Cross - Origin Resource Sharing Vulnerability Testing: Techniques and Implications

Vivek Somi

somivivek@gmail.com

Abstract

This review article examines weaknesses in Cross- Origin Resource Sharing (CORS) and the necessary testing techniques essential for protection of modern online applications. Safeguarding sensitive user data depends on understanding and mitigating CORS-related risks as online applications become more dynamic and linked. Offering a complete basis for vulnerability detection, the paper looks at several testing techniques including human testing, automated tools, and browser developer tools. Moreover, it emphasizes real events that show the serious results of poorly applied CORS rules, therefore stressing the need for strict security measures. Examined are best practices for CORS security—that is, suitable configuration, regular audits, and increased developer awareness—that help companies to raise their security posture. The report presents expected developments in CORS security including machine learning for identifying anomalies and centralized policy management. This mix of strategies and penalties gives businesses trying to keep high levels of safety in a digital context significant new insight.

Keywords: Cross-Origin Resource Sharing (CORS), Vulnerability Testing, Web Application Security, Security Best Practices

1. INTRODUCTION

Cross-origin resource sharing, sometimes referred to as CORS, is a security feature influencing the interaction of online apps among several backgrounds. This provides safe online access to materials. By use of the "same-origin policy," which bans websites from accessing resources stored outside of their domain unless they are explicitly allowed to do so, CORS is quite vital in the management of the interaction between web clients and servers. Although online security depends on this method, incorrect implementation of this one could cause major hazards. Before one can start to understand CORS' ideas, headers, and preflight requests, one must first have a fundamental awareness of them. The CORS method employs particular headers to be used. These headers restrict the access to a resource's sources. These headers consist of `visit-Control-allow-Methods`, which manages HTTP methods, `Access-Control-allow-Headers`, which notes which headers are approved in requests, and `Access-Control-allow-Origin`, which notes which domains are let to visit the website. Under some circumstances a preflight request is triggered, thereby confirming the CORS policy of the server before the real request is made. This method is frequently utilized in situations including requesting custom headers, applying non-simple methods like PUT or DELETE, or managing resources that can possibly lead to security concerns. Although CORS is quite important, it can also introduce flaws, particularly in cases of erroneous rules setting. Many often-occurring CORS flaws are brought on by improper settings. These include too permissive settings allowing access from any source, the use of insecure wildcard symbols (`*`) in the `Access-Control-allow-Origin` header, and the neglect to properly name the sources and approaches that are really required[1].

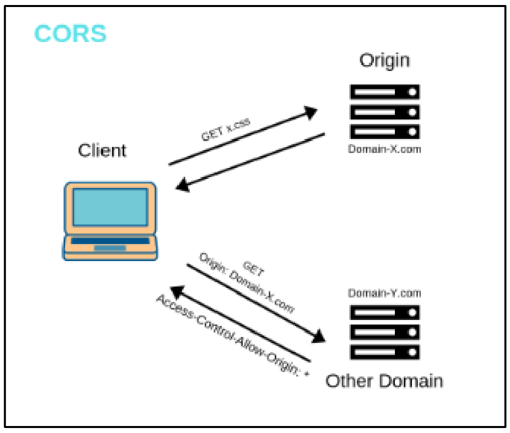

Figure 1 Cross-Origin Resource Sharing (CORS)

These settings enable possible attacks including data exfiltration, in which an attacker could steal sensitive data from one origin by using access from another origin. Cross-site scripting (XSS) attacks—which happen when an assailant inserts harmful code into a vulnerable website, therefore extending the attack surface—can also disclose application vulnerabilities. These situations involve looking at responses to demands coming from several sources. This helps one to find improperly set policies that might be used. Two automatic testing tools often used to search for CORS vulnerabilities are OWASP ZAP and Burp Suite. These tools aim to find vulnerabilities by automating the cross-origin request sending procedure and examining server responses. Regarding large-scale API or application testing, these technologies are simply not comparable to anything else. Furthermore, browser development tools provide a direct way for real-time evaluation of CORS requests and responses, therefore arming developers with the capacity to fix problems as they arise[2]–[4]. These instruments show information about request headers, server responses, and any possible problems connected to CORS rules, so helping to identify vulnerabilities. This helps one to identify weaknesses. It is crucial to underline that the consequences of CORS vulnerabilities go well beyond basic misconfigurations; if they are not checked, they can cause serious security issues. One of the most serious threats CORS weaknesses can provide is data exfiltration. Sensitive data, such personal information or API keys, could be accessed by a malevolent third-party source in this context. If an assailant finds, for instance, that a poorly configured CORS policy allows their origin to access private data from a web server, they can then submit requests that are not authorized and get secret data. Another key issue linked with CORS vulnerabilities is the abuse of cross-site scripting, sometimes referred to as XSS[5]–[7].

