SURVIVAL GUIDE:

Preventing 7 Critical Types of Software Attacks

www.brightsec.com
Introduction

Application security is crucial because applications serve as gateways to valuable data and resources, making them prime targets for cyberattacks. A security breach can lead to financial losses, reputational damage, legal consequences, and compromise of sensitive information.

By prioritizing application security, organizations can mitigate these risks and protect their users, customers, and assets. Robust security measures, such as secure coding practices, regular vulnerability assessments, and strong access controls, help identify and address vulnerabilities before they can be exploited.

While healthcare, financial services and high tech have the highest attack costs, application attacks affect all industries. The average cost of a data breach is now over $4M according to recent research from IBM.¹

Mitigating application attacks requires a multi-layered approach to enhance the security of your applications. Most attacks can be mitigated with secure coding practices. Following secure coding guidelines, such as input validation, output encoding, and proper error handling, to prevent common vulnerabilities like injection attacks (e.g., SQL injection, XSS) is the leading proactive approach to securing applications.

It is important to note that there is no single strategy to avoid and mitigate application attacks but rather a combination of mitigating activities. This Survival Guide will introduce seven of the most common application attacks and suggested mitigation strategies.

Server-Side Request Forgery (SSRF)

Server-side request forgery (SSRF) attacks pose a significant threat, enabling attackers to deceive server-side applications into granting unauthorized access or tampering with files. These attacks exploit the lack of proper input sanitization when reading data from URLs.

When an SSRF vulnerability exists, attackers can manipulate URLs to send HTTP requests to specific domains. They achieve this by modifying URL paths or entirely replacing URLs. Notably, SSRF attacks commonly exploit URLs pointing to internal services within an organization's infrastructure, which should remain inaccessible to outsiders. However, attackers exploit SSRF to gain entry into these sensitive URLs.

Once successful, SSRF attacks can result in unauthorized access to crucial organizational data, including valuable login credentials. The consequences can extend to the compromised web application, the underlying backend system, or even external servers the application interacts with.

SSRF attacks exploit input sanitization flaws to deceive server-side applications, granting unauthorized access and file tampering. Attackers manipulate URLs by modifying paths or replacing them, targeting internal services. Despite their intended inaccessibility, SSRF allows entry to these sensitive URLs. Successful attacks yield unauthorized access to vital organizational data, including valuable login credentials. Consequences affect the compromised web app, backend system, and external servers.

SSRF attacks grant attackers privileged access to various resources. Hackers leverage SSRF to target:

- Private IP addresses
- Internal resources protected by firewalls
- Server loopback interfaces, such as "localhost" (http://127.0.0.1)
- Internal APIs
- Privileged files on vulnerable servers
- Local ports (identified through port scanning)

Furthermore, SSRF aids attackers in concealing the true origin of their connections. By masquerading as the local application's backend, they can avoid detection and gain unauthorized access to protected resources.

Consequences of SSRF Attacks

**SQL injection**
By leveraging the access gained through SSRF, attackers can target the application's database and carry out SQL injection attacks, potentially extracting or manipulating sensitive information.

**SQL injection**
By leveraging the access gained through SSRF, attackers can target the application's database and carry out SQL injection attacks, potentially extracting or manipulating sensitive information.

**Remote code execution (RCE)**
Exploiting SSRF can provide attackers with local access to the server hosting the application. This enables them to gain full shell access and exploit any underlying vulnerabilities in the operating system, granting them extensive control over the compromised system.

**Exploiting other web application vulnerabilities**
In conjunction with SSRF, attackers may combine additional vulnerabilities present in the web application, such as XXE (XML External Entity), XSS (Cross-Site Scripting), or CSRF (Cross-Site Request Forgery). This combination amplifies the impact and scope of the attack, allowing for further compromise of the targeted system.
Mitigating and preventing SSRF attacks

Mitigating and preventing Server-Side Request Forgery (SSRF) attacks is of paramount importance in today's security landscape. SSRF attacks pose a significant threat to web applications, allowing attackers to exploit vulnerabilities and gain unauthorized access to internal resources or sensitive information. By adopting a proactive approach and implementing comprehensive mitigation strategies, businesses can significantly reduce the risk of SSRF attacks and protect their critical systems and data from compromise.

Outlined below are six mitigation strategies. Implementing a combination of these mitigation strategies strengthens the overall defense against SSRF attacks, safeguarding applications and sensitive data.

