

Vexing Vulnerabilities in Web Applications

Joshua France
Florida State University
jaf20b@my.fsu.edu

Mehmet Ozmen
Florida State University
mco20m@my.fsu.edu

Jamel Douglas
Florida State University
jed18c@my.fsu.edu

Aiden Talavera
Florida State University
ajt18b@my.fsu.edu

Adam MacDougall
Florida State University
arm19i@my.fsu.edu

Jordan Northup
Florida State University
jtn19c@my.fsu.edu

Joshua Flashman
Florida State University
jaf19j@my.fsu.edu

Matteo van Zwieten
Florida State University
mjv21g@my.fsu.edu

Abstract

Systems are built to store and protect information in order for users to have availability, but there are people and organizations that work to exploit system vulnerabilities to steal or gain access to information that is confidential. This study focuses on web application vulnerabilities, and the methods tested to gain security, so that unauthorized users cannot gain access to files and information that they should not be privy to. The team has performed a number of penetration testings to assess the information on the system's security and integrity. and The PHP injection method of attack uses the input in the form of shell based commands to get information from a system with ease. We also tested for XSS vulnerabilities by using javascript code to manually check for a possible attack. The results the team found showed that both PHP injections and the XSS attacks have common flaws including lack of input validation and sanitization. We found that by implementing input validation code, the PHP injection vulnerabilities were disappearing, thus keeping the integrity of the system safe.

Keywords: Command Injection, SQL, Web Application Vulnerability, PHP injection, PHP vulnerability, Input Validation, XSS Attacks, Sanitation

Introduction

The debut of the internet to the modern world has been a boon to society, constantly evolving, surpassing, and advancing the world in ways people of the time could not realize. The internet has been used to store data and communicate globally. Many people who were incapable of using the internet in the past now have much easier access. Let us take an example such as making a website. Long ago in the early stages of the internet it would have been quite a feat to actually create and manage a site. Nowadays you have easy access to information to make a basic website without much hassle. In other words, everything is far more streamlined.

With the coming of this new age on the internet the websites you see on the internet now have been streamlined for easy use. User interfaces have become far more friendly to the general public. This came with such luxuries as user generated input. Whereas before the age of streamlined websites you had more or less banks of files and data stored within. Websites are now dynamic because of user input inserted into settings or a query of some sort. These customizable and friendly items have unfortunately given way to many malicious cyber threats. Namely hacking or damaging confidentiality, integrity, or availability of a system. These new and exciting features have created an incomprehensible amount of ways for criminals to get into systems with or without the owner knowing and unfortunately when you fix one problem another will always appear simply because of compromise.

There will always be new ways for hackers to get into a system. SQL injection, format string software attacks, and remote code execution are all a few ways out of many to get into systems where a criminal should not be. System

hijacking, data corruption, or system wipes are just a few ways these common hacking methods can attack systems. Catastrophic damage can be done to websites through this malicious use of code.

The focus of this paper will be on examining a basic vulnerable website, determining critical vulnerabilities, and attempting to solve them. These vulnerabilities will be dissected and explained. Solutions will be presented based on gathered data and an understanding of the issue.

Associated Research

A Classification of SQL Injection Attacks and Countermeasures

There were quite a few scholarly entries regarding this injection method but none were so informative to the team than *A Classification of SQL Injection Attacks and Countermeasures* by William G.J. Halfond, Jeremy Viegas, and Alessandro Orso. This paper classifies and agrees with other scholars and the research team that SQL injection is one of the most prominent threats for web applications. Within *A Classification of SQL Injection Attacks and Countermeasures* it states that web applications that are vulnerable to attack from injection can be subsequently, in the worst cases, either taken control of or corrupt the web applications host. This is definitely a scenario that is extremely dangerous for anyone involved whether sensitive data from a user or the host themselves. Taking a look at the apache server used for this research and running commands, this is a true statement made by *A Classification of SQL Injection Attacks and Countermeasures*.

The team has come to a consensus with *A Classification of SQL Injection Attacks and Countermeasures* when it comes to the main problem “Many of the techniques have problems handling attacks that take advantage of poorly-coded stored procedures and cannot handle attacks that disguise themselves using alternate encodings.”. Code structure is the most important and can be security’s greatest flaw when it comes to SQL injection. As discussed in further sections of this research paper, poorly coded work in itself has dangerous holes to exploit.

The research was particularly useful on solutions for the team. The paper gave out fine solutions ranging from black box tests which is a type of technique for testing web applications for testing vulnerabilities that have to do with injection. Which is a web crawler that injects code into files to see if a vulnerability can be found in the specific area.

The specific technique for prevention and countermeasures that the team had found strongly resonated with our research was the static code checker. The static code checker allows us to check for type correctness and, depending on the static code checker, allows users to check for executive code within a dynamic string entry within the code.

Overall the solution helped the team understand the efficiency and the inner workings of the more in depth vulnerabilities that were occurring within the system that we had prepared. Our new understanding of the static code checker that was expounded upon in *A Classification of SQL Injection Attacks and Countermeasures* allowed us to pull from the similar situations in *A Classification of SQL Injection Attacks and Countermeasures* to realize a similar solution.

DNS Pharming through PHP Injection: Attack Scenario and Investigation

Similar in nature to SQL Injection, PHP Injection attacks take advantage of vulnerabilities in unsanitized input entries. In the article *DNS Pharming through PHP Injection: Attack Scenario and Investigation* by Divya Rishi Sahu and Deepak Singh Tomar of the Maulana Azad National Institute of Technology (MANIT), they discuss a method of DNS pharming using PHP Code Injection, as well as methods of investigating these types of attacks. In terms of mitigation of risks, they clarify that “sanitization of input strings and reduces (of) privileges” ... “are the two main mitigation techniques to mitigate the code injection vulnerability.” (21)

In their attack investigation, they exploit a vulnerability in an input field to create a new unauthorized PHP web page, which is then used to collect DNS information on victims visiting the site. When victims visit the main page, web server, they are then redirected to this unauthorized web page that then collects this data.

As a response to this attack, the writers of the article proposed a complex algorithm that could be used to detect a PHP Injection. This script was successful when tested against the DNS Pharming script, and detected it before it could take control of the system.

The duo's investigation of PHP Injection helped us in analyzing and understanding how this injection method works, and more information about what can be done to help prevent it. PHP Injection vulnerabilities are a big concern that, while easily patchable and preventable, can be very dangerous if not understood and vulnerabilities are left open.

Testing and comparing web vulnerability scanning tools for SQL injection and XSS attacks

This research paper was focused on pointing out the dangers that can occur if a site is vulnerable to SQL injections or XSS attacks. Researchers Marco Vieira, Henrique Madeira, and José Fonseca knew that hackers started to widely use automatic web vulnerability scanners in order to find and attack sites that are easy prey for SQL injections or XSS attacks. They conducted tests and looked into how the vulnerability scanners functioned and what key things they were searching for when detecting the vulnerabilities, so that they could use the research to provide better defense for websites. They present statistics at the beginning to show what a widespread issue a lack of cybersecurity is and how many attacks occur.

