Automated versus Manual Approach of Web Application Penetration Testing

Navneet Singh[1], Vishtasp Meherhomji[2], B. R. Chandavarkar[3]
Department of Computer Science and Engineering
National Institute of Technology Karnataka
Surathkal, Mangalore, India
dwordnavneet@gmail.com[1], vishtasp.sm@gmail.com[2], brcnitk@gmail.com[3]

***Abstract*—The main aim of this work is to find and explain certain scenarios that can demonstrate the differences in automated and manual approaches for penetration testing. There are some scenarios in which manual testing works better than automatic scripts/vulnerability scanners for finding security issues in web applications. In some other scenarios, the opposite may be true. The concepts of various web application vulnerabilities have been used for testing, including OWASP[1] Top 10, using both manual and automatic approaches. Automation tools and scripts have been used and tested to see what could potentially go wrong if attackers exploit such vulnerabilities. Also, certain scenarios have been used which determine whether one approach is better than the other for finding/detecting security issues in web applications. Finally, the work concludes by providing results in the form of pros-and-cons of both approaches, which it realises after carrying this out.**

***Index Terms*—Web Application, Penetration Testing, OWASP, Vulnerability**

I. Introduction

Web applications play a vital role in our daily life. Whether we want to do financial transactions or social conversations, web applications are convenient for all daily needs. Most web applications depend on the trust of their users for them to be profitable for their owners. Web applications that deal with subjects such as online banking, social media, and other similar topics require confidentiality, integrity, and other security measures to keep the trust of users in them.

However, because no technology is entirely secure, some vulnerabilities could allow attackers to harm users in one way or another. If users find that specific web applications are vulnerable to attacks by malicious entities, they will stop visiting and using those web applications. The loss of user trust severely affects the business model of web applications. Hence it is of paramount importance to ensure that the web pages are free of vulnerabilities.

Penetration testing of web applications [1] [2] [3] for finding/detecting their security issues is very crucial in order to protect users' data and trust. It involves the use of various tools and techniques [4] [5], generally employed by penetration testers[2], to determine whether a web application is susceptible to attacks by malicious users. Such testing can be broadly categorised into two approaches: manual and automated [6] [7].

Manual approach for penetration testing [7] mainly uses suitable tools which let penetration testers to apply the concepts they learnt with time and experience. These may be openly available tools or even scripts that the testers themselves write to analyse some particular vulnerabilities. Whatever the case, the manual approach to penetration testing involves a person (or a group of people) who must carefully inspect the website for weaknesses.

On the other hand, automated testing for vulnerabilities [2] [8] [9] is mainly done by automated scripts and vulnerability scanners, which can be used by anyone[3]. These scanners look for generally well-known vulnerabilities, which are often found on websites and can be easily targeted by malicious users for harming websites.

An important area of interest is the study of the differences in these two approaches [7]. While it is possible to say without a doubt that automated testing, using vulnerability scanners, is significantly faster in comparison to the manual method, an important question that arises is this: How effective are they? While many studies in this area discuss their work by providing their comparison results, we provide specific scenarios highlighting the differences in the effectiveness of these approaches. Both methods have the potential for outperforming the other in different situations, and our work focuses on highlighting some of them.

The rest of this paper is structured as follows: Section II discusses some similar work done by our peers in this area and their discussions of specific attack scenarios in real-world applications. That is followed by particular scenarios that demonstrate differences in the effectiveness of the two approaches towards penetration testing in Section III. Section IV discusses mitigation techniques. Section V presents our results and conclusions based on our observations, which we recorded as we carried out this work.

II. Literature Survey

Penetration testing for web applications is a widely explored field within the domain of web security. Many white hat

[1]Open Web Application Security Project; online community dedicated to web security

[2]In this paper, 'tester' refers to a human who is performing manual testing on a web application.

[3]Many automatic vulnerability scanning tools are available for use. For example, the OWASP Zed Attack Proxy (ZAP) is one of the popular ones.

hackers regularly look for and discover vulnerabilities in websites. The means and tools used by them are also a field which is keenly watched and receives significant scrutiny.