Firewalls

Enforcing firewall policies specifying allowed host connections is a common mitigation strategy. However, host-based firewalls may struggle to differentiate between legitimate application connections and those initiated by other software on the same node. Firewalls can also have limitations in blocking outbound connections while allowing connections within the same network segment.

HTTP CONNECT Proxy

Overcoming firewall limitations, an HTTP CONNECT proxy can forward all traffic and apply access control lists (ACLs) to regulate allowed destinations. This approach relies on the application supporting HTTP CONNECT and routing traffic accordingly.

Application Layer Controls

Implementing application-level controls involves checking if a target address is permitted before establishing a connection. However, it is crucial to address "time-of-check" and "time-of-use" vulnerabilities, where attackers manipulate DNS queries to change the target address. Lower-layer hooks with classless inter-domain routing (CIDR) checks and restrictions on HTTP redirects can help mitigate these vulnerabilities.

Whitelists and DNS Resolution

Creating an allowlist of approved hostnames or IP addresses is an effective prevention measure. If an allowlist is not feasible, a denylist can be used with proper validation of user input. Avoid requests to endpoints with private (non-routable) IP addresses and customize the denylist based on your application and environment characteristics.

Authentication on Internal Services

Enabling authentication for all internal services, including caching systems and NoSQL databases, prevents unauthorized access via SSRF. Ensure all services within your local network require authentication, aligning with the zero-trust security approach.

Harden Cloud Services

Cloud service providers like AWS and Azure offer SSRF mitigation measures through hardened configurations. For instance, AWS restricts access to cloud service metadata from containers, reducing the attack surface.
Code Injection

Code injection attacks exploit vulnerabilities in computer programs by introducing malicious code, altering the program's execution. These attacks pose significant risks, enabling the propagation of viruses, worms, data corruption, denial of access, or even complete host takeover.

In the case of the PHP language, the language allows serialization and deserialization of objects. However, when untrusted input is introduced into a deserialization function, it opens the door for attackers to overwrite existing programs and execute malicious attacks.

This vulnerability highlights the importance of implementing robust security measures to prevent unauthorized code injection and protect the integrity and security of PHP applications.
Mitigating and Preventing PHP Code Injection

Preventing PHP code injections requires implementing effective security practices. Here are some recommendations:

**Avoid using direct system commands**

Functions like exec(), shell_exec(), system(), and passthru() provide direct access to the operating environment, making the web server stack vulnerable to malicious activity. Instead, leverage safer alternatives and built-in PHP functions, such as the ZipArchive class for archiving operations.

**Use robust input sanitization**

Ensure proper validation and sanitization of user input to prevent attacks. Avoid weak sanitization methods like strip_tags() and htmlspecialchars(), which may still allow certain malicious payloads. Implement thorough input validation and consider using specialized libraries or frameworks that offer stronger sanitization mechanisms.

**Disable verbose error messages**

Turn off PHP errors in your PHP.ini configuration by disabling the error_reporting modes E_ALL, ~E_NOTICE, and ~E_WARNING. This prevents sensitive information about your PHP application and web server from being exposed through error outputs, reducing the attack surface.

**Employ a PHP security linter**

Utilize a PHP security linter, such as PHPlint, to scan your code for errors and potential security flaws. PHPlint offers comprehensive checks for PHP 7 and PHP 8, providing detailed feedback on discovered issues. Run the linter regularly during development to identify and address security vulnerabilities early on.

By following these best practices, you can significantly enhance the security of your PHP applications and reduce the risk of code injection attacks.
Cross-site request forgery (CSRF) is a dangerous cyber attack technique that involves hackers impersonating legitimate, trusted users to perform unauthorized actions. These attacks can have severe consequences, such as altering firewall settings, injecting malicious data into forums, or conducting fraudulent financial transactions.

One of the most concerning aspects of CSRF attacks is that the targeted users often remain unaware that an attack has taken place. By the time they realize it, the damage may have already occurred, and recovery might be difficult or even impossible.

CSRF attacks exploit a browser-based process that enhances convenience during login to web applications. When a user accesses a site after logging in, the browser typically keeps them signed in by passing an authentication token. This token contains various information, including session cookies, basic authentication credentials, IP address, and even Windows domain credentials.