In order to test the different web scanners they used a two stage testing plan. In the first stage the code of the target web application is examined in order to identify all the points where each type of fault can be injected, resulting in a list of possible faults. When the list of faults is very large (because the application code is extensive, resulting in lots of possible locations for each fault type), only a percentage of the fault locations is used. The second stage comprises the injection of each fault, which corresponds to the insertion of the code change (defined by the fault operator) in the web application. After injecting each fault, the web application is scanned by the tools under evaluation to compare their results. Every test starts with a restoration of the initial setup: the web server is restarted; the database is restored; and the website files are copied from a clean backup. Then, the next fault is injected into the web application. After, the scanner application is started and at the end the results are saved into a file. The file name includes a reference to the delta file, the web application file and the type of fault injected. The Java tool monitors the scanner application to detect when its execution stops before continuing the next test. This procedure is repeated until all the faults are injected. This procedure is also repeated for all the web vulnerability scanners.

After running through the processes they determined that all the scanners had detected some vulnerabilities that none of the others had. The results of the evaluation of three leading web application vulnerability scanners show that different scanners produce quite different results and that all of them leave a considerable percentage of vulnerabilities undetected. The percentage of false positives is very high, ranging from 20% to 77% in the experiments performed. The results obtained also show that the proposed approach allows easy comparison of coverage and false positives of the web vulnerability scanners. In addition to the evaluation and comparison of vulnerability scanners, the proposed approach also can be used to improve the quality of vulnerability scanners, as it easily shows their limitations.

Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis

A novel technique called the Knuth-Morris-Pratt (KMP) string match algorithm is used to defend against both cross-site scripting (XSS) and SQL injections. Development of this tool first required the study of various attack patterns, where a parse tree was then created to visualize these patterns. From these patterns, a "filter()" function was created by using the KMP string matching algorithm. Using the "filter()" function allows the detection of malicious code and prevents all forms of SQL injections or XSS attempts. The way it works is by matching the user's input data with the data of the pattern of the injection data. Every input string is passed through the filter function, and if it returns true then the user is immediately blocked and prompted with a warning message.

Other various server-side and client-side methods do exist as well. A popular method for server-side protection is the use of Perl's taint mode. This interpreter tracks tainted values, and any untrusted sources are marked as potentially malicious and are then propagated. Any upper-level access requests are immediately shut down and the

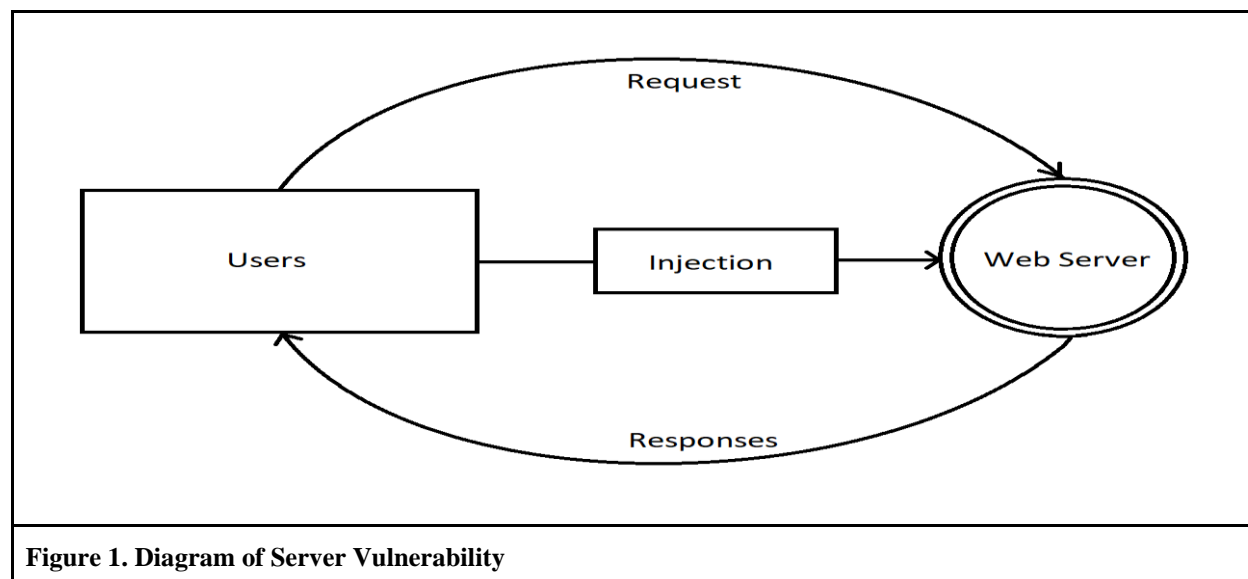
user is prompted with an error. As far as client-side defenses go, proxy services are a user's best friend. Proxies analyze all HTTP traffic that is exchanged between the user and the web server. Clients are first scanned to see if the request contains any special characters (such as the "%" character). If the special characters are found, the request is marked as potentially malicious and the website is considered as vulnerable to XSS. As a result, the special characters must be encoded before the response is delivered to the user's browser, blocking any attempts to attack. Application-level firewalls are another defense method that analyze HTML pages for included hyperlinks that may potentially lead to a potential sensitive data leak.

This was especially helpful to the team in learning and trying to come to an understanding of filtering. Much of the team's research is going to come from input sanitation and filtering. Most of the injection scans that were enacted for the purpose of this research paper uses these words as a basis for solutions.

Methodology

This research team will apply the use of an up to date threat model to accurately research and solve the vulnerability at hand. A threat model according to Victoria Drake from Open Web Application Security Project or OWASP is "A threat model is a structured representation of all the information that affects the security of an application.". The research team has concluded to develop and employ their own threat model to properly evaluate and solve the vulnerability at hand.

The team has employed a threat model defined by these steps: identify, understand, reference, and weaken. The identify step was taken care of immediately as the team had previous testing in injecting commands into a text box and using a vulnerability scanner allowing for the team to immediately come up with the problems that infested the system. The team then created a simple diagram, as shown in figure 1, to create a visual representation of the threats and how it worked within the server.



This method was fleshed out by the use of a vulnerability scanner to identify the threat. The vulnerability scanner was then used to piece together and understand the standing vulnerabilities within the server. The team prioritized the critical problems and vulnerabilities within the apache server. Using the vulnerability scanner and the team's understanding of the now identified vulnerabilities the team got to work by trying to come up with ways these problems could be removed.

It was then necessary to begin trying to reduce or weaken the injection vulnerability and cross site scripting that had been identified and understood. The methodology employed was a simple yet effective method of dealing with the vulnerability at hand.

Server Environment: Ubuntu

Before attempting to resolve injection vulnerabilities, it is necessary to create a server to experiment with and test exploitation methods on. The following section will detail the steps taken to create a simulation environment for both testing exploitations and applying countermeasures.