Much research has gone into exploring the different tools which testers use during their search for vulnerabilities. Hessa Mohammed Zaher Al Shebli et al. [4] have discussed penetration testing along with multiple scenarios that testers must consider during the process. They also discuss various tools and software which web application penetration testing uses.

Automated penetration testing is another lucrative field that researchers explore to a great extent. Researchers are always on the lookout to improve the abilities of automatic penetration testing tools — Kevin P. Haubris et al. [2] attempt to improve penetration testing automation using various existing tools.

Much research goes into analysing the differences in the effectiveness of automated and manual approaches for penetration testing. Nuno Antunes et al. [6], as well as Yaroslav Stefinko et al. [7], in their respective papers, focus on the differences they observe in the two approaches. Sangeeta Nagpure and Sonal Kurkure, within one part of their paper [3], also discuss these differences.

In our paper, we discuss various scenarios wherein the differences between the two approaches are apparent. Naturally, much research goes into the individual vulnerabilities which exist in web applications as well.

Concerning clickjacking, Mohammad R. Faghani and Uyen T. Nguyen [10] discuss its propagation in social media. Daehyun Kim and Hyoungshick Kim [11] discuss the surprisingly high prevalence of clickjacking vulnerabilities in websites. Rakhi Sinha et al. [12], discuss some approaches to defend against this attack.

For cross-site-scripting (XSS), Mohit Dayal et al. [13] discuss it and its impacts. Ankit Shrivastava et al. [14], as well as Shaimaa Khalifa Mahmoud et al. [15] in their respective papers, discuss prevention and mitigation of XSS attacks.

Jin Huang et al. [16] discuss an automated tool for detecting vulnerabilities that arise from uploading files to web applications. Xiao Cheng et al. [17] discuss the detection of vulnerabilities arising out of errors in the business logic of web applications.

Most papers that discuss the differences between automated and manual methods of penetration testing do not give much attention to scenarios in which such differences are generally observed. The motivation behind this paper is to do precisely that. This paper focuses on highlighting some situations which can demonstrate how the behaviours of the two methods differ.

III. Attack Scenarios

This section discusses some vulnerabilities that are frequently exploited by attackers to gain unauthorised access to a website or harm its users in other ways. The general concept of an attack is explained, followed by a specific scenario that can be used by an attacker. There exist many more ways in which attackers attempt to harm websites. However, this section only discusses a select few, which are more prevalent.

A. Clickjacking

In layman's terms, if a website allows a different website to load it into its *iframe* tag, it is vulnerable to clickjacking/UI redressing [10] [11]. A malicious website can load the trusted website into its *iframe*. Then, by using CSS[4], another layer of the user interface can be kept on the real interface. An attacker can make the actual UI invisible by using CSS, which lets users see only the attacker's UI layer. Now users can be fooled to click on this layer's button, unknowingly clicking on the functionality of the trusted website. Observe Fig. 1, which shows the general flow of a clickjacking attack on a vulnerable website.

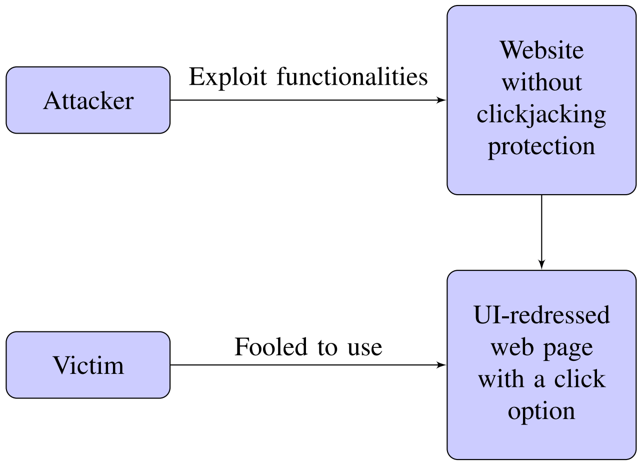
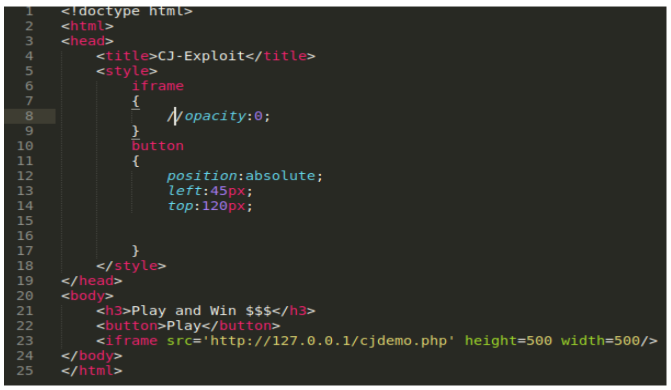

