Initial Static Security Scan of the Hackazon Application

Hayden Eubanks

School of Business, Liberty University

CSIS 486-D01

Prof. Backherms

November 13, 2023

Initial Static Security Scan of the Hackazon Application

Introduction:

In the security testing of web applications, a combination of static and dynamic security scans can be performed to identify vulnerabilities within the software (OWASP, 2023a). The first of these test methodologies, static testing, refers to the testing of software work products that do not require the execution of code such as the analysis of design documentation, requirements, or the source code itself (Nunes et al., 2019). In this way, static testing allows security professionals to discover vulnerabilities early in the software development lifecycle when the vulnerabilities are much easier and cheaper to mitigate (Simpson & Anthill, 2017). Additionally, dynamic and static testing each have use cases for which they are best suited and as such, a combination of static and dynamic testing can be implemented to uncover a greater number of security defects (Nunes et al., 2019). This highlights the importance for security professionals to understand static testing methodologies and through applying these strategies alongside dynamic testing a greater degree of vulnerabilities can be discovered and mitigated increasing the overall security of the software.

As the work products utilized in static testing are ‘static’ automation tools can be implemented to scan through work products and identify any code that potentially introduces a vulnerability into the software (Ferrara et al., 2021). The report generated from this static vulnerability scan then allows a security professional to identify areas of concern throughout the code and implement mitigation strategies before the software can be exploited. However, as static vulnerability scanners do not dynamically interact with the software, they are only able to detect vulnerabilities that they are specifically searching for within the work product (Nunes et al., 2019). This then highlights one of the major determining factors in selecting a vulnerability

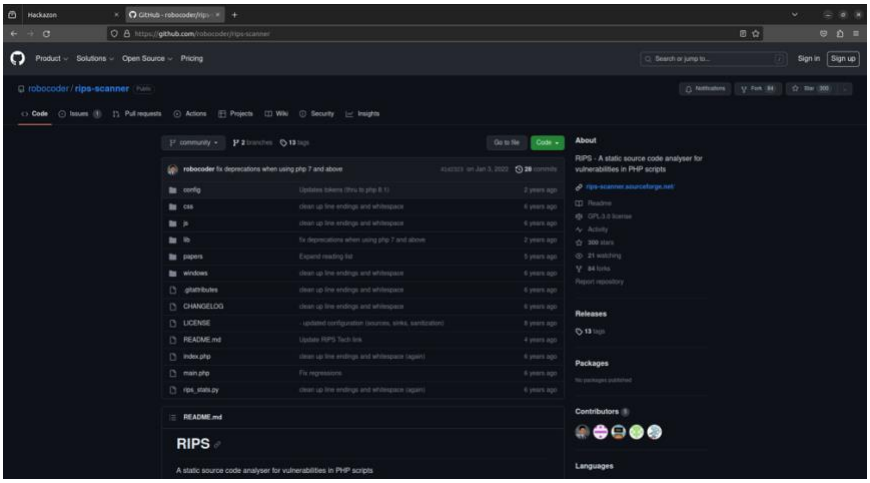
scanner to employ is the number and type of vulnerabilities the scanner is searching for (Ma et al., 2022). With this, the frequency by which these vulnerability lists are updated should also be considered as the untimely updating of known vulnerabilities could result in false negatives being scans that do not identify known vulnerabilities present in the code allowing malicious actors to discover them first. The diversity in vulnerability lists between different scanners then highlights the added value in eliminating false negatives which could be gained from employing a combination of static vulnerability scanners (Nunes et al., 2019). However, a security scanner that is not best suited to the software under test could generate many false positives where errors are being generated where no vulnerability exists (Simpson & Anthill, 2017). With all of these factors in mind, it can be seen that selecting the correct set of tools for a security scan can vastly improve the number of vulnerabilities discovered, and as such it is essential that a security professional understand the benefits and use cases for several static analysis tools.

In this report, the Research and Innovation Promote Security (RIPS) scanner was used to scan the code of the entire Hackazon application and detect several vulnerabilities present within. The primary vulnerability classifications present within the application are vulnerabilities to cross-site scripting (XSS), file manipulation, file disclosure, and command execution with XSS by far being the most common vulnerability discovered. Throughout this report, an examination of these vulnerabilities will be performed with specific attention being placed on the prevention of XSS attacks as this vulnerability is quite severe (OWASP, 2021) and the most prevalent throughout the webpage. The generalized dangers of XSS scripting will then further be explored in the last section of the report with the worst-case scenario of injecting code that monitors and reports user data being examined. The vulnerabilities discovered throughout the security scan of the Hackazon application could prove extremely damaging and as such, reveal the importance of

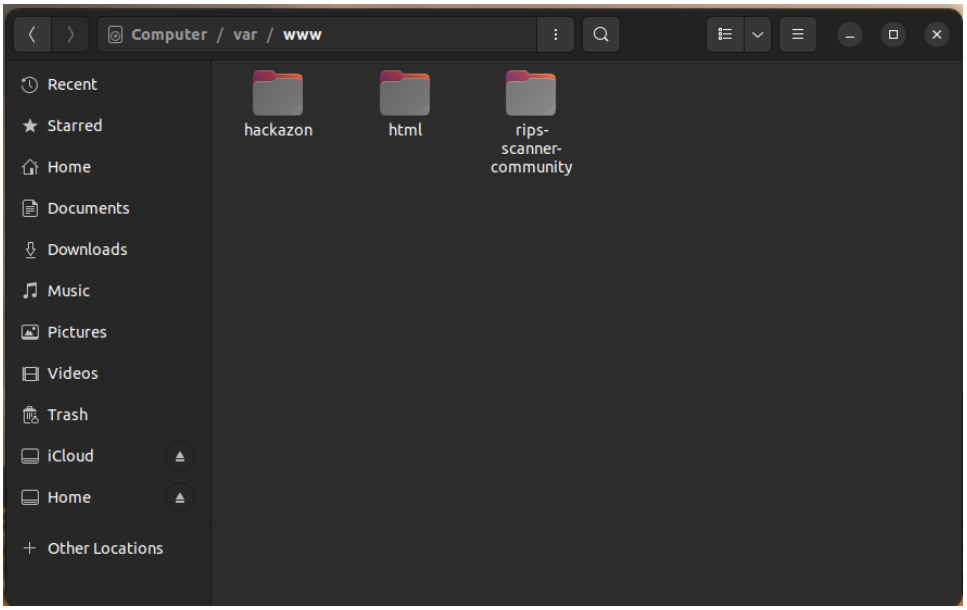
using a static security scanner such as RIPS to detect these vulnerabilities before a malicious actor is able to.