Apache Server Configuration

Our tests involved creating the proper test environment involves the installation and setup of apache as the web server hosting software. Having this server established and configured correctly is critical to the effectiveness of being able to experiment appropriately on a realistic environment.

Apache can be executed and the server can be launched using the command '`sudo systemctl start apache2`'. Once the server is successfully running, it is time to switch to an OS with a graphical interface so that the web server can be visited and displayed correctly. Before leaving Linux, it is important to grab the ip address for the server by running `ifconfig`. An example of this can be seen in Figure 3, with the required IP highlighted.

In a windows environment on the same network, the web server can be tested by connecting to the IP address collected from the Linux device. Once the connection is successful, the website can be viewed and interacted with completely from a Windows or other graphical interface based device.

Understanding Injection

Vulnerability and Risks

PHP Command Line Injection

PHP command line injection is a type of attack where an attacker exploits a vulnerability in the PHP component of a site, which allows attackers to run commands on the server that site is run on. Typically, PHP is used to parse information that is submitted to a form on a website. When the user presses the submit button, the server runs a PHP script which would input the data into a database. Through PHP code injection, attackers can submit commands in a textbox, and the server will execute them. Figure 2 is a diagram that highlights the original submission process, and shows us how the attackers will execute commands.

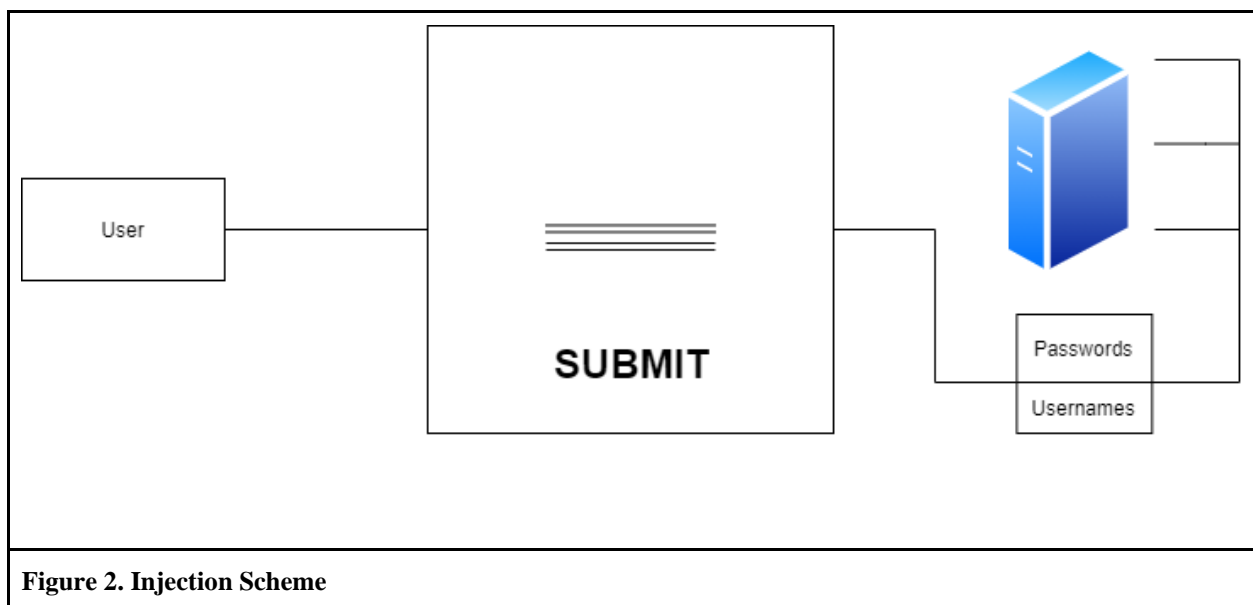


Figure 2. Injection Scheme

Risks

The biggest risk that PHP code injection causes is the ability for attackers to use a text entry box to execute commands on the server. With the ability to run commands on the server, an attacker can do many malicious activities. This can become quite problematic when trying to oversee a server that the host should be in control of but instead it can be overridden by an unseen attacker.

For example, attackers can use commands to access and make their way through the file system. This means that an attacker can see and access files that should be secured and confidential, like service configuration files. Depending on the intent of the attacker, this type of attack is typically used to gain more information about the target machine to possibly aid future attacks.

Other examples of malicious activity that an attacker may do would be extracting data, executing a denial of service attack, and executing remote commands.

Due to the nature of PHP attacks, the possibilities of attacks are only limited by the attacker's imagination and ability. Once a vulnerability or exploit is discovered, it is just a matter of inputting the right sequence of code to attain what the attacker desires. There are several different ways of accomplishing this such as injection through user input, injection through server variables, injection through cookies, and a second-order injection. Depending on what the website does or what it offers, an SQL injection can access confidential information, modify the information displayed on the site, or perform unwanted actions. In a user input injection attackers will use the input box like a terminal to execute code, an injection through server variables uses HTTP, network headers, and environmental variables, and an injection through cookies is when an attacker hijacks a website's cookies to build SQL queries.

A second-order injection is when an injection is made and the code is set to occur at a later time after an action happens on the site. This makes it much more difficult to detect and discover changes. An example would be using a user input injection to modify the SQLIA so that the next time an admin logs in, their password gets changed to one only the attacker knows. While safeguards such as sanitizing the input boxes by disabling special characters and limiting the number of characters a user is able to input can prevent these kinds of injections, there can still be workarounds that aren't apparent. If the database is part of a public package or open sourced then it can be very easy to learn how to bypass certain security measures. Attackers will spend time trying different approaches to private software and learning from the error messages what the restrictions are of each input box and strategize potential workarounds, especially if detection software is not built in.

Understanding Cross-Site Scripting

Cross-site scripting or XSS for short is a type of malicious script injection. This is the most frequently spotted web security vulnerability. These scripts are injected into trusted websites and begin to mess with the site or target the users who access it. Browsers are often tricked into running these scripts because they trust the source and assume it came directly from that trusted site. It is then able to access any cookies, session tokens, or potentially sensitive data that the browser has saved for that site (ex. usernames, passwords, site settings, user data). The two main attack types are "stored" and "reflected" XSS attacks. Stored attacks are where the injected script is stored on the target server(s) permanently. The user then receives the malicious script as soon as their device requests information from the server. In reflected attacks, it is required that the user performs an action before having the script delivered to them. This could occur through clicking a malicious link, submitting a malicious form, or just browsing to a malicious site.

Preventing cross-site scripting can be a very difficult task. Depending on how the site manages user-controllable data, prevention could require many implementations to successfully defend against it. There are many ways this can be accomplished. Filtering input on arrival is one of the more common protection methods, as it filters all user input based on what input is expected or valid. Encoding the output data can be used to prevent this data from being interpreted as active content which could require encoding from a few different sources (ex. HTML, URL, JavaScript, and CSS). This can make it harder for the perpetrator to view the stolen information. A great last-line of defense would be the use of Content Security Policy (CSP). This tool provides a method for the site to use that talks

to the browsers of users, defining which content of the site is allowed to load in the page and in return restricts the attacker.