Attacker
Exploit functionalities
Website without clickjacking protection
Victim
Fooled to use
UI-redressed web page with a click option

Fig. 1. Clickjacking

Simple Attack Scenario: Suppose there is a functionality that allows a user to delete their account on the page displaying account settings. An attacker can load this web page into their malicious *iframe* tag on their website. Further, the attacker can overlap the fake button on top of this delete button, making this user interface invisible. Now, the attacker can send this link to the user. Because the website has already authenticated the user, this *iframe* will load the user's account settings page.

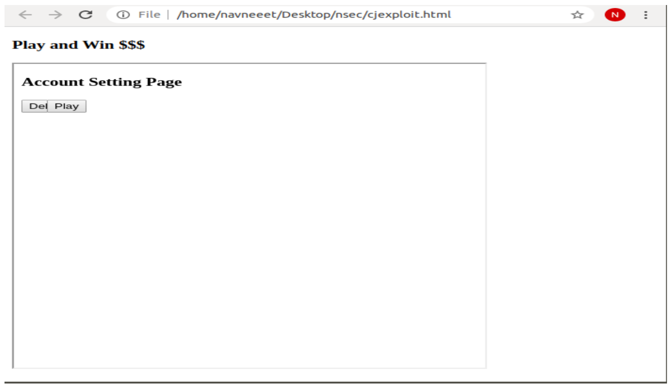
[4]Cascading Style Sheets; used alongside hypertext markup for document presentation.



```
<!doctype html>
<html>
<head>
    <title>CJ-Exploit</title>
    <style>
        iframe
        {
            //opacity:0;
        }
        button
        {
            position:absolute;
            left:45px;
            top:120px;


        }
    </style>
</head>
<body>
    <h3>Play and Win $$$</h3>
    <button>Play</button>
    <iframe src='http://127.0.0.1/cjdemo.php' height=500 width=500/>
</body>
</html>
```

Fig. 2. Code for Exploit



Play and Win $$$

Account Setting Page

Fig. 3. Rendered page of Exploit

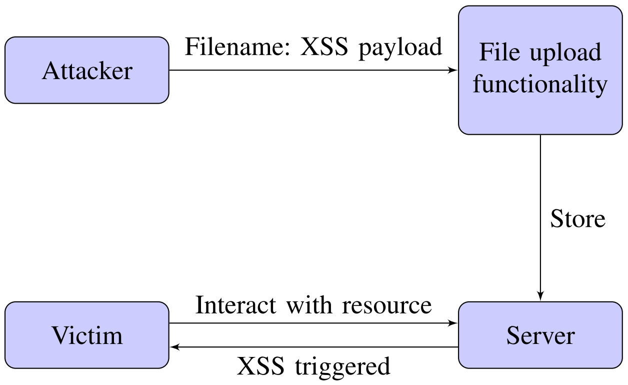


Fig. 4. Cross Site Scripting (XSS)

The user will be tempted to click, unknowingly deleting their account.

Fig. 2 shows the code for the above mentioned simple attack scenario. When a browser renders this, it looks like Fig. 3. Fig. 3 shows two layers: legitimate layer with the heading ‘Settings page’ and ‘Delete account’, and a fake layer with a ‘Play and Win $$$’ heading and a fake ‘Play’ button which overlaps the ‘Delete account’ button. If the attacker makes opacity = 0 in Fig. 1, then the legitimate layer will be invisible, fooling the user to click the button present.