2. LITERATURE REVIEW

Bruno 2022 Instead of re-engineering the Web, the suppliers applied security patches—protocols, techniques—to the Web platform so establishing a more user-friendly and safer environment for web users. Still, these updates not only neglected to completely solve the security problems but also carried extra security risks not known to users or website operators. Original research on two different security patches is presented in this paper to investigate their application in useful environments and find neglected features and factors connected to their use: the security mechanism CORS and the security protocol OAuth. This work emphasizes on offensive strategies in which create automated tools including creative approaches for assessing and quantifying the security aspects of the OAuth protocol and CORS mechanism in useful environments[8].
Zaheri 2022 While proving successful targeted deanonymizing attacks on well-known platforms including Google, Twitter, LinkedIn, Facebook, Instagram, and Reddit, evaluate our assaults on several hardware microarchitectures, diverse operating systems, and multiple browser versions, including the highly secure Tor Browser. Most of the time our assault runs in less than three seconds and may be scaled to target an exponentially high user count. Suggest a thorough defense carried out as a browser plugin to help to reduce

these dangers. Our protection is now available on the Chrome and Firefox app stores to reduce the danger to vulnerable consumers. Have fairly shared our results with the Electronic Frontier Foundation as well as several technological suppliers. In the end, advise users unable to install the extension as well as websites and browser creators[9].

Sprecher 2022 Evaluate our attacks on several hardware microarchitectures, different operating systems, and several browser versions—including the extremely safe Tor Browser—while proving successful targeted deanonymizing attacks on well-known platforms including Google, Twitter, LinkedIn, TikHub, Facebook, Instagram, and Reddit. Usually running in less than three seconds, our attack can be scaled to target an exponentially large number of users. We offer a thorough defense carried out as a browser plugin to help to reduce these risks. Currently available in the Chrome and Firefox app stores, our protection helps to reduce the risk to vulnerable users. Have fairly shared our results with several technological companies as well as the Electronic Frontier Foundation. In the end, advise browsers and websites as well as consumers unable to install the extension[10].

Singanamalla 2022 The domain sharding method used in HTTP/1.1 distributes sub resources across subdomains to improve performance through more connections, therefore impeding the capacity to aggregate requests. In these cases, HTTP/2 clients create new connections to obtain material housed on the same server and start further DNS searches. Originating Frame is an HTTP/2 improvement tool that lets servers tell customers of more domains reachable from the same connection. Supported by only one browser, the extension lacks any known server implementation even though content delivery network (CDN) providers suggested it and the IETF standardized it in 2018. We compile and examine a large dataset. Model connection coalescing using the dataset and find a minimum set of certificate changes that maximize client coalescing possibilities. Then, at a large CDN, ORIGIN Frame support was applied and set up. Five thousand certificates were reissued to evaluate and validate our scaled-down models. Over two weeks, passive observations were conducted on production traffic, during which active measurements were also conducted on 5000 domains[11].

Ravindran 2022 Evaluations supported by penetration testers or security analysts help to equip the company against any security vulnerabilities. Implementing security fixes will help to solve all present faults and vulnerabilities in the web applications of the company after the Vulnerability Assessment and Penetration Testing (VAPT) of the apps. This paper defines common web application security vulnerabilities, prerequisites for doing a security assessment of the web application, and the advised practices and prohibitions connected with each vulnerability throughout the evaluation. This paper looks at several types of security testing and stresses the need of vulnerability assessment and penetration testing (VAPT) in every company[12].