Evidence of RIPS Configuration:

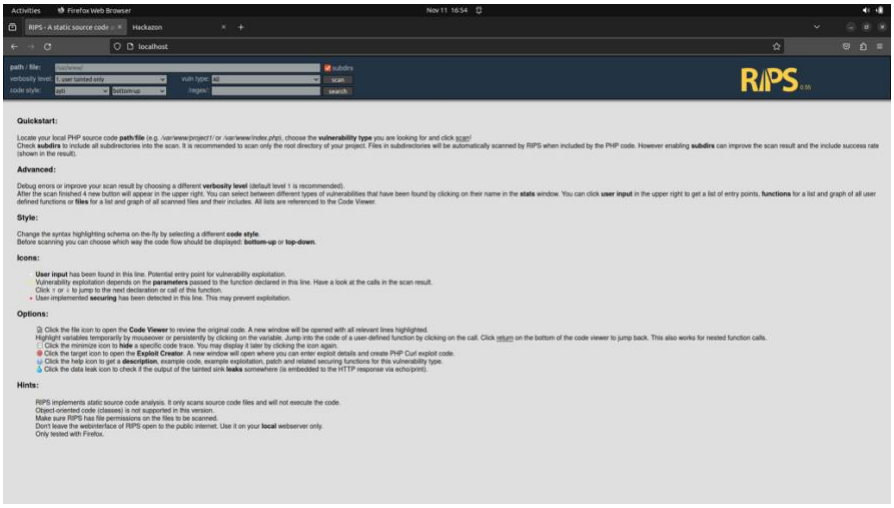
GitHub Repository from which RIPS was retrieved



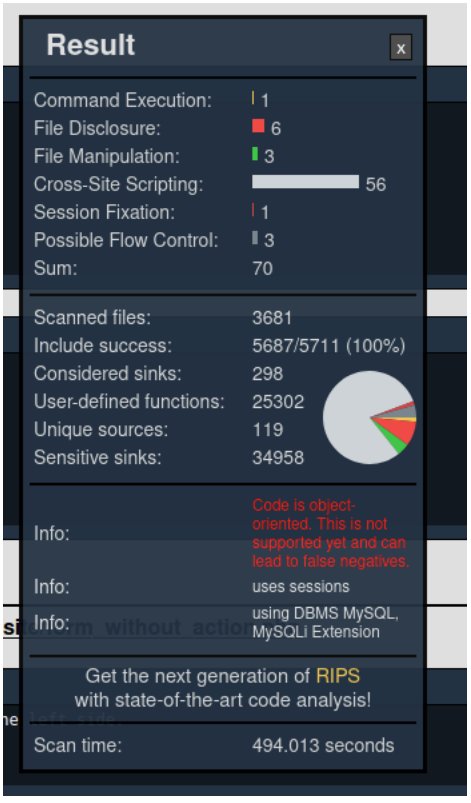
Directory for Apache Websites showing the addition of RIPS scanner



RIPS Scanner launched and configured



Evidence from static security scan



Vulnerability Report:

Through the static vulnerability scan of the Hackazon application, several vulnerabilities were discovered such as vulnerability to cross-site scripting (XSS), file disclosure, file manipulation, session fixation, possible flow control, and code execution. These categories made up the 70 distinct vulnerabilities discovered with XSS by far being the most prevalent accounting for 80% of the vulnerabilities identified (The full list of vulnerable code snippets from the 70 vulnerabilities discovered can be viewed in the appendix at the end of this report). With this, it can be seen that errors such as the vulnerability related to XSS have been replicated across the various pages of the application, and as such an examination of each vulnerability can allow for the vulnerability to be mitigated in each instance. An examination for each of the vulnerability types will be performed within the context of the Hackazon application, and this analysis should then allow for mitigation strategies to be employed that strengthen the security of the application.

The first, and most prominent, vulnerability present within the Hackazon application is the vulnerability to cross-site scripting attacks where a malicious actor could inject their own scripts into a webpage allowing those scripts to execute and carry out a malicious action (OWASP, 2021). XSS attacks will be examined more thoroughly within the following section of the report, but in general XSS attacks can be attributed to a lack of input validation for fields that repeat the user's input back to the webpage (OWASP, 2023b). This is often referred to as a reflected XSS attack as the user's input is reflected off of the database and returned to the user's webpage as part of an error message, search results, a comment in a comment section, or other places where the data would be presented back to the user. This is the most common vulnerability discovered in the Hackazon application as there are several fields for search results and comments where a reflected XSS attack could be performed. However, more dangerous is

the fact that the Hackazon application is also prone to more persistent XSS attacks where the malicious code could be stored on the database to carry out the malicious actor's purposes when other users visit the site. While the worst-case scenario will be further explored in the next section, it can be seen that a vulnerability to XSS attacks is a serious concern and must be mitigated by encoding the special characters used in HTML and JavaScript code so that the user entry cannot be interpreted as executable code (OWASP, 2023b).

The next most common vulnerability in the Hackazon application, file disclosure, can be seen as the vulnerability where a webpage unintentionally reveals information to the user such as files or information regarding the file tree infrastructure (PortSwigger, n.d.). Within the context of the Hackazon application, this security failure arises when user input is used to select a file allowing the user to retrieve files, they otherwise should not have access to. Further, the Hackazon application also allows for information regarding the file tree structure to be displayed when malicious code is injected through XSS scripts (PortSwigger, n.d.). This vulnerability could be amended by changing the code to not directly accept user input for file recovery but instead process the input to ensure the user is only ever returned files they should have access to. Several other vulnerabilities were also highlighted throughout the detailed vulnerability report, and by examining this report mitigation strategies could be created and each vulnerability addressed vastly increasing the security of the Hackazon application.