Steps and Logic

Step 1. Network Topology

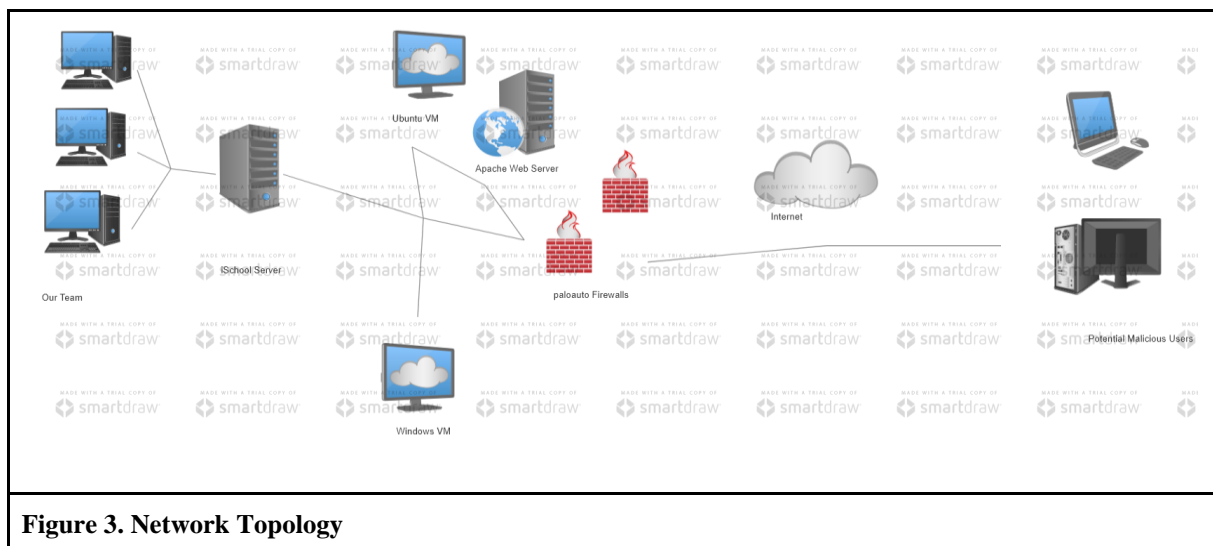


Figure 3. Network Topology

The first step of this process was to map out the network topology so that we could have a better understanding of what would be occurring throughout the process. The topology of a network is the layout of all the components such as computers, switches, firewalls, etc. and how they connect and interact with each other. We have constructed one to illustrate our virtual server and how attacks are performed. To set this up, we have all been given access to a virtual machine, so there are several computers to represent our team. The virtual machine is running Ubuntu and is located on the iSchool server in the SECNET vLab, so that is what our computers are connecting to first. On that server we each have access to three VMs, a windows VM, ubuntu VM, and the shared firewall. We use the Ubuntu VM to extract and run the Apache web server and can use the Windows VM to view the GUI (graphic user interface) of the web server. We also use the windows VM to view and set up the palo alto firewalls. The Firewalls are used to protect against attacks across the internet from potential malicious users.

Step 2. Arachni Vulnerability Scanner and Identifying Vulnerabilities

We decided to use a cyber security tool to run a scan of the vulnerable site to list out any possible vulnerabilities in the site. For this task, we decided to use a program called Arachni, which is a web application security scanner. We decided to use Arachni because of its ability to list out all vulnerabilities in a nice visual report. Knowing how to manually check for vulnerabilities is a fantastic start but having a program to help the team identify and map out the vulnerabilities was paramount to a thorough search of the system.

This first step was to install the scanner onto our Windows 10 virtual machine. Once installed we had to click on a batch file to get everything up and running. From there, we navigated to the web UI login page using a web browser. The web UI for Arachni is going to be located at the localhost on port 9292. Once logged in using the default admin account login, we can begin the process of running a scan with the set of policies we want. We clicked on the scans button on the taskbar and selected new from the dropdown. From there, a page loads that allows us to input our target url, which happens to be the IP address of our Ubuntu virtual machine, and select the configuration file to be used. An example of this is shown in figure 4 below. A description can be added if needed. Once we were ready to

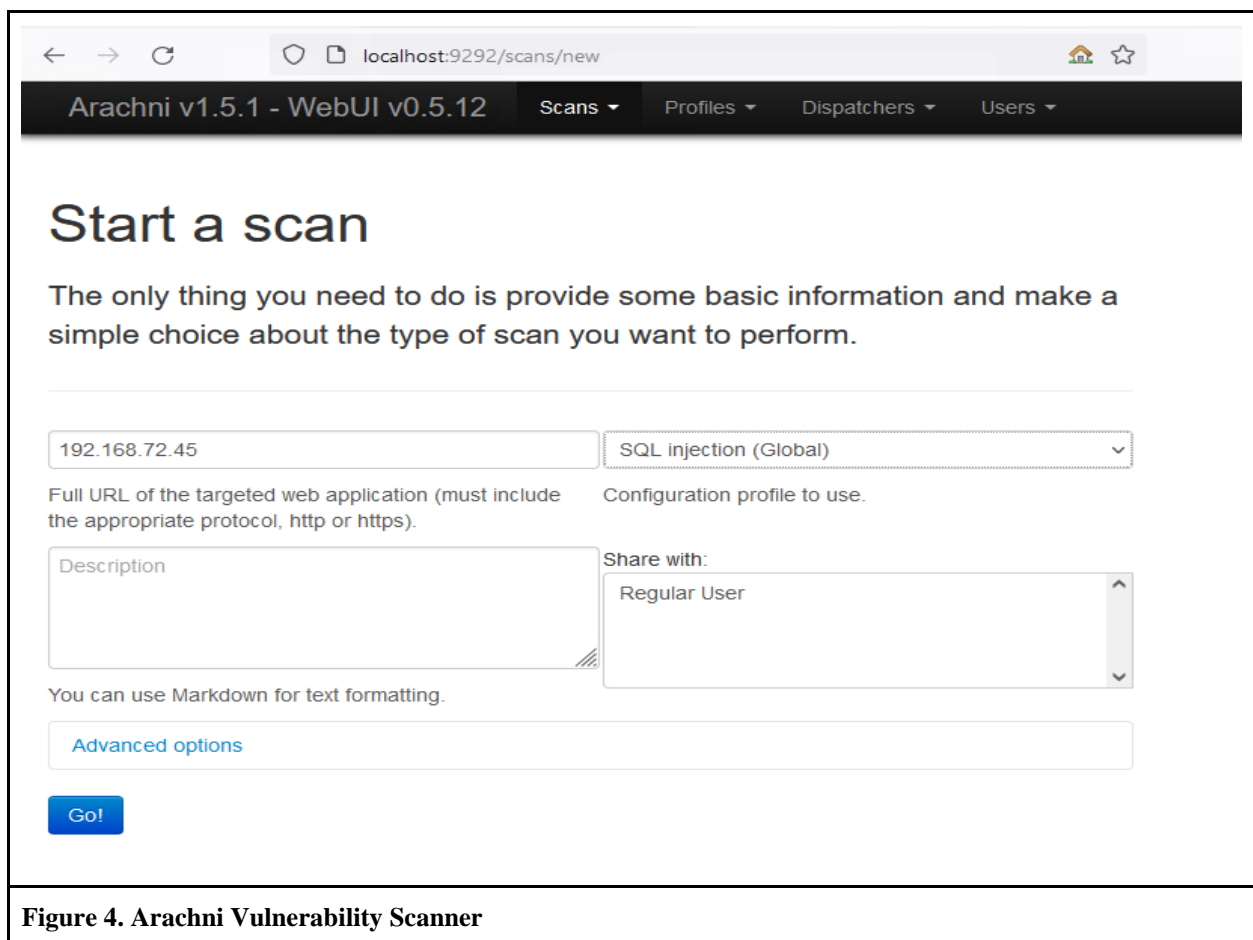
run the scan, we hit the go button at the bottom of the page. Once the scans are completed, the team viewed the vulnerabilities through the report that was produced.

