reconnaisance phase – active

In the second reconnaissance phase, the testing tools interact directly with targeted devices. The goal is to identify used operating systems, running services and potential vulnerabilities. This type of scanning is normally considered as an attack on the network, and has to be authorised by the network owner.

Ettercap [9] is an open source multi-platform tool for network traffic sniffing. It allows the capture of packets and analysis of network protocols. In promiscuous mode, it can capture communication between two users located in the network. The *Ettercap* supports both passive and active scanning and contains several modules for the MitM type of attacks. If the unencrypted traffic is used, the *Ettercap* can be used to gather sensitive information like passwords.

Nmap [10] is a well-known open source tool used for network scanning. It can find connected networks end devices, their open ports, run services, and it can build a network map. Versions of operating systems, services, and running daemons can be found as well. This information can be used in combination with well-known vulnerabilities found in publicly accessible databases.

C. Reconnaisance phase – application scanning

The third reconnaissance phase focuses on an automatized scanning of web applications. The results can present a general idea of the application and give some guidance in exploiting existing flaws. On the other hand, common testing tools are often unable to find all the vulnerabilities due to the usage of modern technologies in common web applications. It is therefore often necessary to manually explore the application source code for discovering further flaws.

Arachni - Web Application Security Scanner Framework [11] is an automatized multi-plaform open sources scanning tool for security audit of modern web applications. The framework has an integrated browser engine, which allows scanning of modern complex web applications using advanced technologies like JavaScript, HTML5, DOM and Ajax. The framework is using asynchronous HTTP requests, parallel processing of JavaScript operations, and multi-thread scanning. The framework therefore maintains a high performance. *Arachni* is also able to generate a detailed analysis report with found vulnerabilities. Its detection abilities are very high with trustworthiness over 90%.

Testing of an application can take up to tens of minutes (depending on the application scale) and during this time, the application performance can be greatly reduced. It is therefore highly recommended not to use this type of scanning on an application used in a commercial sphere deployment.

OWASP ZAP (Zed Attack Proxy) Project [12] by OWASP (The Open Web Application Security Project) is a tool for web application scanning and it contains several modules for exploitation attacks. These modules include: *Proxy* (for communication capturing), *Scanner* (passive and active), *Fuzzer* (sequentially sends potentially dangerous payload in order to identify a vulnerability), *Spider* (traverses all the web pages from the initial URL in order to discover new sequences of the application), and *Forced browsing* (discovers direct access to files stored on the server, using dictionary method).

D. Web Application Exploitation

After the reconnaissance phase is done, the application exploitation phase can begin. This phase can include various tools depending on targeted exploitations. The most common categories and specific tools are described below.

SQL Tools like *SQLmap* and *NoSQLMap*. *SQLmap* [13] is a popular open source tool for testing the database part of a web application. A typical exploitation which can be found is the SQL injection. In the case of a successful exploitation, the *SQLmap* can access the operating system shell. The tool can save analysis results and data gathered from the database into a file. The attack itself can take a few minutes, depending on the scope of the database. The supported databases are: MySQL, Oracle, PostgreSQL, Microsoft SQL Server, Microsoft Access, IBM DB2, SQLite, Firebird, Sybase, SAP MaxDB, and HSQLDB.

NoSQLMap [14] is an open source tool targeting NoSQL databases. Currently it supports only MongoDB, but extensions for other NoSQL databases like CouchDB, Redis, or Cassandra are planned. The time needed for the testing is similar to *SQLmap* and data can be also saved into a file if the attack is successful.

Password attacks – the typical tool for password attacks is the *hashcat* [15] and its derivatives like *oclHashcat* and *cudaHashcat*. The *hashcat* is an open source tool supporting many hash algorithms (including MD, SHA, and bcrypt). The computation can run either on CPU (*hashcat*) or GPU (*cudaHashcat* on Nvidia and *oclHashcat* on AMD). The GPU performance can typically be much higher due to the parallelized architecture of modern graphic cards.