TABLE NO. 1 LITERATURE SUMMARY

Author's name	Methodology used	Problem statement	Research gap
Sprecher 2022 [10]	Monitoring third-party script access to user PII using Firefox browser	Lack of access control over third-party scripts leads to privacy violations	Existing solutions fail to control third-party access to user PII effectively
Meiser 2021 [13]	Studied Tranco Top 5,000 sites, analyzed cross-origin communication prevalence.	Increased attack surface created by cross-origin communication partners' trust.	Limited research on trust implications in cross-origin communication vulnerabilities.
Zografopoulos 2021 [14]	Demonstrated threat modeling to represent CPS elements and vulnerabilities.	Cyber-physical energy systems are vulnerable to	Limited frameworks for high-fidelity modeling

		disruptive, malicious attacks.	and simulating CPES attacks.
Deshmukh 2021[15]	Implemented IDEA algorithm and CORS filters, compared its effectiveness with AES-256 encryption.	Ensuring privacy, integrity, and security in Healthcare Management Systems.	Limited studies comparing IDEA and AES-256 for healthcare data privacy.
Meiser 2021[13]	Analyzing cross-origin communication and building trust relationship graphs	Increased attack surface due to cross-origin communication on modern web applications.	Prior work overlooks trust-related risks in cross-origin communication

3. CROSS-ORIGIN RESOURCE SHARING (CORS)

A security method called cross-origin resource sharing, or CORS, allows web servers to exert management and control over which origins—domains—are allowed access to resources held on the server. Modern web applications must have CORS since it provides controlled access to resources derived from several sources. For application programming interfaces (APIs), online services, and programs depending on outside data, this is especially crucial[16].

A. The Fundamentals of CORS

Web browsers apply this policy, sometimes referred to as the Same- Origin Policy (SOP), to limit the interactions between documents or scripts loaded from one source and resources loaded from another. This approach will help to stop rogue websites from reading private data gathered from other domains. But in contemporary applications, web pages frequently request resources from another source—such as type fonts, pictures, or application programming interfaces (APIs). Standard operating procedure (SOP) would usually forbid this. By allowing a server to specifically name which sources are permitted to access its resources, the Content Object Requesting Service (CORS) offers a safe way to get around this restriction. Delivering specific HTTP headers in response to the browser's queries allows the server to provide this permission. An extra degree of security is provided by CORS, which guarantees that requests coming from many sources are only approved if they are particularly authorized[17].

B. CORS Headers

CORS uses several headers delivered in HTTP requests and answers to control and manage access from several sources. These headers define whether the request should be approved, whether techniques are permitted, and the kinds of headers that can be used. The main CORS headers listed here are Included in the server's response, this header—known as the Access-Control-allow- Origin header—helps to indicate which sources are allowed to access the resource. The situation may call for a specific domain (such as https://example.com) or a wildcard (*) allowing access from anyone. Access-Control-allow-Methods, or this header, decides which HTTP methods—among others GET, POST, PUT, and DELETE—are allowed when visiting the resource from another website. X-Custom-Header is an example of a header that is defined by the Access-Control-Allow-Headers header. This header determines which custom headers are permitted in cross-origin requests. In situations where the client delivers headers that are not standard, this is an important consideration. A header known as Access-Control-Allow-Credentials is responsible for determining whether credentials (such cookies, HTTP authentication, or TLS client certificates) are permitted to be included in requests that originate from different origins. When this is set to true, the browser will only send credentials if the server permits it to do so. Access-Control-Expose-Headers is a header that gives the server the ability to specify which headers can be exposed to the JavaScript code of the client. If this is not present, then some headers will not be displayed by default. This header, known as Access-Control-Max-Age, specifies the maximum amount of time that a browser is able to store the results of a preflight request in its cache[14], [15],

[18].

C. CORS Preflight Requests

CORS preflight requests are automatically sent by the browser before the actual cross-origin request is made. This is especially true for methods or headers that are not regarded to be "simple" (for example, requests to delete or put data). The preflight technique guarantees that the server is aware of the CORS request and provides its permission to the browser before it advances with the request. A preflight request is a kind of OPTIONS request used to find out whether the server can fulfil the main request. Along with the Origin header, which details the method and headers the browser intends to employ, an OPTIONS request is delivered to the server by the browser. Among the headers the server returns to the browser in response to a request to ascertain whether the request is allowed are Access-Control-allow-Methods, Access-Control-allow-HEADERS, and Access-Control-allow-Origin. The browser will first generate an OPTIONS request should a client try to execute a cross-origin PUT request with a custom header, for example. The browser will make the PUT request upon a favorable response from the server displaying the necessary headers. Should the request go unmet, the browser will stop it. Even though CORS preflight requests add overhead, particularly for operations that are complicated, they are necessary for ensuring that security is maintained while also facilitating resource sharing between origins. Both users and servers are protected from interactions that are not authorized or hazardous when preflight requests are implemented. This is accomplished by ensuring that servers explicitly accept cross-origin operations[19]–[21].