However, the problem arises when these authentication tokens lack proper validation. This vulnerability allows attackers to easily steal the token and impersonate the user, as the website fails to differentiate between a fake user request and a legitimate one. To address this issue, CSRF tokens have been implemented in all web frameworks. These tokens enable websites to verify the validity of a session token, thereby providing an additional layer of protection against CSRF attacks.
Mitigating and preventing CSRF

Here are some techniques that can help prevent and mitigate CSRF attacks.

Token Synchronization

CSRF tokens play a crucial role in mitigating CSRF attacks by ensuring that attackers cannot make unauthorised requests to the backend without valid tokens. These tokens should possess certain characteristics to enhance their security: they must be secret, unpredictable, and unique to each user session.

To maximise security, it is recommended that the server-side generates CSRF tokens. Each user request or session should have its own unique token. Using separate tokens per request, rather than per session, offers enhanced security by reducing the window of opportunity for attackers to exploit stolen tokens. By limiting the validity of tokens to specific requests, the impact of token theft can be significantly minimised.

Double-Submitting Cookies

The double-submit cookie method serves as an alternative to managing the CSRF token state on the server-side, addressing potential challenges associated with server-side token management. This stateless technique is easy to implement and involves sending random values twice: once as request parameters and again in cookies. The server then verifies that the two random values from the request parameter and cookie match.

To implement this method, it is advisable to generate a strong, cryptographically random value as a separate cookie on the user's device before authentication. This additional cookie, combined with the session identifier, adds an extra layer of defense, requiring all transaction requests to include the random or pseudo random value. On the server-side, requests are considered legitimate only if both cookies match; otherwise, they are rejected.

To enhance the security of this technique, including tokens in encrypted cookies can be beneficial. Upon decryption, the server compares each cookie with the hidden token in a form field or AJAX parameter call. This approach ensures that subdomains cannot overwrite encrypted cookies unless they possess specific information like an encryption key.

Another option to strengthen the technique is to use cookies containing hash-based message authentication code (HMAC) values. HMAC-based tokens allow the tokens to include secret keys known only to the server. They function similarly to encrypted cookies, relying on server-exclusive information, but with less computational overhead compared to encryption and decryption. Both encrypted and HMAC-based cookies effectively mitigate CSRF attacks as attackers lack the necessary knowledge to recreate cookie values from stolen tokens.

Same-Site Cookies

Same-site cookies are a valuable defense against CSRF attacks as they restrict the cookies sent along with each request, mitigating the risk posed by HTML elements that can transmit cookies. By utilizing same-site cookies, developers can ensure that only specific cookies are allowed to accompany a request.

When a web application sets cookies on a website, the browser stores various elements within the cookies. Apart from the key-value data, cookies contain a domain field that helps differentiate between first-party and third-party cookies. A first-party cookie has a domain field that matches the URL displayed in the browser’s address bar, while the domain of a third-party cookie does not match the URL. First-party cookies are commonly used by web applications to store session information, whereas third-party cookies are often utilized by analytics tools.

Same-site cookies introduce an additional field that specifies whether the browser is permitted to send a first-party cookie with requests originating from HTML elements located on different URLs. This mechanism empowers the application to restrict requests only to sites with matching URLs, enhancing security by preventing unauthorized cross-origin requests.
Enabling User Interaction

Although CSRF attacks typically do not require user interaction, involving users in the security process can help enhance transaction security in certain scenarios. By incorporating user interaction, unauthorized users, including CSRF attackers, can be prevented from performing operations without proper authorization. Requiring user interaction can be achieved through various mechanisms such as re-authentication, CAPTCHA challenges, and one-time tokens, offering robust protection against CSRF attacks.

When implemented effectively, these techniques add an extra layer of security, ensuring that critical operations like financial transactions, account modifications, or password changes can only be executed with explicit user involvement. However, it's important to strike a balance between security measures and user experience, as overly burdensome requirements can negatively impact usability.

Implementing user interaction as part of the CSRF prevention strategy is particularly beneficial for high-risk activities where the potential consequences of unauthorized access are significant. By incorporating user verification steps, organizations can reinforce the security of sensitive operations while still providing a streamlined user experience for routine tasks.

Custom Headers for Requests

Utilizing custom request headers is a highly effective defense mechanism, particularly for securing API and AJAX endpoints against CSRF attacks. This technique leverages the same-origin policies (SOPs) to restrict the inclusion of custom headers. Users can only add custom headers using JavaScript within their own origin, providing an additional layer of protection.