The *hashcat* contains the following attack modes: straight (classical dictionary attack), combination (words connected from multiple dictionaries), brute-force (mask specification allows to omit unused password combinations), permutation (changes positions of each letter in a single word), and table-lookup (each dictionary

word is broken down into single letters and mapped into another table).

Burp Suite - [16] is one of the most reputable platforms for penetration testing of web applications. It consists of the following modules: *Web vulnerability scanner* (continuously updated and therefore able to detect flaws in modern web technologies like REST API, JSON, AJAX, and jQuery), *Proxy* (captures and modify the communication), *Spider* (can create map of the web application and automatically sends forms), *Intruder* (realizes attacks based on performed analysis), *Repeater* (repeatedly modifies HTTP requests and compares their replies), and *Sequencer* (analyses the level of security tokens randomness). The *Burp Suite* is a very complex tool requiring a certain degree of user expertise. Unlike the previous tools, the *Burp Suite* is provided in two versions: free with limited functionality and professional with full features (paid).

BeEF - The Browser Exploitation Framework [17] is a very popular open source framework for penetration testing, focused on XSS attacks. The *BeEF* is also written modularly, so the new attack scenarios can be easily added. The main functionality of the *BeEF* is a hooking process, which allows the takeover of client web browser control. This process can be integrated with the *Metasploit Framework* [18] and found vulnerabilities can be used to gain access to the operating system shell.

IV. The Use Case Penetration Testing

A. Web Application for Tools Testing

In this section, the previously described penetration testing tools will be tested on a custom-made web application. The use case application simulates a typical modern web for E-library, and purposefully contains the most common vulnerabilities described in section 3. The application supports three types of accounts: administrator, librarian, and a customer. The application has the following functions:

- Book reservation
- Credit system for limiting the number of borrowed books
- Options to edit books, credits, and profiles
- Real-time chat based on the Socket.IO technology

The web application uses the following technologies: Node.js (web application back-end), Express (extends module for Node.js), Socket.IO (bidirectional communication between a client and the server), MariaDB (relation database for small to middle sized applications), MongoDB (stores unstructured web content), HTML5 (presentation part), CSS3 (design part), jQuery (local processing on a client side), and Ajax (asynchronous request processing, cooperation with jQuery). These technologies are common in modern web applications and they therefore present a good sample for security testing.

B. Testing Plan

To perform a complete penetration test, a testing plan has to be firstly created. There are many existing approaches and guides; one of the most common is the OWASP Testing Guide v4. This section will describe the sections of this plan, and how to test the most common security vulnerabilities. The complete process will be demonstrated on the use case application. As a testing platform, *Kali Linux 2.0* was chosen for penetration testing.

The testing plan of a private web application should include the five following scenarios:

1. Server and application scanning
2. Input data validation
3. Authentication and authorization
4. Client side vulnerabilities
5. The level of application configuration security

If the application is publicly available over the Internet, the additional scenario (preceding the server and application scanning) – the passive reconnaissance phase – should be added. This scenario will not be described, because the use case application is not publicly deployed and therefore no information could be found about it.

Server and application scanning is the first phase which conducts a search for vulnerabilities, which could be exploited later.

Server scanning (*nmap*) - firstly, the web server should be scanned for used operating system, open ports and running services. In the use case application, the *nmap* tool, suitable for this scanning, discovered the following facts: used operating system (Linux 3.2 - 4.0), open ports (22, 111, 3000, 3306, 27017, 28017, 50892) and running services (OpenSSH, RPC, Node.js, MariaDB). These results are shown in Figure 1.