4. CORS VULNERABILITIES

Misconfigurations or settings that are excessively permissive might result in severe security problems, even though Cross-Origin Resource Sharing (CORS) offers key functionalities for online applications. For developers and security experts to protect their apps from the dangers of possible exploits, they must first fully grasp these weaknesses[22].

A. Misconfigured CORS Policies

Misconfigured CORS policies arise from improper definition of which origins are permitted access to resources by a server. Typical configurations include:

- Inaccurate Origin Whitelisting: Should developers mistakenly set the Access-Control-allow-Origin header, untrusted sites could unintentionally access sensitive pages. For instance, attackers can take advantage of a server's allowing of all sources (*) or inclusion of a wildcard that inadvertently reveals private APIs by requesting illegal access from rogue websites.
- CORS rules should clearly list the precise HTTP methods permitted (GET, POST, PUT, DELETE). Should the server's restrictions be insufficient, an assailant may carry out unplanned operations including data deletion or change on the server.
- Lack of Credentials Control: Should a server mistakenly set the Access-Control-allow-Credentials header, it could let illegal domains send and receive credentials (like cookies) with cross-origin requests, therefore enabling possible session hijacking.

These settings can result from ignorance, mistakes made during deployment, or depending too much on default settings unsuited for the security requirements of the application[23]–[25].

B. Overly Permissive CORS Settings

When a server lets access from all sources without enough thought given the security consequences, overly permissive CORS settings result. This shows up frequently in:

- Using a wildcard (*) for the Access-Control-allow-origin header lets any domain access resources. Especially in cases where the API permits sensitive actions or changes, this exposes delicate APIs and data to possible use.

- Permitting All Methods: Unauthorized behavior can result from the Access-Control-allow-Methods header including all HTTP methods. If a malevolent player can make DELETE requests, for example, they could wipe vital server data.
- Accepting any headers in the Access-Control-allow-Headers without restriction can expose the server's infrastructure to header injection attacks, whereby attackers can control requests to exploit weaknesses in the application.

C. Insecure Wildcard Usage

Insecure wildcard use is the incorrect use of wildcards in CORS environments that results in security issues. These comprise:

- Using a wildcard in the Access-Control-allow- Origin header lets any origin access the resources of the server. If the API lets sensitive actions, this can especially be risky since it lets illegal access possible. Attackers can take advantage of this by creating malevolent websites interacting with the exposed API.
- Combining Wildcards with Credentials: A major security risk results when a server enables credentials (Access-Control-allow-Credentials: true) yet allows wildcard origins. Should a user be authenticated on the site and visit a hostile website capable of making authorized queries to the API, the credentials—that is, session cookies—may be transferred to the attacker, therefore allowing them to pass for another user.
- Lack of Specificity in Wildcard Usage: When wildcards are used without specificity on what resources or methods are available, it can lead to inadvertent exposure of sensitive data or functionalities, therefore enabling attackers to more successfully exploit the program[26].

5. CORS VULNERABILITY TESTING TECHNIQUES AND THEIR IMPLICATIONS

A. Manual Testing Methods

Manual testing for CORS vulnerabilities tackles identification and evaluation of how a web application manages cross-origin requests carefully. This begins with knowing the server's CORS configuration and the resources accessible from many sources. Using tools like cURL or Postman, testers can track the server's responses beginning with HTTP searches to the application from different sources. Analyzing the Access-Control-allow-Origin header helps testers find whether the server appropriately limits access to reliable sources. Testers could try to control the requests to employ several HTTP strategies (GET, POST, DELETE) and track if the server responds suitably for each sort of request. They should also see how the server uses credentials or custom headers to handle requests. This approach lets one fully grasp the server's CORS policy and could highlight possible inappropriate setups or overly liberal settings that might expose the application to security concerns[27]–[29].

B. Automated Testing Tools

Effective screening of mobile applications for CORS vulnerability depends on automated testing approaches. Tools for CORS configuration testing is included in OWASP ZAP and Burp Suite. Testing several sources, techniques, and headers to track server response, these instruments automatically make a range of enquiries to the target application. They can act, for example, enquiring of untrustworthy sources to see whether the program unintentionally grants access to sensitive endpoints. Automated systems can identify any insecure wildcard usage—including Access-Control-allow-Origin: —as well as whether credentials are allowed with such settings. Usually providing thorough data displaying vulnerabilities, these tools enable quick solution of issues for developers. Even while automated technologies can speed up the testing process considerably, hand verification is still necessary to correctly understand the background and implications of the findings[30].