By default, most browsers do not allow JavaScript to include custom headers in cross-origin requests. This restriction can be leveraged to prevent CSRF attacks. Developers can validate the presence of custom headers and their values on the server-side across AJAX endpoints, ensuring the requests originate from trusted sources.

Implementing this approach typically does not require maintaining server-side state data or making significant changes to the user interface. It is particularly well-suited for REST services, as developers can easily add custom headers (along with their corresponding values).

It is important to note that while this method effectively secures AJAX calls, it may not be sufficient for protecting <form> tags, which often require additional security measures like CSRF tokens. Additionally, to ensure the effectiveness of this solution, a robust Cross-Origin Resource Sharing (CORS) configuration must be implemented. If a request from another domain includes a custom header, it will trigger a preflight CORS check, providing an added layer of protection against unauthorized access.

Conduct Regular Web Application Security Tests to Identify CSRF

Addressing vulnerabilities and securing web applications against CSRF attacks is an ongoing process. Even after implementing measures to mitigate CSRF, application updates and code changes can reintroduce potential vulnerabilities. To ensure continuous protection, it is crucial to conduct regular web application security tests that specifically target CSRF vulnerabilities.

Dynamic Application Security Testing (DAST) is a valuable approach for scanning and testing web applications to identify potential security weaknesses, including CSRF vulnerabilities. By employing DAST tools, you can systematically evaluate your application's security posture and identify any gaps or vulnerabilities that may expose it to CSRF attacks.

Regular security testing allows you to proactively detect and remediate CSRF vulnerabilities before they can be exploited by attackers. It provides a comprehensive assessment of your application's security controls and helps ensure that any new features or changes do not inadvertently introduce CSRF risks.

By integrating regular web application security testing, including CSRF-focused assessments, into your development and maintenance processes, you can maintain a strong defense against CSRF attacks and continuously improve the security of your web applications.
Cross-site scripting (XSS) is a malicious technique that involves injecting and executing malicious code within a vulnerable web application. Unlike other attack vectors such as SQL injections, XSS primarily targets the application’s users rather than the application itself.

Successful XSS attacks can have severe consequences, causing significant damage to websites and web applications, tarnishing their reputation and eroding customer trust.

These attacks can lead to various detrimental outcomes, including:

1. **Website defacement:**
   Attackers can modify the appearance and content of websites, defacing them and impacting the user experience.

2. **Compromised user accounts:**
   XSS can be leveraged to steal user credentials, gain unauthorized access to user accounts, and potentially exploit personal information or perform fraudulent activities.

3. **Execution of malicious code:**
   Attackers can inject and execute malicious scripts on web pages, which can further compromise users’ devices, leading to data breaches or unauthorized control.

4. **Session hijacking:**
   If XSS exposes session cookies, attackers can hijack user sessions and impersonate legitimate users. This enables them to perform any actions authorized to the compromised user, including sensitive operations like financial transactions or administrative actions.
XSS: Attack Types

Reflected Cross-Site Scripting

Reflected XSS is a simple form of cross-site scripting that involves an application “reflecting” malicious code received via an HTTP request. As a result of an XSS vulnerability, the application accepts malicious code from the user and includes it in its response.

Stored/Persistent Cross-Site Scripting

Stored XSS involves an application receiving data from a malicious source and storing the data for use in later HTTP responses. This is also known as second-order or persistent XSS, because it persists in the system.

The data can come from any untrusted source that sends an HTTP request to the application, such as comments posted on a blog or an application that displays email messages using SMTP.

DOM-based Cross-site Scripting

DOM-based XSS is an attack that modifies the domain object model (DOM) on the client side (the browser). In a DOM-based attack, the HTTP response on the server side does not change. Rather, a malicious change in the DOM environment causes client code to run unexpectedly.

Mitigating and Preventing XSS Attacks:

1. Implement Content Security Policy (CSP):

To effectively mitigate reflected and stored cross-site scripting, it is important to adopt a whitelist-based approach instead of relying on blacklisting unsafe characters. Here are some recommended practices for input sanitization:

- Utilize a reliable security encoding library to encode all parameters and user input. Apply HTML escaping before inserting parameters/user input data into the HTML body.

- Employ functions like `htmlentities()` to encode characters that can impact the execution context, such as those indicating the start of a script, event, or CSS style.

- When inserting parameters/user input data into HTML attributes, escape the attribute values. Avoid using event handlers or attributes like `href`, style, or `src`.

