International Journal of Economics, Commerce and Management
United Kingdom ISSN 2348 0386 Vol. IX, Issue 2, Feb 2021

http://ijecm.co.uk/

CHAINING LOW-RISK ISSUES INTO HIGH-RISK WEB APPLICATION VULNERABILITIES

Ștefan Nicula
The Bucharest University of Economic Studies, Romania
niculastefan13@stud.ase.ro

Răzvan Daniel Zota
The Bucharest University of Economic Studies, Romania
zota@ase.ro

Abstract

In a Penetration Testing project, low-risk vulnerabilities can play a considerable role in terms of exploiting and identifying other high-risk issues with potentially bigger impact. In companies with a mature level of security for their products, we can note a clear approach in addressing all types of reported vulnerabilities regardless of their standalone impact however, sometimes, these low-risk vulnerabilities can be overlooked. The article presents a research focused on chaining multiple low-risk vulnerabilities in order to reveal high-risk issues that have a bigger exploitation impact. The project was focused on web application specifically, targeting a variety of technologies and included various attack vectors. Key concepts are presented with their corresponding practical examples regardless of their implementation that differs based on the technology stack used. A majority of the examples presented in the paper are examples taken from real-world research projects while others are theoretical or extracted from public records. The results indicated that, in the case of client-side injection attacks, the web application security best practices are oftentimes overlooked or don't have a clear context of exploitation and impact in order to reveal the potential security breach. In a penetration testing project, details such as this are included and reported consistently whereas a bug bounty program will focus on the higher risk vulnerabilities, with a chance of including low risks in their reporting. However, regardless of

the testing methodology, the research showed that low-risk issues can often be combined and chained into multiple vectors that will ultimately result in high-risk finding with real economic and technical impact.
Keywords: penetration testing, web application, high risk, vulnerabilities, exploit, leveraging impact

INTRODUCTION

The number of vulnerabilities affecting a web application can be very broad and usually, dependant on the technology stack used. With technology, we also have specific attack vectors or issues and all of them are categorized based on the likelihood, impact, and overall criticality. The low-risk issues are often referred to as the vulnerabilities that do not have a direct impact on data confidentiality, integrity, and availability which is also known as the CIA Triangle [1]. Other scoring systems such as the CVSS score [2] try to evaluate and reflect the impact using a scoring measurement based on specific factors. Nonetheless, a low-risk issue is usually not something that would threaten the web application, at least not in a direct and practical way, it often comes with theoretical applicabilities or with additional attack vectors or vulnerabilities.

The research project focuses on studying the low-risk issues and their impact on evaluating the security of a web application. The results showed that many times, a low-risk vulnerability is acting as the catalyst or as enablers for specific high-risk vulnerabilities. Moreover, by chaining certain low-risk issues or depending on the exact context, some of them might end up as high-risk vulnerabilities.

All of the metrics and examples described in the article are taken from real-life Pentesting and Bug-bounty projects that are not publicly available. Based on the results analyzed and based on the vulnerabilities identified, we can highlight some of the examples studied:

- CSRF -> XSS in POST body -> Cookies without HTTP Only flag
- CRLF in cookie (session fixation) -> Self XSS
- SSRF -> Open Redirect -> bypassing SSRF domain validation
- Open redirect into Authorization Bearer leak
- MIME-type mismatch -> upload images with XSS payload in EXIF -> obtain XSS

The enumerated situations are presented with a much higher level of detail in the article. We notice a trend in the studied cases, most of them are exploiting some form of Cross-Site Scripting. That’s because many of the analyzed low-risk issues are webserver or HTTP protocol

specifications and injection types of attacks such as Cross-Site Scripting are dependent on browser processing. Those specifics are usually considered low-risk vulnerabilities or security best practices.

CROSS-ORIGIN ISSUES AND HTTP HEADERS

Cross-Site Request Forgery (CSRF) protection can play an important role in the security of a web application against requests coming from external domains. Oftentimes, this protection gets overlooked, however, the main key aspect in exploiting a Cross-Site Scripting with the injection point inside a POST body parameter, is the ability to make the request from foreign origins using HTML forms.