Application scanning - an application should be scanned for vulnerabilities by a complex tool like the *Arachni* or *OWASP ZAP*. If an application contains sections which require a login, it is recommended to use authentication modules. Depending on the application, additional modules should be used. These modules will ensure, that all the application sections will be scanned for vulnerabilities. In our case, the *Arachni* found 44 vulnerabilities in the following categories: 12 high, 7

```
Nmap scan report for eknihovna.knytl (192.168.0.3)

PORT      STATE SERVICE VERSION
22/tcp    open  ssh     OpenSSH 6.7p1 Debian 5+deb8u1 (protocol 2.0)
111/tcp   open  rpcbind 2-4 (RPC #100000)
3000/tcp  open  http    Node.js (Express middleware)
3306/tcp  open  mysql   MariaDB (unauthorized)
27017/tcp open  mongod
28017/tcp open  mongod
50892/tcp open  unknown
MAC Address: B8:88:E3:DD:E0:E1 (Compal Information (kunshan))
Device type: general purpose
Running: Linux 3.X|4.X
OS CPE: cpe:/o:linux:linux_kernel:3 cpe:/o:linux:linux_kernel:4
OS details: Linux 3.2 - 4.0
```

Figure 1. Analysis report from server scanning by the *nmap* tool

medium, 5 low, and 20 informational. For comparison, the use case application was further subjected to a scan by *OWASP ZAP* with the *Proxy* and *Spider* modules. This testing allowed scanning of data flow (including password exchanges and forms submissions) between the server and a client; and detection of potentially hidden parts of the application. The tool found the following vulnerabilities: 2 high, 4 medium, and 6 low (no informational vulnerabilities were found). The *OWASP ZAP* was able to detect the more serious threats: XSS and SQL injection, which were not detected by the *Arachni*. Therefore, in our case, the *OWASP ZAP* results were more accurate and we recommend using this tool.

Input data validation should verify all the application data sources prone to access attacks like XSS (cross-site scripting) or injection. The following four scenarios are the most common areas to conduct an input data validation:

REST API (SQL injection) - each REST API source should be manually identified and their methods and parameters tested by the *OWASP ZAP* with the *Proxy* module. All the HTTP requests should have a valid session ID in order to verify operations requiring authentication. The testing of the use case application revealed parameters prone to SQL injection in all the REST API sources. These vulnerabilities could result in data leaks, unauthorized modification, or application instability.

REST API (XSS) - the same API should be further tested for stored XSS (*api.js* file) vulnerabilities. If some vulnerable API is discovered, HTTP requests can be captured using the *Proxy* module of the *OWASP ZAP* tool. These requests should then be moved into the *Fuzzer* module, where a *XSS.txt* file can be applied on them. This file contains the list of harmful payloads. Each parameter in our application was tested with the harmful payload and compared to response payloads. The comparison was automated with the custom script *restAPI-fuzzer,* but it can be done manually as well. The script was able to detect several unhandled data inputs.

NoSQL injection - incorrect handling of input data should be tested for appropriate databases such as the MongoDB. This database can be used, for example, for chat as in our application. In this case, the HTTP request for chat API was firstly captured with the *Proxy* module. The payload was then modified and sent back to the server. This modification allowed listing of all the messages (instead of listing only messages for a specific user).

Input data (Socket.IO) – the process of the Socket.IO communication testing has to be manually customized for every application. In the use case application, we created a testing script (*socketio-testclient*) and we used the fuzzing method for sending a harmful payload via Socket.IO. This test revealed, that the input data is not secured for XSS, resulting in displaying dialogue windows caused by the harmful payload.

Authentication and authorization is the third phase and should contain at least the following four scenarios:

Level of login component security - this scenario verifies vulnerabilities of a login component. Attacks, like SQL injection, could result in a bypass of the login process. The conducted reports from the first phase should already pointed out if the login form is prone to SQL injection attacks. These found vulnerabilities can be further tested by the *Burp Suite* and its *Intruder* module, or by the *Fuzzer* module of the *OWASP ZAP*. If the vulnerability is confirmed as in our case, the *SQLmap* can be used for database scheme gathering.