Step 3. Analysis

With our completed reports, we completed a team wide analysis of both the injection and XSS scans to understand which vulnerabilities we are experiencing. We compared our results to find correlations between the different vulnerabilities that we researched throughout this paper.

Step 4. Brainstorming Solutions

At this point we got together as a group and discussed different methods to mitigate the vulnerability issues that we found. As we have no knowledge on how the sites were set up, all we can offer are recommendations that we have discovered throughout our research. Also we are not able to complete any of the solutions because that is also out of the scope of our knowledge.



The screenshot displays the Arachni web interface for starting a new scan. The browser address bar shows 'localhost:9292/scans/new'. The page has a dark navigation bar with 'Arachni v1.5.1 - WebUI v0.5.12' and several dropdown menus: 'Scans', 'Profiles', 'Dispatchers', and 'Users'. The main heading is 'Start a scan', followed by the instruction: 'The only thing you need to do is provide some basic information and make a simple choice about the type of scan you want to perform.'

The form contains the following fields and options:

- A text input field containing '192.168.72.45'.
- A dropdown menu for 'Configuration profile to use.' with 'SQL injection (Global)' selected.
- A text input field for 'Full URL of the targeted web application (must include the appropriate protocol, http or https)'.
- A text input field for 'Description'.
- A dropdown menu for 'Share with:' with 'Regular User' selected.
- A note: 'You can use Markdown for text formatting.'
- A link labeled 'Advanced options'.
- A blue 'Go!' button.

Figure 4. Arachni Vulnerability Scanner

Vulnerability Analysis Results

The team ran Arachni to discover the cross site scripting vulnerabilities for the XSS webpage. The results are in figure 5 below.



Figure 5. Global Vulnerability Scan

The results of the vulnerability scan from Arachni showed quite a few high level vulnerabilities in our scans. The high level vulnerabilities are as follows: path traversal, operating system command injection, and file inclusion.

Path Traversal

Path traversal or known similarly as directory traversal is the manipulation of folders and or directories outside of the web root folder. An attacker will typically try to gain access to those files and or directories. Understanding the base cause behind this vulnerability was not too difficult. We simply had to look within the web root folder. Configuration folders should not be stored within the web root folder. Direct shell execution functions, file system calls should not be used when using user input. Attackers will be able to inject commands into user input by manipulating variables in order to gain access to the directories stored on the web server. Manual searching for path traversal vulnerability is similar to XSS vulnerability because it works by either exploiting form input or brute forcing URLs to move around the directories.

File Inclusion

File inclusion is superficially similar to path traversal, in that an attacker will use the vulnerability to manipulate folders and directories. Where it differs is that while a path traversal vulnerability allows an attacker to read inaccessible files, a file inclusion vulnerability allows them to load and execute any file inside of the context of the file that contained the vulnerability. This can be seen directly in the PHP file below in figure 6. The full path allows for these files to be accessed and then carried out as executables. There are two ways for file inclusion which are remote and local similarly to the XSS where it is called stored and reflected. When a PHP wrapper is enabled, an attacker is authorized to upload a reverse shell using the XSS vulnerability of form inputs.

Operating System Command Injection

Operating system command injection can occur when user input is used in code that runs commands on the shell. If these inputs are unsanitized, this can allow a malicious user to run essentially arbitrary commands on the server by exploiting this vulnerability.

```
admin@4774a-ubuntu-04:/var/www/html/extremeinsecure$ cat process.php
<html>

  <head>
    <h1> The results:</h1>
    <br/>
  </head>

<body>
<?php

print "<p>";
$tool = $_POST['tool'];
$tooldir = './products/' . $tool;

print "<b>The files in the tool are:</b>";
system("ls $tooldir");

$filename = $_POST['filename'];
$fullpath = './products/' . $tool . '/' . $filename;
$cmd = 'cat ' . $fullpath;
system("cat $fullpath");
print "</p>";
```

Figure 6. PHP file

Operating system command injection had been found in the team's scan using Arachni. Operating system command injection is in this case PHP injection. To understand how this was possible the team decided to open the PHP file itself that the website runs on.

As we can see in figure 6 above, the function that created the possibility of the command injection from the PHP file is system(). system() function is using "cat \$fullpath" which basically is using linux command with a string parameter and underlying operating system. system() is for executing a system command and displaying the output, which in this case is a string. Since it's a string, users can write any linux command to run in the operating system in the text input box. To be able to prevent these PHP functions from creating command line injection, we should withhold from using direct shell execution functions such as exec(), passthru(), system(). These should not be used since they produce highly unportable code in PHP.

Figure 7 shown below is the input box from the apache server that has the vulnerability of command injection. It can be exploited by typing any unix commands. This unfortunate vulnerability can allow them to run anything they wish that the server host can run as well. An example of this as shown below is ";ls". This command allows a user to see the list of files in a unix directory. After the potential user knows the name of files accessing them is no problem. Usernames, password, or other sensitive information is easy to steal when you have complete access to a server.