A good number of modern web applications from today are using some form of API for their backend calls. Usually, the API implements one or more supported open standard file formats such as JSON and XML. There is a high chance that an API that uses JSON as a template response will also accept and expect JSON format from the backend as well. In both situations, sending XML or JSON using HTML forms is often impossible. However, there are cases where the APIs are accepting and expecting normal POST body parameters. In these circumstances, a strong CSRF implementation should be enforced. CSRF token granularity can be implemented both at session level or at request level meaning that each session will have one CSRF token to be used across the session lifetime or each request made by one user session will have different CSRF tokens each time. The latter would be the best implementation but it also comes with the cost of processing and the difficulty of coding such protection. Usually, a session-unique CSRF token used as a request parameter combined with a CSRF cookie will suffice the need for strong cross-origin protections. Finally, a strong validation of the Origin header and Referer header allows for better cross-origin protection overall. [3]

Depending on the implementation, CSRF protection can be in the form of HTTP headers such as CSRF tokens, cookies, or direct POST or GET URL parameters. As a security best practice, a form of CSRF parameter is always recommended combined with a session-wide CSRF protection such as custom HTTP headers. The Origin and Referer header should be always validated using a whitelist approach.

There are also multiple medium to low-risk vulnerabilities that can affect the implementation of CSRF mitigation. For example, a web application might verify each HTTP request to contain a valid CSRF token or multiple tokens but it will not verify who the token belongs to. In such cases, any valid CSRF token can be used by any user interchangeably. This scenario actually bypasses the CSRF protection implemented because an attacker can use a

self-registered controlled account in order to generate tokens for malicious requests, targeting other user accounts.

On the other hand, speaking of cross-origin requests, there is also the importance of CORS implementation (Cross-Origin-Resource-Sharing). For security reasons, browsers restrict cross-origin HTTP requests initiated from scripts. This means that a web application using those APIs can only request resources from the same origin the application was loaded from unless the response from other origins includes the right CORS headers. This is also known as the Same-Origin Policy. [4]

If the tested web application is not implementing strict CORS policy headers, they can allow an attacker to issue AJAX requests to the target using JavaScript code. This will further enhance an attacker's arsenal in exploiting vulnerabilities such as Cross-Site Scripting using JSON body requests.

If we take a look at the previous example with the CSRF implementation, some of the identified backend API's are expecting JSON input rather than normal POST body parameters.

There is a big difference when it comes to Cross-Site Scripting attacks that are using JSON POST body parameters. Without the capability of issuing a cross-origin request with JavaScript, an attacker cannot exploit the vulnerability and target the users. There are, of course, edge cases that could result in a Persistent Cross-Site Scripting but a Reflected Cross-Site Scripting inside a POST JSON parameter without CORS and CSRF cannot be exploited. There is also the case of the vulnerable functionality being available to authenticated or unauthenticated users which makes a huge difference in the exploitation scenario.

The CORS policy evaluation is usually targeted on two HTTP headers: *Access-Control-Allow-Origin* and *Access-Control-Allow-Credentials*.

The value of the allowed origin can be set to wildcards such as * which will instruct the browser that any origin can issue cross-origin HTTP requests using JavaScript to the targeted web server. [5] The most interesting capability exposed by cross-origin JavaScript requests such as XHR and Fetch API is the ability to make credentialed requests that are aware of HTTP cookies and HTTP Authentication information. By default, in cross-site XMLHttpRequest or Fetch invocations, browsers will not send credentials. However, if the webserver specifically sets the HTTP header *Access-Control-Allow-Credentials* to *true*, then the cross-origin request has the ability to make use of any credentials stored by the browser for the website. This action enables injection types of attacks such as Cross-Site Scripting in POST body parameters that are affecting a functionality accessed only by authenticated users.

Many modern frameworks such as Java Spring Security have CORS implemented by default in the configuration. Browsers have also taken a step further and for each response that contains both allowed origin * and allowed credentials, the cross-origin request will be denied.