- Always use quotes for attributes since quoted attributes can be properly escaped with the corresponding quote. As a general rule, escape all non-alphanumeric characters.

- For dynamically generated JavaScript code, employ JavaScript escaping. Place parameters/user data input only inside quoted values to maintain the proper context and avoid switching contexts.

2. Use HTTPOnly Cookie Flag:

- Set the HTTPOnly flag on session cookies and any custom cookies not accessed by JavaScript code.

- Note that while the HTTPOnly flag is enabled by default in .NET applications, it may require manual enabling in other programming languages.
3. Implement Content Security Policy (CSP):

Content Security Policy (CSP) is a powerful defense mechanism against XSS vulnerabilities. By creating lists that specify access permissions to client-side resources such as JavaScript and CSS, CSP instructs the browser to execute resources only from trusted sources. Consider the following example:

```
Content-Security-Policy:
default-src 'self'; script-src 'self' static.domain.tld
```

This CSP header ensures that all resources are loaded only from trusted sources, as exemplified by static.domain.tld.

4. X-XSS-Protection Header:

- The X-XSS-Protection header is available in popular browsers like Google Chrome and Internet Explorer. It helps filter suspicious content to prevent reflected XSS attacks. However, reliance solely on this header may introduce additional client-side security risks. It is recommended to set the header to X-XSS-Protection: 0, disabling the XSS Auditor and preventing it from following the default response behavior of the browser.

- DOM XSS poses unique challenges as sanitization needs to be performed on the client-side. Consider the following best practices:

5. Sanitizing Inputs:

- HTML escape and then JavaScript escape any parameter or user data input before inserting it into the HTML subcontext within the execution context.

- When inserting into the HTML attribute subcontext within the execution context, apply JavaScript escape.

- Avoid including any volatile data (e.g., parameters/user input) in event handlers and JavaScript code subcontexts within the execution context.

6. Using the Correct Output Method:

- Utilize appropriate browser API output methods. For instance, when accepting content from user inputs in a `\<div>`, use functions like `\innerText` instead of `\innerHTML`, as they sanitize the content.

- Avoid manual encoding of the output. Use `\element.textContent` to display user-provided content, ensuring DOM XSS vulnerabilities are mitigated. An example provided by OWASP is as follows:

```html
<b>Current URL:</b> <span id="contentholder"></span>
...
<script>
    document.getElementById("contentholder").textContent = document.baseURI;
</script>
```

By following these practices, you can significantly enhance the security of your web applications against XSS attacks, both at the server-side and client-side levels.
XML External Entity (XXE)

XXE (XML External Entity Injection) is a critical web-based vulnerability that allows malicious actors to manipulate XML data processing in a web application. By injecting specially crafted XML entities, an attacker can exploit this vulnerability to gain unauthorized access and perform various malicious actions.

One of the primary risks associated with XXE is the potential exposure of sensitive files on the application server’s file system. Attackers can leverage XXE to retrieve valuable information, such as configuration files, credentials, or other sensitive data residing on the server.

Furthermore, XXE can also be used as a stepping stone to launch additional attacks, such as Server-Side Request Forgery (SSRF). By manipulating the XXE payload, an attacker can trick the application into making requests to external systems or internal resources that the application itself has access to. This can lead to severe consequences, including compromise of the underlying server infrastructure or unauthorized interactions with backend systems.

To mitigate XXE vulnerabilities, it is essential to follow security best practices. This includes validating and sanitizing XML inputs, disabling external entity processing, and implementing strict input validation and whitelisting techniques. Additionally, employing a robust web application firewall (WAF) and conducting regular security assessments can help identify and address potential XXE vulnerabilities.

By understanding the risks associated with XXE and implementing proactive security measures, organizations can protect their web applications and the underlying infrastructure from exploitation and unauthorized access.
Impact of XXE Injection

These attacks can lead to various detrimental outcomes, including:

- **Data theft:**
  XXE injections can allow attackers to extract sensitive data, such as passwords, confidential documents, or personal information, from a target system.

- **System compromise:**
  XXE injections can be used to gain unauthorized access to systems and data, allowing attackers to execute malicious code, install malware, or steal sensitive data.

- **Denial of Service (DoS) attacks:**
  XXE injections can be used to launch DoS attacks, overwhelming target systems and making them unavailable to users.

- **Reputation damage**
  XXE injections can result in the loss of sensitive data and the compromise of systems, which can damage an organization's reputation and impact customer trust.