B. File Upload Vulnerabilities

File upload functionalities can lead to many vulnerabilities if websites do not handle them properly. Following is one of the cases wherein XSS is explained.

XSS Through File Upload: Cross-site-scripting (XSS) [13] [14] [15] is a vulnerability which occurs when user inputs are not sanitized. These inputs could be HTML tags that let malicious users inject JavaScript. The addition of this JavaScript could lead to the stealing of cookies or other harmful effects.

Simple attack scenario: Suppose there exists a social-networking website where a user can comment on an article. Other users can then read the comments. A malicious user can inject the following payload:

<img src=“https://someattackerwebsite.com/x?=document. cookie” >

If the website is not handling the user input correctly, this payload will be rendered in the comments section, causing any user who visits this article to lose their cookies to the attacker, thus giving them access to the user’s account.

Consider another scenario, one in which a website has the functionality of uploading an image. If it reflects the name of the uploaded file, then an attacker can rename the image as a payload and then upload this image. Once the image gets uploaded, this payload will get triggered. Even after having sanitised all other inputs, a developer may forget to take care of the one case mentioned above (and as this is something that is rarely paid attention to, it may have slipped from their mind). Observe Fig. 4, which shows how an attacker can trigger an attack at the victim’s side via XSS.

C. Sensitive File Discovery

Websites can contain many files, some with sensitive data, which should not be present there. If an attacker finds them, they can use the data for malicious purposes. Sometimes these sensitive files could be text files that consist of database credentials. Then, the effort of securing the database from different security issues goes in vain. It could easily be a file in a standard format that contains sensitive data that should not be public.

Simple Attack Scenario: Consider a web application that is using GitHub for version control. Since many developers use the freely available public repositories, they may leave private API keys, which can then be accessed by anyone. Someone who wishes to obtain an API key without purchasing can easily search across GitHub for relevant API keys. Since API calls often incur costs to their owners, such an illegitimate user could cause the owner to have to pay for a service they did not use.

D. Business Logic Vulnerability

This class of vulnerability mostly arises due to bad logic rather than poor practice. When developers do not look at web applications’ functioning as a whole, there may be chances of business logic failure leading to business logic vulnerabilities, which could harm the web application. This vulnerability does not have any specific and fixed impact. The impact depends on the functionality of web application and how business logic fails and what an attacker can achieve by exploiting that failure of business logic.

Simple Scenario: The example mentioned below can explain the vulnerability in an effortless yet effective way.

Consider a web application that is designed only for the employees of a company. If an employee leaves the job, that employee should not have access to the web application anymore. However, if this is not taken care of, business logic vulnerability arises, which leads to unauthorised access to the web application because that employee is not authorised to access it anymore.

IV. Mitigation

This section describes the general practices which are required for the mitigation of the vulnerabilities introduced above. The methods described here are insufficient if developers do not implement them correctly, causing them to pass testing despite being vulnerable. This section does not aim to be a complete guide for the mitigation of the attack scenarios discussed. The methods described here are some of the commonly used mitigation techniques, and this is in no way a complete list.

A. Clickjacking

The prevention of clickjacking [12] is essential to ensure the safety of the users of the web application. The most common solution here is to use the *x-frame-options* header to restrict the website from being rendered by different websites into their *iframe*.

x-frame-options header uses the following directives:

- *x-frame-options: deny*
  This directive does not allow any website to load the target website into its *iframe*.
- *x-frame-options: sameorigin*
  This directive only allows a website which has the same origin as the target website.

Thus, the solution is to use the appropriate header to prevent clickjacking.

B. File Upload Vulnerabilities

There must be strict restrictions on uploading the files. The web application should check all the characteristics of a file that needs to be uploaded. As mentioned earlier, the file upload functionality can give rise to different vulnerabilities, and XSS could be one. So, there is a need for restrictions to be placed in all fields, ranging from filename to content type [16].

C. Sensitive File Discovery

Sensitive file discovery is a part of the information gathering process when a penetration tester uses various methods to find sensitive files. The mitigation of it involves educating developers to realise what is confidential and should not be available publicly.