INJECTION TYPES OF ATTACKS AND HTTP HEADERS

Arguably, one of the most common and hunted web application flaws is Cross-Site Scripting (XSS). This vulnerability is caused by a bad or missing client-side input validation and results in malicious HTML of JavaScript user-supplied input to be reflected in the web application response without any sanitization. This vulnerability can be further classified by the manner of execution and by the component that executes the payload. As such, we have XSS issues that are DOM-based meaning that the payload is executed on the client-side, by the JavaScript code used by the targeted web application. A good example would be the angular expression injection [6] that affected all the angular libraries and allowed an attacker to inject expression such as *{{constructor.constructor('alert(1)')()}}* in order to escape the angular sandbox and execute malicious JavaScript code.

There's also the Cross-Site Scripting caused by the missing backend validation and these issues have two forms, reflected and stored. Depending on the manner that the payload is used, an XSS can become a persistent one if the payload is kept and used inside a specific webpage each time the user visits the page. A good example here would be the profile webpage of a web application that usually contains the email or the name of the user. A successful injection inserted in any of these inputs would result in a Persistent Cross-Site Scripting as each time the user will visit the profile page, the XSS payload will execute. The Reflected XSS is a temporary payload that gets executed when the victim visits a specific URL that contains the XSS payload embedded in a URL parameter or, in the case of Reflected XSS in POST parameters, the victim will have to visit a webpage that makes a cross-origin request, either with HTML forms or JavaScript requests.

Content-Security-Policy header

A bad or missing Content-Security-Policy header (CSP) might make the difference between a fully functional Cross-Site Scripting exploit and a non-exploitable one. A primary goal of CSP is to mitigate and report XSS attacks. XSS attacks exploit the browser's trust of the content received from the server. Malicious scripts are executed by the victim's browser because the browser trusts the source of the content, even when it's not coming from where it seems to be coming from. [7] The question raises if the CSP header actually mitigates Cross-Site Scripting. The header does help however, it is not a complete protection by any means. Depending on the

dynamic content processed by the web application, a very strict CSP policy might not actually be achievable. The best mitigation for Cross-Site Scripting still remains the sanitization and validation of user-supplied input although HTTP headers such as CSP add an extra layer of protection.

Host header injection

The case of a Host Header injection can be classified from low-risk to high-risk issues, depending on the vulnerable functionality. This vulnerability refers to the fact that the web application backend is resolving the reflected URLs or other dynamic contents based on the Host header received from the client-side request. As such, if the Host header gets modified somehow, the URLs in the webpage will reflect that value. However, such vulnerability cannot be exploited in conjunction with user interaction because the browser does not allow the Host header to be modified using either JavaScript or HTML code. [8]

There still exists a certain case that actually allows a real case exploitation scenario. Sometimes, functionalities such as forgot password, are also dynamically resolving and computing URLs based on the Host header received from the HTTP request. As such, by sending a password reset with a modified Host header on a specific email address, chances are that the inserted value will actually reflect in the reset password URL sent in the email. This abuse can obviously cause harm as an external attacker can interact with the vulnerable function in an unauthenticated way, by default. Having control of the reset password domain, an attacker can send valid phishing attacks on behalf of the legitimate webserver email service, moreover, the password reset link is computed by the backend beforehand, so any URLs will contain a valid password reset token.

MIME type mismatch

Browsers are implementing what's called "MIME sniffing" which is a functionality that allows browsers to expect a specific Content-Type from the webserver's response by parsing a number of indicators. This functionality can be abused in order to trick the browser into interpreting files that have the wrong or nonexistent MIME type returned by the webserver. In modern browsers, resources such as images, stylesheets, or scripts will be rejected if the MIME type does not match the context of the file. [9] In older versions of the browser, it was possible that an incorrectly stated MIME type of an uploaded file led to Cross-Site Scripting if the file contained XSS payloads inside their body. The most common mistake encountered was the usage of incorrect content-type returned by the webserver. MIME types are case-insensitive as per the

RFC, however faulty implementation could lead to upper case mime types to be interpreted as unknown and prone to MIME sniffing validation resulting in XSS on some browsers.