- **Increased risk of future attacks:**
  XXE injections can create a foothold for attackers within a target system, making it easier for them to carry out additional attacks in the future.

How to Detect XXE Attacks

Detection of XXE attacks can be performed in a few ways:

- **Manually:**
  This involves manually reviewing XML input files, server logs, and network traffic to identify any potential XXE attacks. This can be challenging because it requires expertise in understanding how XML parsers work and the various types of XXE attacks, as well as a good understanding of the specific system being monitored.

- **Using SCA tools:**
  Static Code Analysis (SCA) tools can be used to scan code and identify any potential XXE vulnerabilities before the code is deployed. These tools can help identify common patterns in code that could lead to XXE attacks and provide suggestions for remediation.

- **Code scanning in early development stages:**
  Integrating code scanning into the development process can help identify XXE vulnerabilities early in the development lifecycle. This can reduce the risk of these vulnerabilities being exploited and minimize the potential impact of a successful attack.

- **Reputation damage**
  XXE injections can result in the loss of sensitive data and the compromise of systems, which can damage an organization's reputation and impact customer trust.

- **Increased risk of future attacks:**
  XXE injections can create a foothold for attackers within a target system, making it easier for them to carry out additional attacks in the future.

Regardless of the method used, monitoring and testing is important to ensure that systems remain secure and free from XXE attacks.
Mitigating and preventing XXE

XXE vulnerabilities pose a significant threat to web applications, often arising due to the inherent support for XML features in the underlying parsing library. These features, while useful in certain scenarios, can be dangerous when misused by malicious actors. To mitigate the risk of XXE attacks, disabling these features is crucial and can be achieved through configuration or programmatic means.

First and foremost, it is essential to disable external entities resolution. External entities allow the inclusion of external resources within an XML document, which can be exploited by attackers to disclose sensitive information or launch further attacks. By disabling external entities resolution, the application prevents unauthorized access to external resources.

Similarly, disabling XInclude support is important. XInclude is an XML language feature that enables the inclusion of content from external XML files. This feature, when abused, can lead to XXE vulnerabilities. Disabling XInclude support ensures that the application does not process external XML content, reducing the attack surface.

These preventive measures can be implemented both at the configuration level and programmatically. In the configuration, the XML parsing library settings should be adjusted to disable external entities resolution and XInclude support. By overriding the default behaviors, the application enforces strict restrictions on XML processing and mitigates the risk of XXE attacks.

It is worth noting that careful consideration should be given to the specific XML parsing library and its associated configuration options, as they may vary across frameworks and programming languages. Consulting the documentation and best practices provided by the library vendor is recommended to ensure the correct approach is followed.
DNS (Domain Name System) is a fundamental protocol that plays a crucial role in translating human-readable domain names into the corresponding IP addresses that computers use to communicate on the internet. It acts as a global distributed database that maps domain names to their associated IP addresses, allowing users to access websites and services using familiar names instead of numerical IP addresses.

When a user enters a domain name, such as website.com, into a web browser, the DNS process begins to resolve the domain name to its corresponding IP address. Here’s a breakdown of how it works:

**DNS Resolver:**

The user’s computer or network device contains a DNS resolver, which is responsible for handling DNS queries. The resolver first checks its local cache to see if it already has the IP address for the requested domain name. If it finds a match, it can immediately provide the IP address without further queries.

**DNS Query:**

If the resolver does not have the IP address in its cache, it sends a DNS query to a DNS server. The resolver typically sends the query to the DNS server provided by the user’s internet service provider (ISP). The query includes the domain name that needs to be resolved.

**Recursive Resolution:**

The DNS server receiving the query performs a recursive resolution process. It checks its own cache to see if it has the IP address for the domain name. If not, it starts querying other DNS servers in a recursive manner until it finds a server that can provide the authoritative answer.
**Authoritative DNS Server:**

The recursive DNS server eventually reaches an authoritative DNS server that holds the accurate mapping of the domain name to its IP address. The authoritative server is responsible for a specific domain and has the final answer regarding that domain's IP address.

**Response and Caching:**

Once the authoritative DNS server is found, the IP address is returned in a DNS response. The response is sent back through the chain of DNS servers to the user's resolver. The resolver then provides the IP address to the requesting program, such as a web browser, allowing it to establish a connection to the requested website. Additionally, the resolver caches the IP address locally for future use, reducing the need for repeated DNS queries for the same domain name.