Just because there is no direct link to sensitive files does not mean that no one will be able to access them. As a result, anything which contains confidential information about the web application, the organisation which owns the web application, employees' data, or other crucial information should not be kept in systems connected to the internet.

D. Business Logic Vulnerability

The mitigation of vulnerabilities of this kind is done by developers' rational thinking while developing the web application. Developers should thoroughly test all functionalities of web applications for each case. They must understand which situations could give rise to unintentional business logic failure. There are even attempts [17] at creating automated tools for detecting such vulnerabilities.

Business logic failures can be found by even the people who do not belong to the domain of computer science or technology. General people can also think of a question: What if? What if a customer performs this task; how will the application behave? Will it cause any security hazards to the web application or company? If it is causing any harm, then it is necessary to take care of it.

V. Results

Web application developers are usually careful enough to apply mitigation techniques for common vulnerabilities. However, despite their efforts, some vulnerabilities can remain in the application.

The previous section explained some mitigation techniques. These are generally considered robust methods to prevent attacks, as long as developers implement them with caution. Despite making use of these techniques, vulnerabilities can still exist. Some specific scenarios related to this are discussed here.

The main goal of this section is to describe how well automatic and manual penetration testing methods succeed in detecting these vulnerabilities. They both have their advantages and disadvantages [7], which this section explores.

A. Clickjacking

Consider a website that uses *x-frame-origin:sameorigin* to load itself for some required functionality. If it gives access to this functionality to the user, it could turn into the clickjacking vulnerability.

Consider the following scenario:

Suppose a website has implemented *x-frame-origin:sameorigin*, thus letting only the same website load itself. In other words, no website except for itself can load the target website. Therefore, it is secure against clickjacking.

However, imagine this website has the functionality of an HTML editor, allowing users to create web pages and share them with other users of the same website. A malicious user can then load the settings page of the website (which may contain a button for deleting the user's account) into the *iframe* tag and use CSS to redress the user interface. Now, the malicious user can share this with other users, which would fool an innocent user into clicking the button, causing the deletion of their account.

Automatic vulnerability scanners look for the implementation of the *x-frame-origin* header. If they find it to be present, they decide that the website is free from clickjacking attacks. However, the above scenario shows that this can fail despite the presence of this header. On the other hand, a tester, carrying out manual penetration testing, can detect this vulnerability.

For example, ZAP[5] determined a test website safe against clickjacking by checking for the presence of *x-frame-origin* header, despite the vulnerability mentioned above being present.

[5]Zed Attack Proxy; a popular testing tool for web application security

The above discussion demonstrates one simple scenario wherein automatic vulnerability scanners fail to detect what manual penetration testers would catch, with their experience and lateral thinking.

B. File Upload Vulnerabilities

Vulnerability scanners check for input sanitation. However, consider image inputs. Here a website may wish to display the name of the image file as uploaded by a user. These are often overlooked, which leads to failure in detection by vulnerability scanners. There are many ways in which input sanitation can be overlooked. Manual testing of a target website has a higher chance of detecting such lapses in sanitation by an experienced and sharp tester.

C. Sensitive File Detection

Automatic vulnerability scanners are well equipped to determine when developers or users post sensitive data in a public arena. The most common kinds of sensitive data include passwords, API keys, PINs, etc. These have a familiar structure and are easily identifiable by automatic vulnerability scanners.

As an example, users may push their application's source code to GitHub, and the source code may contain API keys provided by Google. If this occurs, the owner of the key receives an email notification from Google alerting them of the public availability of the key. Automatic scanners accomplish this, keeping vigilance regarding such activities.

While manual testing can obtain the same result, automatic scanners are much faster at scanning through tons of possible locations where such vulnerabilities may be present. It saves a vast amount of workforce and brings such issues to light at the earliest.

D. Business Logic Vulnerability

Computer systems are not smart enough to know precisely how developers wish for them to behave. Systems behave in the precise way in which developers program them. A business logic vulnerability arises when developers make a logical error in their programs. It might even be something as simple as subtracting a value in place of adding it or even something which has been entirely forgotten.