C. Browser Developer Tools for CORS Testing

Testing CORS setups straight from the client-side perspective is made much easier using browser development tools. Most contemporary browsers—including Chrome and Firefox—have built-in developer tools that let testers examine network requests and answers. Testers may instantly see CORS request and response headers using the Network tab. When they connect with the web application, this helps them to verify whether the required CORS headers are available and properly configured. Using the Console or Network panel, testers can also change requests to assess the application's performance with varying headers, approaches, or origins. This practical approach facilitates the identification of any differences in the application's CORS policy implementation relative to expected behavior. Though browser tools are excellent for on-demand testing, they should be matched with manual and automated testing to provide a complete evaluation of possible weaknesses[31].

D. Implications of CORS Vulnerabilities

Data Exfiltration Risks: CORS vulnerabilities let hostile actors access private data kept on a web server, they create serious data exfiltration threats. Attackers can create requests from illegal sources to take advantage of misconfigured or too liberal CORS settings. If an application lets any origin access sensitive endpoints, for example, attackers could access personal data, authentication tokens, or financial information from user sessions. Applications managing sensitive data, such banking or healthcare information, especially call for this kind of consideration. Strict CORS rules that only let trustworthy domains are absolutely necessary for companies since data exfiltration can cause serious reputation harm, regulatory fines, and loss of customer confidence[32].

CORS vulnerabilities could allow Cross-Site Scripting (XSS) attacks—where an assailant embeds harmful code on trustworthy websites. Should a web application allow cross-origin queries without sufficient validation or origin verification, an assailant may take advantage of this weakness by sending a specifically created request using malicious JavaScript. After execution, the script may act unauthorized on the user's behalf, such session cookie stealing or forwarding users to dangerous websites. The ability to run scripts inside a user's environment might seriously compromise the security of the user and the program, therefore facilitating possible data leaks and illegal transactions. To prevent XSS exploitation, then, strict CORS configurations are absolutely necessary[33].

CORS vulnerabilities create major threats to API security. Many modern online programs rely on APIs for the best performance; hence they often enable outside integrations to increase functionality. For these APIs, improperly implemented CORS rules enable illegal access and usage. Malefactors could use this to modify API endpoints, therefore causing illegal data access or modification. An API managing user account settings may be used to alter user information without appropriate authentication if CORS regulations are insufficiently implemented. Therefore, organizations must ensure their APIs enforce stringent CORS regulations and undergo regular testing to prevent any exploitation, thereby safeguarding sensitive user data and preserving the integrity of their services[34].

6. BEST PRACTICES FOR CORS SECURITY

Organizations should follow many best practices emphasizing appropriate configuration, rigorous policies, and frequent audits if they want strong CORS security. Starting with explicitly specifying the permitted origins using the Access-Control-allow-origin header, proper CORS configuration Specify trusted domains to restrict access rather than wildcards (*), which can expose delicate resources to any domain. Use the Access-Control-allow-Methods header also to precisely specify which HTTP methods (GET, POST, PUT, DELETE) are allowed for every resource, therefore reducing the possibility of illegal activity. Strict CORS rules not only whitelisting reliable sources but also guarantee that the Access-Control-allow-Credentials header is only set to true when essential, therefore stopping illegal sites from gaining user credential. One must avoid too liberal environments that could expose vulnerabilities by letting all headers without appropriate validation. Moreover, developers should refrain from letting requests from sources that might be possibly dangerous or dubious. Maintaining CORS security depends mostly on regular security audits. Review CORS setups often to find any possible vulnerabilities or incorrect setups.

7. CASE STUDIES

A. Real-world CORS Vulnerability Incidents

In several well-known cases, CORS flaws have caused major security breaches revealing private user records and undermining user confidence. One such instance happened in 2019 after poorly defined CORS rules let outside websites access private user data on an established social media network. The Access-feature-allow-Origin header of the platform was set to let all origins, thereby allowing attackers to create hostile websites maybe using this capability. Enquiring information from their domains—personal data and private messages among other things—the attackers endangered the integrity of the social network and its users, therefore acquiring user data. Another incident connected to an e-commerce website failing to apply rigorous CORS settings permitted a rogue site run on behalf of registered users without authorization. Attackers may manage account settings and participate in fraudulent behavior by means of misconfiguration, therefore causing major financial losses to the business and its clients. These events highlight how fast companies have to review and change their CORS settings to stop data exfiltration and unwanted access[35].