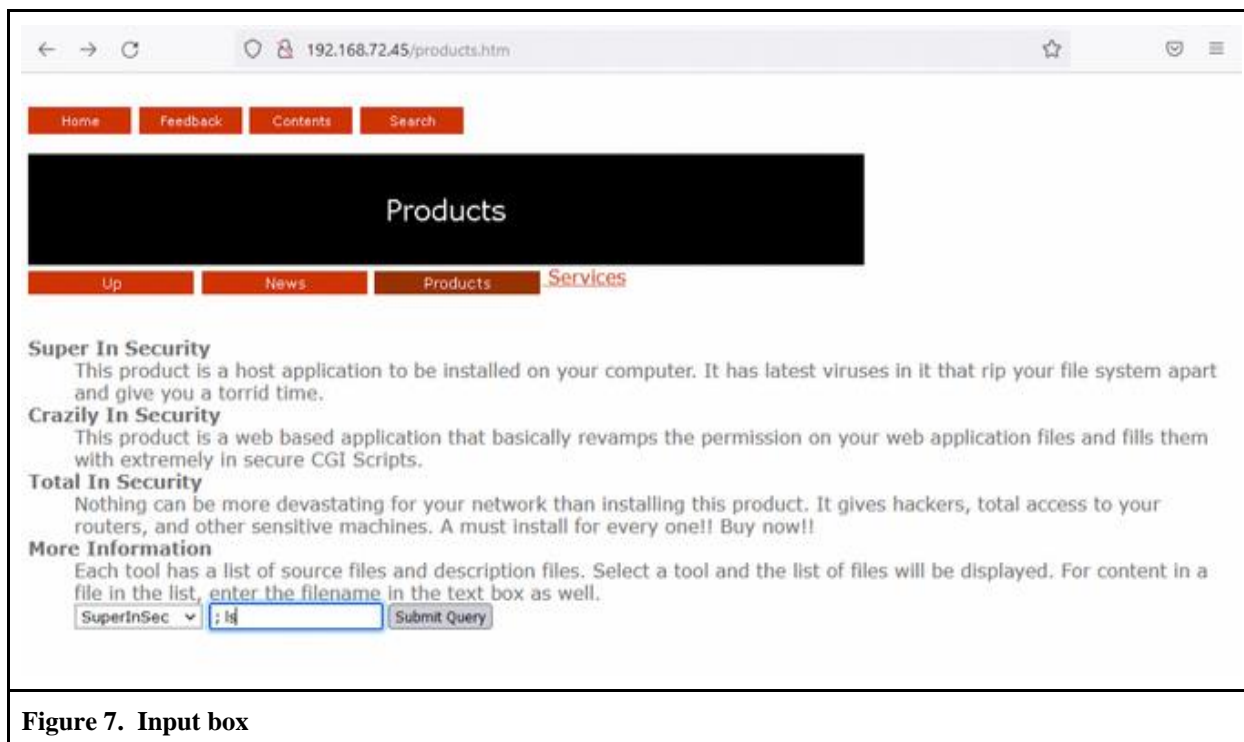


Figure 7. Input box

This query input box will take any shell commands and execute in the real-time environment the server is running. A hijacker can execute an open shell inside the form input box which will give him/her full access to the server's runtime environment.

Figure 8 below shows the results of even a simple command like "ls" and gives the results of the command when run through the website's vulnerability.

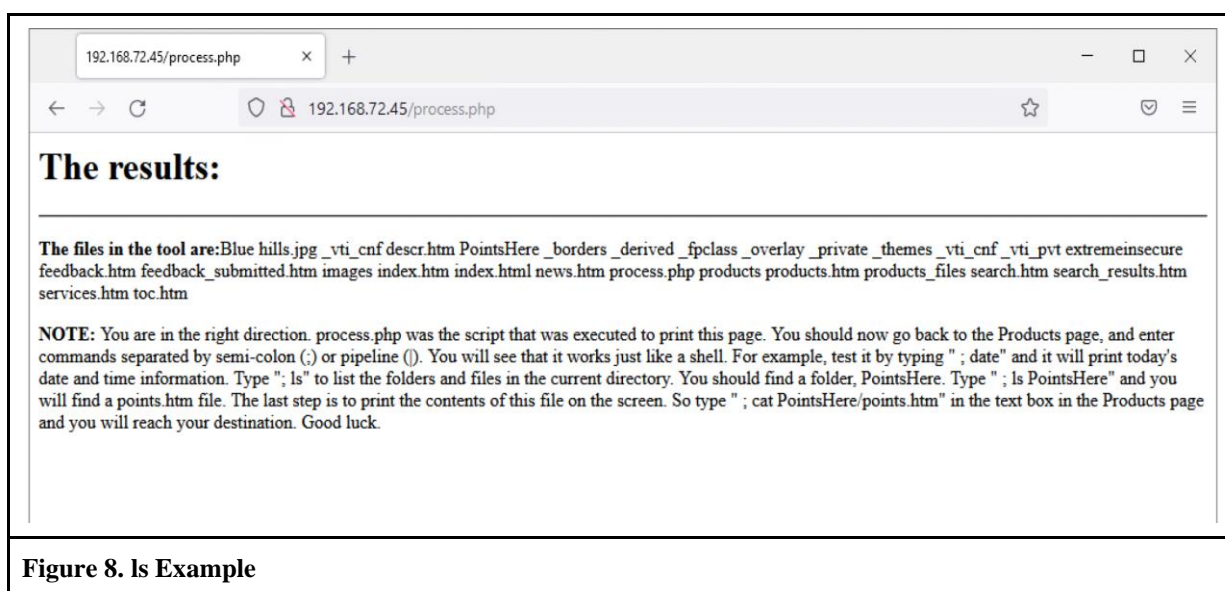


Figure 8. ls Example

The contents of the files inside the server are now being shown at the top of figure 8. This can include anything that has sensitive data to the least important piece of data. These files can be altered and changed to harm either the host or those connected to the host.

Figure 8 above stresses the most important part of the results gained from this research paper. Nothing is well protected unless the foundation of what you have built is solid. The extreme-insecure site had poorly designed code. It was destined to fail from the start. No matter if you have a website that is protecting its users' client side, if the server itself is poorly designed and allows entry from the front door of the server then it does not matter what the other host does, attackers will always get in. The team has come to the conclusion that there is no sanitization for the input.

Cross Site Script Analysis Results

The team ran Arachni to discover the cross site scripting vulnerabilities for the XSS webpage. The results are in figure 9 below.



Figure 9. Arachni XSS Scan

As you can see from figure 9 the team had uncovered two different types of cross site scripting vulnerabilities. We had cross site scripting in script context and cross site scripting. Both of these vulnerabilities can be explained in the same context.