In the example explained earlier, if an employee leaves the company, the system does not automatically know this. A user must deliberately provide this information to it. If the developers design the system such that it revokes all employee permissions as soon as a user notifies it of their leave, then it is good. In case that is not so, someone must manually revoke the employee's permissions.

Either way, no automatic scanner is capable of detecting such a vulnerability. Until such a time when computers become intelligent enough to think as humans do, business logic vulnerabilities must be examined using manual testing.

E. Summary

To summarise all that has been discussed, observe Table I. This work discusses four scenarios that highlight certain situations, not always taken care of by application developers. While manual testing proves to be more effective at detecting a higher number of vulnerabilities, automatic testing is excellent at doing its task in a short time, thus saving a significant amount of workforce. Both have their advantages and disadvantages in different scenarios [7].

Scenario	**Vulnerability**	**Manual**	**Automatic**
Clickjacking	*sameorigin*	✓	✗
File upload	XSS	✓	✗
Sensitive files	public domain	✗	✓
Business logic	logic failure	✓	✗

TABLE I
SUMMARY OF THE RESULTS OBTAINED FOR THE SCENARIOS OBSERVED

The Pros and the Cons: Both methods, i.e., automated testing and manual testing, have their advantages and disadvantages. Following is a list of them for both the methods:

Automated Testing: Pros

- Faster scanning of web applications.
- Enables saving significant workforce over scanning a large number of locations in an application.
- Continuous and quick evaluation of user-provided data is possible.

Automated Testing: Cons

- Unable to catch exceptional cases of vulnerabilities.
- Prone to a significant number of false negatives as well as false positives.

Manual Testing: Pros

- With their experience and lateral thinking, humans can locate vulnerabilities that are missed by automated scanners.
- Can realise alternate security techniques used by developers, thus reducing false positives in the detection of vulnerabilities.

Manual Testing: Cons

- It is significantly slower, and it can take a long time for an application to be thoroughly tested.

VI. Conclusion

The testing of web applications is essential to detect vulnerabilities. While there are two broad categories of vulnerability testing viz. manual testing and automatic testing, they both perform differently under certain different circumstances. As can be observed from the scenarios highlighted above, manual testing proves to be more effective at detecting some unique vulnerabilities. Since automatic scanners are developed using knowledge of commonly occurring vulnerabilities, they often miss the uncommon ones. Undoubtedly, automated scanners prove to be more effective in cases where common vulnerabilities need to be checked for, thus saving a considerable amount of workforce. They are also better at scanning through a large number of locations where sensitive data may be present.

It is essential to mention that this paper does not cover all possible attack scenarios on web applications. A chosen few have been highlighted in this paper. Another significant limitation to keep in mind here is that the tests over these scenarios were conducted using web applications, running locally, explicitly designed to test for them. It is possible that when these scenarios exist in real, live web applications, inexperienced testers performing manual penetration testing may be unable to catch such vulnerabilities.

On the whole, manual testing for vulnerabilities is a better approach since the experience and lateral thinking of people performing manual testing allows for the detection of vulnerabilities that automatic scanners are unable to find. Automated scanners help speed up vulnerability testing, but they are by no means entirely sufficient to ensure that web applications are fully secure. It is thus a wise choice to make use of the advantages of both methods. While manual testing cannot be ignored, automatic testing can enhance its role in web application vulnerability detection. Testers can make use of automated tools to assist them in detecting vulnerabilities. Testers often create their automated scripts to run specific tasks for identifying weaknesses. Commercial vulnerability scanners cannot provide these tailor-made scripts, so it depends on the tester's skill to create them and use them appropriately. Automated tools are especially useful for beginners who are still learning to explore an application and find vulnerabilities. As they gain experience, their dependence on these tools can reduce; however, their use cannot be entirely neglected.

REFERENCES

[1] S. Mohammad and S. Pourdavar, "Penetration test: A case study on remote command execution security hole," in *Fifth International Conference on Digital Information Management (ICDIM)*. Thunder Bay, ON, Canada: IEEE, July 2010.