B. Lessons Learned from CORS Exploits

The knowledge gained from these CORS on vulnerability events emphasizes the need to use strong security policies and periodically monitoring systems. First, businesses must understand that CORS guidelines are a basic feature of online application protection and should not be given second importance. From settings like allowing wildcards or too liberal starts, severe penalties including data breaches and loss of consumer confidence ensue. Consequently, the least privileged strategy should direct the configuration of CORS rules thereby permitting only certain, dependable domains to access delicate resources. Finding likely mistakes also calls for regular observation and experimentation. Regular security audits should combine hand testing with automated technologies to guarantee that configurations stay strong and that any software updates do not unintentionally create risks. Finally, incident response systems should be developed to quickly control likely CORS-related hazards. Reducing risks even further is achieved by teaching developers and security staff about CORS vulnerabilities and by supporting a security-first strategy. Using best practices and learning from past mistakes will help companies greatly improve their security posture against CORS-related risks and safeguard their consumers and data[36].

8. FUTURE TRENDS IN CORS SECURITY

CORS Policy Management Frameworks: One interesting development is centralized CORS policy administration systems. These tools enable businesses to quickly set CORS configuration and implementation across several applications. Simplifying the upkeep of these policies helps businesses to maintain continual security posture and considerably reduce the risk of misconfiguration, which has traditionally been a common vulnerability in CORS deployments[37].

Machine Learning for Anomaly Detection: Another emerging trend is combining machine learning methods to study traffic patterns and spot anomalies in cross-origin requests. By means of machine learning, security systems can identify perhaps dangerous activity outside of accepted norms. This proactive threat detection tool enables businesses to react fast to suspicious behavior, therefore enhancing overall security and resistance against attacks[38].

Implementation of Security Guidelines and Standards: Developing is more focused on using security ideas and guidelines for CORS. Driven by legal criteria and industry norms, these structured recommendations guarantee developers implement safe coding guidelines and generate CORS rules. This approach seeks to provide a consistent security basis, therefore lowering the related CORS vulnerability risks.

Dynamic CORS Configurations: Also ahead are dynamic CORS settings depending on user context or risk assessment. This method allows businesses to impose more flexible access controls that could respond to real-time events, therefore enhancing security without sacrificing usability. By continuously adjusting CORS policies, organizations may effectively balance security issues with user experience, thereby ensuring that approved requests are granted and so lowering any possible risks[39].

9. CONCLUSION

The vital relevance of Cross-Origin Resource Sharing, security in contemporary web application architectures is underlined by this analysis of vulnerability assessment methods. The fast growth of interactive and dynamic web applications depends on a complete knowledge of CORS vulnerabilities, which might endanger program integrity and private user data. Using numerous testing approaches, including manual testing, robots, and browser developer tools, we present a complete framework for the efficient identification and mitigating of CORS-related hazards. Case studies show the specific effects of poorly tuned CORS rules can cause major data leaks and a loss of user confidence. These events highlight how urgently companies should have strong CORS security systems covering the application of strict rules, the behavior of regular audits, and the building of a developer awareness culture. Modern technologies are poised to greatly improve CORS security with centralized CORS administration and machine learning for anomaly detection. Maintaining online applications in an environment growingly linked depends on an active approach for CORS security.