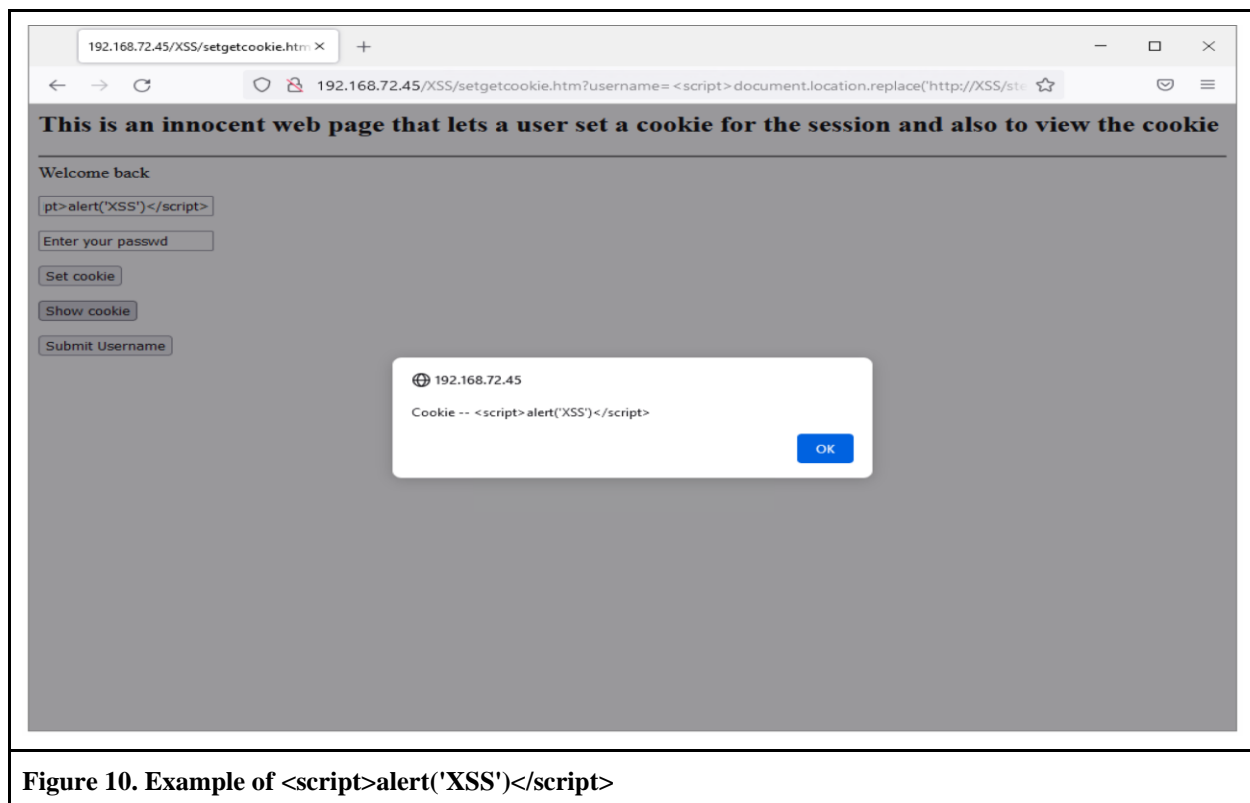


Figure 10. Example of `<script>alert('XSS')</script>`

For the cross site script analysis the team ended up using both the vulnerability scanner Arachni and manually testing for XSS vulnerabilities. Using javascript codes such as `<script>alert('XSS')</script>` will allow the team to manually check for XSS scans besides using website vulnerability scanners. To perform this test we used a reflected XSS attack to test this vulnerability. When we get an alert, we know that this web page has an XSS vulnerability.



Figure 11. Stored Script

This could be used for a variety of attacks. The reason this is possible, as alerted to us by Arachni, is because of the site taking non-trusted data and performing no checks or sanitization of any kind. The above figure 11 vulnerability is an example of reflected XSS vulnerability on our team's web server.

Solutions & Countermeasures

Detection

While the recognition of vulnerabilities and the prevention of a PHP injection or XSS attack is paramount, detecting attempted attacks can make a website aware that there are users attempting to exploit these vulnerabilities in the first place, and potentially identify the users involved. The first form of defense you have is knowing the vulnerability is there at all. The team as seen from the analysis results section of this paper would recommend using a vulnerability scanner to increase the likelihood of a website owner detecting the vulnerabilities before they become a problem. Detecting an attempted PHP injection or XSS attack can be as simple as reviewing a history of user inputs - but for a website that receives thousands of inputs daily, human review is simply not viable.

However, most PHP injections and XSS attacks make use of special characters; a program could be made to parse each input, looking for inputs with these characters, or with greater sophistication, actively look for certain input formats that would cause a malformed query if the input was unsanitized. However, in inputs that regularly make use of these special characters and thousands of inputs a day, this still leaves a great potential for false positives and an overly exhaustive list of flagged inputs for a human to review.

To be secure, file system calls should not be used when taking input from a user. After initializing the server and its code, such methods -- manual or using web application vulnerability scanners -- should be used to test the framework.

Finally, building on the previous method, an injection or XSS attack could be detected by running the unsanitized, unsafe input, rolling back any changes that occurred, and then running the sanitized input; each time, any changes to an associated database would be recorded. Afterwards, the changes could be compared - and if the unsafe and safe input produced different results, the input is most likely an attempted injection attack. This method is most useful when the inputs are modifying a database.

Prevention

Sanitization and Input Validation

Both PHP injections and XSS attacks exploit a common vulnerability: a lack of input validation and sanitization. All of the analysis results can be linked to this. File inclusion, operating system command injection, path traversal, and cross site scripting all suffer from this problem in the instance of the team's environment while researching. Without both of these working in unison, the web application, or even the entire server, is at risk. An input in this context is any data sent to the web application or server by a user - a good example of this is a textbox on an application, but inputs can take a wide range of forms, including URLs via GET request parameters, POST requests, cookies, or even the names of user-generated files. In order to ensure that user input is harmless, it must be validated and sanitized.

Input validation is recognizing malicious code by checking input against a set of criteria, typically using regular expressions in order to parse strings. The way input validation is accomplished depends largely on how the input is intended to be used by a non-malicious user. For example, if an input is provided for a user to write their age, it should only allow input that is less than three characters, is a number, and doesn't contain special characters, syntax, or other whitespace. This type of validation is called whitelist validation, where the only input allowed is checked against criteria that indicate it is clean. It is typically the most secure and robust type of input validation; however it struggles with inputs that are meant to be complex, varied, or unique, or where the complete set of allowed inputs cannot be determined. In these cases, blacklist validation is more appropriate. Blacklist validation is a type of input validation where input is checked against a set of illegal inputs, and discarded if it matches. If the example above used a blacklist, instead of allowing input that is less than three characters, it would discard input that is greater than three characters. A blacklist is generally less secure than a whitelist, as the list of illegal inputs is very large, and so would likely be incomplete. Ideally, black listing and whitelisting would be used in conjunction.

Once an input fails validation, it can either be rejected, producing an error message for the user, or it can be sanitized. Input sanitization is modifying, and thereby ‘cleaning’ inputs of any potential malicious code by removing any illegal inputs while preserving the original content of the input; because of this, it is almost always used with blacklist validation. The most common form of input sanitization is escaping metacharacters; this means identifying and telling the code interpreter to ignore any characters that have a special meaning within the code. For example, when using HTML, if an input contains ‘>’, it could potentially contain an attempted PHP injection or XSS attack, but by replacing the ‘>’ with ‘<’, the potential malicious code is prevented from executing, while also appearing identical when displayed to the user. When the data will be output into HTML, this process can be done automatically in PHP by using the function `htmlspecialchars()` to convert to HTML entities such as ‘<’ whenever the browser will output an input that came from a user, or if the data will be output to SQL, the function `mysql_real_escape_string()` will escape special characters for SQL.