OPEN REDIRECT

Open Redirect is another low-risk issue which is very dependent on the context. An Open Redirect vulnerability can be considered a low to medium risk vulnerability. It allows the ability to redirect the victim to an external domain using a specially crafted URL designed to exploit this issue. For example, the following URL can be considered a valid Open Redirect:

https://exampledomain.foo?redirect=fake.attackersite.com

A specific edge case regarding the Open Redirect is the scenario where an Authentication flow contains a dynamic host combined with DOM JavaScript and results in the session or Authorization header being disclosed inside a DOM parameter. One identified vulnerability of such was found in a project where the Open Auth implementation contained a URL GET parameter resolved and populated by the JavaScript client-side code. The redirector could have been replaced with any other arbitrary host value. When the user successfully authenticated in the platform, the redirections would occur and the Authorization Bearer would be transmitted in a GET request with # URI delimiter.

https://fake.attackersite.com?loginID=8d12hd81hd128#AuthBearer=9d12j9213kf912kf129f

An Open Redirect can also help bypass specific backend validations on Referer header or other internal checks. For example, a case of limited Server-Side Request Forgery (SSRF) with limited access to internal domain paths can be further escalated into a fully exploitable vulnerability [10]. Normally, an SSRF exists where the user-supplied input is used as a path by the backend in order to make another HTTP request. The specific case identified allowed the user to supply only URLs and paths inside the main domain only. However, a specific web application was identified to be vulnerable to an Open Redirect issue. The prerequisite here is that the SSRF will actually follow the redirect and continue the flow which it did. The biggest risk of an SSRF lies in the fact that a legitimate server with possible elevated privilege is making the request, be it privilege inside an internal network or using a privileged session. As such, validation is really important in this case and a scenario of Open Redirect affecting the same host, which is allowed, can help bypass the SSRF under certain circumstances. That’s because, if the initial URL points inside the domain which passes the check but the URL is a redirect which points to another domain outside of the main domain, the validation was already passed and the SSRF will respect the 302 redirect message and forward the request to the host mentioned in the Location HTTP header.

SELF-CROSS-SITE SCRIPTING

A self XSS vulnerability is a vulnerability that can be exploited by the user only and URLs or HTML forms cannot be crafted to exploit this scenario without extensive user interaction or self-injection from the user's side. As such, this vulnerability is considered an informational to low-risk one. Even though this vulnerability is a low-risk, it can actually be exploited under certain circumstances. One edge case found was combining a session fixation with a Self XSS injected in a malicious account in order to obtain JavaScript code execution inside the victim browser.

The Self Cross-Site Scripting was inside the user profile details and the injection endpoint was protected by CSRF tokens. The injected payload would execute after the login, inside the main webpage of the vulnerable application as the profile was parsed and shown directly on the home page.

As such, the full Proof of Concept would be that an attacker will create a malicious account with an already injected XSS payload and it would use that account to authenticate the victims in their browser.

This action can be achieved by abusing the missing CSRF protection on the login endpoint. Furthermore, the web application had multiple login forms present. The main login form was JSON based and as such, no redirection would happen because of CORS as opposed to the second identified login which used HTML redirect into the main webpage if the login credentials were valid. The second login form also allowed normal POST body parameters to be provided.

By chaining these vulnerabilities, using a rogue account with XSS payload injected already, and making an HTML form that authenticates the victim into that account, we are able to execute JavaScript code inside the victim's browser. With this scenario, an attacker can potentially abuse the user’s trust in the web application and present a rogue login webpage that will leak the credentials to a server controlled by the attacker. [11]

CRLF INJECTION

An edge case that worked on older webservers is based on CRLF injection in the HTTP response headers. This attack has a big potential as it allows the user-supplied input to modify the HTTP headers returned by the webserver. The HTTP headers are parsed by the browser and the browser will act accordingly with the specified values. There are cases where cookie values are reflected inside the webpages. Unless there is a session fixation or cookie direct manipulation at some point in the web application, this scenario is highly unlikely to be exploited. However, a specifically crafted CRLF injection could result in a session fixation scenario or a cookie value injection. If CORS also allows cross-origin requests with credentials,

the possibilities are endless. CRLF injection works by injecting new-line characters inside the response values reflected somewhere in the HTTP response headers, usually in the Location or Cookies headers. If these new-lines are interpreted by the backend or webserver, it will allow the attacker to define HTTP response headers such as a new Cookie header or Location, or even both. This exploitation has its limitations in terms of the position of the injection inside the HTTP response headers. Also multiple headers of the same type are not allowed and the first one will be the one parsed and accepted by the browser. Even so, exploitation scenarios do exist. [12]