Examples of XSS:

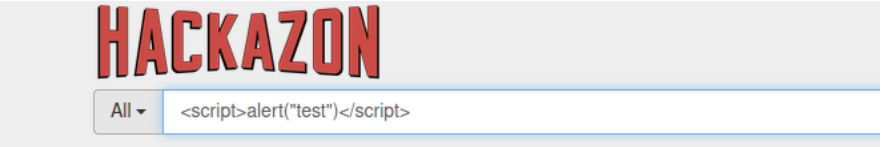
Cross Site Scripting (XSS) is the most common vulnerability discovered within the Hackazon application, and as such it is important to examine how XSS attacks work as well as how they are mitigated. To accomplish this, XSS attacks will be examined on the general level before closely examining specific examples from the Hackazon application to gain a clear understanding of the discovered vulnerabilities. Following this, the worst-case scenario will be explored to show the severity of this exploit and the effect it could have on the business and end users. Finally, mitigation strategies will be explored to discuss how XSS attacks can be mitigated increasing the security of web applications such as Hackazon. This examination of XSS attacks will then allow a security professional to understand, detect, and mitigate XSS vulnerabilities highlighting the importance of understanding XSS attacks.

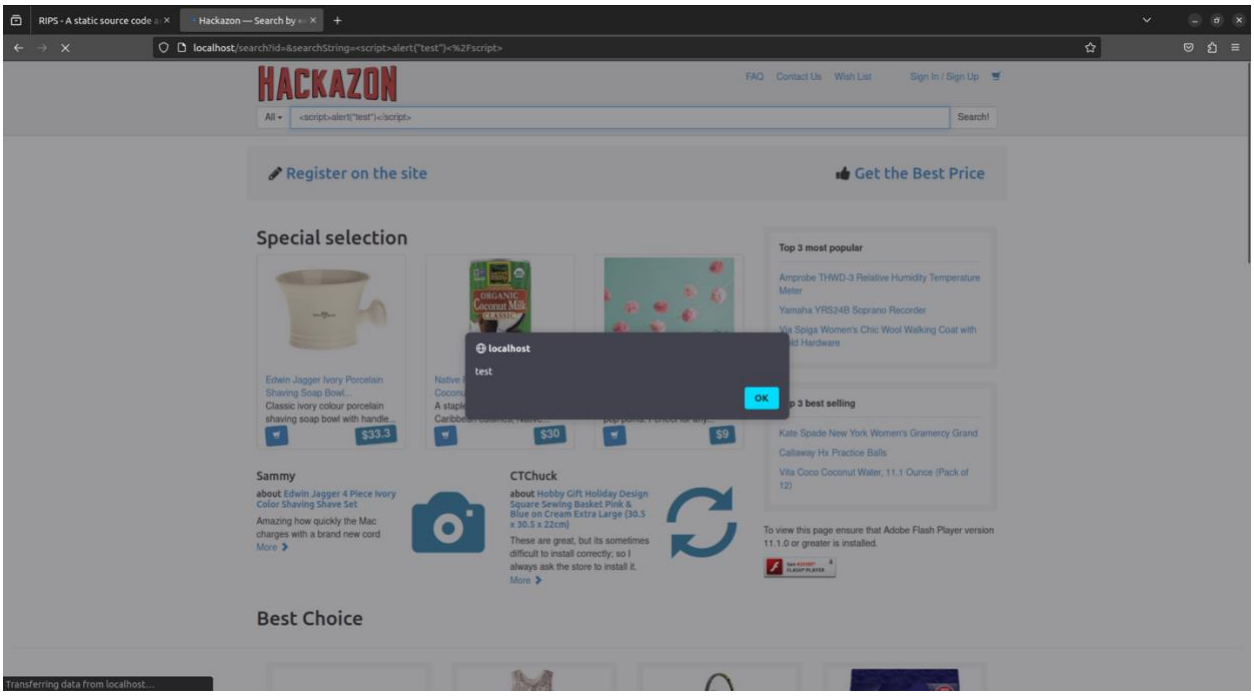
XSS attacks can be described as a form of injection attack where a malicious actor injects HTML or JavaScript code into a webpage through a field that echoes user input back to the webpage or to the database itself (Rodríguez et al., 2022). These attacks can then be broken down into their two most prominent classifications being reflected and stored XSS attacks. Reflected XSS attacks result when the malicious code is reflected off of the database and injected back into the webpage (OWASP, 2023b). For example, if the malicious code is injected into the search bar where the user input is echoed back to the screen, the code typically responsible for posting the user's input then places the code into the webpage where instead of presenting as text, the code is executed. In contrast to this, stored XSS attacks are more persistent attacks where the malicious code is stored on a database and then executed when that database value is retrieved and displayed on the screen (OWASP, 2023b). One example of this within the Hackazon application could be the storing of malicious code within the description of a posted

item for sale. When a user then goes to view this item the code is executed resulting in the XSS attack being performed.

To better understand these XSS attacks, specific examples from the Hackazon application will be examined. The first of these examples can be viewed on the index.php home page at line 592 of the code where the user input for the item search bar is processed.

This vulnerability then means that code can be injected into the search bar and executed on the system when the search results are returned. For example, when the JavaScript code is entered for the alert command surrounded by the script identifiers, the code is then executed on the page.





While this specific example is relatively benign, more malicious code could be injected to enumerate the system or attempt to steal data regarding the system or other users.

Another example in the code where an XSS vulnerability can be observed is in the cart/view.php file on line 11 where the HTTP environment variable for the host is echoed from the server.

This vulnerability means that if an attacker was able to manipulate this HTTP variable through another injection attack, then they would be able to execute an XSS attack when this input is echoed back to the user. Again, this highlights a severe vulnerability within the system that must be mitigated to ensure system security and integrity.

A third example of an XSS vulnerability within the Hackazon application can then be observed in the description of a product. This XSS vulnerability example highlights an example of the more dangerous stored XSS attack as this attack would be executed upon retrieving the malicious code embedded in the description from the database. An example of this can be viewed on line 109 of the index.php file where a description of a product on the home page is displayed.

Using this example as a basis for the worst-case scenario, it can be seen that code injected into the description of a product displayed on the home page would prove extremely dangerous as all users who visit the home page would then have the code executed on their page (Wibowo & Sulaksono, 2021). A malicious user may inject a window to pop up suggesting that the user must re-enter their login credentials and as the user has not left the trusted site, they may be inclined to enter the credentials allowing the malicious actor to be sent the credentials. If this idea was then extended to more sensitive information such as payment details, the vast potential damages of an XSS attack such as this can be seen. XSS attacks are a serious threat facing online applications today (Nunes et al., 2019) and for this reason, security professionals must understand XSS attacks and be able to mitigate them when they are discovered.