Finally, there are certain PHP functions that should be avoided if possible in contexts where user input is used, as they increase the vulnerability of PHP injection by taking input, directly executing shell commands with that input, and displaying the output. These include, but are not limited to, functions such as `system()`, `passthru()`, `show_source()`, `exec()`, `shell_exec()`, `proc_open()`, `popen()`, `curl_exec()`, `curl_multi_exec()`, and `parse_ini_file()`, which if compromised would allow users to run any shell command they input.

An added input to this solution is to also use a code checker. A static code checker can help you from making fatal mistakes in the development process of the website. A human can only do so much and while a code checker may not gather all executive code running without sanitation it does at catch some vulnerabilities in the code you may have had to deal with if you had not run one.

Firewall

This is one of the most basic steps for securing your server. The team has seen based on the network topology, this server does not have a firewall outside of it protecting it from the networks outside. There is nothing to block the ports which allows packets to be sent outside. Although you could directly see the files from the website, could you directly send them out if there were mail ports blocked? No, the team surmised that when using a firewall to block the packets from getting outside it would at least help cover some of the vulnerability by not allowing sensitive data to be sent outside of the network via an SMTP. While it may not fully cover them from being able to see the data or get their hands on it, it would at least not allow it to be directly sent from the server to the attacker’s network. If you have no need for certain ports make sure to secure them.

Conclusion

PHP Injection and Cross-Site Scripting attacks are both vulnerabilities that, initially, seem relatively simple to patch. Despite this, they’ve affected Google, Facebook, and virtually every other big name tech company – simply because it only takes a single, innocuous, improperly sanitized user-supplied input to grant a malicious attack access and put server-side or client-side data at risk. The team through our initial analysis was led to the issue at the root of both of these vulnerabilities is poor input validation and sanitization on any code that uses parameters that are external to the code – be they inputs into an online form, or an external resource – and executes a command using those parameters. As this team has shown, exploiting this can be used to view pages or files that are restricted or not normally visible, as well as copying information such as passwords, injecting arbitrary code in order to change the website visible to users or execute shell commands; in short, these attacks are extremely serious and can be very damaging.

Within this paper we conducted research into vulnerabilities on the website. Our first step had the team analyze the network topology to understand how threats could get inside in the first place. The team then had to process a vulnerability scanner to identify the key vulnerabilities within the system. We then analyzed the steps ourselves and broke them down to understand the report from the vulnerability scanner. We also thought of solutions to either negate or stymie the amount of vulnerabilities within the website. Lastly we summarized the ways to help prevent vulnerabilities from occurring on the system.

Recommendations

Because vulnerabilities exist where unprotected code uses external data that can be modified to hijack it, interacting with user data only when absolutely necessary is the best defense. Barring that, ensuring every area where data from a user is used or outputted, no matter how minor, is thoroughly validated and sanitized will make it much more difficult for an attacker to find an exploit. After a website is properly defended, testing these defenses is also very important, as not every vulnerability will be obvious; this can be done with an application security scanner framework such as Arachni. Additionally, avoid using PHP functions that directly execute shell commands with user input.

Implications

PHP injection and cross site scripting are among the most common and serious vulnerabilities in web applications at present, and have been for some time. It remains to be seen how these security threats will evolve over time, as attackers find new ways to detect vulnerabilities and subvert existing countermeasures to prevent code injections, just as new advances in detecting and patching vulnerabilities emerge. Future research could be done on how well, with specific statistics, these methods could actually work regarding preventing vulnerabilities.

References

- A Pentester's guide to code injection: Cobalt blog*. Cobalt. (n.d.). Retrieved November 18, 2021, from <https://cobalt.io/blog/a-pentesters-guide-to-code-injection>
- Abikoye, O. C., Abubakar, A., Dokoro, A. H., Akande, O. N., & Kayode, A. A. (2020). A novel technique to prevent SQL injection and cross-site scripting attacks using Knuth-Morris-Pratt string match algorithm. *EURASIP Journal on Information Security*, 2020(1), 14. <https://link.springer.com/article/10.1186/s13635-020-00113-y>
- Apache Http Server: CVE security vulnerabilities, versions and detailed reports. (n.d.). Retrieved October 5, 2021, from https://www.cvedetails.com/product/66/Apache-Http-Server.html?vendor_id=45
- Diagrams.net*. App Diagrams. <https://app.diagrams.net/>. Accessed 1 October, 2021.
- Drake, Victoria. *Threat Modeling*. 1 December, 2001. https://owasp.org/www-community/Threat_Modeling. Accessed 26 September, 2021
- Fonseca, José, et al. "Testing and Comparing Web Vulnerability Scanning ... - IPG." *Testing and Comparing Web Vulnerability Scanning Tools for SQL Injection and XSS Attacks*, <http://bdigital.ipg.pt/dspace/bitstream/10314/3533/1/Fonseca-CompSQLXSS.pdf>.
- Halfond, William. Orso, Alessandro. Viegas, Jeremy. *A Classification of SQL Injection Attacks and Countermeasures*. Georgia Institute of Technology, 2006, <https://www.cc.gatech.edu/fac/Alex.Orso/papers/halfond.viegas.orso.ISSSE06.pdf>. Accessed 26 September, 2021
- PHP Command Injection: Examples and Prevention. (n.d.). StackHawk. Retrieved October 4, 2021, from <https://www.stackhawk.com/blog/php-command-injection/>
- PHP: SQL Injection—Manual*. (n.d.). Retrieved October 2, 2021, from <https://www.php.net/manual/en/security.database.sql-injection.php>
- Pauli, J. J. (2013). *The basics of web hacking: Tools and techniques to attack the Web*. Syngress, an imprint of Elsevier. Retrieved October 3, 2021, from [http://index-of.co.uk/Hacking/Josh%20Pauli%20\(Auth.\)-The%20Basics%20of%20Web%20Hacking.%20Tools%20and%20Techniques%20to%20Attack%20the%20Web%20\(2013\).pdf](http://index-of.co.uk/Hacking/Josh%20Pauli%20(Auth.)-The%20Basics%20of%20Web%20Hacking.%20Tools%20and%20Techniques%20to%20Attack%20the%20Web%20(2013).pdf)
- Sahu, D. R., & Tomar, D. S. (n.d.). *DNS Pharming through PHP Injection: Attack Scenario and Investigation*. <https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.993.174&rep=rep1&type=pdf>
- Visual Paradigm*. Online-Visual Paradigm, <https://online.visual-paradigm.com/app/diagrams/#diagram:proj=0&type=ThreatModelDiagram&gallery=/repository/b025dca9-184d-4f18-9d80-abfadb176ec3.xml&name=Website%20Threat%20Modeling>. Accessed 1 October , 2021.
- Vogt, P., Nentwich, F., Jovanovic, N., Kirda, E., Kruegel, C., & Vigna, G. (n.d.). *Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis*. 12. https://people.scs.carleton.ca/~soma/id-2007w/readings/ndss07_xssprevent.pdf