[2] K. P. Haubris and J. J. Pauli, "Improving the efficiency and effectiveness of penetration test automation," in *10th International Conference on Information Technology: New Generations*. Las Vegas, NV, USA: IEEE, April 2013.

[3] S. Nagpure and S. Kurkure, "Vulnerability assessment and penetration testing of web application," in *International Conference on Computing, Communication, Control and Automation (ICCUBEA)*. Pune, India: IEEE, August 2017.

[4] H. M. Z. A. Shebli and B. D. Beheshti, "A study on penetration testing process and tools," in *IEEE Long Island Systems, Applications and Technology Conference (LISAT)*. Farmingdale, NY, USA: IEEE, May 2018.

[5] A. Tetskyi, V. Kharchenko, and D. Uzun, "Neural networks based choice of tools for penetration testing of web applications," in *IEEE 9th International Conference on Dependable Systems, Services and Technologies (DESSERT)*. Kiev, Ukraine: IEEE, May 2018.

[6] N. Antunes and M. Vieira, "Comparing the effectiveness of penetration testing and static code analysis on the detection of sql injection vulnerabilities in web services," in *15th IEEE Pacific Rim International Symposium on Dependable Computing*. Shanghai, China: IEEE, November 2009.

[7] Y. Stefinko, A. Piskozub, and R. Banakh, "Manual and automated penetration testing. benefits and drawbacks. modern tendency," in *13th International Conference on Modern Problems of Radio Engineering, Telecommunications and Computer Science (TCSET)*. Lviv, Ukraine: IEEE, February 2016.

[8] G. Chu and A. Lisitsa, "Poster: Agent-based (bdi) modeling for automation of penetration testing," in *16th Annual Conference on Privacy, Security and Trust (PST)*. Belfast, UK: IEEE, August 2018.

[9] L. Li and L. Wei, "Automatic xss detection and automatic anti-anti-virus payload generation," in *International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC)*. Guilin, China: IEEE, October 2019.

[10] M. R. Faghani and U. T. Nguyen, "A study of clickjacking worm propagation in online social networks," in *Proceedings of the 2014 IEEE 15th International Conference on Information Reuse and Integration (IEEE IRI)*. Redwood City, CA, USA: IEEE, August 2014.

[11] D. Kim and H. Kim, "Performing clickjacking attacks in the wild: 99% are still vulnerable!" in *First International Conference on Software Security and Assurance*. Suwon, South Korea: IEEE, July 2015.

[12] R. Sinha, D. Uppal, D. Singh, and R. Rathi, "Clickjacking: Existing defenses and some novel approaches," in *International Conference on Signal Propagation and Computer Technology (ICSPCT)*. Ajmer, India: IEEE, July 2014.

[13] M. Dayal, N. Singh, and R. S. Raw, "A comprehensive inspection of cross site scripting attack," in *International Conference on Computing, Communication and Automation (ICCCA)*. Noida, India: IEEE, April 2016.

[14] A. Shrivastava, S. Choudhary, and A. Kumar, "Xss vulnerability assessment and prevention in web application," in *2nd International Conference on Next Generation Computing Technologies (NGCT)*. Dehradun, India: IEEE, October 2016.

[15] S. K. Mahmoud, M. Alfonse, M. I. Roushdy, and A.-B. M. Salem, "A comparative analysis of cross site scripting (xss) detecting and defensive techniques," in *The 8th IEEE International Conference on Intelligent Computing and Information Systems (ICICIS)*. Cairo, Egypt: IEEE, December 2017.

[16] J. Huang, Y. Li, J. Zhang, and R. Dai, "Uchecker: Automatically detecting php-based unrestricted file upload vulnerabilities," in *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. Portland, OR, USA: IEEE, June 2019.

[17] X. Cheng, H. Wang, J. Hua, M. Zhang, G. Xu, L. Yi, and Y. Sui, "Static detection of control-flow-related vulnerabilities using graph embedding," in *24th International Conference on Engineering of Complex Computer Systems (ICECCS)*. Guangzhou, China: IEEE, November 2019.