Access to unauthorized sections of an application – is the OWASP Top 10 A7 vulnerability and should be thoroughly tested. One of the approaches is to use the *OWASP ZAP* with the *Proxy* module for user login (with client credentials). Afterwards, all sections available to this user can be accessed. Consequently, the *Forced Browse* attack can be conducted with the default *OWASP ZAP* dictionary. In our case, 708 520 requests were sent and the attack found 4 sections, which should be accessible only to the administrator. This indicates, that some sections of the application are not using authorization verification. This was confirmed by the following authorization verification conducted by the *Proxy* and *Intruder* modules of the *Burp Suite*.

Session hijacking – the goal of this attack is to discover the session ID of a connected client. The attacker can then login to the application without the knowledge of user credentials. In custom applications, the automated scanning tools are typically unable to detect the session ID. This happened in our application as well, due to the different application session signature. In this case, the manual approach via a packet capturing tool (for example the *Wireshark*) has to be used to discover the session ID.

Socket.IO authentication – this manual test verifies if the Socket.IO is accessible only after a successful authentication. In the use case application, the customized script *socketio-testclient* was connected to the URL: *http://192.168.0.3:3000*. After the script was launched, the console response showed: *"Connecting to the socket was successful"*, indicating, that no authentication was necessary. This could result in subsequent attacks.

Client side vulnerabilities tests an important part of the application security – the client side. Two basic scenarios should be tested:

XSS vulnerability exploitation - this scenario uses an unsecured input of unfiltered XSS. The *BeEF* tool with the *hook.js* script can be used to exploit the vulnerability. In order to run the script, a link to the *hook.js* file had to be firstly put into the application database. This can be done using many approaches. In our test, we simply sent the link to the user chat. After the successful hooking process, the information about the client's browser and its stored cookies can be gathered. Additionally, the web content can be spoofed as well.

CSRF (Cross-Site Request Forgery) exploitation – is a malicious JavaScript code, which executes an attack when it is accessed. Such a code was added into the form on the *Profile* page. If a user is logged into the application and accesses this page (the link can be sent by the chat), the inserted JavaScript executes the attack. Our attack

contained a hidden form with request to change the user password.

The level of application configuration security is the last phase and should be tested in the following four scenarios:

Stolen hashed passwords - this test verifies a situation where a text file with hashed password is stolen. Based on the hash password length analysis, the length of the hash function can be determined. In our application, the 40 hex long password corresponds to 40 * 4 (hex) = 160 bits. Then, the used hashed function can be guessed (in our case SHA1). Finally, an appropriate tool can be used to break the passwords. We used the *hashcat* tool with dictionary *rockyou.txt* and we also tested various breaking methods. The *Straight* method was able to break 13 of 25 passwords in 5 seconds. The same number of passwords was broken by the *Table-lookup* method in 67 seconds. The last method, *Combination*, was able to break only 6 passwords in 90 minutes (and the remaining time was estimated to 12 hours).

Sensitive data exposure - the test verifies MitM attack, which can capture usernames and passwords when a user is logging into an application via the HTTP. The *Ettercap* tool can be used for sniffing the connections via the ARP poisoning. In the use case application, this attack was able to capture the username and password for every client logging into the application.

Sensitive data theft - this scenario verifies the possibility of access to sensitive data. Because it depends heavily on the application context, this analysis has to be conducted manually. The *Burp Suite* can be used to map all the HTTP requests of the application's REST API. In the use case application, the captured JSON files showed hash of the user passwords, which could then be misused.

MongoDB security – if a database is used in an application, its security should be tested. Firstly, the *NoSQLMap* can be used to scan an application's sub network for discovering the database's local IP address and port. The database can then be exploited with the *NoSQL Web App* attack. In the use case application, the database was successfully found and data was cloned into a local file. All the chat messages could therefore be exploited.

C. Testing Summary

Tested vulnerabilities and used tools are summarized in the table 1. The scenarios, where threats could be found using automatized tools are marked as *Automatic,* otherwise they are marked as *Manual* or *Combination*. In these later cases, the tools had to be combined with methods of manual code analysis, or custom made scripts, requiring the more consistent knowledge about the security issue.