The DNS system operates in a distributed manner, with numerous DNS servers worldwide, ensuring efficient and reliable resolution of domain names to IP addresses. This translation process occurs seamlessly behind the scenes, enabling users to navigate the internet using domain names while relying on the underlying DNS infrastructure to facilitate the connection between human-readable names and machine-understandable IP addresses.

**What Are the 5 Major DNS Attack Types?**

**DNS Tunneling**

DNS tunneling is a technique where attackers encode data from various programs or protocols into DNS queries and responses. By leveraging compromised systems and external network connectivity, they gain access to internal DNS servers. This enables them to take control of the DNS server and execute data payloads, allowing remote server management and application manipulation.

DNS tunneling relies on controlling a server and domain that acts as an authoritative server for executing data payload programs and facilitating server-side tunneling. This method poses a significant security risk as it bypasses traditional security controls. Implementing robust monitoring, filtering, and education measures can help mitigate the threats associated with DNS tunneling.

**DNS Amplification**

DNS amplification attacks are a form of Distributed Denial of Service (DDoS) attacks that aim to overwhelm a target server. These attacks exploit open DNS servers, which are publicly accessible, to flood the target with a high volume of DNS response traffic.

In a DNS amplification attack, the attacker begins by sending a DNS lookup request to the open DNS server, falsifying the source address to appear as the target's address. The DNS server then responds with a DNS record, which is redirected to the actual target controlled by the attacker.

By leveraging the amplification effect, where a small request generates a large response, attackers can amplify their traffic and exhaust the target's resources, leading to service disruption. Mitigation measures, such as DNS rate limiting and network filtering, can help protect against DNS amplification attacks.
DNS Flood Attack

DNS flood attacks leverage the DNS protocol to execute UDP floods, overwhelming the target. Attackers generate a large volume of spoofed, yet valid, DNS request packets and create a vast group of source IP addresses.

The targeted DNS servers respond to the flood of requests, believing them to be legitimate. However, the massive influx of requests can overload the DNS server, depleting its resources. This exhaustion of network resources leads to the target's DNS infrastructure being taken offline, causing a loss of internet access for the target.

Preventing and mitigating DNS flood attacks require robust network infrastructure, traffic filtering, and rate limiting mechanisms.

DNS Spoofing

DNS spoofing, also known as DNS cache poisoning, manipulates DNS records to redirect traffic to a fake website posing as the legitimate destination. Users unknowingly visit the fraudulent site and are prompted to enter their login credentials. By doing so, they unwittingly provide the threat actor with access to their account and any sensitive information shared on the counterfeit login page.

Moreover, these malicious websites may exploit vulnerabilities to install viruses or worms on users' computers, granting the threat actor persistent access to the compromised machine and its stored data. Protecting against DNS spoofing requires implementing secure DNS configurations, monitoring DNS responses for anomalies, and educating users about potential phishing attempts.

Mitigating and preventing DNS attacks

There are several ways that can help you protect your organization against DNS attacks:

- **Keep DNS Resolver Private and Protected**

  Restrict DNS resolver usage to only users on the network and never leave it open to external users. This can prevent its cache from being poisoned by external actors.

- **Configure Your DNS Against Cache Poisoning**

  Configure security into your DNS software in order to protect your organization against cache poisoning. You can add variability to outgoing requests in order to make it difficult for threat actors to slip in a bogus response and get it accepted. Try randomizing the query ID, for example, or use a random source port instead of UDP port 53.

- **Securely Manage Your DNS servers**

  Authoritative servers can be hosted in-house, by a service provider, or through the help of a domain registrar. If you have the required skills and expertise for in-house hosting, you can have full control. If you do not have the required skills and scale, you might benefit from outsourcing this aspect.

NXDOMAIN Attack

A DNS NXDOMAIN attack floods the DNS server with numerous requests for non-existent records, aiming to exhaust server resources. DNS proxy servers are often targeted, causing them to consume their resources querying the authoritative server. This overload leads to a slowdown and eventual cessation of responses to legitimate requests, resulting in service disruption.
Local file inclusion is an attack technique in which attackers trick a web application to run or expose files on a web server. LFI attacks can expose sensitive information, and in severe cases, they can lead to cross-site scripting (XSS) and remote code execution.

File inclusions are a key to any server-side scripting language, and allow the content of files to be used as part of web application code.