REFERENCES

1. M. Squarcina, S. Calzavara, and M. Maffei, “The Remote on the Local: Exacerbating Web Attacks Via Service Workers Caches,” *Proc. - 2021 IEEE Symp. Secur. Priv. Work. SPW 2021*, pp. 432–443, 2021, doi: 10.1109/SPW53761.2021.00062.
2. I. Homoliak, S. Venugopalan, D. Reijsbergen, Q. Hum, R. Schumi, and P. Szalachowski, “The Security Reference Architecture for Blockchains: Toward a Standardized Model for Studying Vulnerabilities, Threats, and Defenses,” *IEEE Commun. Surv. Tutorials*, vol. 23, no. 1, pp. 341–390, 2021, doi: 10.1109/COMST.2020.3033665.
3. A. Tiberkak, “Lightweight Remote Control of Distributed Web-of-Things Platforms : First Prototype,” no. April, 2021, doi: 10.1109/IoTaIS50849.2021.9359718.
4. A. A. Pranata *et al.*, “Misconfiguration Discovery with Principal Component Analysis for Cloud-Native Services To cite this version : HAL Id : hal-03137874 Miscon guration Discovery with Principal Component Analysis for Cloud-Native Services,” 2021.
5. N. Korzhitskii, “Characterizing the Root Landscape of Certificate Transparency Logs,” no. June, 2020.
6. A. Jurcut, T. Niculcea, P. Ranaweera, N. An, and L. Khac, “Security Considerations for Internet of Things : A Survey,” *SN Comput. Sci.*, pp. 1–19, 2020, doi: 10.1007/s42979-020-00201-3.
7. A. Sudhodanan, S. Khodayari, and J. Caballero, “Cross-Origin State Inference (COSI) Attacks: Leaking Web Site States through XS-Leaks,” *27th Annu. Netw. Distrib. Syst. Secur. Symp. NDSS 2020*, 2020, doi: 10.14722/ndss.2020.24278.
8. Bruno, “DEPARTMENT OF INFORMATION ENGINEERING AND COMPUTER SCIENCE Analysis of Oauth and CORS vulnerabilities in the wild . Elham Arshad,” no. July, 2022.
9. M. Zaheri, Y. Oren, R. Curtmola, and U. S. Symposium, “Targeted Deanonymization via the Cache Side Channel : Attacks and Defenses,” 2022.
10. S. Sprecher, C. Kerschbaumer, and E. Kirda, “SoK: All or Nothing - A Postmortem of Solutions to the Third-Party Script Inclusion Permission Model and a Path Forward,” *Proc. - 7th IEEE Eur. Symp. Secur. Privacy, Euro S P 2022*, pp. 206–222, 2022, doi: 10.1109/EuroSP53844.2022.00021.
11. S. Singanamalla *et al.*, “Respect the ORIGIN! A Best-case Evaluation of Connection Coalescing in The Wild,” *Proc. ACM SIGCOMM Internet Meas. Conf. IMC*, pp. 664–678, 2022, doi: 10.1145/3517745.3561453.
12. U. Ravindran and R. V. Potukuchi, “A Review on Web Application Vulnerability Assessment and

Penetration Testing," *Rev. Comput. Eng. Stud.*, vol. 9, no. 1, pp. 1–22, 2022, doi: 10.18280/rces.090101.