Only the vulnerabilities from the server and application scanning part can be found using automatized tools. The reason why using automated tools in other categories is not enough, is the complexity and novelty of modern web technologies. The automated tools can be able to detect these vulnerabilities only if they are updated frequently, which is not always the case. For this reason,

TABLE I. PENETRATION TEST SUMMARY

Tested vulnerability	Used tools	Method
Open ports	Nmap	Automatic
Vulnerability scanning	Arachni, OWASP ZAP	Automatic
SQL Injection	OWASP ZAP (Proxy), SQLmap	Automatic
Data validation (XSS)	OWASP ZAP (Proxy), Fuzzer	Combination
NoSQL Injection	OWASP ZAP (Proxy)	Automatic
Data validation (Socket.IO)	Code analysis, testing scripts	Manual
SQL Injection - login	Burp Suite (Intruder), SQLmap	Automatic
Authorization	OWASP ZAP (Proxy), Forced Browse	Combination
Session hijacking	Wireshark, Burp Suite	Manual
Socket.IO vulnerability	Testing scripts	Manual
XSS exploitation	BeEF	Combination
CSRF exploitation	Burp Suite / Ajax, JavaScript	Manual
Password encryption strength	Hashcat	Automatic
User credential theft	Ettercap	Automatic
Sensitive data theft	Burp Suite	Combination
MongoDB security	NoSQLmap	Automatic

there will always be a delay between the introduction of new web technology and implementation of threat detection into these automated tools.

V. SECURITY RECOMMENDATIONS FOR THE APPLICATION

Based on the typical vulnerabilities from the section 4, the following recommendations were created to significantly increase the security of web applications.

Server and application scanning part - Opened database ports should be disabled for remote access. This can be accomplished by binding these ports on the loopback interface (127.0.0.1). The password autocompletion should be disabled in all the input password fields (set autocomplete parameter to off). To protect all the replies against *clickjacking attack*, X-Frame-Options should be sent in a reply header. This can be done with the following setting: *app.use (helmet.xframe('deny'));* To protect the session against the XSS attack, it has to be set to inaccessible for JavaScript. This can be done by setting a *HttpOnly* flag for all the HTTP responses: *cookie: {httpOnly: true, secure: true}*.

Input data validation part – for elimination of SQL injection attacks, the database queries should use parameter bindings instead of their ad-hoc creation. An alternative is to use "escaping" of input parameters. NoSQL injection should be prevented by using input data validation for example with the *mongo-sanitize* module and its *sanitize* function. In the case of numeric input data, these variables should be explicitly cast into numeric data types. Additional data validation can be conducted by using regular expressions, or by the *XSS* module.

Authentication and authorization part - The input parameters of all functions should be "escaped" (for example: *conn.escape(req.body.name))* or used via a parametrized query. Unauthorized access can be mitigated by using an ACL module, or by implementing a custom authorization middleware. Session ID can be effectively protected by using HTTPS and an already mentioned secured cookie. To protect the Socket.IO access via authentication, the module *socketio-auth* should be implemented.

Client side vulnerabilities part - validation of input parameters (manually, or with the *XSS* module) has to be implemented to protect the application. Prevention against the CSRF attack can be done by implementing a hidden authorization token. This randomly generated number will ensure the uniqueness of every request. The implementation example is the *csurf* module.

The level of application configuration security part - better protection of stored passwords can be ensured by the *password salting*. This will increase the password length and add randomness into the stored passwords. An additional recommended measure is to enforce the password policies like minimal password length, and a need to include lower-case, upper-case, and special characters. To protect against sensitive data exposure, the HTTPS protocol should be used. Exposure of sensitive data is a logical flaw of an application design. This error should be detected and corrected in the application development phase. The *Burp Suite*, or the *OWASP ZAP,* can be used to test HTTP requests and responses to detect such flaws. The used databases (*MongoDB)* should be secured by their binding on the local loopback. If a remote access to the database is required, the database should work in the *secure* mode. This mode will ensure secure authentication of clients accessing the database.