**How Does Local File Inclusions Work?**

When an application uses a file path as an input, the app treats that input as trusted and safe. A local file can then be injected into the included statement. This happens when your code is vulnerable. In this case, a hacker makes a request that fools the app into executing a malicious PHP script (web shell for example).

In some cases, if the application provides the ability to upload files, attackers can run any server-side malicious code they want. Most applications do not provide this capability, and even if they do, the attacker cannot guarantee that the app saves the file on the server where the LFI vulnerability is located. The attacker will also need to know the file path to their uploaded file on the server file system.
The impact of a Local File Inclusion

The impact of a Local File Inclusion attack can vary based on the exploitation and the read permissions of the webserver user. Based on these factors, an attacker can gather usernames via an /etc/passwd file, harvest useful information from log files, or combine this vulnerability with other attack vectors (such as file upload vulnerability) to execute commands remotely.

Let’s take a closer look at three possible outcomes of local file inclusion:

Information disclosure:

Although not the worst outcome of a local file inclusion vulnerability, information disclosure can reveal important information about the application and its configuration. That information can be valuable to an attacker to gain a deeper understanding of the application and can help them detect and exploit other vulnerabilities.

Directory Traversal:

A local file inclusion vulnerability can lead to Directory Traversal attacks, where an attacker will try to find and access files on the web server to gain more useful information, such as log files. Log files can reveal the structure of the application or expose paths to sensitive files.

Remote Code Execution:

Combined with a file upload vulnerability, a Local File vulnerability can lead to remote code execution. In this case the attacker would use LFI to execute the unwanted file. To compound matters, an attacker can upload a file to the server to gain the ability to execute commands remotely, resulting in the attacker being able to control the whole server remotely.

Mitigating and preventing Local File Inclusion vulnerabilities

To prevent LFI attacks, several measures can be implemented:

- **ID assignment**
  Save your file paths in a secure database and give an ID for every single one, this way users only get to see their ID without viewing or altering the path.

- **Whitelisting**
  Use verified and secured whitelist files and ignore everything else.

- **Use databases**
  Don’t include files on a web server that can be compromised, use a database instead.

- **Better server instructions**
  Make the server send download headers automatically instead of executing files in a specified directory.
Detecting/ responding to Local File Inclusion vulnerabilities

Responding to local file inclusion vulnerabilities requires a comprehensive approach to mitigate the risk and protect your web application. Here are some steps you can take:

**Patch and update:**
Keep your web application and all associated components, frameworks, and libraries up to date with the latest security patches. Regularly check for updates and apply them promptly.

**Input validation and sanitization**
Implement strict input validation and sanitization techniques to prevent malicious input from being processed or executed as code. Validate and sanitize user-supplied data before using it in file inclusion operations.

**Whitelist allowed file paths:**
Restrict the file paths that can be accessed by the application. Use a whitelist approach, allowing only specific directories and files to be included. This helps prevent unauthorized access to sensitive files.

**Disable directory listing:**
Disable directory listing on the web server to prevent attackers from obtaining information about the file system structure.

**Use secure coding practices:**
Follow secure coding practices, such as avoiding the use of user-supplied input in file inclusion functions, utilizing secure file access methods, and implementing access controls to restrict file inclusion to authorized resources.

**Implement server-side controls:**
Configure the server to enforce security measures, such as disabling unnecessary file inclusion functions, setting proper file permissions, and implementing file access restrictions.

**Conduct security testing:**
Regularly perform vulnerability assessments and penetration testing to identify and address any remaining vulnerabilities or misconfigurations.

**Monitor and log:**
Implement logging mechanisms to monitor file inclusion activities and detect any suspicious or malicious behavior. Analyze the logs regularly to identify potential attacks or anomalies.

**Educate developers:**
Provide training and awareness programs to educate developers about secure coding practices and the risks associated with local file inclusion vulnerabilities.

By following these steps, you can enhance the security of your web application and minimize the potential impact of local file inclusion vulnerabilities.
Conclusion

Many techniques to avoid the most common application attacks were covered in this guide. There is no “silver bullet” to avoid application attacks and it requires a number of mitigating activities.

Best practices to avoid application attacks include implementing secure coding practices, regularly update and patch applications, enforce strong access controls, utilize secure configurations, conduct thorough security testing, deploy web application firewalls, provide security education and training, establish an incident response plan, ensure the security of third-party libraries, and maintaining continuous monitoring and logging of application activities to detect and respond to potential attacks promptly.

1 https://www.ibm.com/reports/data-breach