CONCLUSION

Oftentimes, low-risk issues are overlooked or not reported in bug bounty programs by researchers because low-risk issues are not rewarded mainly due to the overall convention and the nature of the program. It would not be, indeed, cost-effective for companies to pay every single low-risk issue reported without a valid proof of concept that shows an impact. However, if a high-risk vulnerability is found, any low-risk issue that accompanies or helps in the exploitation scenario should be mentioned and reported accordingly, together with the high-risk issue identified. In this way, not only the low risks will be taken in consideration but also will have their risks and usefulness highlighted given the context of the high-risk impact scenario. In a pentest engagement or a bug bounty program, a web application that does not enforce security best practices often feels much more exposed, as an attacker, the room to leverage and interact with the web application is more open.

The article focused mostly on injection types of attacks that leverage specific low-risk issues because usually these types of attacks require additional circumstances in order to be exploited. Especially, attacks that are using some form of browser processing do have requirements that need to be met for a successful proof of concept. In a penetration testing project, low-risk issues are reported as stand-alone, without a possible full chain exploit. This is a good thing but is specific to these types of projects because the cost of a pentest includes all the identified vulnerabilities, and all the low-risk issues are not needed to be reported and paid individually. The more low-risks are patched, the less attack-surface would be exposed and this could also possibly lower the high-risk vulnerabilities impact up to the point that they could no longer be exploited. Future research includes the analysis for potentially integrating low risk issues reporting in the bug bounty programs and their effect on the cost and the efficiency of the engagements. Automated scans can help reveal such issues. However, they need to be assigned a context and a theoretical impact for them to be considered.

REFERENCES

[1] Josh Fruhlinger, The CIA triad: Definition, components and examples, Feb 10, 2020, https://www.csoonline.com/article/3519908/the-cia-triad-definition-components-and-examples.html, last accessed on date: June, 2020;

[2] Common Vulnerability Scoring System version 3.1, June 2019, https://www.first.org/cvss/specification-document, last accessed on date: Jan, 2020;

[3] M. E. Masri and N. Vlajic, "Current state of client-side extensions aimed at protecting against CSRF-like attacks," 2017 IEEE Conference on Communications and Network Security (CNS), Las Vegas, NV, 2017, pp. 390-391, doi: 10.1109/CNS.2017.8228690.

[4] Jesse Ruderman, Same-origin policy, Nov 9, 2020, https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy, last accessed on date: Nov, 2020;

[5] MDN contributors, Cross-Origin Resource Sharing (CORS), Nov 8, 2020 https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS, last accessed on date: Nov, 2020;

[6] Gareth Heyes, XSS without HTML: Client-Side Template Injection with AngularJS, January 27, 2016, https://portswigger.net/research/xss-without-html-client-side-template-injection-with-angularjs, last accessed on date: Dec, 2020;

[7] MDN contributors, Content Security Policy (CSP), Jun 2, 2020, https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP, last accessed on date: Dec, 2020;

[8] Peter Yaworski, Real-World Bug Hunting: A Field Guide to Web Hacking, No Starch Press, July 9, 2019

[9] Christoph Kerschbaumer, Mitigating MIME Confusion Attacks in Firefox, August 26, 2016, https://blog.mozilla.org/security/2016/08/26/mitigating-mime-confusion-attacks-in-firefox/,

[10] Harsh Jaiswal, Vimeo SSRF with code execution potential, Mar 8, 2019, https://medium.com/@rootxharsh_90844/vimeo-ssrf-with-code-execution-potential-68c774ba7c1e, last accessed on date: Jan, 2020;

[11] Jack, Uber Bug Bounty: Turning Self-XSS into Good-XSS, March 22, 2016, https://whitton.io/articles/uber-turning-self-xss-into-good-xss/, last accessed on date: Dec, 2019;

[12] filedescriptor, HTTP Response Splitting (CRLF injection) in report_story, April 21, 2015, https://hackerone.com/reports/52042, last accessed on date: Aug, 2020;