VI. Conclusion

The performed penetration testing verified the usability of current automated scanning tools. While these tools were able to find most of the vulnerabilities, the testing also confirmed, that some vulnerabilities were not detected. This is in most cases, caused by usage of modern web technologies, which are not yet implemented in these scanning tools. Therefore, the usage of modern technologies does not typically ensure the maximal security, if only the automated tools are used for security testing.

Performing high quality penetration testing is a time-consuming task requiring knowledge of various technologies. Automatized scanning tools can be used to quickly gain a general idea about the application security status, but cannot present a complex analysis. To verify all the aspects of the application, more specialized tools have to be used together with manual code analysis, and writing of custom scripts. It is also important to mention the influence of the penetration testing on an application performance. In more intrusive testing, the application should not be deployed in the production.

Acknowledgment

We would like to thank Radek Knytl for his contribution to this paper. This work and contribution is supported by the project of the student grant competition of the University of Pardubice, Faculty of Electrical Engineering and Informatics.

References

[1] S. Rafique, Humayun, M., Hamid, B., Abbas, A., Akhtar, M., Iqbal, K., "Web application security vulnerabilities detection approaches: A systematic mapping study," in: 2015 IEEE/ACIS 16th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD). pp. 1–6 (June 2015)

[2] M. Alenezi, Javed, Y., "Open source web application security: A static analysis approach," in: 2016 International Conference on Engineering MIS (ICEMIS). pp. 1–5 (Sept 2016)

[3] J. Sohn, Ryoo, J., "Securing web applications with better "patches": An architectural approach for systematic input validation with security patterns," in: 2015 10th International Conference on Availability, Reliability and Security. pp. 486–492 (Aug 2015)

[4] H. Hakim, Sellami, A., Abdallah, H.B., "Evaluating security in web application design using functional and structural size measurements," in: 2016 Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement (IWSM-MENSURA). pp. 182–190 (Oct 2016)

[5] Kumar, R., "Analysis of key critical requirements for enhancing security of web applications," in: 2015 International Conference on Computers, Communications, and Systems (ICCCS). pp. 241–245 (Nov 2015)

[6] A. Al-Bajjari, Yuan, L., "Optimized authentication scheme for web application," in: 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA). pp. 52–58 (Nov 2016)

[7] Paterva: Maltego (2016), [online]. Available: https://www.paterva.com/web7/index.php

[8] L. Baird, Discover - github (dec 2016), [online]. Available: https://github.com/leebaird/discover

[9] A. Ornaghi, "Ettercap home page" (dec 2016) [online]. Available: https://ettercap.github.io/ettercap/

[10] G. Lyon, "Nmap: the network mapper" (dec 2016) [online]. Available: https://nmap.org/

[11] A. Laskos, "Nmap: the network mapper" (dec 2016) [online]. Available: http://www.arachni-scanner.com/

[12] OWASP: Owasp zed attack proxy project (dec 2016) [online]. Available: https://www.owasp.org/index.php/ZAP

[13] B. Damele, Stampar., M., "sqlmap: automatic sql injection and database takeover tool," (dec 2016) [online]. Available: http://sqlmap.org/

[14] Nosqlmap (dec 2016) [online]. Available: https://github.com/tcstool/nosqlmap

[15] J. Steube, "hashcat - advanced password recovery," (dec 2016) [online]. Available: https://hashcat.net/hashcat/

[16] PortSwigger Ltd, "Burp suite" (dec 2016) [online]. Available: https://portswigger.net/burp/

[17] W. Alcorn, "Beef - the browser exploitation framework project," (dec 2016) [online]. Available: http://beefproject.com/

[18] Rapid7, "metasploit," (2016) [online]. Available: https://www.metasploit.com