13. G. Meiser, P. Laperdrix, and B. Stock, "Careful Who You Trust: Studying the Pitfalls of Cross-Origin Communication," *ASIA CCS 2021 - Proc. 2021 ACM Asia Conf. Comput. Commun. Secur.*, pp. 110–122, 2021, doi: 10.1145/3433210.3437510.
14. I. Zografopoulos and G. S. Member, "Cyber-Physical Energy Systems Security : Threat Modeling , Risk Assessment , Resources , Metrics , and Case Studies," pp. 29775–29818, 2021, doi: 10.1109/ACCESS.2021.3058403.
15. K. Deshmukh and P. Pramila, "Data Integrity , Security and Privacy in Healthcare Management System Data Integrity , Security and Privacy in Healthcare Management System," no. June, 2021.
16. K. Deshmukh and P. M. Chawan, "Data Integrity and Privacy in Healthcare Management System: A Survey," *Int. Res. J. ...*, no. June, pp. 405–408, 2020, [Online]. Available: https://www.researchgate.net/profile/Pramila-Chawan/publication/349482563_Data_Integrity_and_Privacy_in_Healthcare_Management_System_A_Survey/links/60325a3c92851c4ed5893c10/Data-Integrity-and-Privacy-in-Healthcare-Management-System-A-Survey.pdf
17. A. Fongen, K. Helkala, and M. S. Lund, "Trust Through Origin and Integrity : Protection of Client Code for Improved Cloud Security," no. c, pp. 16–21, 2020.
18. F. Ram and C. Mera-g, "An Empirical Study on Microservice Software Development," pp. 16–23, 2021, doi: 10.1109/SESoS-WDES52566.2021.00008.
19. J. Ruohonen, J. Salovaara, and V. Leppänen, "On the integrity of cross-origin javascripts," *IFIP Adv. Inf. Commun. Technol.*, vol. 529, pp. 385–398, 2018, doi: 10.1007/978-3-319-99828-2_27.
20. I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci, "Vulnerable open source dependencies: Counting those that matter," *Int. Symp. Empir. Softw. Eng. Meas.*, 2018, doi: 10.1145/3239235.3268920.
21. T. Bhuiyan, A. Begum, S. Rahman, and I. Hadid, "API vulnerabilities: Current status and dependencies," *Int. J. Eng. Technol.*, vol. 7, no. 2, pp. 9–13, 2018, doi: 10.14419/ijet.v7i2.3.9957.
22. S. A. Kolopaking, "Threat Modeling and Countermeasures of Continuously Operating Reference Stations Network Backbone to Improve Precise Positioning Service Security," 2018.
23. J. Chen *et al.*, "We still don't have secure cross-domain requests: An empirical study of CORS," *Proc. 27th USENIX Secur. Symp.*, pp. 1079–1093, 2018.
24. X. Deng, T. Wu, J. Yan, and J. Zhang, "Combinatorial Testing on Implementations of HTML5 Support," *Proc. - 10th IEEE Int. Conf. Softw. Testing, Verif. Valid. Work. ICSTW 2017*, vol. 2017, pp. 262–271, 2017, doi: 10.1109/ICSTW.2017.47.
25. C. Kerschbaumer, "Enforcing Content Security by Default within Web Browsers," *Proc. - 2016 IEEE Cybersecurity Dev. SecDev 2016*, pp. 101–106, 2017, doi: 10.1109/SecDev.2016.033.
26. J. Muller, V. Mladenov, J. Somorovsky, and J. Schwenk, "SoK: Exploiting Network Printers," *Proc. - IEEE Symp. Secur. Priv.*, pp. 213–230, 2017, doi: 10.1109/SP.2017.47.
27. T. Huttunen, "Browser cross-origin timing attacks," 2016.
28. B. R. Amarasekara and A. Mathrani, "Controlling risks and fraud in affiliate marketing: A simulation and testing environment," *2016 14th Annu. Conf. Privacy, Secur. Trust. PST 2016*, no. December 2016, pp. 353–360, 2016, doi: 10.1109/PST.2016.7906986.
29. R. Stevens, J. Crussell, and H. Chen, "On the origin of mobile apps: Network provenance for android applications," *CODASPY 2016 - Proc. 6th ACM Conf. Data Appl. Secur. Priv.*, pp. 160–171, 2016, doi: 10.1145/2857705.2857712.
30. A. Agarwal and J. Kim, "Spook . js : Attacking Chrome Strict Site Isolation via Speculative Execution".
31. M. Scarlato *et al.*, "BATDIV : A Blockchain-based Approach for Tourism Data Insertion and Visualization".

32. S. Agarwal, *Helping or Hindering ? How Browser Extensions Undermine Security*, vol. 1, no. 1. Association for Computing Machinery. doi: 10.1145/3548606.3560685.
33. A. Van Kesteren, "Cross-Origin Resource Sharing," *W3C Work. Draft*, no. February 2004, pp. 1–20, 2010, [Online]. Available: http://www.w3.org/TR/access-control/
34. T. Wu, Y. Song, F. Zhang, S. Gao, and B. Chen, "My Site Knows Where You Are : A Novel Browser Fingerprint to Track User Position".
35. T. Van Goethem, W. Joosen, and N. Nikiforakis, "The clock is still ticking: Timing attacks in the modern web," *Proc. ACM Conf. Comput. Commun. Secur.*, vol. 2015-Octob, pp. 1382–1393, 2015, doi: 10.1145/2810103.2813632.
36. H. Plate, S. E. Ponta, and A. Sabetta, "Impact assessment for vulnerabilities in open-source software libraries," *2015 IEEE 31st Int. Conf. Softw. Maint. Evol. ICSME 2015 - Proc.*, pp. 411–420, 2015, doi: 10.1109/ICSM.2015.7332492.
37. A. E. Bodenmiller, A. E. Bodenmiller, and A. K. S. P. D, "A High-Level Cloud Architecture for Community Software as a Service (CSaaS) A High-Level Cloud Architecture for Community Software as a Service (CSaaS)," no. MAY 2014, 2015.
38. F. Lebeau, B. Legeard, F. Peureux, and A. Vernotte, "Model-based vulnerability testing for web applications," *Proc. - IEEE 6th Int. Conf. Softw. Testing, Verif. Valid. Work. ICSTW 2013*, pp. 445–452, 2013, doi: 10.1109/ICSTW.2013.58.
39. L. S. Huang, Z. Weinberg, C. Evans, and C. Jackson, "Protecting browsers from cross-origin CSS attacks," *Proc. ACM Conf. Comput. Commun. Secur.*, pp. 619–629, 2010, doi: 10.1145/1866307.1866376.