Fortunately, XSS attacks while incredibly dangerous are relatively easy to mitigate. The first step in mitigating XSS attacks involves the careful validation of user input so that all special

characters used in HTML or JavaScript code are encoded to benign values that will not result in code execution (OWASP, 2023b). This encoding should also hold true for the processing of values such as HTTP environment variables or any other input that could have been modified by a malicious actor. Additionally, a security professional can utilize tools such as a static vulnerability scanner to detect any segments of code that may have been missed and ensure that any remaining vulnerabilities are mitigated (Simpson & Anthill, 2017). As XSS vulnerabilities result from flaws in the source code, static analysis tools can be used to effectively detect these vulnerabilities proving their value to security professionals seeking to protect an application such as Hackazon.

References

Ferrara, P., Mandal, A. K., Cortesi, A., & Spoto, F. (2021). Static analysis for discovering IoT vulnerabilities. *International Journal on Software Tools for Technology Transfer, 23*(1), 71-88. https://doi.org/10.1007/s10009-020-00592-x

Ma, L., Yang, H., Xu, J., Yang, Z., Lao, Q., & Yuan, D. (2022). Code analysis with static application security testing for python program. *Journal of Signal Processing Systems*, *94*(11), 1169-1182. https://doi.org/10.1007/s11265-022-01740-z

Nunes, P., Medeiros, I., Fonseca, J., Neves, N., Correia, M., & Vieira, M. (2019). An empirical study on combining diverse static analysis tools for web security vulnerabilities based on development scenarios. *Computing*, *101*(2), 161-185. https://doi.org/10.1007/s00607-018-0664-z

OWASP. (2023a). *Source Code Analysis Tools*. OWASP. https://owasp.org/www-community/Source_Code_Analysis_Tools

OWASP. (2023b). *Cross site scripting (XSS)*. OWASP. https://owasp.org/www-community/attacks/xss/

OWASP. (2021). *OWASP top ten: Top 10 web application security risks*. OWASP. https://owasp.org/www-project-top-ten/

PortSwigger. (n.d.). *Information disclosure vulnerabilities*. PortSwigger. https://portswigger.net/web-security/information-disclosure

Rodríguez, G. E., Torres, J. G., Flores, P., & Benavides, D. E. (2020). Cross-site scripting (XSS) attacks and mitigation: A survey. *Computer Networks (Amsterdam, Netherlands: 1999), 166*, 106960. https://doi.org/10.1016/j.comnet.2019.106960

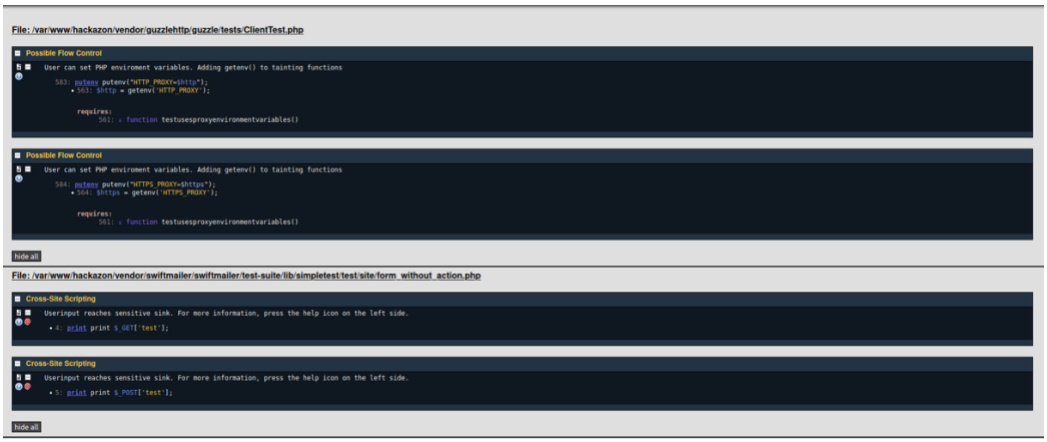
Simpson, M. T., Anthill, N. (2017). *Hands-on ethical hacking (3rd ed.)*. Cengage Learning. https://ng.cengage.com/static/nb/ui/evo/index.html?deploymentId=5681612456081340358553076923&eISBN=9781337271721&id=1937025370&snapshotId=3720027&

Wibowo, R. M., & Sulaksono, A. (2021). Web vulnerability through cross site scripting (XSS) detection with OWASP security shepherd. *Indonesian Journal of Information Systems*, *3*(2), 149-159. https://doi.org/10.24002/ijis.v3i2.4192

Appendix:

Full Vulnerability Report

Screenshot1:

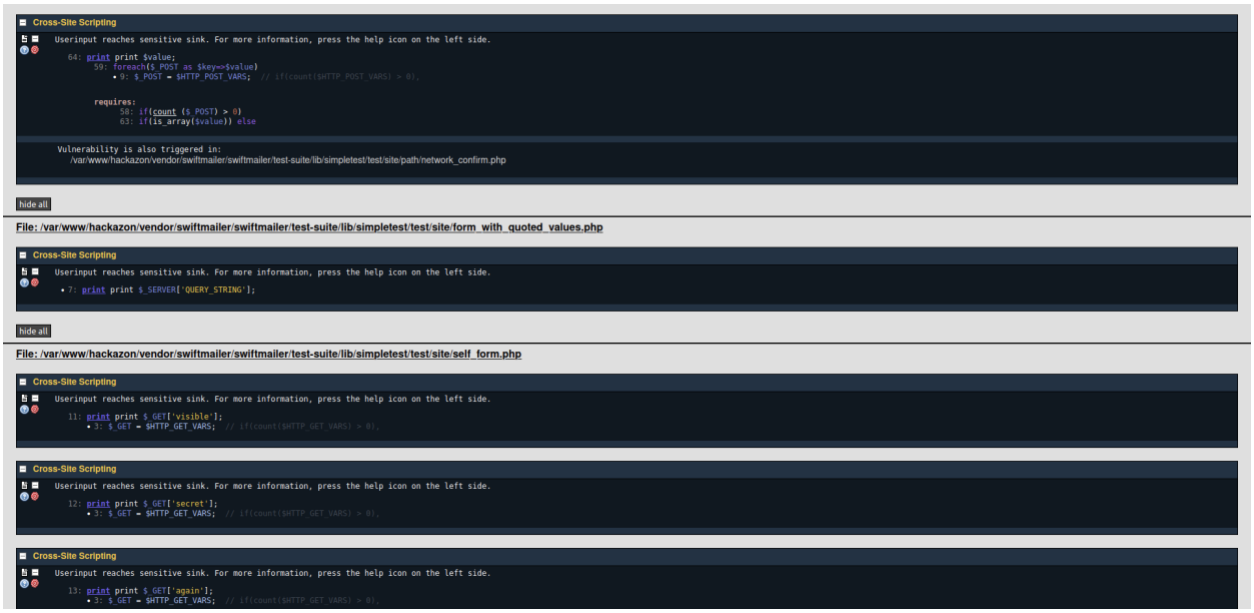


Screenshot2:

Screenshot3:

Screenshot4:

Screenshot5:



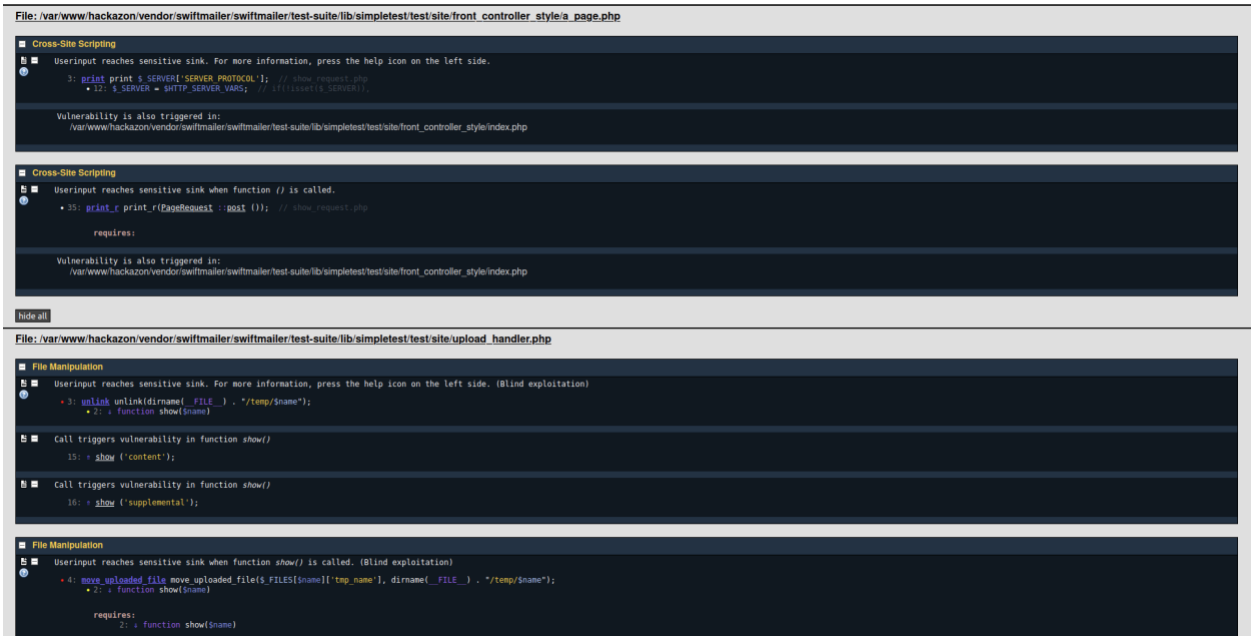
Screenshot6:

Screenshot7:

Screenshot8:

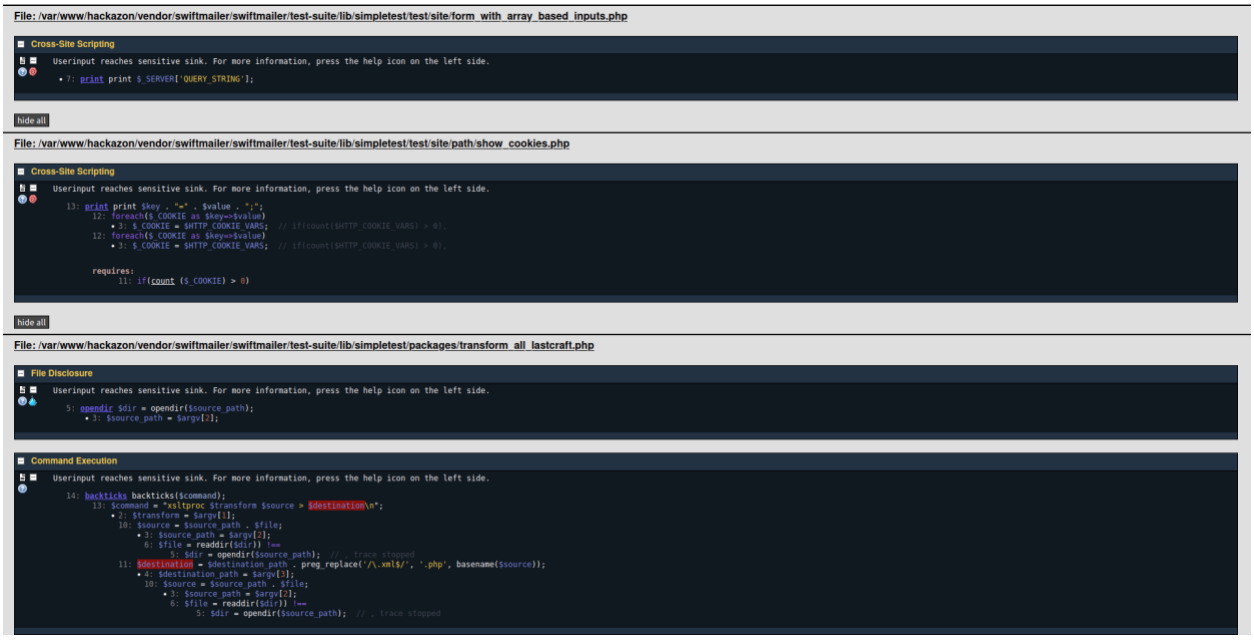
Screenshot9:

Screenshot10:



Screenshot11:

Screenshot12:

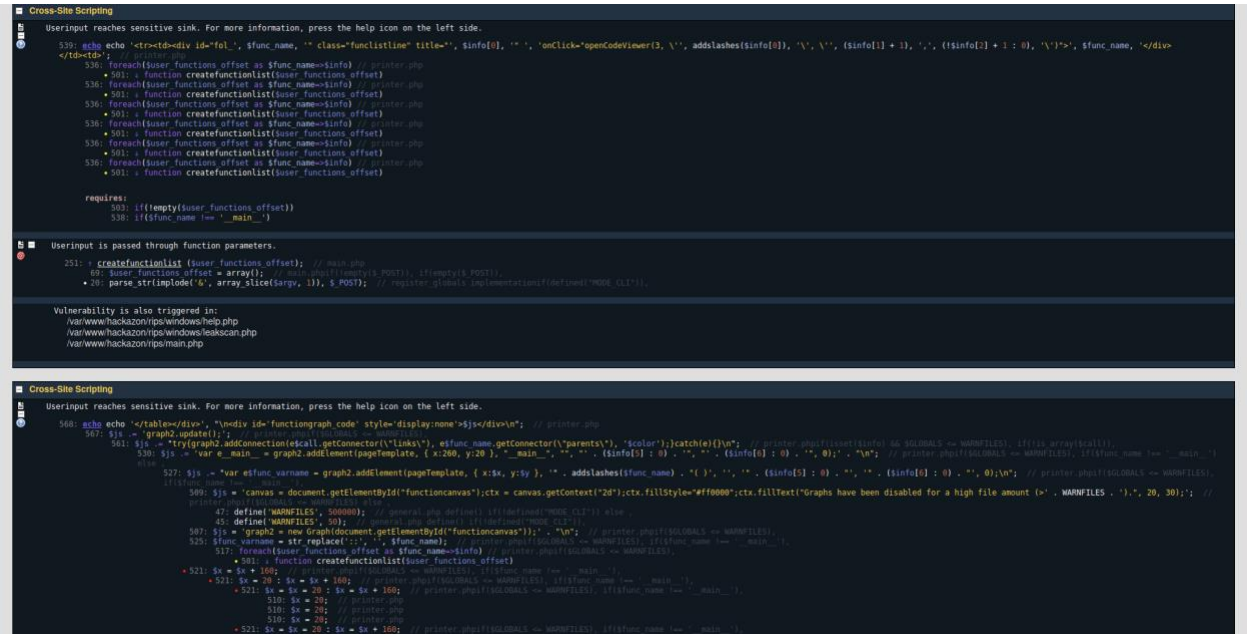


Screenshot13:

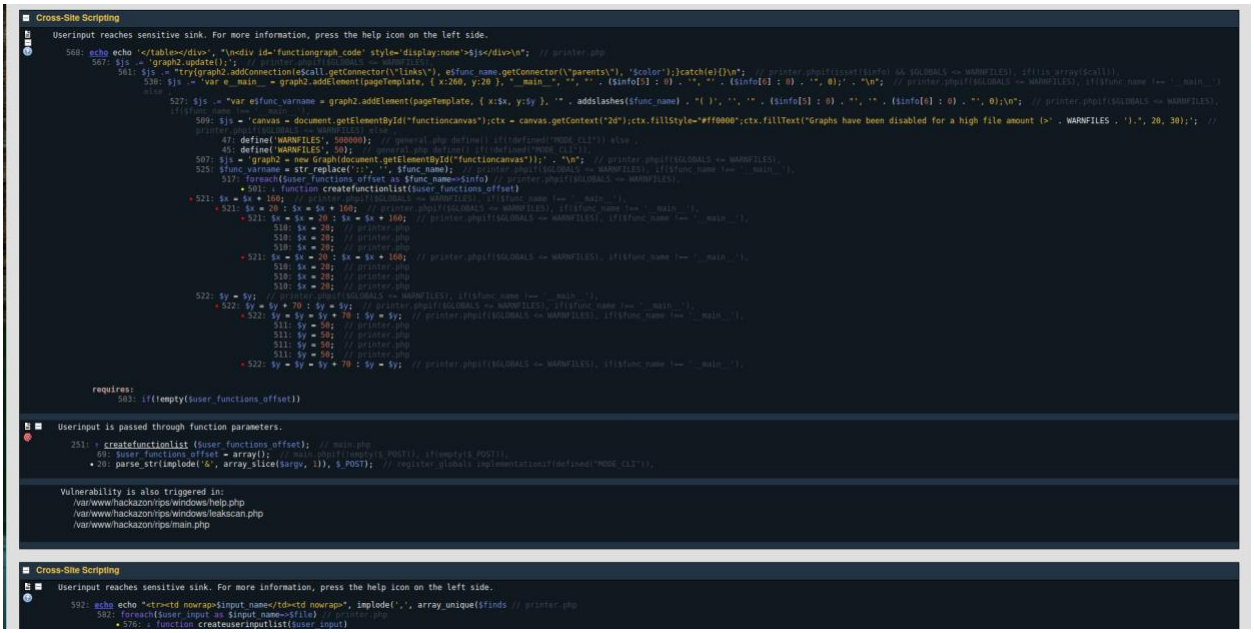
Screenshot14:

Screenshot15:

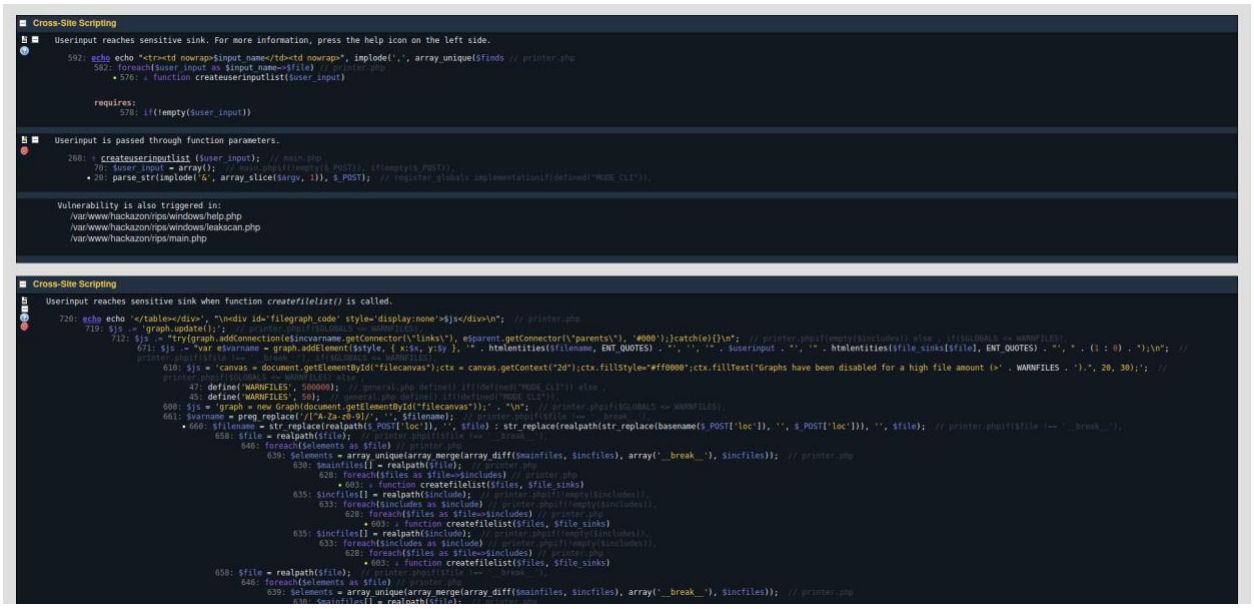
Screenshot16:



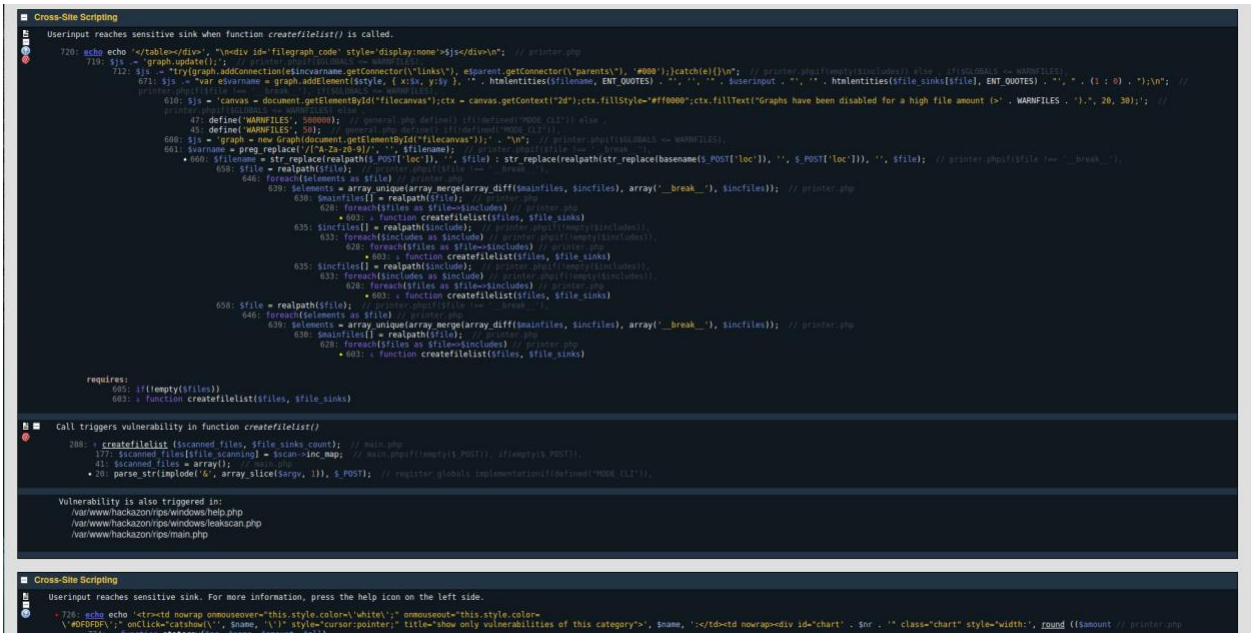
Screenshot17:



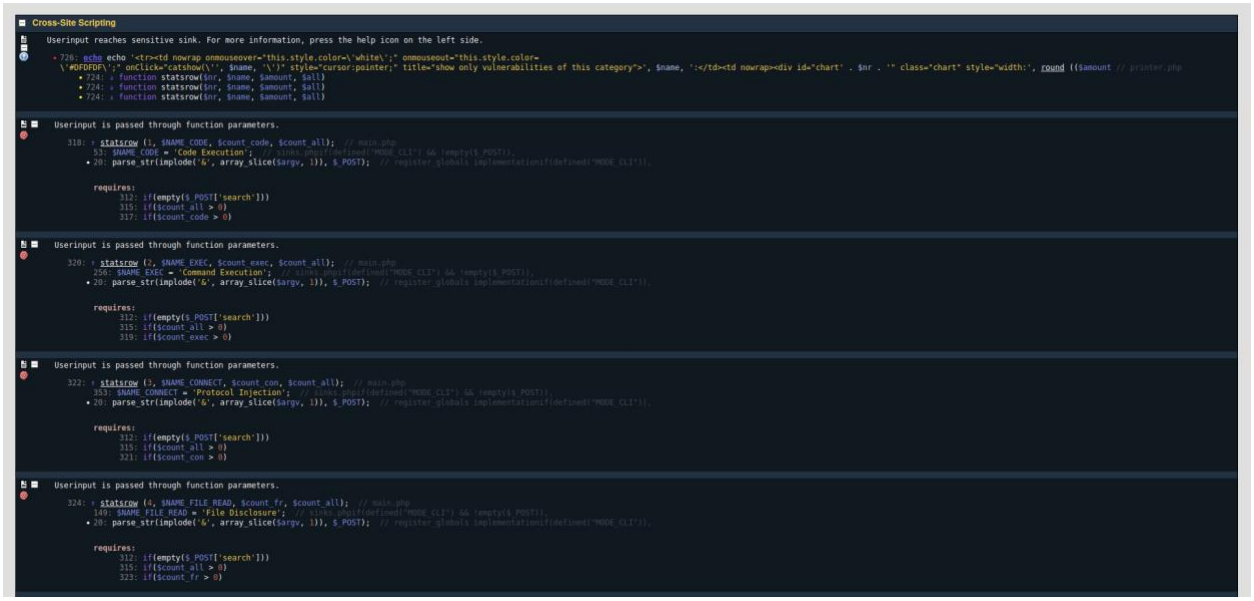
Screenshot18:



Screenshot19:



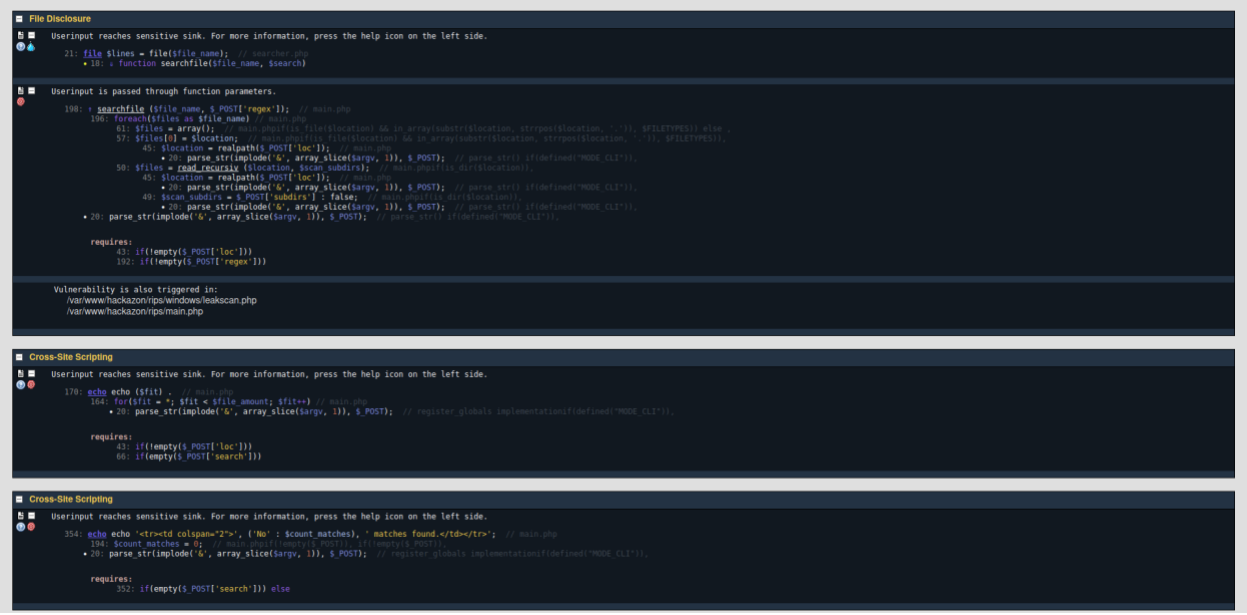
Screenshot20:



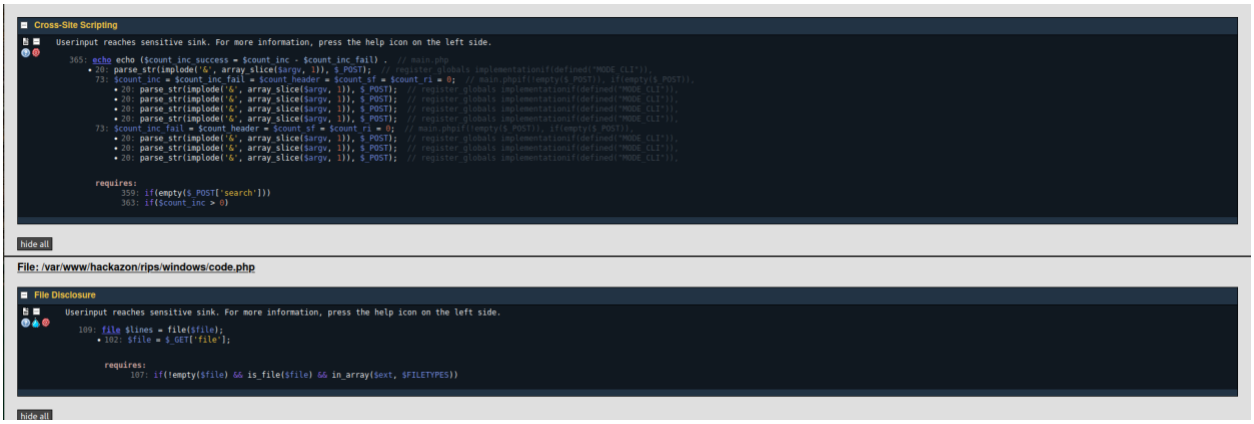
Screenshot21:

Screenshot22:

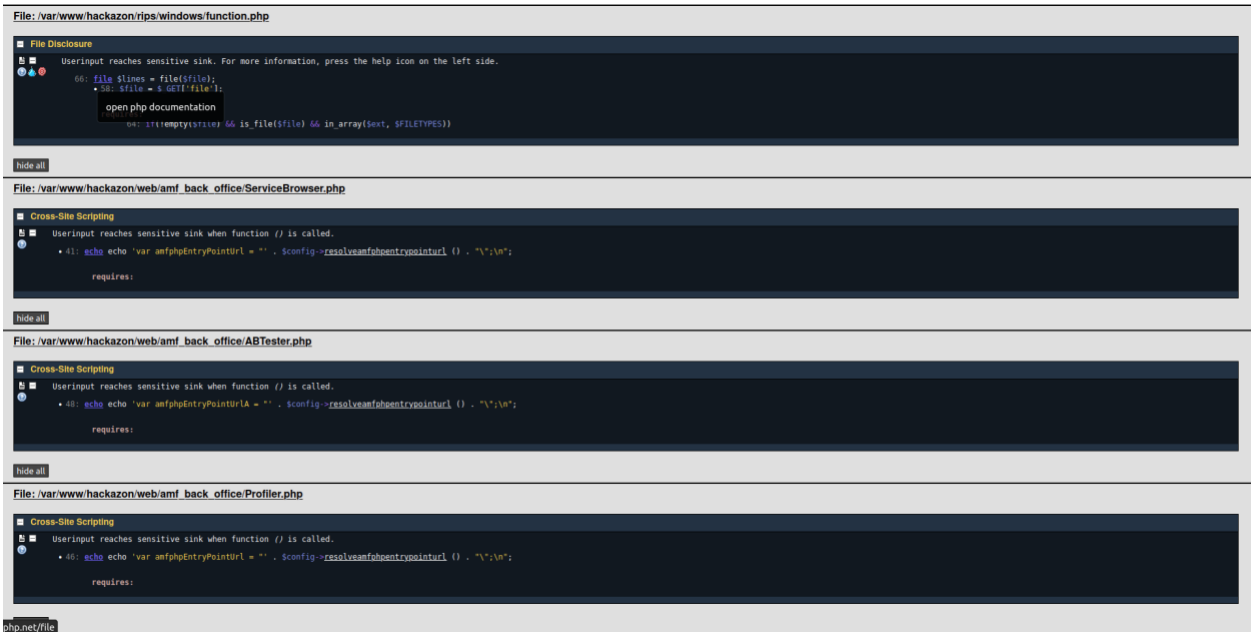
Screenshot23:



Screenshot24:



Screenshot25:



Screenshot26:

Screenshot27: